FIFTH GENERATION COMPUTER SYSTEMS 1992

Edited by
Institute for New Generation
Computer Technology (ICOT)

Volume 2

Ohmsha, Ltd. IOS Press

FIFTH GENERATION COMPUTER SYSTEMS 1992

Edited by
Institute for New Generation
Computer Technology (ICOT)

Volume 2

Ohmsha, Ltd. IOS Press

FIFTH GENERATION COMPUTER SYSTEMS 1992

ISBN 4-274-07724-1 (Ohmsha)
ISBN 90-5199-099-5 (IOS Press)

Library of Congress Catalog Card Number: 92-073166

Published and distributed in *Japan* by
Ohmsha, Ltd.
3-1 Kanda Nishiki-cho, Chiyoda-ku, Tokyo 101, Japan

Distributed in *North America* by
IOS Press, Inc.
Postal Drawer 10558, Burke, VA 22009-0558, U.S.A.

United Kingdom by
IOS Press
73 Lime Walk, Headington, Oxford OX3 7AD, England

Europe and the rest of the world by
IOS Press
Van Diemenstraat 94, 1013 CN Amsterdam, Netherlands

Far East jointly by
Ohmsha, Ltd., IOS Press

Printed in Japan

CONTENTS OF VOLUME 1

PLENARY SESSIONS

ICOT SESSIONS

CONTENTS OF VOLUME 2

FOUNDATIONS

AND-Parallelism and OR-Parallelism

Implementation Techniques

Extension of Logic Programming

Task Scheduling and Load Analysis

Concurrency

Databases and Distributed Systems

FOUNDATIONS

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Logic Program Synthesis from First Order Logic Specifications

Tadashi KAWAMURA
Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
tkawamur@icot.or.jp

Abstract
In this paper, a logic program synthesis method from first order logic specifications is described. The specifications are described by Horn clauses extended by universally quantified implicational formulae. Those formulae are transformed into definite clause programs by meaning-preserving unfold/fold transformation. We show some classes of first order formulae which can be successfully transformed into definite clauses automatically by unfold/fold transformation.

1 Introduction

Logic program synthesis based on unfold/fold transformation [1] is a standard method and has been investigated by many researchers [2, 3, 5, 6, 11, 12, 19]. As for the correctness of unfold/fold rules in logic programming, Tamaki and Sato proposed meaning-preserving unfold/fold rules for definite clause programs [20]. Then, Kanamori and Horiuchi proposed unfold/fold rules for a class of first order formulae [7]. Recently, Sato proposed unfold/fold rules for full first order formulae [18].

In the studies of program synthesis, unfold/fold rules are used to eliminate quantifiers by folding to obtain definite clause programs from first order formulae. However, in most of those studies, unfold/fold rules were applied nondeterministically and general methods to derive definite clauses were not known. Recently, Dayantis [3] showed a deterministic method to derive logic programs from a class of first order formulae. Sato and Tamaki [19] also showed a deterministic method by incorporating the concept of continuation.

This paper shows another characterization of classes of first order formulae from which definite clause programs can be derived automatically. Those formulae are described by Horn clauses extended by universally quantified implicational formulae. As for transformation rules, Kanamori and Horiuchi's unfold/fold rules are adopted. A synthesis procedure based on unfold/fold rules is given, and with some syntactic restrictions, those formulae are successfully transformed into equivalent definite clause programs. This study is also an extension of those by Pettorossi and Proietti [14, 15, 16] on logic program transformations.

The rest of this paper is organized as follows. Section 2 describes unfold/fold rules and formalizes the synthesis process. Section 3 describes a program synthesis procedure and proves that definite clause programs can be successfully derived from some classes of first order formulae using this procedure. Section 4 discusses the relations to other works and Section 5 gives a conclusion.

In the following, familiarity with the basic terminologies of logic programming is assumed[13]. As syntactical variables, X, Y, Z, U, V are used for variables, A, B, H for atoms and F, G for formulae, possibly with primes and subscripts. In addition, θ is used for a substitution, $F\theta$ for the formula obtained from formula F by applying substitution θ, $\overline{X}$ for a vector of variables and $F_G[G']$ for replacement of an occurrence of subformula G of formula F with formula G'.

2 Unfold/Fold Transformation for Logic Program Synthesis

In this section, preliminary notions of our logic program synthesis are shown.

2.1 Preliminaries

Preliminary notions are described first.

A formula is called an *implicational goal* when it is of the form $F_1 \rightarrow F_2$, where F_1 and F_2 are conjunctions of atoms.

Definition 2.1 Definite Formula
Formula C is called a *definite formula* when C is of the form

$$A \leftarrow G_1 \wedge G_2 \wedge \cdots \wedge G_n (n \geq 0),$$

where G_i is a (possibly universally quantified) conjunction of implicational goals for $i = 1, 2, \ldots, n$. A is called the *head of* C, $G_1 \wedge G_2 \wedge \ldots \wedge G_n$ is called the *body of* C and each G_i is called a *goal in the body of* C.

Note that the notion of a definite formula is a restricted form of that in [7].

A set of definite formulae is called a *definite formula program*, while a set of definite clauses is called a *definite clause program*. We may simply say *programs* instead of definite formula (or clause) programs when it is obvious to which we are referring.

Definition 2.2 Definition Formula

Let P be a definite formula program. A definite formula D is called a *definition formula for* P when all the predicates appearing in D's body are defined by definite clauses in P and the predicate of D's head does not appear in P. The predicate of D's head is called a *new predicate*, while those defined by definite clauses in P are *old predicates*. A set of formulae $\mathcal{D}$ is called a *definition formula set for* P when every element D of $\mathcal{D}$ is a definition formula for P and the predicate of D's head appears only once in $\mathcal{D}$.

Atoms with new predicates are called *new atoms*, while those with old predicates are called *old atoms*.

2.2 Unfold/Fold Transformation

In this subsection, unfold/fold transformation rules are shown following [7]. Below, we assume that the logical constant *true* implicitly appears in the body of every unit clause. Further, we assume that a goal is always deleted from the body of a definite formula when it is the logical constant *true*, and a definite formula is always deleted when some goal in its body is the logical constant $false$.

Further, we introduce the reduction of implicational goals with logical constant *true* and $false$, such as $\neg true \Rightarrow false, true \wedge F \Rightarrow F$, and so on. (See [7] for details.) Let G be an implicational goal. The reduced form of G, denoted by $G\downarrow$, is the normal form in the above reduction system.

Variables not quantified in formula F are called *global variables of* F. Atoms appearing positively (negatively) in formula F are called *positive (negative)* atoms of F.

Definition 2.3 Positive Unfolding

Let P_i be a program, C be a definite formula in P_i, G be a goal in the body of C and A be a positive old atom of G containing no universally quantified variable. Then, let G_0 be $G_A[false]\downarrow$ and C'_0 be the definite formula obtained from C by replacing G with G_0. Further, let $C_1, C_2, \ldots, C_k$ be all the definite clauses in P_i whose heads are unifiable with A, say by mgu's $\theta_1, \theta_2, \ldots, \theta_k$. Let G_j be the reduced form of $G\theta_j$ after replacing $A\theta_j$ in $G\theta_j$ with the body of $C_j\theta_j$, and C'_j be the definite formula obtained from $C\theta_j$ by replacing $G\theta_j$ in the body with G_j. (New variables introduced from C_j are global variables of G_j.) Then, $P_{i+1} = (P_i - \{C\}) \cup \{C'_0, C'_1, C'_2, \ldots, C'_k\}$. $C'_0, C'_1, C'_2, \ldots, C'_k$ are called the *results of positive unfolding* C *at* A (or G).

Example 2.1 Let P be a definite clause program as follows :

C_1 : list([]).
C_2 : list([X|L]) ← list(L).
C_3 : 0 < suc(Y).
C_4 : suc(X) < suc(Y) ← X < Y.
C_5 : member(U,[U|L]).
C_6 : member(U,[V|L]) ← member(U,L).

Let C_7 be a definition formula for P as follows :

C_7 : less-than-all(X,L) ←
list(L) ∧ ∀ Y(member(Y,L) → X<Y).

Suppose that $P_0 = P \cup \{C_7\}$. Then, by unfolding C_7 at list(L), program $P_1 = P \cup \{C_8, C_9\}$ is obtained, where

C_8 : less-than-all(X,[]) ← ∀ Y(member(Y,[]) → X<Y).
C_9 : less-than-all(X,[Z|L]) ←
list(L) ∧ ∀ Y(member(Y,[Z|L]) → X<Y).

Before showing the negative unfolding rule, we introduce the notion of *terminating atoms*. Intuitively, atom A is terminating when every derivation path of A is finite. See [7] for the precise definition.

Definition 2.4 Negative Unfolding

Let P_i be a program, C be a definite formula in P_i, G be a goal in the body of C and A be a negative old atom of G such that every atom obtained from A by instantiating all global variables in A to ground is terminating. Let $C_1, C_2, \ldots, C_k$ be all the definite clauses in P_i whose heads are unifiable with A, say by mgu's $\theta_1, \theta_2, \ldots, \theta_k$, where θ_j instantiates no global variable in G. Let G_0 be $G_A[false]\downarrow$ and G_j be the reduced form of $G\theta_j$ after replacing $A\theta_j$ in $G\theta_j$ with the body of $C_j\theta_j$. (New variables introduced from C_i are universally quantified variables in G_i.) Let C' be the definite formula obtained from C by replacing G in the body of C with $G_0 \wedge G_1 \wedge \ldots \wedge G_k$. Then, $P_{i+1} = (P_i - \{C\}) \cup \{C'\}$. C' is called the *results of negative unfolding* C *at* A (or G).

Example 2.2 Let P and P_1 be programs in Example 2.1. By unfolding C_8 at member(X,[]), $P_2 = P \cup \{C_9, C_{10}\}$ is obtained, where

C_{10} : less-than-all(X,[]) ← ∀ Y ($false$ → X < Y)↓.

that is,

C_{10} : less-than-all(X,[]).

Further, by unfolding C_9 at member(X,[Z|L]), $P_3 = P \cup \{C_{10}, C_{11}\}$ is obtained, where

C_{11} : less-than-all(X,[Z|L]) ← list(L) ∧
∀ Y($false$ → X<Y)↓ ∧
∀ Y($true$ → X<Z)↓ ∧
∀ Y (member(Y,L) → X<Y)↓.

that is,

C_{11} : less-than-all(X,[Z|L]) ← list(L) ∧
X < Z ∧ ∀ Y (member(Y,L) → X < Y).

Definition 2.5 Folding

Let P_i be a definite formula program, C be a definite formula in P_i of the form $A \leftarrow K \wedge L$ and D be a definite

formula of the form $B \leftarrow K'$, where K,K' and L are conjunctions of goals. Suppose that there exists a substitution θ such that $K'\theta = K$ holds. Let C' be a clause of the form $A \leftarrow B\theta, L$. Then $P_{i+1} = (P_i - \{C\}) \cup \{C'\}$.

Note that when applying folding, some conditions have to be satisfied to preserve the meanings of programs. See [7] for details.

Example 2.3 Let P and P_3 be programs in Example 2.2. By folding C_{11} by C_7, $P_4 = P \cup \{C_{10}, C_{12}\}$ is obtained, where

C_{12} : less-than-all(X,[Y|L]) $\leftarrow$
X < Y $\wedge$ less-than-all(X,L)

2.3 Program Synthesis by Unfold/Fold Transformation

In this subsection, our program synthesis problem is formalized. Firstly, several notions are defined to formalize the program synthesis processes.

Definition 2.6 Descendant and Ancestor Formula

Let P be a definite formula program, C be a definite formula in P and P' be a definite formula program obtained from P by successively applying positive or negative unfolding to P. A definite formula C' in P' is called a *descendant formula* of C when

(a) C' is identical to C, or
(b) C' is the result of positive or negative unfolding of a descendant formula of C.

Conversely, C is called an *ancestor formula* of C'.

Example 2.4 In Examples 2.1 – 2.3, definite formulae $C_7, C_8, \ldots, C_{11}$ are descendant formulae of C_7.

Definition 2.7 U-selection Rule

A rule that determines what transformation should be applied to a definite formula program is called a *selection rule*. Let P be a definite formula program and C be a definite formula in P. A selection rule R is called a *U-selection rule for P rooted on C* when R always selects positive or negative unfolding applied to a descendant formula of C. C is called the *root formula for R* (or *of the transformation*.) A definite formula program obtained from P by successively applying transformation rules according to R is called a definite formula program *obtained from P via R*.

Definition 2.8 Closed Program

Let P be a definite clause program, C be a definition formula for P, $\mathcal{D}$ be a definition formula set for P and R be a U-selection rule for $P \cup \{C\}$ rooted on C. Let P' be a definite formula program obtained from $P \cup \{C\}$ via R. P' is said to be *closed with respect to* triple $< P, C, \mathcal{D} >$ when every descendant formula C' of C in P' satisfies one of the following:

(a) C' is a definite clause.
(b) There exists a goal G consisting of positive atoms only in the body of C' such that an old atom in G is not unifiable with the head of any definite clause in P'.
(c) By successively folding C' by clauses in $\{C\} \cup \mathcal{D}$, a definite clause can be obtained.

$P \cup \{C\}$ is said to be *closed with respect to $\mathcal{D}$* when there exists a closed program with respect to $< P, C, \mathcal{D} >$ and for every definition formula D in $\mathcal{D}$ there exists a closed program with respect to $< P, D, \mathcal{D} \cup \{C\} >$.

Example 2.5 Let P and P_3 be programs in Example 2.2. Then, P_3 is closed w.r.t. $< P, C_7, \emptyset >$. Further, $P \cup \{C_7\}$ is closed w.r.t. $\emptyset$.

The above framework is an extension of the one shown in [8], and also a modification of the one Pettorossi and Proietti proposed [14, 15, 16] in their studies of program transformation.

Now, our problem can be formalized as follows: for given definite clause program P and definition formula C for P, find a finite definition formula set $\mathcal{D}$ for P such that $P \cup \{C\}$ is closed with respect to $\mathcal{D}$.

3 Some Classes of First Order Formulae from Which Logic Programs Can Be Derived

In this section, we specify some classes of first order formulae from which definite clause programs can be derived by unfold/fold transformation.

3.1 A Program Synthesis Procedure

In this subsection, we show a naive program synthesis procedure. In the following, we borrow some notions about programs in [15, 16]. We consider definite formula (clause) programs with predicate =, which have no explicit definition in the programs. Predicate = is called a *base predicate*, while other predicates are called *defined predicates*. Atoms with base predicates are called *base atoms*, while those with defined predicates are called *defined atoms*. Transformation rules can be applied to defined atoms only.

A formula containing base atoms can be reduced by unifying arguments of =. When a universally quantified variable and a global variable are unified, the global variable is substituted for the universal one. The above reduction is called the *reduction with respect to* =. We assume that no formulae are reduced w.r.t. = unless this is explicitly mentioned.

Further, we assume that the following operations are always applied implicitly to the results of positive or negative unfolding. Goals G is said to be *connected* when at most one universally quantified implicational goal G'

appears in G and each atom in G' has common universally quantified variables with at least one another atom in G'. Let C be a definite formula such that all the goals in its body are connected. Let C' be one of the results of positive or negative unfolding C at some goal. By logical deduction, definite formulae $C'_1, C'_2, \ldots, C'_m (m \geq 1)$ are obtained from C' such that all the goals in the body of C'_i are connected. (Note that some goal G in the body of C' is of the form $F_1 \rightarrow F_2$ or $F_1 \vee F_2$ and no universally quantified variables appear in both F_1 and F_2, C' can be split into two formulae by replacing G in C' with $\neg F_1$ (or F_1) and F_2.)

Before showing our program synthesis procedure, a notion is defined.

Definition 3.1 Sound Unfolding

Suppose that positive or negative unfolding is applied to a definite formula at atom A. Then, the application of unfolding is said to be *sound* when no two distinct universally quantified variables in A are unified when reducing the result of unfolding with respect to $=$.

Some syntactic restrictions on programs ensure the soundness of all possible applications of unfolding. In fact, the restriction shown in [3] ensures the soundness. However, in the following, we assume that every application of unfolding is sound, without giving any syntactic restriction, for simplicity.

Now, we show our program synthesis procedure, which is similar to partial evaluation procedures(cf.[9, 10]). First, a procedure to synthesize new predicates is shown.

Procedure 3.1 *Synthesis of New Predicates*

Suppose that definite formula program P and definite formula C in P of the form $A \leftarrow G_1, G_2, \ldots, G_n$ are given. Let G'_i be the reduced formula obtained from G_i by removing all base atoms and by replacing all universally quantified variables appearing in every base atom with distinct fresh global variables if global variables are substituted for them when reducing G_i w.r.t. $=$. Let D_i be of the form $H_i \leftarrow G'_i$ for $i = 1, 2, \ldots, n$, where H_i is an atom whose predicate does not appear in P or H_j for $i \neq j$ and whose arguments are all global variables of C appearing in G'_i. Then, $D_1, D_2, \ldots, D_n$ are returned.

Note that in Procedure 3.1, C can be folded by $D_1, D_2, \ldots, D_n$ after reducing it w.r.t. $=$ when C is the result of sound unfolding, and the result of the folding is a definite clause.

Example 3.1 Let P be a program as follows.

C_1 : all-less-than(L,M) $\leftarrow$ list(L) $\wedge$ list(M) $\wedge$
$\forall$ U,V (member(U,L) $\wedge$ member(V,M) $\rightarrow$ U < V).

C_2 : member(U,[V|X]) $\leftarrow$ U = V.

C_3 : member(U,[V|X]) $\leftarrow$ member(U,X).

The definition of '<' is given in Example 2.1. Suppose that C's body consists of only one goal. By applying positive unfolding and negative unfolding to C successively, the following formulae are obtained. (The reduction w.r.t. $=$ is done when no universally quantified variable appears as an argument of $=$.)

C_4 : all-less-than([],M) $\leftarrow$ list(M).

C_5 : all-less-than([X|L],M) $\leftarrow$ (list(L) $\wedge$ list(M)) $\wedge$
(list(L) $\wedge$ list(M) $\wedge$
$\forall$ U,V (U = X $\wedge$ member(V,M) $\rightarrow$ U < V)) $\wedge$
(list(L) $\wedge$ list(M) $\wedge$
$\forall$ U,V (member(U,L)$\wedge$member(V,M) $\rightarrow$ U < V)).

Then, by Procedure 3.1, the following new predicates are defined from C_5.

D_1 : new1(X,L,M) $\leftarrow$ list(L) $\wedge$ list(M) $\wedge$
$\forall$ V (member(V,M) $\rightarrow$ X < V).

D_2 : new2(L,M) $\leftarrow$ list(L) $\wedge$ list(M) $\wedge$
$\forall$ U,V (member(U,L) $\wedge$ member(V,M) $\rightarrow$ U < V).

Next, the whole procedure for program synthesis is shown.

Procedure 3.2 *A Program Synthesis Procedure*

Suppose that definite clause program P and definition formula C for P are given. Let $\mathcal{D}$ be the set $\{C\}$.

(a) If there exist no unmarked formulae in $\mathcal{D}$, then return P and stop.

(b) Select an unmarked definition formula D from $\mathcal{D}$. Mark D 'selected.' Let P' be the set $\{D\}$.

(c) If there exist no formulae in P' which do not satisfy conditions (a) and (b) in Definition 2.8, then $P := P \cup P'$ and go to (a).

(d) Select a definite formula C' from P'. Apply positive or negative unfolding to C'. Let $C_1, \ldots, C_n$ be the results. Remove C' from P'.

(e) Apply Procedure 3.1 to $C_1, \ldots, C_n$. Let $D_1, \ldots, D_m$ be the outputs. Add D_i to $\mathcal{D}$ if it is not a definite clause and there exists no formula in $\mathcal{D}$ which is identical to D_i except for the predicate of the head. Fold $C_1, \ldots, C_n$ by the formulae in $\mathcal{D}$ and add the results to P'.

(f) Go to (c).

Example 3.2 Consider the program in Example 3.1 again. We see that D_2 is identical to C except for the predicate of the head. C_5 can be folded by D_1 and C after reduction w.r.t. $=$. The result is as follows.

C_6 : all-less-than([X|L],M) $\leftarrow$ list(L) $\wedge$ list(M) $\wedge$
new1(X,L,M) $\wedge$ all-less-than(L,M).

Similar operations are applied to D_1, and finally, the following clauses are obtained.

D_3 : new1(X,L,[]) $\leftarrow$ list(L).

D_4 : new1(X,L,[Y|M]) $\leftarrow$ X < Y $\wedge$ new1(X,L,M).

Note that Procedure 3.2 does not necessarily derive a definite clause program from a definite formula program. For example, when the following program is given as input, Procedure 3.2 does not halt.

C_1 : p(X,Y) $\leftarrow$ p(X,Z) $\wedge$ p(Z,Y)

C_2 : h(X,Y) $\leftarrow$ $\forall$ Z (p(X,Z) $\rightarrow$ p(Y,Z))

3.2 Classes of First Order Formulae

In this section, we show some classes of definite formula programs which can be transformed into equivalent definite clause programs by Procedure 3.2.

Throughout this subsection, we assume that unfolding is always applicable to every definite formula at an atom when there exist definite clauses whose heads are unifiable with the atom. Note that the above assumption does not always hold. This problem will be discussed in 3.3.

After giving a notion, we show a theorem which is an extension of the results shown in [15]. A *simple expression* is either a term or an atom.

Definition 3.2 Depth of Symbol in Simple Expression

Let X be a variable or a constant and E be a simple expression in which X appears. The depth of X in E, denoted by depth(X,E), is defined as follows.

(a) depth(X,X) = 1.

(b) depth(X,E) = max{depth(X,t_i)|X appears in t_i for $i = 1,\ldots,n$} + 1, if E is either $f(t_1,\ldots,t_n)$ or $p(t_1,\ldots,t_n)$, for any function symbol f or any predicate symbol p.

The deepest variable or constant in E is denoted by maxdepth(E).

Theorem 3.1 Let P be a definite clause program. Suppose that for any definition formula C for P, there exists a U-selection rule R for $P \cup \{C\}$ rooted on C such that R is defined for all descendant clauses of C in which at least one defined atom appears. Suppose also that there exist two positive integers H and W such that every descendant clause C' of C in every program P' obtained from $P \cup \{C\}$ via R satisfies the following two conditions.

(a) The depth of every term appearing in every goal in the body of C' is less than H.

(b) Let $G_1, G_2, \ldots, G_n$ be connected goals in the body of C'. Then, the number of atoms appearing in G_i is less than W, for $i = 1, 2, \ldots, n$.

Then, there exists a finite definition formula set $\mathcal{D}$ for P such that $P \cup \{C\}$ is closed with respect to $\mathcal{D}$.

Proof. From hypothesis (a), only a finite number of distinct atoms (modulo renaming of variables) can appear in the goals of all the descendant formulae of C. Then, apply Procedure 3.2 to P and C. Note that every goal in the body of every descendant formula of C is connected. Then, for every goal of every descendant formula of C, the number of atoms appearing in the goal is less than W, from hypothesis (b). Hence, only a finite number of distinct goals can appear in all the descendant formulae of C. Thus, we can obtain a finite definition formula set $\mathcal{D}_0$ for P such that there exists a closed program P' w.r.t. $< P, C, \mathcal{D}_0 >$.

The above discussion holds for all the definition formulae in $\mathcal{D}_0$, since those formulae are constructed from bodies of the descendant formulae of C. Evidently, only a finite number of distinct definition formulae can be defined. Thus, there exists a finite definition formula set $\mathcal{D}$ for P such that $P \cup \{C\}$ is closed w.r.t. $\mathcal{D}$. □

Theorem 3.1 shows that Procedure 3.2 can derive a definite clause program when (a) a term of infinite depth can not appear, or (b) an infinite number of atoms can not appear in a connected goal during a transformation process. In the following, we show some syntactic restrictions on programs which satisfy the above conditions.

Proietti and Pettorossi showed some classes of definite clause programs which satisfy the conditions in Theorem 3.1 in their studies of program transformation [15]. We show that some extensions of their results are applicable to our problem.

The following definitions are according to [15]. The set of variables occurring in simple expression E is denoted by var(E).

Definition 3.3 Linear Term Formula and Program

A simple expression or a formula is said to be *linear* when no variable appears in it more than once. A definite formula (clause) is called a *linear term formula (clause)* when every atom appearing in it is linear. A definite formula (clause) program is called a *linear term program* when it consists of linear term formulae (clauses) only.

A linear term formula (clause) is called a *strongly linear term formula (clause)* when its body is linear. A definite formula (clause) program is called a *strongly linear term program* when it consists of strongly linear term formulae (clauses) only.

Note that the following definite clause is not a linear term clause.

member(X,[X|L]).

However, it is easy to obtain an equivalent linear term clause as follows :

member(X,[Y|L])← X=Y.

Definition 3.4 A Relation $\leq$ between Linear Simple Expressions

Let E_1 and E_2 be linear simple expressions. When depth(X,E_1)≤depth(X,E_2) holds for every variable X in var(E_1)∩var(E_2), we write $E_1 \leq E_2$. (Both $E_1 \leq E_2$ and $E_2 \leq E_1$ hold when var(E_1)∩var(E_2)= $\emptyset$.)

Definition 3.5 Non-Ascending Formula and Program

Let C be a linear term formula and H be the head of C. C is said to be *non-ascending* when $A \leq H$ holds for every defined atom A appearing in the body of C. A linear term program is said to be *non-ascending* when it consists of non-ascending formulae only.

A definite formula (clause) is said to be *strongly non-ascending* when it is a strongly linear term formula (clause) and non-ascending. A definite formula (clause) program is said to be *strongly non-ascending* when it

consists of strongly non-ascending formulae (clauses) only.

Definition 3.6 Synchronized Descent Rule

Let P be a linear term program, R be a U-selection rule for P and C be any descendant formula of the root formula for R. Let $A_1, A_2, \ldots, A_n$ be all the atoms appearing in the body of C. Then, R is called a *synchronized descent rule* when

(a) R selects the application of positive or negative unfolding to C at A_i if and only if $A_j \leq A_i$ holds for $j = 1, \ldots, n$, and

(b) R is not defined for C, otherwise.

Note that synchronized descent rules are not necessarily defined uniquely for given programs and definition formulae.

The following theorem is an extension of the one shown in [15, 16].

Lemma 3.2 Let P be a non-ascending definite clause program, C be a linear term definition formula for P, and R be a synchronized descent rule rooted on C. Let P' be a program obtained from $P \cup \{C\}$ via R. For each defined atom A appearing in the body of every descendant clause of C in P', the following holds :

$\mathrm{maxdepth}(A) \leq$
$\max\{\mathrm{maxdepth}(B) |\ B$ is a defined atom in $P \cup \{C\}\}$

Proof. By induction on the number of applications of unfolding. □

Now we show some classes of definite formula programs which satisfy the hypotheses of Theorem 3.1. In the following, for simplicity, we deal with definition formulae with only one universally quantified implicational goal in the body. The results are easily extended to the definite formulae with a conjunction of universally quantified implicational goals.

The following results are also extensions of those shown in [15].

Theorem 3.3 Let P be a strongly non-ascending definite clause program and C be a linear term definition formula for P of the form $H \leftarrow A_1 \wedge \forall \overline{X}(A_2 \rightarrow A_3)$, such that the following hold.

(a) For every clause D in P of the form $H_D \leftarrow B_1 \wedge \ldots \wedge B_n \wedge B'_1 \wedge \ldots \wedge B'_m$, where $B_1, \ldots, B_n$ are defined atoms and $B'_1, \ldots, B'_m$ are base atoms, the following hold.

(a-1) Let t_H be any argument of H_D. For every argument t_i of B_i, if t_H contains a common variable with t_i, then t_i is a subterm of t_H.

(a-2) For every argument t_i of B_i, if t_i is a subterm of an argument t_H of H_D, then no other argument of B_i is a subterm of t_H.

(b) There exist two arguments t_i and s_i of some A_i ($t_i \neq s_i, i = 1, 2$ or 3) such that the following hold.

(b-1) There exists an argument t_j of A_j ($i \neq j$) such that
- $\mathrm{vars}(A_i) \cap \mathrm{vars}(A_j) = \mathrm{vars}(t_i) \cap \mathrm{vars}(t_j)$, and
- either t_i is a subterm of t_j, t_j is a subterm of t_i or $\mathrm{vars}(t_i) \cap \mathrm{vars}(t_j) = \emptyset$.

(b-2) There exists an argument s_k of A_k ($k \neq i, j$) such that the same relations as above hold for s_i and s_k.

(b-3) A_j contains no common variable with A_k.

Then, there exists a definition formula set $\mathcal{D}$ for P such that $P \cup \{C\}$ is closed with respect to $\mathcal{D}$.

Proof. Note that there exists an atom A in the body of C s.t. an argument of A is a maximal term in the body of C w.r.t. subterm ordering relation. Let C' be any result of unfolding C at A and G be any connected goal in the body of C' of the form $F_1 \wedge \forall \overline{X}(F_2 \rightarrow F_3)$, where F_i is a conjunction of atoms. Then, from the hypothesis, it can be shown that a similar property to hypothesis (b) holds for G. Note that the number of implicational goals dose not increase by applying positive unfolding and no global variables are instantiated by applying negative unfolding. Then, again there exists an atom in the body of C' s.t. one of its arguments is a maximal term in the body of C' w.r.t. subterm ordering relation. By induction on the number of applications of unfolding, a synchronized descent rule can be defined for every descendant formula of C. Then, from Lemma 3.2, the depth of every term appearing in every descendant clause of C is bounded.

Note that the number of different subterms of a term is bounded. Then, from the hypothesis, the number of atoms appearing in every connected goal in the body of every descendant formula of C is bounded. Thus, P and C satisfy the hypotheses of Theorem 3.1. Hence, there exists a definition formula set $\mathcal{D}$ for P such that $P \cup \{C\}$ is closed with respect to $\mathcal{D}$. □

Note that Theorem 3.3 holds for any nondeterministic choice of synchronized descent rules in the above proof. Note also that any program can be modified to satisfy hypothesis (a) of Theorem 3.3 by introducing atoms with $=$ in the body.

Corollary 3.4 Let P be a strongly non-ascending definite clause program and P' be a definite clause program such that no predicate appears in both P and P'. Let C be a linear term definition formula for $P \cup P'$ of the form $H \leftarrow A_1 \wedge \forall \overline{X}(A_2 \rightarrow A_3)$, where the predicates of A_1 and A_2 are defined in P and that of A_3 is defined in P'. Suppose that the following hold.

(a) Hypothesis (a) of Theorem 3.3 holds for every clause D in P.

(b) There exist arguments t_1 of A_1 and t_2 of A_2 such that the following hold.

(b-1) $\mathrm{vars}(A_1) \cap \mathrm{vars}(A_2) = \mathrm{vars}(t_1) \cap \mathrm{vars}(t_2)$.

(b-2) Either t_1 is a subterm of t_2, t_2 is a subterm of t_1 or $\mathrm{vars}(t_1) \cap \mathrm{vars}(t_2) = \emptyset$.

(c) No variable in A_3 is instantiated by applying positive or negative unfolding to C successively.

Then, there exists a definition formula set $\mathcal{D}$ for $P \cup P'$ such that $P \cup P' \cup \{C\}$ is closed with respect to $\mathcal{D}$.

Proof. Suppose that unfolding is never applied at A_3. A synchronized descent rule can be defined by neglecting A_3. Since variables in A_3 are never instantiated, no other atoms are derived from A_3. Thus, the corollary holds. □

In Corollary 3.4, no restrictions are required on the definition of A_3. This result corresponds to that in [3]. Note that any program can be modified to satisfy hypothesis (c) of Corollary 3.4 by introducing atoms with $=$ in the body.

Example 3.3 The program and the definition formula in Example 2.1 satisfy the hypotheses of Theorem 3.3 and Corollary 3.4, if clause C_5 is replaced with the equivalent clause :

C_5' : member(U,[V|L]) ← U=V.

In fact, a definite clause program can be obtained, as shown in subsection 2.2.

Next, we show an extension of the results shown in Theorem 3.3. Let P be a non-ascending definite clause program and C be a definition formula for P of the form $H \leftarrow A \wedge \forall \overline{X}(F_1 \rightarrow F_2)$, where A is an atom, and F_1 and F_2 are conjunctions of atoms. Let D_i be the definition clause for P of the form $H_i \leftarrow F_i$ for $i = 1, 2$. If D_i can be transformed into a set of definite clauses which satisfies the hypotheses of Theorem 3.3, by replacing F_i with H_i, we can show that $P \cup \{C\}$ can be transformed into an equivalent definite clause program.

The above problem is related to the foldability problem in [16]. The foldability problem is described informally as follows. Let P be a definite clause program and C be a definition clause for P. Then, find program P' obtained from $P \cup \{C\}$ which satisfies the following : for every descendant clause C' of C in P', there exists an ancestor clause D of C' such that C''s body is an instance of D's.

Proietti and Pettorossi showed some classes of definite clause programs such that the foldability problem can be solved [16]. We show that their results are also available to our problem.

A definite clause program P is said to be *linear recursive* when at most one defined atom appears in the body of each clause in P. Note that a linear recursive and linear term program (clause) is a strongly linear term program (clause).

Lemma 3.5 Let P be a linear recursive non-ascending program and C be a non-ascending definition clause for P of the form $H \leftarrow A_1 \wedge A_2 \wedge B_1 \wedge \ldots \wedge B_n$, where A_1 and A_2 are defined atoms and $B_1, \ldots, B_n$ are base atoms. Suppose that the following hold.

(a) For every clause D in P of the form $H_D \leftarrow A_D \wedge B_1' \wedge \ldots \wedge B_n'$, where A_D is the only defined atom in the body of D, the following hold.

(a-1) Let t_H be any argument of H_D. For every argument t_A of A_D, if t_H contains a common variable with t_A, then t_A is a subterm of t_H.

(a-2) For every argument t_A of A_D, if t_A is a subterm of an argument t_H of H_D, then no other argument of A_D is a subterm of t_H.

(b) There exist arguments t_1 of A_1 and t_2 of A_2 such that the following hold.

(b-1) $\mathrm{vars}(A_1) \cap \mathrm{vars}(A_2) = \mathrm{vars}(t_1) \cap \mathrm{vars}(t_2)$.

(b-2) Either t_1 is a subterm of t_2, t_2 is a subterm of t_1 or $\mathrm{vars}(t_1) \cap \mathrm{vars}(t_2) = \emptyset$.

Then, from $P \cup \{C\}$, we can obtain a linear recursive non-ascending program which define the predicate of H by unfold/fold transformation.

Proof. As shown in [16], we can get a solution of the foldability problem for P and C. Then, obviously, a linear recursive program is obtained. □

Example 3.4 Let P be a linear recursive non-ascending program as follows.

C_1 : subseq([],L).
C_2 : subseq([X|L],[Y|M]) ← X = Y ∧ subseq(L,M).
C_3 : subseq([X|L],[Y|M]) ← subseq([X|L],M).

Let C be a non-ascending definition clause for P as follows.

C : csub(X,Y,Z) ← subseq(X,Y), subseq(X,Z).

Then, $P \cup \{C\}$ can be transformed into a linear recursive non-ascending program as follows.

csub([],Y,Z).
csub([A|X],[B|Y],Z) ← A = B ∧ cs(A,X,Y.Z).
csub([A|X],[B|Y],Z) ← csub([A|X],Y,Z).
cs(A,X,Y,[B|Z]) ← A = B ∧ csub(X,Y,Z).
cs(A,X,Y,[B|Z]) ← cs(A,X,Y,Z).

Though Proietti and Pettrossi showed one more class [16], we will not discuss this here.

Now, we get the following theorem.

Theorem 3.6 Let P be a linear recursive non-ascending program and C be a linear term definition formula for P of the form $H \leftarrow A_1 \wedge \forall \overline{X}(A_2 \wedge B_2 \rightarrow A_3 \wedge B_3)$, such that the following hold.

(a) Hypothesis (a) of Lemma 3.5 holds for P.

(b) Let S_1 be the set of all the arguments of A_1, and S_i be the set of all the arguments of A_i and B_i for $i = 2, 3$. Then, there exist two terms t_j and s_j in some S_j ($t_j \neq s_j, j = 1, 2$ or 3) such that the following hold.

(b-1) there exists a term t_k in S_k ($j \neq k$) such that
· $\mathrm{vars}(S_j) \cap \mathrm{vars}(S_k) = \mathrm{vars}(t_j) \cap \mathrm{vars}(t_k)$, and

· either t_j is a subterm of t_k, t_k is a subterm of t_j or vars(t_j)∩vars(t_k)=∅.

(b-2) There exists a term s_l of S_l ($l \neq j, k$) such that the same relations as above hold for s_j and s_l.

(b-3) S_k contains no common variable with S_l.

Then, there exists a definition formula set $\mathcal{D}$ for P such that $P \cup \{C\}$ is closed with respect to $\mathcal{D}$.

Proof. Obvious from Theorem 3.3 and Lemma 3.5. □

Note that it is easy to extend the result of Theorem 3.6 to allow the conjunction of an arbitrary number of atoms to appear in the body of the definition formula. Note also that it is possible to extend the result to allow arbitrary definition of A_3 and B_3, in a similar way to Corollary 3.4.

3.3 Further Consideration about Syntactic Restrictions

As described in 3.2, the application of unfolding may be prohibited in Kanamori and Horiuchi's framework. In this subsection, we discuss some methods to avoid prohibition, though we do not necessarily give the precise syntactic restriction. (Due to space limitations, we do not refer to the terminating property, though several sufficient conditions are known to guarantee it.)

(1) Universally Quantified Variables Appearing in Positive Atoms

Positive unfolding can not be applied to definite formulae at positive atoms with universally quantified variables. Thus, we have the following two problems.

(a) Synchronized descent rules can not be defined when universally quantified variables are instantiated by negative unfolding.

(b) We can not unfold formulae of the form $\forall \overline{X} A$ when A is an atom and some variables in $\overline{X}$ appear in A.

To avoid case (a), the following restriction is sufficient. When applying negative unfolding, no universally quantified variable is instantiated. Though the restriction seems to be strong, most of significant examples of program synthesis can be dealt with under the restriction.

Case (b) corresponds to the compilation failure in Sato and Tamaki's first order compiler [19]. They restricted their language as follows. For every implicational goal $F_1 \rightarrow F_2$ appearing in a formula, uvar(F_1)⊇uvar(F_2) holds, where uvar(F_i) means the set of universally quantified variables appearing in F_i.

The above condition is available for our problem. Note that the application of positive unfolding does not affect the condition. When applying negative unfolding at atom A in universally quantified implicational goal G, the following restrictions are also required. All the universally quantified variables appearing in A also appear in some negative defined atom in each result of negative unfolding G, or they are unified with terms consisting of constants and global variables by reduction w.r.t. =.

We believe that techniques such as mode analysis are available to guarantee that every applicable negative unfolding satisfies the above conditions.

(2) Global Variables Appearing in Negative Atoms

Negative unfolding should be applied without instantiating global variables. In some cases, this restriction may be critical. However, we can deal with most of those cases by adding positive atoms to the formula such that the global variables can be instantiated by applying positive unfolding at those atoms. Atoms with predicates which specify data types (cf. list) are available. For example, with the definitions of 'member' and '<' in Example 2.1, negative unfolding can not be applied to the definite formula below.

less-than-all(X,L) ← ∀ Y(member(Y,L) → X<Y).

However, we can apply negative unfolding to the formula below, after positive unfolding list(L).

less-than-all(X,L) ←
list(L) ∧ ∀ Y(member(Y,L) → X<Y).

(3) Sato's Unfold/Fold Transformation

Recently, Sato proposed unfold/fold transformation rules for full first order programs [18]. Their unfolding operation does not require conditions like Kanamori and Horiuchi's. On the other hand, more complex conditions are required when applying folding. Thus, when we adopt Sato's rules in place of Kanamori and Horiuchi's, we need not consider the restrictions discussed in (1) and (2) above, while some other difficulties are introduced to satisfy the folding conditions.

4 Discussion

The work described here is an extension of Pettorossi and Proietti's work on program transformation [14, 15, 16]. They formalized the successful unfold/fold transformation in three ways, and showed that the problem of whether a given program can be transformed successfully or not is unsolvable. They also showed some classes of definite clause programs which can be transformed successfully. Our results owe much to their work, though currently we do not know whether our problem is decidable.

Proietti and Pettorossi also showed that any definite clause program can be transformed successfully by performing suitable generalization of the atoms to be folded [15, 16]. However, the generalization technique is not available for our problem. Folding by a definition formula obtained by generalizing atoms with universally quantified variables may not satisfy the conditions for

folding [7], since universally quantified variables can not appear in the head of the formula.

Proietti and Pettorossi also showed a transformation procedure called loop absorption [15, 16]. In this procedure, they found clause C and its descendant clause C' such that C''s body is an instance of C's (or a subset of C''s body is identical to C's body). Then, a new definition clause whose body is identical to that of C is constructed. They also showed a procedure to eliminate unnecessary variables [17]. We can modify our naive procedure described in 3.1 by incorporating the loop absorption and the elimination of unnecessary variables. Programs obtained by the modified procedure are expected to be more efficient and have less code than those obtained by the naive procedure.

There have been several studies on logic program synthesis from universally quantified implicational formulae [3, 4, 19]. Our work is closely related to that of Dayantis [3]. There, program synthesis was also considered from formulae of the form $H \leftarrow \forall \overline{X}(A \rightarrow B)$. They showed that a class of those formulae can be transformed into definite clauses by deductive derivation. They also discussed the generality of the class using several examples. Their deductive method is analogous to unfold/fold transformation and the derivation processes almost correspond to those by our procedure when our procedure does not apply positive unfolding. They also mechanized their derivation processes. Our notion of the soundness of the application of unfolding is ensured by part of their syntactic restrictions on the arguments of formulae, though we have not discussed how this is ensured. However, the classes we have shown are still wider than those they showed after we incorporate those restrictions.

Sato and Tamaki showed a deterministic algorithm to transform logic programs with universally quantified implicational formulae into definite clause programs [19]. In their method, unfold/fold transformation is applied to universal continuation forms. Their method can be applied to a wider class of first order formulas than ours, while the results of the compilation are not necessarily efficient and the code sizes of those results increase generally.

5 Conclusion

A logic program synthesis method from some classes of first order logic specifications have been shown. The method is based on unfold/fold transformation. Some classes of first order formulae which can be transformed into definite clause programs by unfold/fold transformation have been shown.

Acknowledgments

I would like to thank Tadashi Kanamori and anonymous referees for helpful comments. I would also like to thank Koichi Furukawa and Ryuzo Hasegawa for their advice, and Kazuhiro Fuchi for giving me the opportunity to do this research.

References

[1] Burstall, R.M. and J.Darlington, "A Transformation System for Developing Recursive Programs", J.ACM, Vol.24, No.1, pp.44–67, 1977.

[2] Clark, K.L. and S. Sickel, "Predicate Logic: A Calculus for Deriving Programs", Proc. of 5th International Joint Conference on Artificial Intelligence, pp.419–420, 1977.

[3] Dayantis, G., "Logic Program Derivation for a Class of First Order Logic Relations", Proc. of 10th International Joint Conference on Artificial Intelligence, pp.9–14, Italy, 1987.

[4] Fribourg, L., "Extracting Logic Programs from Proofs that Use Extended Prolog Execution and Induction", Proc. of 7th International Conference on Logic Programming, pp.685–699, Jerusalem, 1990.

[5] Hansson, A. and Tarnlund, S.A., "A Natural Programming Calculus", Proc. of 6th International Joint Conference on Artificial Intelligence, pp.348–355, 1979.

[6] Hogger, C.J., "Derivation of Logic Programs", J.ACM, Vol.28, pp.372–392, 1981.

[7] Kanamori, T. and K. Horiuchi, "Construction of Logic Programs Based on Generalized Unfold/Fold Rules", Proc. of 4th International Conference on Logic Programming, pp.744–768, Melbourne, 1987.

[8] Kawamura, T., "Derivation of Efficient Logic Programs by Synthesizing New Predicates", Proc. of 1991 International Logic Programming Symposium, pp.611–625, San Diego, 1991.

[9] Komorowski, J., "Partial Evaluation As A Means for Inferencing Data Structures in An Applicative Language : A Theory And Implementation in The Case of Prolog", Proc. of the ACM Symposium on Principles of Programming Languages, pp.255–267, 1982.

[10] Komorowski, J., "Towards a Programming Methodology Founded on Partial Deduction", Proc. of the European Conference on Artificial Intelligence, pp.404–409,1990.

[11] Lau, K.K. and S. D. Prestwich, "Top-down Synthesis of Recursive Logic Procedures from First-order Logic Specifications", Proc. of 7th International Conference on Logic Programming, pp.667–684, Jerusalem, 1990.

[12] Lau, K.K. and S. D. Prestwich, "Synthesis of a Family of Recursive Sorting Procedures", Proc. of 1991 International Logic Programming Symposium, pp.641–658, San Diego, 1991.

[13] Lloyd, J. W., "Foundations of Logic Programming", Springer-Verlag, 2nd Edition, Berlin, Heidelberg, New York, 1987.

[14] Pettorossi, A. and M. Proietti, "Decidability Results and Characterization of Strategies for the Development of Logic Programs", Proc. of 6th International Conference on Logic Programming, pp.539–553, Lisboa, 1989.

[15] Proietti, M. and A. Pettorossi, "Construction of Efficient Logic Programs by Loop Absorption and Generalization", Proc. of the Second Workshop on Meta-programming in Logic, pp.57–81, Leuven, 1990.

[16] Proietti, M. and A. Pettorossi, "Synthesis of Eureka Predicates for Developing Logic Programs", Proc. of 3rd European Symposium on Programming, Copenhagen, LNCS 432, Springer-Verlag, pp.307–325,1990.

[17] Proietti, M. and A. Pettorossi, "Unfolding - Definition - Folding, In This Order, For Avoiding Unnecessary Variables In Logic Programs", Proc. of 3rd International Symposium on Programming Language Implementation and Logic Programming, Passau, LNCS 528, Springer-Verlag, pp.347–358,1991.

[18] Sato, T., "An Equivalence Preserving First Order Unfold/fold Transformation System", Algebraic and Logic Programming, Proceedings, LNCS 463, Springer-Verlag, pp.173–188, 1990.

[19] Sato, T. and H. Tamaki, "First Order Compiler : A Deterministic Logic Program Synthesis Algorithm", J.Symbolic Computation, Vol.8, pp.605–627, 1989.

[20] Tamaki, H. and T. Sato, "Unfold/Fold Transformation of Logic Programs", Proc. of 2nd International Logic Programming Conference, pp.127–138, Uppsala, 1984.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Sound and Complete Partial Deduction with Unfolding Based on Well-Founded Measures *

Bern Martens Danny De Schreye Maurice Bruynooghe†

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: {bern,dannyd,maurice}@cs.kuleuven.ac.be

Abstract

We present a procedure for partial deduction of logic programs, based on an automatic unfolding algorithm which guarantees the construction of sensibly and strongly expanded, finite SLD-trees. We prove that the partial deduction procedure terminates for all definite logic programs and queries. We show that the resulting program satisfies important soundness and completeness criteria with respect to the original program, while retaining the essentially desired amount of specialisation.

1 Introduction

Since its introduction in logic programming by Komorowski ([Komorowski, 1981]), partial evaluation has attracted the attention of many researchers in the field. Some, e.g. [Venken, 1984], [Venken and Demoen, 1988], [Sahlin, 1990], have addressed pragmatic issues related to the impurities of Prolog. Others were attracted by the perspective of eliminating the overhead associated with meta interpreters. Some examples are: [Gallagher, 1986], [Levi and Sardu, 1988], [Safra and Shapiro, 1986], [Sterling and Beer, 1989] and [Takeuchi and Furukawa, 1986]. Finally, a firm theoretical basis for the subject was described in [Lloyd and Shepherdson, 1991].

Just as in [Bruynooghe *et al.*, 1991a], we use the term "partial deduction" in this paper, rather than the more familiar "partial evaluation". Following [Komorowski, 1989], we do so because we want to leave the latter term for works taking into account the non-logical features of Prolog and the order in which answers are produced. In the present paper, we adhere to the viewpoint taken in [Lloyd and Shepherdson, 1991] which states that the specialised program should have the same answers as the original one.

Indeed, the authors of [Lloyd and Shepherdson, 1991] present important criteria which, when satisfied by the specialised program, guarantee this to be the case. A partial deduction procedure imposing these criteria, is described in [Benkerimi and Lloyd, 1990]. However, termination of this procedure is not guaranteed, not even for definite logic programs. In this paper, we propose an alternative method which does terminate for all definite logic programs. A central part of any partial deduction procedure is an unfolding algorithm which builds the SLD(NF)-trees used as starting point for synthesising specialised clauses. In general, termination of this unfolding process is problematic in its own right. In [Bruynooghe *et al.*, 1991a], a general criterion for avoiding infinite unfolding is presented. In the present paper, we build on those results for formulating a terminating procedure for partial deduction, respecting the soundness and completeness conditions of [Lloyd and Shepherdson, 1991].

The paper is organised as follows. In section 2, we recapitulate (and adapt) some basic concepts in partial deduction from [Lloyd and Shepherdson, 1991], as well as the criteria for soundness and completeness presented there. We sketch the partial deduction method from [Benkerimi and Lloyd, 1990] and show an example on which the unfolding rules mentioned there do not terminate. In section 3, we introduce an automatic algorithm for finite unfolding, adapted from [Bruynooghe *et al.*, 1991a]. Next, in section 4, our partial deduction procedure is presented. We give an algorithm which implements it and prove its termination. Moreover, we prove that the method satisfies the criteria introduced in [Lloyd and Shepherdson, 1991]. We also show that the intended specialisation is indeed obtained. We conclude the paper in section 5 with a short discussion, including a brief comparison with the approach of [Benkerimi and Lloyd, 1990] and some directions for further research.

*work partially supported by ESPRIT BRA COMPULOG (project 3012)

†All authors are supported by the Belgian National Fund for Scientific Research.

2 Partial Deduction

2.1 Basic concepts, soundness and completeness

We assume familiarity with the basics of logic programming. Definitions of the following concepts can be found in [Lloyd and Shepherdson, 1991] and [Benkerimi and Lloyd, 1990]: *most specific generalisation (msg)*, *incomplete* SLD-tree, *resultant* of a derivation, *partial deduction for an atom in a program*, *partial deduction for a set of atoms in a program*, *partial deduction of a program wrt a set of atoms*, *independence* of a set of atoms, **A**-*closedness* of a set of formulas, **A**-*coveredness* of a program and goal. In [Lloyd and Shepherdson, 1991] and [Benkerimi and Lloyd, 1990], the definitions are given for normal programs and using the term "partial *evaluation*". In the present paper, we restrict ourselves to definite programs and goals and, as mentioned above, use the term "partial *deduction*". The necessary adaptations are straightforward (as exemplified in [Bruynooghe *et al.*, 1991a]).

We adapt the following theorem from [Lloyd and Shepherdson, 1991].

Theorem 2.1 Let P be a definite logic program, G a definite goal, **A** a finite, independent set of atoms, and P' a partial deduction of P wrt **A** such that $P' \cup \{G\}$ is **A**-covered. Then the following hold:

- $P' \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
- $P' \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

In other words, under the conditions stated in this theorem, computation with a partial deduction of a program is sound and complete wrt computation with the original program. This is clearly a very desirable characteristic of any procedure for partial deduction. It is therefore important to devise methods for partial deduction that ensure the conditions of theorem 2.1 are satisfied.

In [Benkerimi and Lloyd, 1990], one such method is presented. Basically, it proceeds as follows. For a given goal G and program P, a partial deduction for G in P is computed. This is repeated for any goal occurring in the resulting clauses which is not an instance of one already processed. Assuming the procedure terminates, one gets in this way a set of clauses **S** and a set **A** of partially deduced atoms such that **S** is **A**-closed. But one also wants **A** to be independent. In order to achieve this, the procedure is modified as follows. Whenever a goal occurring in **S** is not an instance (nor a variant) of one in **A**, but has a common instance with it, the latter is removed from **A** and a partial deduction is computed for their msg (which itself is therefore added to **A**) and added to **S**. The original partial deduction for the removed goal is itself also removed from **S**. The process stops if **A** becomes independent and **S** **A**-closed. **S** can then be used to synthesize a partial deduction of P wrt **A** which satisfies the conditions of theorem 2.1 for any goal G' which is an instance of G.

However, the tactic of taking msgs to make **A** independent causes an unacceptable loss of specialisation in the resulting partial deduction. To remedy this, the authors of [Benkerimi and Lloyd, 1990] introduce a renaming transformation as a pre-processing stage before running their algorithm. It amounts to duplicating and renaming the definitions of those predicates, occurring in the original goal G, which are likely to pose specialisation problems. The details can be found in [Benkerimi and Lloyd, 1990].

2.2 Unfolding

One question is left more or less unanswered until now: How to obtain the (incomplete) SLD-trees used as a basis for producing partial deductions? In other words, which computation rule should be used for building these trees (including the question of deciding when to stop the unfolding)? [Benkerimi and Lloyd, 1990] mentions 4 criteria and proposes the following one as the best : The computation rule R_v selects the leftmost atom which is not a variant of an atom already selected on the branch down to the current goal. However, this rule fails to guarantee the production of finite SLD-trees in all cases. We present a counter-example. It is the well-known "reverse" program with accumulating parameter.

Example 2.2

source program:

```
reverse([],L,L).
reverse([X|Xs],Ys,Zs) ← reverse(Xs,[X|Ys],Zs).
```

query:

```
←reverse([1,2|Xs],[],Zs).
```

The reader can verify that R_v generates an infinite SLD-tree.

Some authors have therefore combined R_v or other computation rules with a depth bound: (a.o.) [Levi and Sardu, 1988], [Sterling and Beer, 1986], [Takeuchi and Furukawa, 1986]. This does of course guarantee finiteness, but it seems a rather ad-hoc solution which does not reflect any properties of the given unfolding problem. We therefore proposed an alternative solution in [Bruynooghe *et al.*, 1991a]. (An extended version of this paper can be found in [Bruynooghe *et al.*, 1991b].)

3 An Algorithm for Finite Unfolding

In [Bruynooghe *et al.*, 1991a], a general criterion for avoiding infinite unfolding during partial deduction and a terminating unfolding algorithm based on it, are presented. In this section, we introduce a fully automatic version of that algorithm, tuned towards unfolding object-level definite logic programs. A slightly more sophisticated approach may be desirable when dealing with meta interpreters. We will not address that point in the present paper and concentrate on object-level programs. Although a slightly more accurate presentation of the algorithm itself is given, most of what follows now is adapted from [Bruynooghe *et al.*, 1991a]. The interested reader is referred to that paper for a full (and more general) account with all the technical details on the well-founded measures underlying our approach. Here, we only introduce what is necessary for a good understanding of algorithm 3.6.

For technical reasons, we will assume a numbering on the nodes of an SLD-tree (e.g. left-to-right, top-down and breadth-first). We will use the following notation for nodes in an SLD-tree: (G, i) where G is a goal of the tree having i as its associated number. (The notations "(G, i)" and "G" will be used interchangeably, as the context requires.)

We first define a weight-function on terms. It counts the number of functors in its argument.

Definition 3.1 Let **Term** denote the set of terms in the first order language used to define the theory P. We define $|.| : \mathbf{Term} \rightarrow I\!N$ as follows:

If $t = f(t_1, \ldots, t_n), n > 0$
then $|t| = 1 + |t_1| + \cdots + |t_n|$
else $|t| = 0$

It is then possible to introduce weight-functions on atoms.

Definition 3.2 Let p be a predicate of arity n and $\mathbf{S}= \{a_1, \ldots, a_m\}, 1 \leq a_k \leq n, 1 \leq k \leq m$, a set of argument positions for p. We define $|.|_{p,S} : \{A | A \text{ is an atom with predicate symbol } p\} \rightarrow I\!N$ as follows:

$$|p(t_1, \ldots, t_n)|_{p,S} = |t_{a_1}| + \cdots + |t_{a_m}|$$

The next two definitions introduce useful relations on literals and goals in an SLD-tree.

Definition 3.3 Let $(G, i) = ((\leftarrow A_1, \ldots, A_j, \ldots, A_n), i)$ be a node in an SLD-tree τ, let $R(G) = A_j$ be the call selected by the computation rule R, let $H \leftarrow B_1, \ldots, B_m$ be a clause whose head unifies with A_j and let $\theta = mgu(A_j, H)$ be the most general unifier. Then (G, i) has a son (G', k) in τ, $(G', k) = ((\leftarrow A_1, \ldots, A_{j-1}, B_1, \ldots, B_m, A_{j+1}, \ldots, A_n)\theta, k)$. We say that $B_1\theta, \ldots, B_m\theta$ in G' are *direct descendents* of A_j in G and that A_j in G is a *direct ancestor* of $B_1\theta, \ldots, B_m\theta$ in G'.

The binary relations *descendent* and *ancestor*, defined on atoms in goals, are the transitive closures of the direct descendent and direct ancestor relations respectively. For A an atom in G and B an atom in G', A is an ancestor of B is denoted as $A >_{pr} B$ ("pr" stands for proof tree).

Notice that we also speak about one *goal* G' being an ancestor (or descendent) of another *goal* G. This terminology refers to the obvious relationships between goals in an SLD-tree and should not be confused with the proof-tree based relationships between literals, introduced in the previous definition. The following definition does introduce a relationship between goals, based on definition 3.3.

Definition 3.4 Let G and G' denote two different nodes in an SLD-tree τ. Let R be the computation rule used in τ. Then G' *covers* G iff

1. $R(G')$ and $R(G)$ are atoms with the same predicate
2. $R(G') >_{pr} R(G)$

Notice that G' covers G implies that G' is an ancestor of G.

We need one more piece of terminology.

Definition 3.5 Let G and G' denote two different nodes in an SLD-tree τ. We call G' the *youngest covering ancestor* of G iff

1. G' covers G
2. For any other node G'' such that G'' covers G, we have that G'' covers G'

We are now finally able to formulate the following algorithm:

Algorithm 3.6

Input
a definite program P
a definite goal $\leftarrow A$

Output
a finite SLD-tree τ for $P \cup \{\leftarrow A\}$

Initialisation
$\tau := \{(\leftarrow A, 1)\}$
$Pr := \emptyset$
$Terminated := \emptyset$
$Failed := \emptyset$
For each recursive predicate p/n in P and
for the derivation D in τ:
$S_{p,D} := \{1, \ldots, n\}$

While there exists a derivation D in τ such that $D \notin Terminated$ **do**
Let (G, i) name the leaf of D

Select the leftmost atom $p(t_1, \ldots, t_n)$ in G
satisfying the following condition:
If p is recursive and there is
a youngest covering ancestor (G', j) of (G, i) in D
then $|R(G')|_{p,S_{p,D}{}^{new}} > |p(t_1, \ldots, t_n)|_{p,S_{p,D}{}^{new}}$ where
$S_{p,D}{}^{new} = S_{p,D} \setminus S_{p,D}{}^{remove}$ and
$S_{p,D}{}^{remove} =$
$\{a_k \in S_{p,D} \mid |p(t_1, \ldots, t_n)|_{p,\{a_k\}} > |R(G')|_{p,\{a_k\}}\}$
If such an atom $p(t_1, \ldots, t_n)$ can be found
then
$R(G) := p(t_1, \ldots, t_n)$
Let $Derive(G, i)$ name the set of all derivation steps
that can be performed
If $Derive(G, i) = \emptyset$
then
Add D to $Terminated$ and $Failed$
else
Let $Descend(R(G), i)$ name the set of
all pairs $((R(G), i), (B\theta, j))$, where
— B is an atom in the body of a clause
applied in an element of $Derive(G, i)$
— θ is the corresponding m.g.u.
— j is the number of the corresponding
descendent of (G, i)
Expand D in τ with the elements of $Derive(G, i)$
Add the elements of $Descend(R(G), i)$ to Pr
For every newly created extension D' of D and
for every recursive predicate q in P:
if $q = p$ and (G, i) has a covering ancestor in D
then $S_{q,D'} := S_{q,D}{}^{new}$
else $S_{q,D'} := S_{q,D}$
else
Add D to $Terminated$
Endwhile

We have the following theorem.

Theorem 3.7 Algorithm 3.6 terminates. If a definite program P and a definite goal $\leftarrow A$ are given as inputs, its output τ is a finite (possibly incomplete) SLD-tree for $P \cup \{\leftarrow A\}$.

Proof The theorem is an immediate consequence of proposition 3.1 in [Bruynooghe *et al.*, 1991a]. □

Example 3.8 The SLD-tree generated by algorithm 3.6 for the program and the query from example 2.2, are depicted in figure 1. ("reverse" has been abbreviated to "rev".)

4 Combining These Techniques

4.1 Introduction

In the previous section, we introduced an algorithm for the automatic construction of (incomplete) finite SLD-trees. In this section, we present sound and complete

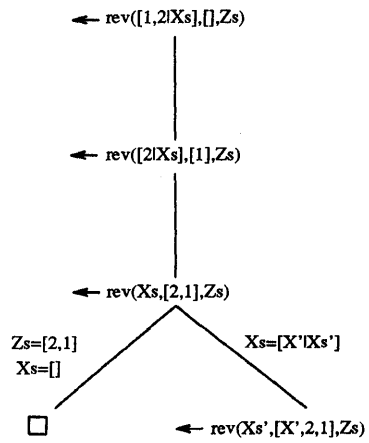


Figure 1: The SLD-tree for example 3.8.

partial deduction methods, based on it. Moreover, these methods are guaranteed to terminate. The following example shows that this latter property is not obvious, even when termination of the basic unfolding procedure is ensured. We use the basic partial deduction algorithm from [Benkerimi and Lloyd, 1990], together with our unfolding algorithm.

Example 4.1 For the reverse program with accumulating parameter (see example 2.2 for the program and the starting query), an infinite number of (finite) SLD-trees is produced (see figure 2). This behaviour is caused by the constant generation of "fresh" body-literals which, because of the growing accumulating parameter, are not an instance of any atom that was obtained before.

In [Benkerimi and Lloyd, 1989], it is remarked that a solution to this kind of problems can be truncating atoms put into **A** at some fixed depth bound. However, this again seems to have an ad-hoc flavour to it, and we therefore devised an alternative method, described in the next section.

4.2 An algorithm for partial deduction

We first introduce some useful definitions and prove a lemma.

Definition 4.2 Let P be a definite program and p a predicate symbol of the language underlying P. Then a *pp'-renaming of P* is any program obtained in the following way:

- Take P together with a fresh—duplicate—copy of the clauses defining p.
- Replace p in the heads of these new clauses by some new (predicate) symbol p' (of the same arity as p).

- Replace p by p' in any number of goals in the bodies of (old and new) clauses.

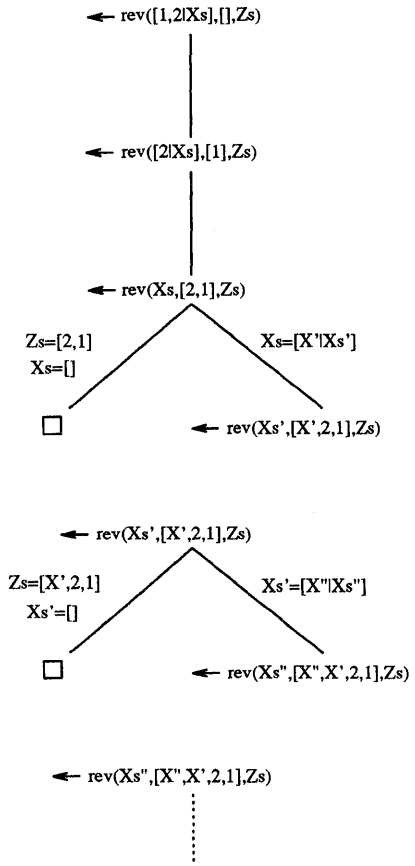


Figure 2: An infinite number of (finite) SLD-trees.

Lemma 4.3 Let P be a definite program and P_r a pp'-renaming of P. Let G be a definite goal in the language underlying P. Then the following hold:

- $P_r \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
- $P_r \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

Proof There is an obvious equivalence between SLD-derivations and -trees for P and P_r. □

Definition 4.4 Let P be a definite program and p a predicate symbol of the language underlying P. Then the *complete pp'-renaming of P* is the pp'-renaming of P where p has been replaced by p' in all goals in the bodies of clauses.

Our method for partial deduction can then be formulated as the following algorithm.

Algorithm 4.5

Input
a definite program P
a definite goal $\leftarrow A = \leftarrow p(t_1, \ldots, t_n)$
in the language underlying P
a predicate symbol p', of the same arity as p,
not in the language underlying P

Output
a set of atoms **A**
a partial deduction P_r' of P_r,
the complete pp'-renaming of P, wrt **A**

Initialisation
$P_r :=$ the complete pp'-renaming of P
A $:= \{A\}$ and label A unmarked

While there is an unmarked atom B in **A** **do**
Apply algorithm 3.6 with P_r and $\leftarrow B$ as inputs
Let τ_B name the resulting SLD-tree
Form P_{rB}, a partial deduction for B in P_r, from τ_B
Label B marked
Let $\mathbf{A}_B$ name the set of body literals in P_{rB}
For each predicate q appearing in an atom in $\mathbf{A}_B$
Let msg_q name an msg of all atoms having q
as predicate symbol in **A** and $\mathbf{A}_B$
If there is an atom in **A** having q as predicate
symbol and it is less general than msg_q
then remove this atom from **A**
If now there is no atom in **A** having q as
predicate symbol
then add msg_q to **A** and label it unmarked
Endfor
Endwhile
Finally, construct the partial deduction P_r' of P_r wrt **A**: Replace the definitions of the partially deduced predicates by the union of the partial deductions P_{rB} for the elements B of **A**.

We illustrate the algorithm on our running example.

Example 4.6

complete renaming of the reverse program:
reverse([],L,L).
reverse([X|Xs],Ys,Zs) ← reverse'(Xs,[X|Ys],Zs).
reverse'([],L,L).
reverse'([X|Xs],Ys,Zs) ← reverse'(Xs,[X|Ys],Zs).

partial deduction for ←reverse([1,2|Xs],[],Zs):
reverse([1,2],[],[2,1]).
reverse([1,2,X|Xs],[],Zs) ← reverse'(Xs,[X,2,1],Zs).

partial deduction for ←reverse'(Xs,[X,2,1],Zs):
reverse'([],[X,2,1],[X,2,1]).
reverse'([X'|Xs],[X,2,1],Zs) ←
reverse'(Xs,[X',X,2,1],Zs).

msg of reverse'(Xs,[X,2,1],Zs) and
reverse'(Xs,[X',X,2,1],Zs): reverse'(Xs,[X,Y,Z|Ys],Zs)

partial deduction for ←reverse'(Xs,[X,Y,Z|Ys],Zs):
reverse'([],[X,Y,Z|Ys],[X,Y,Z|Ys]).
reverse'([X'|Xs],[X,Y,Z|Ys],Zs) ←
reverse'(Xs,[X',X,Y,Z|Ys],Zs).

resulting set **A**:
{reverse([1,2|Xs],[],Zs),reverse'(Xs,[X,Y,Z|Ys],Zs)}

resulting partial deduction:
reverse([1,2],[],[2,1]).
reverse([1,2,X|Xs],[],Zs) ← reverse'(Xs,[X,2,1],Zs).
reverse'([],[X,Y,Z|Ys],[X,Y,Z|Ys]).
reverse'([X'|Xs],[X,Y,Z|Ys],Zs) ←
reverse'(Xs,[X',X,Y,Z|Ys],Zs).

We can prove the following interesting properties of algorithm 4.5.

Theorem 4.7 Algorithm 4.5 terminates.

Proof Due to space restrictions, we refer to [Martens and De Schreye, 1992]. □

Theorem 4.8 Let P be a definite program, $A = p(t_1, \ldots, t_n)$ be an atom and p' be a predicate symbol used as inputs to algorithm 4.5. Let **A** be the (finite) set of atoms and P_r' be the program output by algorithm 4.5. Then the following hold:

- **A** is independent.
- For any goal $G = \leftarrow A_1, \ldots, A_m$ consisting of atoms that are instances of atoms in **A**, $P_r' \cup \{G\}$ is **A**-covered.

Proof

- We first prove that **A** is independent.
  From the way **A** is constructed in the For-loop, it is obvious that **A** cannot contain two atoms with the same predicate symbol. Independence of **A** is an immediate consequence of this.
- To prove the second part of the theorem, let P_r^* be the subprogram of P_r' consisting of the definitions of the predicates in P_r' upon which G depends. We show that $P_r^* \cup \{G\}$ is **A**-closed.
  Let A be an atom in **A**. Then the For-loop in algorithm 4.5 ensures there is in **A** a generalisation of any body literal in the computed partial deduction for A in P_r'. The **A**-closedness of $P_r^* \cup \{G\}$ now follows from the following two facts:
  1. P_r' is a partial deduction of a program (P_r) wrt **A**.
  2. All atoms in G are instances of atoms in **A**.

□

Corollary 4.9 Let P be a definite program, $A = p(t_1, \ldots, t_n)$ be an atom and p' be a predicate symbol used as inputs to algorithm 4.5. Let **A** be the set of atoms and P_r' be the program output by algorithm 4.5. Let $G = \leftarrow A_1, \ldots, A_m$ be a goal in the language underlying P, consisting of atoms that are instances of atoms in **A**. Then the following hold:

- $P_r' \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
- $P_r' \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

Proof The corollary is an immediate consequence of lemma 4.3 and theorems 2.1 and 4.8. □

Proposition 4.10 Let P be a definite program and A be an atom used as inputs to algorithm 4.5. Let **A** be the set of atoms output by algorithm 4.5. Then $A \in \mathbf{A}$.

Proof A is put into **A** in the initialisation phase. From definition 4.4, it follows that no clause in P_r contains a condition literal with the same predicate symbol as A. Therefore, A will never be removed from **A**. □

This proposition ensures us that algorithm 4.5 does not suffer from the kind of specialisation loss mentioned in section 2.1: The definition of the predicate which appears in the query $\leftarrow A$, used as starting input for the partial deduction, will indeed be replaced by a partial deduction for A in P in the program output by the algorithm.

Finally, we have:

Corollary 4.11 Let P be a definite program, $A = p(t_1, \ldots, t_n)$ be an atom and p' be a predicate symbol used as inputs to algorithm 4.5. Let P_r' be the program output by algorithm 4.5. Then the following hold for any instance A' of A:

- $P_r' \cup \{\leftarrow A'\}$ has an SLD-refutation with computed answer θ iff $P \cup \{\leftarrow A'\}$ does.
- $P_r' \cup \{\leftarrow A'\}$ has a finitely failed SLD-tree iff $P \cup \{\leftarrow A'\}$ does.

Proof The corollary immediately follows from corollary 4.9 and proposition 4.10. □

Theorem 4.7 and corollary 4.11 are the most important results of this paper. In words, their contents can be stated as follows. Given a program and a goal, algorithm 4.5 produces a program which provides the same answers as the original program to the given query and any instances of it. Moreover, computing this (hopefully more efficient) program terminates in all cases.

5 Discussion and Conclusion

In [Lloyd and Shepherdson, 1991], important criteria ensuring soundness and completeness of partial deduction are introduced. In the present paper, we started from a recently proposed strategy for finite unfolding ([Bruynooghe *et al.*, 1991a]) and developed a procedure for partial deduction of definite logic programs. We proved this procedure produces programs satisfying the mentioned criteria and, in an important sense, showing the desired specialisation. Moreover, the algorithm terminates on all definite programs and goals.

The unfolding method as it is presented in section 3 was proposed in [Bruynooghe *et al.*, 1991a], but appears here for the first time in this detailed and automatisable form, specialised for object level programs. It tries to maximise unfolding while retaining termination. We know, however, of two classes of programs where the first goal is not achieved. First, meta programs require a somewhat more refined control of unfolding. This issue is addressed in [Bruynooghe *et al.*, 1991a]. We refer the interested reader to that paper (or to [Bruynooghe *et al.*, 1991b]) for further comments on this topic. Second, (datalog) programs where the information contained in constants appearing in the program text plays an important role, are not treated in a satisfactory way. Further research is necessary to improve the unfolding in this case. (A combination of our rule with the R_v computation rule seems promising.) As far as the used unfolding strategy does maximise unfolding, however, it probably diminishes or eliminates the need for dynamic renaming as proposed in [Benkerimi and Hill, 1989].

We now compare briefly algorithm 4.5 with the partial deduction procedure with static renaming presented in [Benkerimi and Lloyd, 1990]. First, we showed above that our procedure terminates for all definite programs and queries while the latter does not. The culprit of this difference in behaviour is (apart from the unfolding strategy used) the way in which msg's are taken. We do this predicatewise, while the authors of [Benkerimi and Lloyd, 1990] only take an msg when this is necessary to keep **A** independent. This may keep more specialisation (though only for predicates different from the one in the starting goal), but causes non-termination whenever an infinite, independent set **A** is generated (as illustrated in example 4.1). Observe, moreover, that we have kept a clear separation between the issues of control of unfolding and of ensuring soundness and completeness. The use of algorithm 3.6 — or further refinements (see above) — guarantees that all sensible unfolding — and therefore specialisation — is obtained. The way in which algorithm 4.5, in addition, ensures soundness and completeness, takes care that none of the obtained specialisation is undone. Therefore, it does not seem worthwhile to consider more than one msg per predicate. Note that one can even consider restricting the partial deduction to the predicate in the starting query and simply retaining the original clauses for all other predicates in the result program. This can perhaps be formalised as a partial deduction where only a 1-step trivial unfolding is performed for these predicates.

Next, the method in [Benkerimi and Lloyd, 1990] is formulated in a somewhat more general framework than the one presented here. A reformulation of the latter incorporating the concept of L-selectability and allowing more than one literal in the starting query seems straightforward. However, a generalisation to normal programs and queries and SLDNF-resolution while retaining the termination property, is not immediate. In e.g. [Benkerimi and Lloyd, 1990], it is proposed that during unfolding, negated calls can be executed when ground and remain in the resultant when non-ground. This of course jeopardises termination, since termination of "ordinary" ground logic program execution is not guaranteed in general. One solution is restricting attention to specific subclasses of programs (e.g. acyclic or acceptable programs, see [Apt and Bezem, 1990], [Apt and Pedreschi, 1990]). Another might be to use an adapted version of our unfolding criterion in the evaluation of the ground negative call, and to keep the latter one in the resultant whenever the SLD(NF)-tree produced is not a complete one. Yet a third way might be offered by the use of more powerful techniques related to constructive negation (see [Chan and Wallace, 1989]).

Finally, [Gallagher and Bruynooghe, 1990] presents another approach to partial deduction focusing both on soundness and completeness and on control of unfolding. The main difference is the control of unfolding by a condition based on maximal deterministic paths, where our approach is based on maximal data consumption, monitored through well-founded measures.

References

[Apt and Bezem, 1990] K. R. Apt and M. Bezem. Acyclic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings ICLP'90*, pages 617–633, Jerusalem, June 1990. The MIT Press. Revised version in *New Generation Computing*, 9(3 & 4):335–364.

[Apt and Pedreschi, 1990] K. R. Apt and D. Pedreschi. Studies in pure prolog: Termination. In J. W. Lloyd, editor, *Proceedings of the Esprit Symposium on Computational Logic*, pages 150–176. Springer-Verlag, November 1990.

[Benkerimi and Hill, 1989] K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. Technical report, Department of Computer Science, University of Bristol, Great-Britain, 1989.

[Benkerimi and Lloyd, 1989] K. Benkerimi and J. W. Lloyd. A procedure for the partial evaluation of logic programs. Technical Report TR-89-04, Department of Computer Science, University of Bristol, Great-Britain, May 1989.

[Benkerimi and Lloyd, 1990] K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings NACLP'90*, pages 343–358. The MIT Press, October 1990.

[Bruynooghe *et al.*, 1991a] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction of logic programs. In V. Saraswat and K. Ueda, editors, *Proceedings ILPS'91*, pages 117–131, October 1991.

[Bruynooghe *et al.*, 1991b] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. Technical Report CW-126, Departement Computerwetenschappen, K.U.Leuven, Belgium, March 1991.

[Chan and Wallace, 1989] D. Chan and M. Wallace. A treatment of negation during partial evaluation. In H. D. Abramson and M. H. Rogers, editors, *Proceedings Meta'88*, pages 299–318. MIT Press, 1989.

[Gallagher and Bruynooghe, 1990] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. In D. H. D. Warren and P. Szeredi, editors, *Proceedings ICLP'90*, pages 732–746, Jerusalem, June 1990. Revised version in *New Generation Computing*, 9(3 & 4):305–334.

[Gallagher, 1986] J. Gallagher. Transforming logic programs by specialising interpreters. In *Proceedings ECAI'86*, pages 109–122, 1986.

[Komorowski, 1981] H. J. Komorowski. A specification of an abstract Prolog machine and its application to partial evaluation. Technical Report LSST69, Linkoping University, 1981.

[Komorowski, 1989] H. J. Komorowski. Synthesis of programs in the framework of partial deduction. Technical Report Ser.A, No.81, Departments of Computer Science and Mathematics, Abo Akademi, Finland, 1989.

[Levi and Sardu, 1988] G. Levi and G. Sardu. Partial evaluation of metaprograms in a multiple worlds logic language. *New Generation Computing*, 6(2 & 3), 1988.

[Lloyd and Shepherdson, 1991] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3 & 4):217–242, 1991.

[Martens and De Schreye, 1992] B. Martens and D. De Schreye. Sound and complete partial deduction with unfolding based on well-founded measures. Technical Report CW-137, Departement Computerwetenschappen, K.U.Leuven, Belgium, January 1992.

[Safra and Shapiro, 1986] S. Safra and E. Shapiro. Meta interpreters for real. In *Information Processing 86*, pages 271–278, 1986.

[Sahlin, 1990] D. Sahlin. The Mixtus approach to automatic partial evaluation of full Prolog. In S. Debray and M. Hermenegildo, editors, *Proceedings NACLP'90*, pages 377–398, 1990.

[Sterling and Beer, 1986] L. Sterling and R. D. Beer. Incremental flavor-mixing of meta-interpreters for expert system construction. In *Proceedings ILPS'86*, pages 20–27. IEEE Comp. Society Press, 1986.

[Sterling and Beer, 1989] L. Sterling and R. D. Beer. Metainterpreters for expert system construction. *Journal of Logic Programming*, pages 163–178, 1989.

[Takeuchi and Furukawa, 1986] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to metaprogramming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.

[Venken and Demoen, 1988] R. Venken and B. Demoen. A partial evaluation system for Prolog : Some practical considerations. *New Generation Computing*, 6(2 & 3):279–290, 1988.

[Venken, 1984] R. Venken. A Prolog meta interpreter for partial evaluation and its application to source to source transformation and query optimization. In *Proceedings ECAI'84*, pages 91–100. North-Holland, 1984.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Framework for Analysing the Termination of Definite Logic Programs with respect to Call Patterns

Danny De Schreye* Kristof Verschaetse† Maurice Bruynooghe*

Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium.
e-mail: {dannyd,kristof,maurice}@cs.kuleuven.ac.be

Abstract

We extend the notions *'recurrency'* and *'acceptability'* of a logic program, which were respectively defined in the work of M. Bezem and the work of K. R. Apt and D. Pedreschi, and which were shown to be equivalent to respectively termination under an arbitrary computation rule and termination under the Prolog computation rule. We show that these equivalences still hold for the extended definitions. The main idea is that instead of measuring ground instances of atoms, all possible calls are measured (which are not necessarily ground). By doing so, a more practical technique is obtained, in the sense that "more natural" measures can be used, which can easily be found automatically.

1 Introduction

In the last few years, a strong research effort in the field of logic programming has addressed the issue of termination. From the more theoretical point of view, the results obtained by Vasak and Potter [1986], Baudinet [1988], Bezem [1989], Cavedon [1989], Apt and Pedreschi [1990], and Bossi *et al.* [1991] have provided several frameworks and basic techniques to formulate and solve questions regarding the termination of logic programs in semantically clear and general terms. Other researchers, such as Ullman and Van Gelder [1988], Plümer [1990], Wang and Shyamasundar [1990], Verschaetse and De Schreye [1991], and Sohn and Van Gelder [1991] have provided practical and automatable techniques for proving the termination of logic programs with respect to certain classes of queries at compile time.

In this paper, we propose an extension of the theoretical frameworks for the characterisation of terminating programs and queries proposed in [Bezem 1989] and [Apt and Pedreschi 1990]. The framework does not only provide slightly more general results, but also increases the practicality of the techniques in view of automation.

*Supported by the National Fund for Scientific Research.
†Supported by ESPRIT BRA COMPULOG project nr. 3012.

Let us recall some definitions from [Bezem 1989] in order to explain our motivation and the intuition behind our approach.

Definition 1.1 (see [Bezem 1989]; Definition 2.1) A *level mapping* for a definite logic program P is a mapping $|.| : B_P \rightarrow I\!N$.

Definition 1.2 (see [Bezem 1989]; Definition 2.2) A definite logic program P is *recurrent* if there exists a level mapping $|.|$, such that for each ground instance $A \leftarrow B_1, \ldots, B_n$ of a clause in P, $|A| > |B_i|$, for each $i = 1, \ldots, n$.

Definition 1.3 (see [Bezem 1989]; Definition 2.7) A definite logic program P is *terminating* if all SLD-derivations for $(P, \leftarrow G)$, where G is a ground goal, are finite.

One of the basic results of [Bezem 1989] is that a program is recurrent if and only if it is terminating. Although this result is very interesting from a theoretical perspective, it is not a very practical one in terms of automated detection of terminating programs and queries. The problem comes from the fact that the definition of recurrency requires that the level mapping "compares" the head of each ground instance of a clause with *every* corresponding atom in the body and imposes a decrease. Intuitively, what would be preferable is to obtain a well-founding based on a measure function (or level mapping), which only decreases on each recursive call to a same predicate. This corresponds better to our intuition, since nontermination (for pure logic programs) can only be caused by infinite recursion.

As we stated above, the problem is not merely related to our intuition on the cause of nontermination, but more importantly to the practicality of level mappings. Consider the following example.

Example 1.4

$$\begin{array}{l} p([]). \\ p([H|T]) \leftarrow q([H|T]), p(T). \end{array}$$

$$\begin{array}{l} q([]). \\ q([H|T]) \leftarrow q(T). \end{array}$$

It is not possible to take as level mapping a function that maps ground instances $p(x)$ and $q(x)$ to the same level, namely *list-length*(x) if x is a ground list, and 0 otherwise. Instead, the definition of recurrency obliges us to take a level mapping that has a "unnatural" offset (1 in this case).

$$|p(x)| = \textit{list-length}(x) + 1$$
$$|q(x)| = \textit{list-length}(x).$$

In a naive attempt to improve on the results of [Bezem 1989], one could try to start from an adapted definition for a recurrent program, in which the relation $|A| > |B_i|$ would only be required if A and B_i are atoms with the same predicate symbol. However, the equivalence with termination would immediately be lost — even for programs having only direct recursion — as the following example shows.

Example 1.5

$$\texttt{append}([], L, L).$$
$$\texttt{append}([H|S], T, [H|U]) \leftarrow \texttt{append}(S, T, U).$$

$$\texttt{p}([H|T]) \leftarrow \texttt{append}(X, Y, Z), \texttt{p}(T).$$

An "extended" notion of recurrency, where the level mapping only relates the measure of ground instances of the recursive calls, would hold with respect to the level mapping:

$$|\texttt{p}(x)| = \textit{list-length}(x)$$
$$|\texttt{append}(x, y, z)| = \textit{list-length}(x).$$

On the other hand, the program is clearly not terminating — if it would be terminating, then we would have shown that append/3 terminates for a call with all three arguments free.

The heart of the problem is that in the definition of recurrency, the level mapping is used for two quite distinct purposes at the same time. First, the level mapping does ensure that on each derivation step, the measure of a recursive descending call is smaller than the measure of the ancestor call (or at least: for each ground instance of such a derivation step). Second, since we are only given that the top level goal is ground (or, in a more general version of the theorem, bounded) — but we have no information on the instantiation of any of the descending calls — the level mapping is also used to ensure that we have some upper limit on the measures for the calls of the (independent) recursive subcomputation evoked by the original call. In the current definition, this is done by imposing that the level also decreases between a call and its descendants that are not related through recursion.

The way in which we address the problem here, differs from the approach in [Bezem 1989] in three ways:

1. We first compute all atoms that can occur as calls during any SLD-derivation for the top-level goal(s) under consideration.
2. We use an extended notion of level mapping, defined on all such atoms — not only the ground ones.
3. We have an adapted definition of recurrency, with as its most important features:
   (a) the condition $|A| > |B_i|$ is not imposed on ground instances of a clause, but instead, on each instance obtained after unification with a (possible) call,
   (b) the decrease $|A| > |B_i|$ is only imposed if A and B_i are calls to the same predicate symbol. (This is for direct recursion — in the context of indirect recursion, the condition is more complex).

One of the side effects of taking this approach is that there is no more necessity to start the analysis for one ground or bounded goal. The technique works equally well when we start from any general set of atoms. The additional advantage that we gain here is that in practice, we are usually interested in the termination properties of a program with respect to some call pattern. Such call patterns can always be specified in terms of abstract properties of the arguments in the goals through mode information, type information or combined (rigid or integrated) mode and type information (see [Janssens and Bruynooghe 1990]). Any such call pattern corresponds to a set of atoms in the concrete domain, and can therefore be analysed with our approach.

The paper is organised as follows. In the next section we extend the equivalence theorem of [Bezem 1989] in the way described above. In section 3 we take a completely similar approach to extend results of [Apt and Pedreschi 1990] on left termination. In section 4, we illustrate the improved practicality of the new framework. We also indicate how some simple extensions are likely to provide full theoretical support for the automated technique proposed in [Verschaetse and De Schreye 1991].

All proofs have been omitted from the paper. They can be found in [De Schreye and Verschaetse 1992].

2 Recurrency with respect to a set of atoms

We first introduce some conventions and recall some basic terminology. Throughout the paper, P will denote a definite logic program. The extended Herbrand Universe, U_P^E, and the extended Herbrand Base, B_P^E, associated to a program P, were introduced in

[Falaschi *et al.* 1989]. They are defined as follows. Let $Term_P$ and $Atom_P$ denote the sets of respectively all terms and all atoms that can be constructed from the alphabet underlying to P. The variant relation, denoted $\approx$, defines an equivalence. U_P^E and B_P^E are respectively the quotient sets $Term_P/\approx$ and $Atom_P/\approx$. For any term t (or atom A), we denote its class in U_P^E (B_P^E) as $\bar{t}$ ($\bar{A}$). There is a natural partial order on U_P^E (and B_P^E), defined as: $\bar{s} \leq \bar{t}$ if there exist representants s' of $\bar{s}$ and t' of $\bar{t}$ in $Term_P$ and a substitution θ, such that $s' = t'\theta$. Throughout the paper, S will denote a subset of B_P^E. We define its closure under $\leq$ as: $S^c = \{\bar{A} \in B_P^E \mid \exists \bar{B} \in S : \bar{A} \leq \bar{B}\}$.

Definition 2.1 P is *terminating with respect to* S if for any representant A' of any element $\bar{A}$ of S, every SLD-tree for $(P, \leftarrow A')$ is finite.

Denoting the classical notion of a Herbrand Base (of ground atoms) over P as B_P, then with the terminology of [Bezem 1989] we have:

Lemma 2.2 P is terminating if and only if it is terminating with respect to B_P.

Lemma 2.3 If all SLD-derivations for $(P, \leftarrow A)$ are finite, and θ is any substitution, then all SLD-derivations for $(P, \leftarrow A\theta)$ are finite.

From lemma 2.3 it follows that in order to verify definition 2.1 for a set $S \subseteq B_P^E$, it suffices to verify the finiteness of the SLD-trees for $(P, \leftarrow A)$ for only one representant of each element in $\bar{A}$. It also follows that P is terminating with respect to a set $S \subseteq B_P^E$ if and only if it is terminating with respect to S^c. In fact, given that P terminates with respect to S, it will in general be terminating with respect to a larger set of atoms than those in S^c. It is clear that if all SLD-trees for $(P, \leftarrow A)$ are finite, and if $H \leftarrow B_1, \ldots, B_n$ is a clause in P, such that A and H unify, then all SLD-trees for $(P, \leftarrow B_i\theta), i = 1, \ldots, n$, where $\theta = mgu(A, H)$, are finite. We can characterise the complete set of terminating atoms associated to a given set S as follows.

Definition 2.4 For any $T \subseteq B_P^E$, define $T_P^{-1}(T) = \{\overline{B_i\theta} \in B_P^E \mid A'$ is a representant of $\bar{A} \in T$, $H \leftarrow B_1, \ldots, B_n$ is a clause in P, $\theta = mgu(A', H)$ and $1 \leq i \leq n\}$.

Denote $\mathcal{H}_S = \{T \in 2^{B_P^E} \mid S^c \subseteq T\}$. $\mathcal{H}_S$ is a complete lattice with bottom element S^c.

Definition 2.5 $R_S : \mathcal{H}_S \rightarrow \mathcal{H}_S : R_S(T) = T \cup T_P^{-1}(T)^c$.

Lemma 2.6 R_S is continuous.

As a result, the least fix-point for R_S is $R_S{\uparrow}\omega$.

Lemma 2.7 P is terminating with respect to S if and only if P is terminating with respect to $R_S{\uparrow}\omega$.

As a result of our construction (in fact: as the very purpose of it), $R_S{\uparrow}\omega$ contains every call in every SLD-tree for any atomic goal of S. Formally:

Proposition 2.8 Let $call(P, S)$ denote the set of all atoms B, such that B is the subgoal selected by the computation rule in some goal of some SLD-tree for a pair $(P, \leftarrow A)$, with A the representant of an element of S. Then, $call(P, S) \subseteq R_S{\uparrow}\omega$.

We now introduce a variant of the definition of a level mapping, where the mapping is defined on equivalence classes of calls.

Definition 2.9 (level mapping)
A *level mapping with respect to a set* $S \subseteq B_P^E$ is a function $|.| : R_S{\uparrow}\omega \rightarrow \mathbb{N}$. A level mapping $|.|$ is called *rigid* if for all $\bar{A} \in R_S{\uparrow}\omega$ and for any substitution θ, $|\bar{A}| = |\overline{A\theta}|$, i.e. the level of an atom remains invariant under substitution.

With slight abuse of notation, we will often write $|A|$, where A is a representant of $\bar{A} \in B_P^E$. The associated notion of recurrency with respect to S will not be defined on ground instances of clauses, but instead on all instances $(H \leftarrow B_1, \ldots, B_n)\theta$ of clauses $H \leftarrow B_1, \ldots, B_n$ of P, such that $\theta = mgu(A, H)$, where A is a representant of an element of $R_S{\uparrow}\omega$. The definition in [Bezem 1989] does not explicitly impose a decrease of the level mapping at each inference step. The level mapping's values should only decrease for ground instances of clauses. By considering more general instances of clauses (as above), we can explicitly impose a decrease of the level mapping's value during (recursive) inference steps. As a result, the adapted level mapping no longer needs to perform different functionalities at once, and we can concentrate on the real structure of the recursion.

Now, concerning this recursive structure, there are a number of different possibilities for a new definition of recurrency, depending on how we aim to deal with indirect recursion. In order not to confuse all issues involved, we first provide a definition for programs P, relying only on direct recursion.

Definition 2.10 A (directly recursive) program P is *recurrent with respect to* S, if there exists a level mapping $|.|$ with respect to S, such that:

- for any A' representant of $\bar{A} \in R_S{\uparrow}\omega$,
- for any clause $H \leftarrow B_1, \ldots, B_n$ in P, such that $mgu(A', H) = \theta$ exists,
- for any atom $B_i, 1 \leq i \leq n$, with the same predicate symbol as H: $|A'| > |B_i\theta|$.

What is expressed in this definition is that for any two recursively descending calls with a same predicate symbol in any SLD-tree for (representants of) atoms in S, the level mapping's value should decrease. This condition has the advantage of being perfectly natural and therefore, of being easy to verify in an automated way. The only possible problem in view of automation is that it requires the computation of $R_S\uparrow\omega$. But, this problem is precisely the type of problem that can easily be solved (or approximated) through abstract interpretation (see section 4).

In the presence of indirect recursion, we need a more complex definition, that deals with the problem that a recursive call with a same predicate symbol as an ancestor call may only appear after a finite number of inference steps (instead of in the body of the particular instance of the applied clause). This can be done in several ways. We first provide a definition related to the concept of a resultant of a finite (incomplete) derivation. Based on this definition, we prove the equivalence with termination. After that, we provide a more practical condition, of which definition 2.10 is an obvious instance for the case of direct recursion.

First, we need some additional terminology.

Definition 2.11 Let A be an atom and $(G_0 = \leftarrow A)$, G_1, $G_2, \ldots$, G_n, $(n > 0)$, a finite, incomplete SLD-derivation for $(P, \leftarrow A)$. Let $\theta_1, \ldots, \theta_n$ be the corresponding sequence of substitutions, and let $\theta = \theta_1\theta_2\cdots\theta_n$ and $G_n = \leftarrow B_1, \ldots, B_m$. With the terminology of [Lloyd and Shepherdson 1991] we say that $A\theta \leftarrow B_1, \ldots, B_m$ is the *resultant* of the derivation.

Definition 2.12 A resultant $A\theta \leftarrow B_1, \ldots, B_m$ of a derivation $(G_0 = \leftarrow A), G_1, \ldots, G_n$, is a *recursive resultant for* A if there exists i $(1 \leq i \leq m)$, such that B_i has the same predicate symbol as A.

Definition 2.13 (recurrency wrt a set of atoms)
A program P is *recurrent with respect to* S, if there exists a level mapping, $|.|$, with respect to S, such that:

- for any A' representant of $\bar{A} \in R_S\uparrow\omega$,
- for any recursive resultant $A'\theta \leftarrow B_1, \ldots, B_m$, for A',
- for any atom B_i, $1 \leq i \leq m$, with the same predicate symbol as A': $|A'| > |B_i|$.

Proposition 2.14 If P is recurrent with respect to S, then P terminates with respect to S.

Just as in the framework of Bezem, the converse statement holds as well.

Theorem 2.15
P is recurrent with respect to S if and only if it is terminating with respect to S.

One of the nice consequences of this result is that we can now relate the concept of a recurrent program in the sense of [Bezem 1989] to recurrency with respect to a set of (ground) atoms.

Corollary 2.16 P is recurrent if and only if it is recurrent with respect to B_P.

It may seem surprising to the reader that two apparently very different notions such as recurrency and recurrency with respect to B_P coincide. It is our experience from our work in termination of unfolding in the context of partial deduction ([Bruynooghe *et al.* 1991]) that this is not unusual. The reason is that conditions occurring in these contexts require the *"existence"* of some well-founded measure. The specific properties of such measures can take totally different form without loosing the termination property. The only real difference lies in the practicality.

We conclude the section by introducing a condition that implies definition 2.13. This condition has the advantage over definition 2.13 that it does not rely on the verification of some property for each of a potentially infinite number of recursive resultants. Instead it only requires such a verification for a finite number of clauses, which can be characterised through the minimal, cyclic collections of P.

Definition 2.17 (minimal cyclic collection)
A minimal cyclic collection of P is a finite sequence of clauses of P:

$$\begin{array}{lcl} A_1 & \leftarrow & B_1^1, \ldots, A_2', \ldots, B_{n_1}^1 \\ & \vdots & \\ A_m & \leftarrow & B_1^m, \ldots, A_{m+1}', \ldots, B_{n_m}^m \end{array}$$

such that:

- for each pair $(i \neq j)$, the heads of the clauses, A_i and A_j, are atoms with distinct predicate symbols,
- A_i and A_i' have the same predicate symbols $(1 < i \leq m)$,
- A_{m+1}' has the same predicate symbol as A_1.

Only a finite number of minimal cyclic collections exists. They can easily be characterised and computed from the predicate dependency graph for P.

Proposition 2.18
Let $S \subseteq B_P^E$ and $|.|$ a *rigid* level mapping with respect to S, such that for any minimal cyclic collection of P (after standardizing apart),

$$\begin{array}{lcl} A_1 & \leftarrow & B_1^1, \ldots, A_2', \ldots, B_{n_1}^1 \\ & \vdots & \\ A_m & \leftarrow & B_1^m, \ldots, A_{m+1}', \ldots, B_{n_m}^m \end{array}$$

and for any $\bar{A}_1, \ldots, \bar{A}_m \in R_S\uparrow\omega$, with $A''_1, \ldots, A''_m$ as their respective representants, and $\theta_i = mgu(A_i, A''_i)$, $(1 \leq i \leq m)$, the following condition holds:

$$\left\{ \begin{array}{lcl} |A'_2\theta_1| & \geq & |A''_2| \\ & \vdots & \\ |A'_m\theta_{m-1}| & \geq & |A''_m| \end{array} \right\}$$
$$\Downarrow$$
$$|A''_1| > |A'_{m+1}\theta_m|.$$

Then, P is recurrent with respect to S.

The conditions in proposition 2.18 seem rather unnatural at first sight and need some clarification. First, observe that in the case of direct recursion — except for the rigidity of the level mapping — the conditions coincide with those of definition 2.10.

For the case of indirect recursion, the conditions that one would intuitively expect, are that for each minimal cyclic collection

$$\begin{array}{rcl} A_1 & \leftarrow & B^1_1, \ldots, A'_2, \ldots, B^1_{n_1} \\ & \vdots & \\ A_m & \leftarrow & B^m_1, \ldots, A'_{m+1}, \ldots, B^m_{n_m} \end{array}$$

and each A''_1 representant of $\bar{A}_1 \in R_S\uparrow\omega$, such that $\theta = mgu(A''_1, A_1)$ and $\theta_i = mgu(A'_i, A_i)$, $1 < i \leq m$, exist and are consistent, we have

$$|A''_1| > |A'_{m+1}\theta\theta_1 \cdots \theta_m|.$$

The problem is that such a condition is not correct. Consider the clauses:

$$\begin{array}{lcll} p(a,[\_|X]) & \leftarrow & p(b,X). & (cl1) \\ p(b,X) & \leftarrow & q(a,[\_|X]). & (cl2) \\ q(b,X) & \leftarrow & p(a,[\_|X]). & (cl3) \\ q(a,[\_|X]) & \leftarrow & q(b,X). & (cl4) \end{array}$$

There are 4 associated minimal collections: (cl1), (cl2,cl3), (cl3,cl2) and (cl4). Consider for instance the derivation $\leftarrow p(a,[\_,\_])$, $\leftarrow p(b,[\_])$, $\leftarrow q(a,[\_,\_])$, $\leftarrow q(b,[\_])$, $\leftarrow p(a,[\_,\_])$.

The problem is caused by resultants associated to derivations that start with a clause from one minimal cyclic collection — say (cl2) in the collection (cl2,cl3) — then shift to applying another collection, (cl4), and only after this resume the first collection and apply clause (cl3). The head of the third clause, $q(b,X)$, does not unify with $q(a,[\_|X'])$, and therefore, the condition on the cycle (cl2,cl3) can not be applied.

So, we have to impose the condition in proposition 2.18. It states that, even if the next call in the traversal of a minimal collection (A''_i) is not really related — as an instance — to a call we obtained earlier ($A'_i\theta_{i-1}$), but if — through the intermediate computation in another minimal collection — the level between these two has decreased anyway, then the final conclusion between the original call to the collection and the indirectly depending one must still hold. We will not discuss the condition any further here, but we will return to its practicality in section 4.

3 Acceptability with respect to a set of atoms

All definitions and propositions from the previous section can be specialised for the Prolog computation rule. Following [Apt and Pedreschi 1990], we call an SLD-derivation that uses Prolog's left-to-right computation rule, an *LD-derivation.*

Definition 3.1 (left termination wrt S) Let S be a subset of B^E_P. A program P is left-terminating with respect to S if for any representant A of any element of S, every LD-derivation is finite.

Recall definitions 2.4 and 2.5. The motivation behind these definitions was finding an overestimation of all calls that are possible in any SLD-derivation using an arbitrary computation rule. The fact that no fixed computation rule is used, forces us to take the closure under all possible instantiations in definition 2.5, and hence $R_S\uparrow\omega$ contains in general a lot more calls than can really occur when a particular computation rule is chosen.

In this section, we focus our analysis on computations that use Prolog's left-to-right computation rule. Therefore, adapted definitions of the $\mathcal{T}_P^{-1}$ and R_S functions are needed.

Definition 3.2 For any $T \subseteq B^E_P$, define: $\mathcal{P}_P^{-1}(T) = \{\overline{B_i\theta\sigma_1\cdots\sigma_{i-1}} \in B^E_P \mid A'$ is a representant of $\bar{A} \in T$, $H \leftarrow B_1, \ldots, B_n$ is a clause in P, $\theta = mgu(A', H)$, $1 \leq i \leq n$, $\exists\sigma_1, \ldots, \sigma_{i-1}$, such that $\forall j = 1, \ldots, i-1$: σ_j is an answer for $(P, \leftarrow B_j\theta\sigma_1\cdots\sigma_{j-1})\}$.

The answer substitutions σ_j are computed using LD-resolution. Let $\mathcal{H}^{l-r}_S$ denote $\{T \in 2^{B^E_P} \mid S \subseteq T\}$.

Definition 3.3 $R^{l-r}_S : \mathcal{H}^{l-r}_S \rightarrow \mathcal{H}^{l-r}_S : R^{l-r}_S(T) = T \cup \mathcal{P}_P^{-1}(T)$

In a completely analogous way as in the previous section, we find that R^{l-r}_S is continuous. Hence, the least fix point $R^{l-r}_S\uparrow\omega$ contains all atoms that can possibly occur as a call when P is executed under the Prolog computation rule, and when a representant of an element from S is used as query.

Level mappings are now defined on R^{l-r}_S. Recursive resultants are constructed using the left-to-right computation rule. This allows us to consider only recursive resultants of the form $p(s_1, \ldots, s_n) \leftarrow p(t_1, \ldots, t_n), B_2, \ldots, B_m$. The analogue of recurrency with respect to a set S of atoms, is *acceptability with respect to* S.

Definition 3.4 (acceptability wrt a set of atoms) A program P is *acceptable with respect to* S, if there exists a level mapping $|.|$ with respect to S, such that for any $p(s_1, \ldots, s_n)$, representant of an element in $R^{l-r}_S\uparrow\omega$, and for any recursive resultant $p(s_1, \ldots, s_n)\theta \leftarrow p(t_1, \ldots, t_n), B_2, \ldots, B_m$: $|p(s_1, \ldots, s_n)| > |p(t_1, \ldots, t_n)|$.

Theorem 3.5
P is acceptable with respect to S if and only if it is left-terminating with respect to S.

As in section 2, we provide a more practical, sufficient condition. The result is completely analogous to proposition 2.18.

Proposition 3.6
Let $S \subseteq B_P^E$ and $|.|$ a level mapping with respect to S, such that for any minimal cyclic collection of P (after standardizing apart),

$$\begin{array}{rcl} A_1 & \leftarrow & B_1^1, \ldots, B_{i_1}^1, A_2', \ldots, B_{n_1}^1 \\ & \vdots & \\ A_m & \leftarrow & B_1^m, \ldots, B_{i_m}^m, A_{m+1}', \ldots, B_{n_m}^m \end{array}$$

and for any $\bar{A}_1, \ldots, \bar{A}_m \in R_S^{l-r}\uparrow\omega$, with $A_1'', \ldots, A_m''$ as their respective representants, and with $\theta_j = mgu(A_j, A_j'')$ $(1 \leq j \leq m)$ and σ_k^j is a computed answer substitution for $(P, \leftarrow B_k^j\theta_j\sigma_1^j \cdots \sigma_{k-1}^j)$ $(1 \leq k \leq i_j)$, the following condition holds:

$$\left\{ \begin{array}{lcl} |A_2'\theta_1\sigma_1^1 \cdots \sigma_{i_1}^1| & \geq & |A_2''| \\ & \vdots & \\ |A_m'\theta_{m-1}\sigma_1^{m-1} \cdots \sigma_{i_{m-1}}^{m-1}| & \geq & |A_m''| \end{array} \right\}$$
$$\Downarrow$$
$$|A_1''| > |A_{m+1}'\theta_m\sigma_1^m \cdots \sigma_{i_m}^m|,$$

Then, P is acceptable with respect to S.

4 Practicality and automation

A fully automated technique needs to address the following issues:

- safe approximations of $R_S\uparrow\omega$ and $R_S^{l-r}\uparrow\omega$ must be computed,
- precise and natural level mappings are needed, and
- the conditions in propositions 2.18 and 3.6 must be automatically verifiable.

For left termination, there is one extra issue:

- some properties of the answer substitutions for the atoms in $R_S^{l-r}\uparrow\omega$ are needed; in particular, after application of a computed answer substitution we want an estimation of the relationship between the sizes of the arguments of the atoms in $R_S^{l-r}\uparrow\omega$.

Concerning the first issue, observe that in practice, the sets of atoms S in the framework are likely to be specified in terms of call patterns over some abstract domain. The framework contains no implicit restriction on the kind of abstractions that are used for this purpose. They could be either expressing mode or type information, or even combined mode and type information — as in the rigid or integrated types of [Janssens and Bruynooghe 1990]. Abstract interpretation can be applied to automatically infer a safe approximation of $R_S\uparrow\omega$ or $R_S^{l-r}\uparrow\omega$ (see [Janssens and Bruynooghe 1990]).

Automated techniques for proving termination use various types of *norms*. A *norm* is a mapping $||.|| : U_P^E \rightarrow \mathbb{N}$. Several examples of norms can be found in the literature. When dealing with lists, it is often appropriate to use *list-length*, which gives the depth of the rightmost branch in the tree representation of the term. A more general norm is *term-size*, which counts the number of function symbols in a term. Another frequently used norm is *term-depth*, which gives the maximum depth of (the tree representation of) a term.

However, we restrict ourselves to *semi-linear* norms, which were defined in [Bossi *et al.* 1991].

Definition 4.1 (semi-linear norm)
A norm $||.||$ is *semi-linear* if it satisfies the folowing conditions:

- $||V|| = 0$ if V is a variable, and
- $||f(t_1, \ldots, t_n)|| = c + ||t_{i_1}|| + \cdots + ||t_{i_m}||$ where $c \in \mathbb{N}$, $1 \leq i_1 < \cdots < i_m \leq n$ and $c, i_1, \ldots, i_m$ depend only on f/n.

Examples of semi-linear norms are *list-length* and *term-size*.

As was pointed out in [Bossi *et al.* 1991], proving termination is significantly facilitated if the norm of a term remains invariant under substitution. Such terms are called *rigid*.

Definition 4.2 (rigid term; see [Bossi *et al.* 1991])
Let $||.||$ be a (semi-linear) norm. A term t is *rigid* with respect to $||.||$ if for any substitution σ, $||t\sigma|| = ||t||$.

Rigidity is a generalisation of groundness; by using this concept it is possible to avoid restricting the definition of a norm to ground terms only, a restriction that is often found in the literature.

Given a semi-linear norm and a set of atoms S, a very natural level mapping with respect to S can be associated to them.

Definition 4.3 (natural level mapping)
Given is a semi-linear norm $||.||$ and a set of atoms S. $|.|_{nat}$, the *natural level mapping induced by* S, is defined as follows: $\forall p(t_1, \ldots, t_n) \in R_S\uparrow\omega$:

$$\begin{array}{rcll} |p(t_1, \ldots, t_n)|_{nat} & = & \sum_{i \in I} ||t_i||, & \text{if } I \neq \emptyset \\ & = & 0 & \text{otherwise,} \end{array}$$

with $I = \{i \mid \forall p(u_1, \ldots, u_n) \in R_S\uparrow\omega : u_i \text{ is rigid}\}$.

Let us illustrate the practicality of such mappings — and of the framework itself — with some examples.

Example 4.4
Reconsider example 1.4 from the introduction. Assume that $S = \{p(x) \mid x \text{ is a nil-terminated list}\}$. Let $||.||_l$ be the *list-length* norm. The argument positions of all atoms in $R_S\uparrow\omega$ are rigid under this norm. So, $|p(x)|_{nat} = ||x||_l$ and $|q(x)|_{nat} = ||x||_l$. The program is directly recursive, so that it suffices to verify the conditions of definition 2.10.

For the clause $p([H|T]) \leftarrow q([H|T]), p(T)$ and for each call $p(x) \in R_S\uparrow\omega$, with $\theta = mgu(x, [H|T])$, we have $|p(x)|_{nat} > |p(T)\theta|_{nat}$. By the same argument, the condition on the clause $q([H|T]) \leftarrow q(T)$ holds as well. Thus, the program is recurrent with respect to S under the natural, *list-length* level mapping with respect to S.

As a second example, we take a program with indirect recursion. It defines some form of well-formed expressions built from integers and the function symbols $+/2$, $*/2$ and $-/1$.

Example 4.5

$$\begin{array}{lcll} e(X+Y) & \leftarrow & f(X), e(Y). & (cl1) \\ e(X) & \leftarrow & f(X). & (cl2) \\ \\ f(X*Y) & \leftarrow & g(X), f(Y). & (cl3) \\ f(X) & \leftarrow & g(X). & (cl4) \\ \\ g(-(X)) & \leftarrow & e(X). & (cl5) \\ g(X) & \leftarrow & integer(X). & (cl6) \end{array}$$

The obvious choice for a level mapping for this program is *term-size*. However, the program is not recurrent in the sense of [Bezem 1989] with respect to this norm. Since it is clearly terminating, a level mapping exists. The most natural mapping (in the sense of [Bezem 1989]) we were able to come up with is:

$$\begin{array}{lcl} |e(x)| & = & 3 \times term\text{-}size(x) + 2 \\ |f(x)| & = & 3 \times term\text{-}size(x) + 1 \\ |g(x)| & = & 3 \times term\text{-}size(x). \end{array}$$

In the context of our framework, consider the set $S = \{e(x) \mid x \text{ is ground}\}$. Through abstract interpretation, we can find that $R_S\uparrow\omega \subseteq B_P$.

Let $||.||_t$ be the *term-size* norm. Again, the argument positions of all atoms in $R_S\uparrow\omega$ are rigid (even ground) under this norm. Thus, $|e(x)|_{nat} = ||x||_t$, $|f(x)|_{nat} = ||x||_t$ and $|g(x)|_{nat} = ||x||_t$. The program contains essentially[1] 6 minimal, cyclic collections: (cl1), (cl3), (cl1, cl3, cl5), (cl1, cl4, cl5), (cl2, cl3, cl5), (cl2, cl4, cl5).

Let us consider, as an example, the third collection:

$$\begin{array}{lcl} e(X+Y) & \leftarrow & f(X), e(Y). \\ f(X'*Y') & \leftarrow & g(X'), f(Y'). \\ g(-(X'')) & \leftarrow & e(X''). \end{array}$$

[1]Since collections are sequences of clauses, cyclic permutations should be considered as well.

Assume that $e(x)$, $f(y)$ and $g(z)$ are any atoms with ground terms x, y and z, and that:

$$\begin{array}{l} \theta_1 = mgu(e(x), e(X+Y)) \\ \theta_2 = mgu(f(y), f(X'*Y')) \\ \theta_3 = mgu(g(z), g(-(X''))). \end{array}$$

Also assume that $|f(X)\theta_1| \geq |f(y)|$ and $|g(X')\theta_2| \geq |g(z)|$. We then have $|e(x)| > |f(X)\theta_1| \geq |f(y)| > |g(X')\theta_2| \geq |g(z)| > |e(X'')\theta_3|$, so that $|e(x)| > |e(X'')\theta_3|$, and the conditions of proposition 2.18 (for the third cycle) are fulfilled. All other cycles can be verified in a similar way. The conclusion is that the program is recurrent with respect to S and the very natural *term-size* level mapping.

In the context of left termination, definition 4.3 can be adapted to produce equally natural level mappings with respect to a set S. Obviously, $R_S\uparrow\omega$ should be replaced by $R_S^{l-r}\uparrow\omega$. In the context of left termination there is an extra issue, namely, (an approximation of) the set of possible answer substitutions for an atom is needed. The next example illustrates how this is handled.

Example 4.6

$$\begin{array}{l} p([],[]). \\ p([H|T],[G|S]) \leftarrow d(G,[H|T],U), p(U,S). \\ \\ d(H,[H|T],T). \\ d(G,[H|T],[H|U]) \leftarrow d(G,T,U). \end{array}$$

Assume that $S = \{p(x,y) \mid x \text{ is a nil-terminated list and } y \text{ is free}\}$. Notice that $R_S\uparrow\omega$ contains the set $\{p(x,y) \mid x \text{ and } y \text{ are free variables}\}$. We are not able to define a level mapping on $R_S\uparrow\omega$ that can be used to prove recurrency with respect to S. This is not surprising, since P is not terminating with respect to S.

However, program P is left terminating with respect to S. We prove this by showing that P is acceptable with respect to S. The set $R_S^{l-r}\uparrow\omega$ is the union of $\{p(x,y) \mid x \text{ is a nil-terminated list and } y \text{ is free}\}$ and $\{d(x,y,z) \mid x \text{ and } z \text{ are free variables and } y \text{ is a nil-terminated list}\}$. This can be found by using abstract interpretation. Since there is only direct recursion in program P, it suffices to show that: (1) for any $p(x,y) \in R_S^{l-r}\uparrow\omega$, $|p(x,y)| > |p(U,S)\theta\sigma|$, where $\theta = mgu(p(x,y), p([H|T],[G|S]))$ and σ is a computed answer substitution for $(P, \leftarrow d(G,[H|T],U)\theta)$, and (2) for any $d(x,y,z) \in R_S^{l-r}\uparrow\omega$, $|d(x,y,z)| > |d(G,T,U)\theta|$, where $\theta = mgu(d(x,y,z), d(G,[H|T],[H|U]))$.

Now, in practice, the statement "σ is a computed answer substitution for $(P, \leftarrow d(G,[H|T],U)\theta)$" can be replaced by "$||[H|T]\theta\sigma||_l = ||U\theta\sigma||_l + 1$". This latter statement is a so-called *linear size relation*, which expresses a relation between the norms of the arguments of the atoms in the success set of the program. Alternatively, it can also be interpreted as a (non-Herbrand)

model of the program. For more details we refer to [Verschaetse and De Schreye 1992], where we describe an automated technique for deriving linear size relations.

By taking this information into account, and by taking $|p(x,y)| = ||x||_l$ for any $p(x,y) \in R_S^{l-r} \uparrow \omega$ — notice that x is rigid with respect to $||.||_l$ — we find: $|p(x,y)| = ||x||_l = ||[H|T]\theta||_l = ||[H|T]\theta\sigma||_l = ||U\theta\sigma||_l + 1 > ||U\theta\sigma||_l = |p(U,S)\theta\sigma|$.

The second inequality, $|d(x,y,z)| > |d(G,T,U)\theta|$, is more easy to prove. This time, the *list-length* of the second argument can be taken as level mapping. Since both inequalities hold, we can conclude that the program is acceptable with respect to the set of atoms that is considered.

Automatic verification of the conditions for recurrency and acceptability is handled by reformulating them into a problem of checking the solvability of a linear system of inequalities. This part of the work is described in more detail in [De Schreye and Verschaetse 1992].

References

[Apt and Pedreschi 1990] K. R. Apt and D. Pedreschi. Studies in pure Prolog: termination. In *Proceedings Esprit symposium on computational logic*, pages 150–176, Brussels, November 1990.

[Baudinet 1988] M. Baudinet. Proving termination properties of Prolog programs: a semantic approach. In *Proceedings of the 3rd IEEE symposium on logic in computer science*, pages 336–347, Edinburgh, July 1988. Revised version to appear in Journal of Logic Programming.

[Bezem 1989] M. Bezem. Characterizing termination of logic programs with level mappings. In *Proceedings NACLP'89*, pages 69–80, 1989.

[Bossi *et al.* 1991] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. Technical Report 4/29, CNR, Department of Mathematics, University of Padova, March 1991.

[Bruynooghe *et al.* 1991] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction of logic programs. In *Proceedings ILPS'91*, pages 117–131, San Diego, October 1991. MIT Press.

[Cavedon 1989] L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In *Proceedings ICLP'89*, pages 571–584, June 1989.

[De Schreye and Verschaetse 1992] D. De Schreye and K. Verschaetse. Termination analysis of definite logic programs with respect to call patterns. Technical Report CW 138, Department Computer Science, K.U.Leuven, January 1992.

[Falaschi *et al.* 1989] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[Janssens and Bruynooghe 1990] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. Technical Report CW 107, Department of Computer Science, K.U.Leuven, March 1990. To appear in Journal of Logic Programming, in print.

[Lloyd and Shepherdson 1991] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3 & 4):217–242, October/November 1991.

[Plümer 1990] L. Plümer. *Termination proofs for logic programs.* Lecture Notes in Artificial Intelligence 446. Springer-Verlag, 1990.

[Sohn and Van Gelder 1991] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Proceedings 10th symposium on principles of database systems*, pages 216–226. Acm Press, May 1991.

[Ullman and Van Gelder 1988] J. D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal ACM*, 35(2):345–373, April 1988.

[Vasak and Potter 1986] T. Vasak and J. Potter. Characterisation of terminating logic programs. In *Proceedings 1986 symposium on logic programming*, pages 140–147, Salt Lake City, 1986.

[Verschaetse and De Schreye 1991] K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In *Proceedings ICLP'91*, pages 301–315, Paris, June 1991. MIT Press.

[Verschaetse and De Schreye 1992] K. Verschaetse and D. De Schreye. Automatic derivation of linear size relations. Technical Report CW 139, Department Computer Science, K.U.Leuven, January 1992.

[Wang and Shyamasundar 1990] B. Wang and R. K. Shyamasundar. Towards a characterization of termination of logic programs. In *Proceedings of international workshop PLILP'90*, Lecture Notes in Computer Science 456, pages 204–221, Linköping, August 1990. Springer-Verlag.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Automatic Verification of GHC-Programs: Termination

Lutz Plümer
Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik III
D-5300 Bonn 1, Römerstr. 164
lutz@uran.informatik.uni-bonn.de

Abstract

We present an efficient technique for the automatic generation of termination proofs for concurrent logic programs, taking Guarded Horn Clauses (GHC) as an example. In contrast to Prolog's strict left to right order of evaluation, termination proofs for concurrent languages are complicated by a more sophisticated mechanism of subgoal selection. We introduce the notion of directed GHC programs and show that for this class of programs goal reductions can be simulated by Prolog-like derivations. We give a sufficient criterion for directedness. Static program analysis techniques developed for Prolog can thus be applied, albeit with some important modifications.

1. Introduction

With regard to termination it is useful to distinguish between two types of software systems or programs: transformational and reactive [HAP85]. A transformational system receives an input at the beginning of its operation and yields an output at the end. If the problem at hand is decidable, termination of the process is surely a desirable property. Reactive systems, on the other hand, are designed to maintain some interaction with their environment. Some of them, for instance operating systems and database management systems, ideally never terminate and do not yield a final result at all. Based on the process interpretation of Horn clause logic, concurrent logic programming systems have been designed for many different applications including reactive systems and transformational parallel systems. While for some of them termination is not a desirable property, for others it is. In this paper we discuss how automatic termination proofs for concurrent logic programs can be achieved automatically.

Automatic proof techniques for pure Prolog programs have been described in several papers including [ULG88] and [PLU90a]. Prolog is characterized by a fixed computation rule which always selects the leftmost atom. Deterministic subgoal selection and strict left to right order of evaluation cannot be assumed for the concurrent languages.

Static program analysis techniques, which are well established for sequential Prolog, such as abstract interpretation, inductive assertions and termination proof techniques, substantially depend on the strict left to right order of evaluation in most cases and thus cannot easily be applied to concurrent languages. Concurrent languages delay subgoals which are not sufficiently instantiated. Goals which loop forever when evaluated by a Prolog interpreter may deadlock in the context of a concurrent language. These phenomena may suggest that termination proofs for concurrent logic programs require a different approach. This paper, however, shows that techniques which have been established for pure Prolog are still useful in the context of concurrency.

Our starting point is the question under which conditions reductions of a concurrent logic program can be simulated by Prolog-like derivations. We take Guarded Horn Clauses (GHC, see [UED86]) as an example, but our results can easily be extended to other concurrent logic programming languages such as PARLOG, (Flat) Concurrent Prolog or FCP(:). Our basic assumptions are the restriction of unification to input matching, nondeterministic subgoal selection and resuming of subgoals which are not sufficiently instantiated. Since we consider all possible derivations, the commit operator does not need special attention.

In general simulation is not possible: if there is a GHC-derivation of g' from g, g' cannot necessarily be derived with Prolog's computation rule.

One could now try to augment simulation by program transformation. Let, for instance, P' be derived from P by including all clause body permutations. Although P' may be exponentially larger than P, there are still derivations which are not captured.

Example 1.1:

Program: $p \leftarrow q,r.$ $q \leftarrow s,t.$ $r \leftarrow u,v.$
$s.$ $v.$

Goal: $\leftarrow p$

This goal can be reduced to $\leftarrow t,u$ by nondeterministic subgoal selection, but not by a Prolog like computation, even after adding the following clauses:

$p \leftarrow r,q.$ $q \leftarrow t,s.$ $r \leftarrow v,u.$

The reason is that in order to derive $\leftarrow t,u$, the subderivations of $\leftarrow q$ and $\leftarrow r$ have to be interleaved.

The question arises whether there is an interesting subclass for which appropriate simulations can be defined. Such a class of programs will be discussed in Section 3. The main idea is to assume that if a subgoal p may produce some output on which evaluation of another subgoal q depends, then p is smaller w.r.t. some partial ordering. Whether a program maintains such a property, which we will call directedness, is undecidable. We will then introduce the stronger notion of well-formedness which can be checked syntactically. Well-formedness is related to directionality, which is discussed in [GRE87]. Well-formedness is sufficient but not necessary for directedness, and it will turn out that quite a lot of nontrivial programs (including for instance systolic programs as discussed in [SHA87a] and most of the examples given in [TIC91]) fall into this category. In Section 5 we will demonstrate how termination proof techniques which have been established for pure Prolog can be generalized such that they apply to well-formed GHC programs.

The rest of this paper is organized as follows. Section 2 provides basic notions. Section 3 introduces the notion of directed programs and shows that this property is undecidable. It provides the notion of well-formedness and shows that it is sufficient for directedness. Section 4 discusses oriented and data driven computation and shows that after some simple program transformation derivations with directed GHC-programs can be simulated by Prolog-like derivations. Using the notion of S-models introduced in [FLP89], Sections 5 and 6 show how termination proofs can be achieved automatically.

2. Basic Notions

We use standard notation and terminology of Lloyd [Llo87] or Apt [APT90]. Following [APP90] we will say LD-resolution (LD-derivation, LD-refutation LD-tree) for SLD-resolution (SLD-derivation, SLD-refutation SLD-tree) with the leftmost selection rule characteristic for Prolog.

Next we define GHC programs following [UED87] and [UED88].

A GHC program is a set of guarded Horn clauses of the following form:

$$H \leftarrow G_1,\ldots,G_m \mid B_1,\ldots,B_n. \qquad (m > 0, n > 0)$$

where H, $G_1,\ldots,G_m$ and $B_1,\ldots,B_n$ are atomic formulas. H is called a clause head, the G_i's are called guard goals and the B_i's are called body goals. The part of a clause before '|' is called a guard, and the part after '|' is called a body. One predicate, namely '=', is predefined by the language. It unifies two terms.

Declaratively, the commitment operator '|' denotes conjunction, and the above guarded Horn clause is read as "H is implied by $G_1,\ldots,G_m$ and $B_1,\ldots,B_n$". The operational semantics of GHC is given by parallel input resolution restricted by the following two rules:

Rule of Suspension:

- Unification invoked directly or indirectly in the guard of a clause C called by a goal G (i.e. unification of G with the head of C and any unification invoked by solving the guard goals of C) cannot instantiate the goal G.
- Unification invoked directly or indirectly in the body of a clause C called by a goal G cannot instantiate the guard of C or G until C is selected for commitment.

Rule of Commitment:

- When some clause C called by a goal G succeeds in solving (see below) its guard, the clause C tries to be selected for subsequent execution (i.e., proof) of G. To be selected, C must first confirm that no other clauses in the program have been selected for G. If confirmed, C is selected indivisibly, and the execution of G is said to be committed to the clause C.

An important consequence is that any unification intended to export bindings to the calling goal must be specified in the clause body and use the predefined predicate '='.

The operational semantics of GHC is a sound - albeit not complete - proof procedure for Horn clause programs: if $\leftarrow B$ succeeds with answer substitution θ, then $\forall(B\theta)$ is a logical consequence of the program.

Subsequently, we may find it convenient to denote a goal g by the pair $\langle G;\theta\rangle$, i.e. $g = G\theta$. A single derivation step reducing the i-th atom of G using clause C and applying mgu θ' is denoted by $\langle G;\theta\rangle \rightarrow_{i;C} \langle G';\theta\theta'\rangle$. Subscripts may be omitted.

3. Directed Programs

An *annotation* d_p for an n-ary predicate symbol p is a function from {1,...,n} to {+,-} where '+' stands for input and '-' for output. We will write p(+,+,-) in order to state that the first two arguments of p are input and the last is output.

A goal atom A *generates (consumes)* a variable v if v occurs at an output (input) position of A. A *is generator* for B, if some variable v occurs at an output position of A and at an input position of B; in this case, B is *consumer of* A.

Let $\hat{t}$ denote a tuple of terms. A derivation $\langle p(\hat{t});\varepsilon\rangle \rightarrow^* \langle G;\theta\rangle$ *respects* the input annotation of p if $v\theta = v$ for every variable v occurring at an input position of $p(\hat{t})$.

A *goal* is *directed* if there is a linear ordering among its atoms such that if A_i is generator for A_j then A_i precedes A_j in that ordering. A *program* is *directed*, if all its derivations *respect directedness*, i.e., all goals derived from a directed goal are directed. Note that directedness of a goal is a static

property which can be checked syntactically. Directedness of a program, however, is a dynamic property.

Theorem 3.1: It is undecidable, whether a program is directed.

Proof: Let $t_M(X)$ be a directed GHC simulation of a Turing machine M for a language L which binds X to halt if and only if M applied to the empty tape halts. Such a simulation is for instance described in [PLU90b]. Next consider the following procedures p_M and q:

$p_M(X,Y) \leftarrow t_M(A)$, q(A,X,Y).
q(halt,X,X).

and the (directed) goal

$\leftarrow$ r(X,Y), s(Y,Z), p_M(X,Z).

The following annotations are given:

t_M(-). q(+,-,-). p_M(-,-). r(+,-). s(+,-).

If M halts on the empty tape, t_M(A) will bind A to 'halt', p_M(X,Y) will identify X and Y and thus the given goal can be reduced to the undirected goal $\leftarrow$ r(X,Y), s(Y,X). Decidability of program directedness would thus imply solvability of the halting problem: contradiction. ■

Next we introduce the notion of well-formedness of a program w.r.t. a given annotation and show that this property is sufficient for directedness.

A *goal* is *well-formed* if it is directed, generators precede consumers in its textual ordering, and its output is unrestricted. Output of a *goal* is *unrestricted* if all its output arguments are distinct variables which do not occur (i) at an output position of another goal atom and (ii) at an input position of the same atom.

A *program* P is *well-formed* if the following conditions are satisfied by each clause $H \leftarrow G_1,\ldots,G_m \mid B_1,\ldots,B_n$ in P:

- $\leftarrow B_1,\ldots,B_n$ is well-formed
- the input variables of H do not occur at output positions of body atoms.

The predicate '=' has the annotation '- = -'. It is convenient to have two related primitives: '==' (test) and '$\Leftarrow$' (matching) which have the same declarative reading as '=' but different annotations, namely '+ == +' and '- $\Leftarrow$ +'.

Note that the goal $\leftarrow$ r(X,Y),s(Y,Z), p_M(X,Z) is not well-formed because its output is restricted: Z has two output occurrences.

The next example is taken from [UED86]:

Example 1: Generating primes

primes(*Max*,**Ps**) $\leftarrow$ true |
gen(2,*Max*,**Ns**),sift(*Ns*,**Ps**).
gen(*N*,*Max*,**Ns**) $\leftarrow$ *N* $\leq$ *Max* | **N1** $\Leftarrow$ *N* + *1*,
gen(*N1*,*Max*,**Ns1**), **Ns** $\Leftarrow$ *[N/Ns1]*.
gen(*N*,*Max*,**Ns**) $\leftarrow$ *N* > *Max* | **Ns** $\Leftarrow$ *[]*.
sift(*[P/Xs]*,**Zs**) $\leftarrow$ filter(*P*,*Xs*,**Ys**),sift(*Ys*,**Zs1**),
Zs $\Leftarrow$ *[P/Zs1]*.
sift(*[]*,**Zs**) $\leftarrow$ **Zs** $\Leftarrow$ *[]*.
filter(*P*,*[X/Xs]*,**Ys**) $\leftarrow$ *X mod P == 0* | filter(*P*,*Xs*,**Ys**).
filter(*P*,*[X/Xs]*,**Ys**) $\leftarrow$ *X mod P* $\neq$ *0* | filter(*P*,*Xs*,**Ys1**),
Ys $\Leftarrow$ *[X/Ys1]*.
filter(*P*,*[]*,**Ys**) $\leftarrow$ **Ys** $\Leftarrow$ *[]*.

primes(+,-). gen(+,+,-). sift(+,-). filter(+,+,-).

The call primes(Max,Ps) returns through Ps a stream of primes up to Max. The stream of primes is generated from a stream of integers by filtering out the multiples of primes. For each prime P, a filter goal filter(P,Xs,Ys) is generated which filters out the multiples of P from the stream Xs, yielding Ys.

In this example all input terms are italic and all output terms are bold. It can easily be seen that this program is well-formed.

Another example for a well-formed program is quicksort. The call qsort([H|L],S) returns through S an ordered version of the list [H|L]. To sort [H|L] L is split into two lists L_1 and L_2 which are itself sorted by recursive calls to qsort.

Example 2: Quicksort

q_1: qsort(*[]*,**L**) $\leftarrow$ **L** $\Leftarrow$ *[]*.
q_2: qsort(*[H/L]*,**S**) $\leftarrow$ split(*L*,*H*,**A**,**B**),
qsort(*A*,$\mathbf{A_1}$), qsort(*B*,$\mathbf{B_1}$),
append(A_1,*[H/*B_1*]*,**S**).
s_1: split(*[]*,*X*,$\mathbf{L_1}$,$\mathbf{L_2}$) $\leftarrow$ $\mathbf{L_1} \Leftarrow$ *[]*, $\mathbf{L_2} \Leftarrow$ *[]*.
s_2: split(*[X/Xs]*,*Y*,$\mathbf{L_1'}$,$\mathbf{L_2}$) $\leftarrow$ *X* $\leq$ *Y* |
split(*Xs*,*Y*,$\mathbf{L_1}$,$\mathbf{L_2}$),
$\mathbf{L_1'} \Leftarrow$ *[X/*L_1*]*.
s_3: split(*[X/Xs]*,*Y*,$\mathbf{L_1}$,$\mathbf{L_2'}$) $\leftarrow$ *X* > *Y* | split(*Xs*,*Y*,$\mathbf{L_1}$,$\mathbf{L_2}$),
$\mathbf{L_2'} \Leftarrow$ *[X/*L_2*]*.
a_1: append(*[]*,L_1,$\mathbf{L_2}$) $\leftarrow$ $\mathbf{L_2} \Leftarrow L_1$.
a_2: append(*[H/*L_1*]*,L_2,$\mathbf{L_3}$) $\leftarrow$ append(L_1,L_2,$\mathbf{L_3'}$),
$\mathbf{L_3} \Leftarrow$ *[H/*L_3*']*.
split(+,+,-,-). qsort(+,-). append(+,+,-).

Theorem 3.2: Let P be a well-formed program, g a well-formed goal and g $\rightarrow^*$ g' a GHC-derivation. Then g' is well-formed.

Proof: See [PLU92].

Well-formed programs respect input annotations:

Theorem 3.3: Let $\langle p(\hat{t}),\varepsilon\rangle \rightarrow^* \langle G';\theta\rangle$ be a derivation and v an input variable of $p(\hat{t})$. Then $v\theta = v$.

Proof: Goal variables can only be bound by transitions applying '=' or '$\Leftarrow$', since in the other cases matching substitutions are applied. Since both arguments of '=' are output, and '$\Leftarrow$' also binds only output variables, input variables cannot be bound. ■

4. Oriented and Data Driven Computations

Our next aim is to show that derivations of directed programs can be simulated by derivations which are similar to LD-derivations. In this context we find it convenient to use the notational framework of SLD-resolution and to regard GHC-derivations as a special case.

We say that an SLD-derivation is *data driven*, if for each resolution step with selected atom A, applied clause C and mgu θ either C is the unit clause ($X = X \leftarrow$ true.) or C is $B \leftarrow B_1,\ldots,B_n$ and $A = B\theta$. Data driven derivations are the same as GHC derivations of programs with empty guards. The assumption that guards are empty is without loss of generality in this context.

Next we consider oriented computation rules. Oriented computation rules are similar to LD-resolution in the sense that goal reduction strictly proceeds from left to right. They are more general since the selected atom is not necessarily the leftmost one. However, if the selected atom is not leftmost, its left neighbors will not be selected in any future derivation step.

More formally, we define: A computation rule R is *oriented*, if every derivation $\langle G_0;\varepsilon\rangle \rightarrow \ldots\langle G_i;\theta_i\rangle \rightarrow \ldots$ via R satisfies the following property: If in G_i an atom A_k is selected, and A_j ($j < k$), is an atom on the left of A_k, no further instantiated version of A_j will be selected in any future derivation step.

Our next aim is to show that, for directed programs, any data driven derivation can be simulated by an equivalent data driven derivation which is oriented. To prove the following theorem, we need a slightly generalized version of the switching lemma given in [LLO87]. Here $g \rightarrow_{i;C;\theta} g'$ denotes a single derivation step where the i-th atom of g is resolved with clause C using mgu θ.

Lemma 4.1: Let g_{k+2} be derived from g_k via $g_k \rightarrow_{i;C_{k+1};\theta_{k+1}} g_{k+1} \rightarrow_{j;C_{k+2};\theta_{k+2}} g_{k+2}$. Then there is a derivation $g_k \rightarrow_{j;C_{k+2}';\theta_{k+1}'} g_{k+1}' \rightarrow_{i;C_{k+1}';\theta_{k+2}'} g_{k+2}'$ such that g_{k+2}' is a variant of g_{k+2} and C_{k+1}', C_{k+2}' are variants of C_{k+2} and C_{k+1}.

Proof: [LLO87] The difference between this and Lloyds version is that the latter refers to SLD-refutations, while ours refers to (possibly partial) derivations. His proof, however, also applies to our version. ■

Theorem 4.2: Let P be a directed program and $\langle G_0;\varepsilon\rangle$ a directed goal. Let $D = \langle G_0;\varepsilon\rangle\rightarrow\ldots\langle G_k;\theta_k\rangle$ be a data driven derivation using the clause sequence $C_1,\ldots,C_k$. Then there is another data driven derivation D': $\langle G_0;\varepsilon\rangle\rightarrow\ldots\langle G_k';\theta_k'\rangle$ using a clause sequence $C_{i_1}',\ldots,C_{i_k}'$, where $\langle i_1,\ldots,i_k\rangle$ is a permutation of $\langle 1,\ldots,k\rangle$, each C_i' is a variant of C_i and $G_k'\theta_k'$ is a variant of $G_k\theta_k$, and D' is oriented.

Proof: Let g_j be the first goal in D where orientation is violated, i.e. there is the following situation:

$g_i : \langle B_1,\ldots,R,\ldots,R',\ldots\ldots;\theta_i\rangle$

...

$g_j : \langle B_1,\ldots,R,\ldots\ldots;\theta_j\rangle$

R' is selected in g_i and R is selected in g_j. Now we switch subgoal selection in g_{j-1} and g_j and get a new derivation D*. In D* we look again for the first goal violating the orientation. After a finite number of iterations, we arrive at a derivation D' which is oriented. It remains to be shown that D* (and thus D') is still data driven.

Note that up to g_{j-1} both derivations are identical. Above, the switching lemma implies that, from g_{j+1} on, the goals of D' are variants of those of D.

Now let Q be the selected goal of G_{j-1}. Since orientation is violated for the first time in G_j, Q is to the right of R. (If $i = j - 1$ then $Q = R'$, and otherwise j-1 would have the first violation of orientation.) Since $g_{j-1} = \langle G_{j-1};\theta_{j-1}\rangle$ is directed, $Q\theta_{j-1}$ is not a generator of $R\theta_{j-1}$ and thus $R\theta_{j-1}$ and $R\theta_j$ are variants. Let H be the head of the clause applied to resolve R in $\langle G_j;\theta_j\rangle$. Since D is data driven, $R\theta_{j-1} = H\sigma$ for some σ, and so $R\theta_j = H\sigma'$ for some σ'. Thus D' is data driven. ■

Corollary 4.3: Let P be a directed program and g a directed goal. Then g has an infinite data driven derivation if and only if it has an infinite data driven derivation which is oriented.

According to Corollary 4.3, in our context it is sufficient to consider data driven derivations which are oriented. Such derivations are still not always LD-derivations since the selected atom is not necessarily leftmost. If it is not, however, its left neighbors will never be reactivated in future derivation steps; thus w.r.t. termination they can simply be ignored. The same effect can be achieved by a simple program transformation proposed in [FAL88]:

$Pr_G(P) =$ {$p(\bar{\bar{X}}) \leftarrow$ | p is an n-ary predicate appearing in the body or the head of some clause of P and $\bar{\bar{X}}$ is an n-tuple of distinct variables}

$Part_G(P) =$ $P \cup Pr_G(P)$

Simulation Lemma 4.4: Let $D = G_0 \rightarrow\ldots G_{i-1} \rightarrow G_i$ be an oriented SLD-derivation of G_0 and P where

$G_{i-1} = \leftarrow B_1,\ldots,B_j\ldots,B_n$ and

$G_i = \leftarrow (B_1,\ldots,B_{j-1},C_i^+,B_{j+1},\ldots,B_n)\theta_i$.

C_i^+ is the body of the C_j applied to resolve B_j. Then there is an LD-derivation

$D' = G_0 \ldots\rightarrow\ldots G_{k-1}'\rightarrow G_k'$ with $Part_G(P)$, where

$G_{k-1}' = \leftarrow B_j\ldots,B_n$ and

$G_k' = \leftarrow (C_i^+,B_{j+1}\ldots,B_n)\theta_i$.

Proof: Whenever an atom B is selected in D which is not the leftmost one, first the atoms to the left of B are resolved

away in D' with clauses in $Pr_G(P)$, and then D' resolves B in the same way as D. ■

An immediate implication is the following:

Theorem 4.5: If g has a non-terminating data driven oriented derivation with P, then it has a nonterminating LD-derivation with $Part_G(P)$.

The converse, however, is not true. Consider, for instance, the quicksort example from above, extended by the following clauses

q_0: qsort(_,_).
s_0: split(_,_,_,_).
a_0: append(_,_,_).

While the LD-tree for ← qsort([2,1],X) is finite in the context of the standard definition of qsort, it is no longer true for the extended program. Consider the following infinite LD-derivation:

← qsort([2,1],X)
by q_2: ← split([1],2,A,B), qsort(A,A_1), qsort(B,B_1), append(A_1,[H|B_1],S).
by s_0: ← qsort(A,A_1), qsort(B,B_1), append(A_1,[H|B_1],S).
by q_2: ← split(_,_,_,_), ...
by s_0: ← qsort(_,_),...

This derivation, however, is not data driven: resolving qsort(A,A_1) in the third goal with q_2 yields an mgu which is not a matching substitution.

For *data driven* LD-derivations we get a stronger result:

Theorem 4.6: There is a nonterminating data driven oriented derivation for g with P if and only if there is a nonterminating data driven LD-derivation for g with $Part_G(P)$.

Proof: The only-if part is implied by the simulation lemma. For the if-part, consider a nonterminating, data driven LD-derivation D. By removing all applications of clauses in $Pr_G(P)$, one gets another derivation D'. D' is a nonterminating data driven oriented derivation. ■

Restriction to LD-derivations which are data-driven enlarges the class of goal/program pairs which do not loop forever. In the general case, termination of quicksort requires that the first argument is a list. Termination of append requires that the first or the third argument is a list. Restriction to data-driven LD-derivation implies that no queries of quicksort or append (and many other procedures which have finite LD-derivations only for certain modes) loop forever. However, goals like ← append(X,Y,Z) or ← quicksort(A,B) deadlock immediately.

5. Termination Proofs

In this section, we will give a sufficient condition for terminating data driven LD-derivations. We will concentrate on programs without mutual recursion. In [PLU90b] we have demonstrated how mutual recursion can be transformed into direct recursion. We need some further notions.

For a set T of terms, a *norm* is a mapping $|...|: T \to N$. The mapping $||...||: A \to N$ is an input norm on (annotated) atoms, if for all $B = p(t_1,...,t_n)$, $||B|| = \sum_{i \in I} |t_i|$, where I is a subset of the input arguments of B.

Let P be a well-formed program without mutual recursion. P is safe, if there is an input norm on atoms such that for all clauses $c = B_0 \leftarrow B_1,...,B_i,...,B_n$ the following holds: If B_i is a recursive literal (B_0 and B_i have the same predicate symbol), σ a substitution the domain of which is a subset of the input variables of B_0 and θ is a computed answer for $\leftarrow (B_1,...,B_{i-1})\sigma$, then $||B_0\sigma\theta|| > ||B_i\sigma\theta||$.

We can now state the following theorem:

Theorem 5.1: If P is a safe program and $G = \leftarrow A$ is well-formed, then all data driven LD-derivations for G are finite.

PROOF: By contradiction. Assume that there is an infinite data driven LD-derivation D. Then there is an infinite subsequence D' of D containing all elements of D starting with the same predicate symbol p. Let d_i and d_{i+1} be two consecutive elements of D' and

d_i = $\leftarrow p(t_1,...,t_r)$, ...
d_{i+1} = $\leftarrow p(t'_1,...,t'_r)$, ...
and c_i =
$p(s_1,...,s_r) \leftarrow B_1,...,B_k,p(s'_1,...,s'_r),...$

be the clause applied to resolve the first literal of d_i, θ_i the corresponding mgu. Then there is a computed answer substitution θ' for $\leftarrow (B_1,...,B_k)\theta_i$ such that $p(t'_1,...,t'_r) = p(s'_1,...,s'_r)\theta_i\theta'$.

Since D is data driven, θ_i is a matching substitution, i.e. $p(t_1,...,t_r) = p(t_1,...,t_r)\theta_i$. Since P is well-formed, Theorem 3.3 further implies $p(t_1,...,t_r) = p(t_1,...,t_r)\theta_i\theta'$. We also have $p(t_1,...,t_r)\theta_i\theta' = p(s_1,...,s_r)\theta_i\theta'$.

Since P is a safe program
$||p(s_1,...,s_r)\theta_i\theta'|| > ||p(s'_1,...,s'_r)\theta_i\theta'||$ and thus
$||p(t_1,...,t_r)\theta_i\theta'|| > ||p(t'_1,...,t'_r)\theta_i\theta'||$. Since the range of ||...|| is a well-founded set, D' cannot be infinite.
Contradiction. ■

The next question is how termination proofs for data driven LD-derivations can be automated. In [PLU90b] and [PLU91], a technique for automatic termination proofs for Prolog programs is described. It uses an approximation of the program's semantics to reason about its operational behavior. The key concept are predicate inequalities which relate the argument sizes of the atoms in the minimal Herbrand model of the program. Now in any program $Part_G(P)$ for every predicate symbol p occurring in P there is a unit clause $p(\bar{X})$. Thus the minimal Herbrand model M_P of P equals the Herbrand base B_P of P, a semantics which is

not helpful. To overcome this difficulty, we will consider S-models which have been proposed in [FLP89] in order to model the operational behaviour of logic programs more closely. The S-model of a logic program P can be characterized as the least fixpoint of an operator T_s which is defined as follows:

$T_s(I) = \{B \mid \exists\ B_0 \leftarrow B_1,\ldots,B_k$ in P, $\exists\ B_1',\ldots,B_k' \in I$, $\exists\ \delta = mgu((B_1,\ldots,B_k),(B_1',\ldots,B_k'))$, and $B = B_0\delta\}$.

We need some notions defined in [BCF90] and [PLU91].

Let Δ be a mapping from a set of function symbols F to **N** which is not zero everywhere. A norm $|\ldots|$ for T is said to be *semi-linear* if it can be defined by the following scheme:

$|t| = 0$ if t is a variable

$|t| = \Delta(f) + \sum_{i\in I} |t_i|$ if $t = f(t_1, \ldots t_n)$,

where $I \subseteq \{1,\ldots,n\}$ and I depends on f.

A subterm t_i is called selected if $i \in I$.

A term t is *rigid* w.r.t. a norm $|\ldots|$ if $|t| = |t\theta|$ for all substitutions θ. Let $t[v_{(i)} \leftarrow s]$ denote the term derived from t by replacing the i-th occurrence of v by s. An occurrence $v_{(i)}$ of a variable v in a term t is *relevant* w.r.t. $|\ldots|$ if $|t[v_{(i)} \leftarrow s]| \neq |t|$ for some s. Variable occurrences which are not relevant are called irrelevant. A variable is relevant if it has a relevant occurrence. Rvars(t) denotes the multiset of relevant variable occurrences in the term t.

Proposition 5.2: Let t be a term, $t\theta$ be a rigid term and V be the multiset of relevant variable occurrences in t. Then for a semi-linear norm $|\ldots|$ we have $|t\theta| = |t| + \sum_{v\in V} |v\theta|$

Corollary 5.3: $|t\theta| \geq |t|$.

Proof: [PLU91]

For an n-ary predicate p in a program P, a *linear predicate inequality* LI_p has the form $\sum_{i\in I} p_i + c \geq \sum_{j\in J} p_j$, where I and J are disjoint sets of arguments of p, and c, the offset of LI_p, is either a natural number or ∞ or a special symbol like γ. I and J are called input resp. output positions of p (w.r.t. LI_p).

Let M_S be the S-model of P. LI_p is called *valid* (for a linear norm $|\ldots|$) if $p(t_1,\ldots,t_n) \in M_S$ implies $\sum_{i\in I} |t_i| + c \geq \sum_{j\in J} |t_j|$.

Let $A = p(t_1,\ldots,t_n)$. With the notations from above we further define:

- $F(A,LI_p) = \sum_{i\in I} |t_i| - \sum_{j\in J} |t_j| + c$.
- $V_{in}(A,LI_p) = \cup\, rvars(t_i)$
- $V_{out}(A,LI_p) = \cup\, rvars(t_j)$
- $F_{in}(A,LI_p) = \sum_{i\in I} |t_i|$
- $F_{out}(A,LI_p) = \sum_{j\in J} |t_j|$

$F(A,LI_p)$ is called the offset of A w.r.t. LI_p.

Theorem 5.4: Let $\sum_{i\in I} p_i + c \geq \sum_{j\in J} p_j$ be a valid linear predicate inequality, $G = \leftarrow p(t_1,\ldots,t_n)\sigma$ a well-formed goal, V and W the multisets of relevant input resp. output variable occurrences of $p(t_1,\ldots,t_n)$ and θ a computed answer for G. Then the following holds:

i) $\sum_{i\in I} |t_i\sigma\theta| + c \geq \sum_{j\in J} |t_j\sigma\theta|$.

ii) $\sum_{v\in V} |v\sigma\theta| + F(p(t_1,\ldots,t_n),LI_p) \geq \sum_{w\in W} |w\sigma\theta|$.

Proof: According to [FLP89], $p(t_1,\ldots,t_n)\sigma\theta$ is an instance of an atom $p(s_1,\ldots,s_n)$ in the S-model M_S of P. Since the output of G is unrestricted, $t_j\sigma\theta = s_j$ for all $j\in J$. Proposition 5.2 implies $|t_i\sigma\theta| \geq |t_i|$ for all $i\in I$. Thus $\sum_{i\in I} |t_i\sigma\theta| \geq \sum_{i\in I} |s_i|$ and $\sum_{j\in J} |t_j\sigma\theta| = \sum_{j\in J} |s_j|$ which proves the first part of the theorem. The second part is implied by Prop. 5.2. ■

Theorem 5.4 gives a valid inequality relating variables occurring in a single literal goal. Next we give an algorithm for the derivation of a valid inequality relating variables in a compound goal.

Algorithm 5.5 *goal_inequality(G,LI,U,W,Δ,b)*

Input: A well-formed goal $G = \leftarrow B_1,\ldots,B_n$, a set LI with one inequality for each predicate in G, and two multisets U and W of variable occurrences.

Output: A boolean variable b which will be true if a valid inequality relating U and W could be derived, and an integer Δ which is the offset of that inequality.

begin
  M := W; Δ := 0; V := U;
  For i := n **to** 1 **do:**
  If $M \cap V_{out}(B_i,LI_p) \neq \emptyset$ **then**
    $M := (M \setminus V_{out}(B_i,LI_p)) \cup (V_{in}(B_i,LI_p) \setminus V)$;
    $V := V \setminus V_{in}(B_i,LI_p)$;
    $\Delta := \Delta + F(B_i,LI_p)$. **fi**
  If $M = \emptyset$ **then** b:= true **else** b:= false **fi**
end.

Next we show that the algorithm is correct:

Theorem 5.6: Assume that the inequalities in LI are valid and b is true, σ is an arbitrary substitution such that $G\sigma$ is well-formed and θ is a computed answer substitution for $G\theta$. Then $\sum_{v\in V} |v\sigma\theta| + \Delta \geq \sum_{w\in W} |w\sigma\theta|$ holds.

Proof: See [PLU92].

Algorithm 5.5 takes time O(m) where m is the length of G.

[PLU90b] gives an algorithm for the automatic derivation of inequalities for compound goals based on and/or-dataflow graphs which has exponential runtime in the worst case. Algorithm 5.5 makes substantial use of the fact that G is well-formed: each variable has at most one generator, which makes the derivation of inequalities deterministic.

6. Derivation of inequalities for S-models

In Aection 5 it has been assumed that linear inequalities are given for the predicates of a program P. We now show how these inequalities can be derived automatically. We assume that P is well-formed and free of mutual recursion. Let $p <_\pi q$ if $p \neq q$ and p occurs in one of the clauses defining q. Absence of mutual recursion in P implies that $<_\pi$ defines a partial order which can be embedded into a linear order. Thus there is an enumeration $\{p_1,\ldots,p_n\}$ of the predicates of p such that $p_i <_\pi p_j$ implies $i < j$. We will process the predicates of P in that order, thus in analyzing p we can assume that for all predicates on which the definition of p depends valid inequalities have already been derived. Note that a trivial inequality with offset ∞ always holds.

Let in(A) and out(A) denote the sets of input resp. output variables of an atom or a set of atoms according to the annotation of the given programs.

Algorithm 6.1: predicate_inequalities(P,LI):

Input: A well-formed program P defining $p_1,\ldots,p_n$.

Output: A set LI of valid inequalities for the predicates of P.

begin
 LI := $\emptyset$
 For i:= 1 to n do:
  begin
   Let $c_1,\ldots,c_m$ be the clauses defining p_i.
   Let M, N be the input resp. output arguments of p_i.
   li := $\sum_{\mu\in M}|p_\mu| + \gamma \geq \sum_{\nu\in N}|p_\nu|$.
   b_i := true.
   For j:= 1 to m do:
   begin
    Let c_j be $B_0 \leftarrow B_1,\ldots,B_k$.
    goal_inequality($(\leftarrow B_1,\ldots,B_k)$,
     LI$\cup${li},$V_{in}(B_0)$,$V_{out}(B_0)$, Δ_i,b_i)
    c:= $\Delta_i + F_{out}(B_0,li) - F_{in}(B_0,li)$.
    $\Phi_i := b_i$
    If c contains '∞' **then** $\Phi_i := \Phi_i \wedge$ false
(*) **elseif** c is an integer **then** $\Phi_i := \Phi_i \wedge (\gamma \geq c)$
(**) **elseif** $c = \gamma + d \wedge d \leq 0$ **then** $\Phi_i := \Phi_i \wedge$ true
    elseif $c = \gamma + d \wedge d > 0$ **then** $\Phi_i := \Phi_i \wedge$ false
(***) **elseif** $c = k * \gamma + n \wedge k > 1$,
    then $\Phi_i := \Phi_i \wedge (\gamma \leq n/(1-k))$.
   end
   If Φ_i is satisfiable **then** let δ_i be the smallest value for
     γ which satisfies Φ_i
   else let δ_i be '∞'.
   Replace γ in li by δ_i.
   LI := LI $\cup$ {li}
  end
end

Theorem 6.2: The inequalities derived by the algorithm are valid.

Proof: By induction on the number of predicates n in P. The case n = 0 is immediate. For the inductive case, assume that the derived inequalities for the predicates $p_1,\ldots,p_{n-1}$ are valid. Let I_0 be the minimal S-model of P restricted to the predicates $p_1,\ldots,p_{n-1}$. In the context of the program which consists of the definition of p_n only, let $T_s^0 = I_0$ and $T_s^m = T_s(T_s^{m-1})$. Its limes equals the minimal S-model of P restricted to the predicates $p_1,\ldots,p_n$. Now we have to show that the inequality li derived for p_n is valid w.r.t. T_s^m. The proof is now by induction on m. The case m = 0 is implied by the induction assumption on n. Assume that the theorem holds for n - 1. We have to show that the inequality for p_n holds for the elements of T_s^m. Now lett $B \in T_s^m$ and $B_0 \leftarrow B_1,\ldots,B_k$ be the clause applied to derive B. We have $B = B_0\theta$, where θ is a computed answer substitution for $\leftarrow B_1,\ldots,B_k$, which is a well-formed goal. Let $V = in(B_0)$ and $W = out(B_0)$. Let LI be the set of inequalities derived by Algorithm 6.1, and Δ be the result of calling goal_inequality($(\leftarrow B_1,\ldots,B_k)$,LI,V,W, Δ, b_i). Theorem 5.6 and the induction assumption imply

(‡) $\sum_{v\in V}|v\theta| + \Delta \geq \sum_{w\in W}|w\theta|$

Since $B = B_0\theta$, we have $F_{in}(B,li) = F_{in}(B_0,li) + \sum_{v\in V}|v\theta|$ and $F_{out}(B,li) = F_{out}(B_0,li) + \sum_{w\in W}|w\theta|$. Let α be the offset of li. We have to show

(‡‡) $F_{in}(B,li) + \alpha \geq F_{out}(B,li)$.

If b_i is false or Δ is ∞, we are done since in that case α is ∞. Three more cases remain. (*) and (**) immediately imply

(‡‡‡) $\alpha \geq \Delta + F_{out}(B_0,li) - F_{in}(B_0,li)$.

(***) implies $\alpha \leq n/(1-k)$ and thus $\alpha \geq n + k*\alpha$ for some n such that $n + k*\alpha = \Delta + F_{out}(B_0,li) - F_{in}(B_0,li)$. Again (‡‡‡) follows. (‡) and (‡‡‡) together now imply (‡‡). ■

Note that Algorithm 6.1 again has run-time complexity O(n), where n is the length of the given program P.

Algorithm 6.1 is not yet able to derive $p_1 \geq p_2$ for a unit clause like p(X,Y) with mode(p(+,-)). This inequality, however, holds since in a well-formed goal the output argument of p will always be unbound. To overcome this difficulty, we assume that before calling predicate_inequalities(P,LI), P will be transformed to P' in the following way:

Define freevars($B_0 \leftarrow B_1,\ldots,B_n$) = $(out(B_0) \setminus out(B_1,\ldots,B_n)) \cup in(B_1,\ldots,B_n) \setminus in(B_0))$.
Now for the clause $c = B_0 \leftarrow B_1,\ldots,B_n$ in P let freevars(c) = $\{Y_1,\ldots,Y_m\}$. Replace c by $B_0 \leftarrow q(Y_1,\ldots,Y_m),B_1,\ldots,B_n$ where a new predicate q is defined by the unit clause $q(X_1,\ldots,X_m)$ with mode(q(+,...,+)). Note that, after that transformation, P' is well-formed if P is well-formed, and if an inequality is valid for P' it is valid for P as well. In the example mentioned above, input for Algorithm 6.1 will be the program P = {q(X). , p(X,Y) $\leftarrow$ q(Y)} and the output will be $\{0 \geq q_1, p_1 \geq p_2\}$.

Another improvement can be made by considering subsets of the input arguments in order to achieve stronger inequalities. This, however, makes the algorithm less efficient.

7. Example

We finally discuss how, with the techniques given so far, it can be shown that the GHC program for quicksort specified in Section 3 terminates for arbitrary goals.

Corollary 4.3 and Theorem 4.5 imply that is suffices to consider data-driven LD-derivations of the extended program for qsort including the clauses s_0, a_0 and q_0. According to Theorem 5.1 we only have to show that the three predicates of the program are safe. This is easy to show for split and append. In fact these procedures are structural recursive. It is more difficult to prove of qsort because in q_2 both recursive calls contain the local variables A and B. For this reason we need a linear predicate inequality for split which has the form $split_1 + \gamma \geq split_3 + split_4$. After the transforamtion mentioned at the end of the last paragraph s_0 will have the following form:

s_0: $split(L_1,L_2,L_3,L_4) \quad \leftarrow q(L_3, L_4)$

Now s_0 and s_1 give $\gamma \geq 0$ (case * in Algorithm 6.1), while s_2 and s_3 give 'true' (case **). Thus we get $split_1 + 0 \geq split_3 + split_4$. In order to prove safety of qsort, we only have to consider q_2. Using this inequality Algorithm 5.5 immediately shows $||qsort([H|L],S)\theta|| > ||qsort(A,A_1)\theta||$ and $||qsort([H|L],S)\theta| > ||qsort(B,B_1)\theta||$ for all answer substitutions θ for split(H,L,A,B). Thus qsort is safe.

Acknowledgment

Part of this work was performed while I was visiting CWI. K. R. Apt stimulated my interest in concurrent logic programming.

References

[APP90] Apt, K. R., Pedreschi, D., Studies in pure Prolog: Termination, Technical Report CS-R9048, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[APT90] Apt, K. R., Introduction to logic programming, in Leeuwen (ed.), *The Handbook of Theoretical Computer Science*, North Holland 1990.

[BCF90] Bossi, A., Cocco, N., Fabris, M., Proving Termination of Logic Programs by Exploiting Term Properties, Technical Report Dip. di Matematica Pura e Applicata, Universita di Padova, 1990.

[FAL88] Falaschi, M., Levi, G., Finite failures and partial computations in concurrent logic languages, *Proc. of the Int. Conf. of Fifth Gen. Comp. Systems*, ICOT 1988.

[FLP89] Falaschi, M., Levi, G., Palamidessi, C., Martelli, M., Declarative Modeling of the Operational Behavior of Logic Languages, *Theoretical Computer Science 69*, 1989.

[GRE87] Gregory, S., Parallel Logic Programming in PARLOG, Addison Wesley, 1987.

[HAP85] Harel, D., Pnueli, A., On the development of reactive systems, in Apt, K. R. (ed.) *Logics and Models of Concurrent Systems*, Springer 1985.

[LLO87] Lloyd, J., Foundations of Logic Programming, Springer Verlag, Berlin, second edition, 1987.

[PLU90a] Plümer, L., Termination proofs for logic programs based on predicate inequalities, in Warren, D.H.D., Szeredi, P. (eds.), *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press 1990.

[PLU90b] Plümer, L., Termination Proofs for Logic Programs, Springer Lecture Notes in Artificial Intelligence 446, Berlin 1990.

[PLU91] Plümer, L., Termination proofs for Prolog programs operating on nonground terms, *1991 International Logic Programming Symposium, San Diego, California*, 1991.

[PLU92] Plümer, L., Automatic Verification of GHC-Programs: Termination, Technical Report, Universität Bonn, 1992.

[SHA87] Shapiro, E., Concurrent Prolog, Collected Papers, MIT Press 1987.

[SHA87a] Shapiro, E., Systolic Programming: A paradigm of parallel processing, in [SHA87].

[TIC91] Tick, E., Parallel Logic Programming, MIT Press 1991.

[UED88] Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, in Nivat, M., Fuchi, K. (eds.), *Programming of Future Generation Computers*, North-Holland 1988.

[UED86] Ueda, K., Guarded Horn Clauses, in [SHA87].

[ULG88] Ullman, J. D., van Gelder, A., Efficient Tests for Top-Down Termination of Logical Rules, *Journal of the ACM* 35, 2, 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Analogical Generalization

Takenao OHKAWA† Toshiaki MORI‡ Noboru BABAGUCHI‡ Yoshikazu TEZUKA‡

†*Education Center for Information Processing, Osaka University*
‡*Dept. of Communication Eng., Faculty of Eng., Osaka University*
2-1, Yamadaoka, Suita, Osaka, 565 Japan
e-mail: ohkawa@oucom5.oucom.osaka-u.ac.jp

Abstract

Approaches to learning by examples have focused on generating general knowledge from a lot of examples. In this paper we describe a new learning method, called analogical generalization, which is capable of generating a new rule which specifies a given target concept from a single example and existing rules. Firstly we formulate analogical generalization based on the similarity between a given example and existing rules from the logical viewpoint. Secondly, we give a new procedure of inductive learning with analogical generalization, called ANGEL. The procedure consists of the following five steps: (1) extending a given example, (2) extracting atoms from the example and selecting a base rule out of the set of existing rules, (3) generalizing the extracted atoms by means of the selected rule as a guide. (4) replacing predicates, and (5) generating a rule. Through the experiment for the system for parsing English sentences, we have clarified that ANGEL is useful for acquiring rules on knowledge based systems.

1 Introduction

Machine learning has a great contribution to improving performance through automated knowledge acquisition and refinement, and so far, various types of machine learning paradigms have been considered. In particular, learning from examples, which can form general knowledge from specific cases given as input examples, has been well studied and a lot of concerned methods have been proposed[Mitchell 1977, Dietterich and Michalski 1983, Ohkawa *et al.* 1991].

Generally, in learning from examples, we have to give a lot of examples to the learner. Why are so many examples required? We think the reason for this is that the bias for restricting the generalization is relatively weak, because it is independent of the domain. However, when a human being acquires new knowledge, he would not always require a lot of examples. As the case may be, he can learn from one example. We think this is because he decides a strong bias for the generalization according to the domain, and generalizes the examples based on the bias. That is, in order to generalize a few examples appropriately, a strong bias which depends on the domain is indispensable.

It is necessary to consider how the strong bias should be provided. Let us recall the behavior of a human being again. When acquiring new knowledge, he often utilizes similar knowledge which is already known. In other words, the existence of similar knowledge may help for him to associate new knowledge. This process is called analogy. Analogy is considered promising to realize learning from a few examples. Since analogy will be regarded as one of the most effective way for restriction on generalization, modeling its process will make it possible to provide a domain dependent bias.

In this paper, we propose a new learning method, called ANGEL (ANalogical GEneraLization), which is capable of generating a new rule from a single example. In ANGEL, both the rules and the examples are represented as logical formulas. We introduce the notion of analogy[Winston 1980], namely, the similarity between the example and the existing rules as the bias for the generalization[Mori *et al.* 1991]. The similarity is determined by comparing the atoms of both the example and the existing rules. Based on the similarity, firstly, ANGEL extracts atoms from the example and selects a rule out of the existing rules; next, it generates a new rule by generalizing the extracted atoms by means of the selected rule as a guide.

The next section describes the definition of analogical generalization. In this section we consider analogical generalization from the logical viewpoint. Section 3 gives the procedure of ANGEL which is a method for learning based on analogical generalization. In this section, we also give consideration to the experimental result of learning by ANGEL. Finally in section 4, we clarify the originality of ANGEL through its comparison to other related works.

2 Analogical generalization

To represent knowledge, we use the form which conforms to first order predicate logic. Two kinds of forms, called a fact and a rule, are provided. A fact is represented as an atom, while a rule is represented as a Horn clause, which is expressed in the form of

$$\alpha \leftarrow \beta_1, \ldots, \beta_n,$$

where $\alpha, \beta_1, \ldots, \beta_n$ are atoms. Letting r be a rule $\alpha \leftarrow \beta_1, \ldots, \beta_n$, we denote the consequence of rule r, namely α, by $cons(r)$, and denote the premise of rule r, namely $\beta_1, \ldots, \beta_n$, by $prem(r)$.

The underlying notion of analogical generalization is that a new rule is generated by generalizing an input example, which consists of facts, based on the similarity between the example and the existing rules. Before formulating analogical generalization, we define the similarity between two atoms,

and next formalize the similarity between two finite sets of atoms.

2.1 Similarity between two atoms

First, we define some basic notations. A substitution is a finite set of the pair v/t, where v is a variable, t is a term, and the variables are distinct. Let $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ be a substitution and e be an expression, which is either a literal or a conjunction or disjunction of literals. Then $e\theta$ is the expression obtained from e by replacing each occurrence of the variable v_i in e by the term t_i. If S is a finite set of expressions and θ is a substitution, $S\theta$ denotes the set $\{e\theta \mid e \in S\}$.

Let θ be a substitution and S be a finite set of atoms. If $S\theta$ is a singleton, S is unifiable by θ and we write *unifiable*(S).

Now, we give the following two functions, and define the similarity between atoms by means of these functions. Let R be a set of existing rules, and α and α' be atoms.

Definition 1 (R-deducible set)

$$\Phi(R,\alpha) \stackrel{\text{def}}{=} \{\beta \mid R \cup \{\alpha\} \vdash \beta, \beta \textit{ is an atom}\}.$$

Definition 2 (R-similar set)

$$\Psi(R,\alpha,\alpha') \stackrel{\text{def}}{=} \{\beta \mid \beta \in \Phi(R,\alpha), {}^{\exists}\beta' \in \Phi(R,\alpha'), \textit{unifiable}(\{\beta,\beta'\})\}.$$

R-deducible set means all of newly obtained information when a certain fact has been known. Thus the intuitive meaning of R-similar set is newly obtained information in common when each of two distinct facts has been known. Therefore we can say that R-similar set represents the relevance between two facts under the background knowledge.

Definition 3 (Similarity between atoms) *Let α, α_1 and α_2 be atoms. If the following relation holds, α* **is more similar to α_2 than α_1 with respect to R.**

$$\Psi(R,\alpha,\alpha_1) \subset \Psi(R,\alpha,\alpha_2)$$

And if the following holds, **the similarity between α and α_1 is equal to the similarity between α and α_2 with respect to R.**

$$\Psi(R,\alpha,\alpha_1) = \Psi(R,\alpha,\alpha_2)$$

Since R-similar set reflects the relevance between two given facts, the similarity between a certain fact and two distinct facts can be evaluated in terms of the subsumption relation between R-similar sets reasonably .

For example, let R_1 be a set of rules shown as follows.

$$\begin{aligned} R_1 = \{&\texttt{parent}(x,y) \leftarrow \texttt{father}(x,y),\\ &\texttt{parent}(x,y) \leftarrow \texttt{mother}(x,y),\\ &\texttt{family}(x,y) \leftarrow \texttt{parent}(x,y),\\ &\texttt{family}(x,y) \leftarrow \texttt{brother}(x,y),\\ &\texttt{hates}(x,y) \leftarrow \texttt{kills}(x,y),\\ &\texttt{hates}(x,y) \leftarrow \texttt{hurts}(x,y),\\ &\texttt{hates}(x,y) \leftarrow \texttt{strikes}(x,y)\} \end{aligned}$$

Let us consider the similarity of $\texttt{father}(x,y)$ to $\texttt{mother}(\texttt{Jim},\texttt{Betty})$ and $\texttt{brother}(\texttt{Tom},\texttt{Joe})$. For each atom, the following R-deducible sets are derived as

$$\begin{aligned} &\Phi(R_1,\texttt{father}(x,y)) = \{\texttt{father}(x,y),\texttt{parent}(x,y),\texttt{family}(x,y)\}\\ &\Phi(R_1,\texttt{mother}(\texttt{Jim},\texttt{Betty}))\\ &\qquad = \{\texttt{mother}(\texttt{Jim},\texttt{Betty}),\texttt{parent}(\texttt{Jim},\texttt{Betty}),\\ &\qquad\quad \texttt{family}(\texttt{Jim},\texttt{Betty})\}\\ &\Phi(R_1,\texttt{brother}(\texttt{Tom},\texttt{Joe}))\\ &\qquad = \{\texttt{brother}(\texttt{Tom},\texttt{Joe}),\texttt{family}(\texttt{Tom},\texttt{Joe})\}. \end{aligned}$$

R-similar sets of $\texttt{father}(x,y)$ for $\texttt{mother}(\texttt{Jim},\texttt{Betty})$ and $\texttt{brother}(\texttt{Tom},\texttt{Joe})$ are as follows.

$$\begin{aligned} &\Psi(R_1,\texttt{father}(x,y),\texttt{mother}(\texttt{Jim},\texttt{Betty}))\\ &\qquad = \{\texttt{parent}(x,y),\texttt{family}(x,y)\}\\ &\Psi(R_1,\texttt{father}(x,y),\texttt{brother}(\texttt{Tom},\texttt{Joe})) = \{\texttt{family}(x,y)\} \end{aligned}$$

Accordingly $\texttt{father}(x,y)$ is more similar to $\texttt{mother}(\texttt{Jim},\texttt{Betty})$ than $\texttt{brother}(\texttt{Tom},\texttt{Joe})$ with respect to R_1. This result matches our intuition very well.

2.2 Similarity between two finite sets of atoms

The similarity between two finite sets of atoms is determined by the similarity between elements of each set. In this case, we also have to consider the matching between atoms in each set. We begin with the definition of correspondence between two sets of atoms.

Definition 4 (Correspondence) *Let A and B be finite sets of atoms.* **Correspondence φ of A to B** *is defined as follows,*

1. *φ is a relation on A and B.*
2. *There is a substitution θ and for all $(\alpha,\beta) \in \varphi\theta$,*

$$\begin{aligned} &\textit{arity}(\alpha) = \textit{arity}(\beta),\\ &\textit{arg}(\alpha,n) = \textit{arg}(\beta,n) \quad (n = 1,2,\ldots), \end{aligned}$$

   where $\textit{arity}(\alpha)$ indicates the number of arguments of α, and $\textit{arg}(\alpha,n)$ indicates the value of n-th argument of α.

3. *For all $\alpha \in A$, there is an atom β such that $(\alpha,\beta) \in \varphi$. And for all $\beta \in B$, there is an atom α such that $(\alpha,\beta) \in \varphi$.*

For example, let A_1 and B_1 be sets of atoms shown as follows.

$$\begin{aligned} A_1 &= \{\texttt{father}(x,y),\texttt{kills}(y,z)\}\\ B_1 &= \{\texttt{mother}(\texttt{Jim},\texttt{Betty}),\texttt{hurts}(\texttt{Betty},\texttt{Jim})\} \end{aligned}$$

In this case, two correspondences φ_1, φ_2 of A_1 to B_1 are obtained.

$$\begin{aligned} \varphi_1 = \{&(\texttt{father}(x,y),\texttt{mother}(\texttt{Jim},\texttt{Betty})),\\ &(\texttt{kills}(y,z),\texttt{hurts}(\texttt{Betty},\texttt{Jim}))\}\\ \varphi_2 = \{&(\texttt{father}(x,y),\texttt{hurts}(\texttt{Betty},\texttt{Jim})),\\ &(\texttt{kills}(y,z),\texttt{mother}(\texttt{Jim},\texttt{Betty}))\} \end{aligned}$$

Definition 5 (Precedence of correspondence) *Let A and B be sets of atoms, φ_1 and φ_2 be two distinct correspondences of A to B. Then*

- *For all α in A, α is similar to β_1 such that $(\alpha, \beta_1) \in \varphi_1$ than β_2 such that $(\alpha, \beta_2) \in \varphi_2$, or the similarity between α and β_1 is equal to the similarity between α and β_2 with respect to R, and*

- *There exists α in A, which is similar to β_1 such that $(\alpha, \beta_1) \in \varphi_1$ than β_2 such that $(\alpha, \beta_2) \in \varphi_2$, with respect to R,*

if and only if we say that **correspondence φ_1 precedes φ_2 with respect to R.** *For a correspondence φ of A to B, if there is no correspondence that precedes φ, we call φ a* **maximally preceding correspondence of A to B with respect to R.**

Maximally preceding correspondence represents the matching between the most similar atoms in two sets of atoms with binding variables consistently.

In the above example, φ_1 precedes another correspondence, namely, φ_2, with respect to R_1, because `father`(x, y) is more similar to `mother(Jim,Betty)` than `hurts(Betty,Jim)` and likewise `kills`(y, z) is more similar to `hurts(Betty,Jim)` than `mother(Jim,Betty)`. Therefore φ_1 is a maximally preceding correspondence of A_1 to B_1 with respect to R_1.

Definition 6 (Similarity between sets of atoms) *Let A, A', B and C be sets of atoms, φ_B be a maximally preceding correspondence of A to B with respect to R and φ_C be a maximally preceding correspondence of A' to C with respect to R. Then*

- *For all α in $A \cap A'$, α is similar to β_B such that $(\alpha, \beta_B) \in \varphi_B$ than β_C such that $(\alpha, \beta_C) \in \varphi_C$, or the similarity between α and β_B is equal to the similarity between α and β_C with respect to R, and*

- *There exists α in $A \cap A'$, which is similar to β_B such that $(\alpha, \beta_B) \in \varphi_B$ than β_C such that $(\alpha, \beta_C) \in \varphi_C$, with respect to R,*

if and only if we say that the similarity between A and B **is stronger than** *the similarity between A' and C* **with respect to** *R, denoted by*

$$[A : B] \overset{R}{\succ} [A' : C].$$

Now, we assume C_1 is the following set of atoms.

$$C_1 = \{\texttt{brother(Tom,Joe)}, \texttt{strikes(Joe,Mark)}\}$$

A maximally preceding correspondence of A_1 to C_1 with respect to R_1 is shown as

$$\{(\texttt{father}(x, y), \texttt{brother(Tom, Joe)}), \\ (\texttt{kills}(y, z), \texttt{strikes(Joe, Mark)})\},$$

and therefore,

$$[A_1 : B_1] \overset{R}{\succ} [A_1 : C_1].$$

2.3 Formulation of analogical generalization

In this section, we proceed to formulate analogical generalization. First we give a logical consideration on analogical generalization under five conditions to generate a rule, discussing these conditions briefly.

Let τ be a non-ground atom which represents a target concept, and E be an example, that is, a set of ground atoms which is relevant to the target concept. In this case a non-ground atom is an atom containing variables and a ground atom is an atom containing no variable. We assume that E contains τ', called target instance, such that *unifiable*($\{\tau, \tau'\}$). Let E' be a set given by removing target instance τ' from E, and E'' be a set of ground atoms deduced by $R \cup E$. Analogical generalization is formulated as follows.

Definition 7 (Analogical generalization) *Given R, E, τ, and if*

$$R \cup E' \not\vdash \tau', \tag{1}$$

then generating a rule r such that

$$R \cup E' \cup \{r\} \vdash \tau', \tag{2}$$

$$R \cup E' \cup \{r\} \text{ is consistent, and} \tag{3}$$

$$r \text{ satisfies the following five conditions,} \tag{4}$$

is called **analogical generalization.**

- *Selection condition*
  There is a substitution θ such that

  $$\Pi(r)\theta \subseteq E'',$$

  $$cons(r)\theta = \tau',$$

  where $\Pi(r)$ denotes a set of all atoms that constitute r.

- *Similarity condition*
  There is a rule $r' (\in R)$, provided that

  1. *There is a correspondence of $\Pi(r')$ to $\Pi(r)\theta$, which contains $(cons(r'), \tau')$* [1].
  2. *For an arbitrary set of atoms $A (\subseteq E'')$, the following relation does not hold.*

     $$[\Pi(r') : A] \overset{R}{\succ} [\Pi(r') : \Pi(r)\theta].$$

  3. *For an arbitrary rule $r'' (\in R)$ and an arbitrary set of atoms $A (\subseteq E'')$, the following relation does not hold.*

     $$[A : \Pi(r'')] \overset{R}{\succ} [\Pi(r)\theta : \Pi(r')].$$

- *Significance condition*
  For a rule r' which satisfies similarity condition [2], *letting φ be a correspondence of $\Pi(r')$ to $\Pi(r)\theta$,*

  $$\bigcup_{(\alpha,\beta)\in\varphi} \Psi(R, \alpha, \beta) \neq \emptyset.$$

[1] θ indicates the same substitution in selection condition.
[2] We call r' a base rule.

- *Generality condition*
  For a base rule r', letting φ be a correspondence of $\Pi(r')$ to $\Pi(r)$,

$$^{\forall}(\alpha,\beta) \in \varphi,\ \arg(\alpha,n) = \arg(\beta,n)\ (n = 1,2,\ldots).$$

- *Applicability condition*
  For a base rule r', let φ_1 be a correspondence of $\Pi(r')$ to $\Pi(r)\theta$. Let φ_2 be a correspondence of $\Pi(r')$ to $A(\subseteq E'')$ which contains τ', provided that φ_2 contains $(cons(r'),\tau')$. For all $\alpha \in \Pi(r')$, if $R \cup \{\alpha\} \not\vdash \beta_2$ or $\{\alpha\} \vdash \beta_2$ such that $(\alpha,\beta_2) \in \varphi_2$, $R \cup \{\beta_1\} \not\vdash \beta_2$ or $\beta_1 = \beta_2$ such that $(\alpha,\beta_1) \in \varphi_1$ has to holds.

Since there are, in general, many rules satisfying the equation (2) and (3), we have introduced the five conditions as constraints for the rule r.

Selection condition means that the rule r is generated making use of predicates which are used for representing given examples and existing rules.

Similarity condition is a condition for the purpose of generating a rule which is similar to an existing rule. A base rule, which is the most similar rule to a given example in existing rules, is selected appropriately due to this condition. Moreover, it guarantees that, with respect to the similarity, relevant atoms are extracted from the example for the selected base rule. That is, this condition is regarded as a bias depending on the domain specific knowledge.

Similarity condition is a condition for checking the validity of a base rule based on a relative comparison of the similarities between a base rule and an example, while significance condition investigates absolutely the relevance between a base rule and an example by means of R-similar set. Rules not satisfying significance condition should be regarded as absurd rules.

Generality condition removes constants which occur in an example from the generated rule. It aims at the versatility of the generated rule.

If an atom α forms a rule r and $R \cup \{\alpha\}$ is able to deduce another atom α', a rule formed by an atom α' instead of α also satisfies the equation (2) and (3). In this case, the latter rule is more applicable than the former. Applicability condition guarantees the most applicable rule can be adopted.

3 ANGEL

3.1 Procedure

This section presents ANGEL in detail. If the set of existing rules R, an example E and target concept τ are given, ANGEL generate a new rule by means of analogical generalization. We show the overview of ANGEL in Figure 1.

If R consists of recursive rules, R-deducible set will be infinite. Then, we assume R has no recursive rule for computing the similarity between atoms practically.

The procedure of ANGEL consists of five steps: (1) extending an example, (2) extracting atoms from the example and selecting a base rule out of the set of existing rules, (3) generalizing the extracted atoms, (4) replacing predicates,

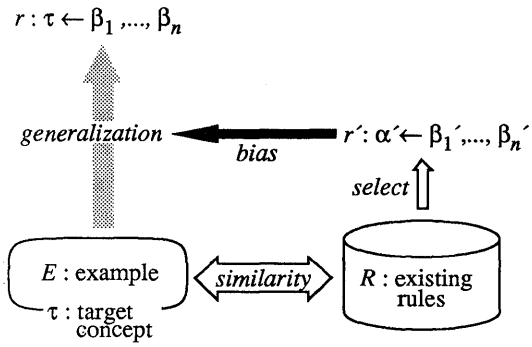


Figure 1: Overview of ANGEL

and (5) generating a rule. We show briefly each step as below.

STEP1 Extending an example
Generate a set of ground atoms which are deduced by $R \cup E$ and denote it by $\tilde{E}$. If an atom $\alpha(\in \tilde{E})$ can be deduced by $R \cup \{\alpha'\}$ $(\alpha' \neq \alpha, \alpha' \in \tilde{E})$, remove the atom α from $\tilde{E}$.

STEP2 Extracting atoms and selecting a base rule
For each rule $r' \in R$, make correspondences of $\Pi(r')$ to A which is an arbitrary subset of $\tilde{E}$. At this time, $cons(r')$ will certainly correspond to the target instance. If a set $A'(\neq A)$ such that,

$$[\Pi(r') : A'] \overset{R}{\succ} [\Pi(r') : A],$$
$$A' \subseteq \tilde{E}$$

does not exist, regard the correspondence of $\Pi(r')$ to A as a candidate of useful correspondence; otherwise abandon the set A. Note that once abandoned sets for a certain rule are never adopted for other rules.

For all candidates of useful correspondences, evaluate the similarities between subsets of an example and rules. And if a correspondence of A' to $\Pi(r'')$ such that,

$$[A' : \Pi(r'')] \overset{R}{\succ} [A : \Pi(r')],$$
$$A' \subseteq \tilde{E},$$
$$r'' \in R$$

does not exist, adopt the correspondence of A to $\Pi(r')$ as a useful correspondence.

STEP3 Generalizing atoms
Generalization is performed by turning constants to variables. As a result of STEP2, there is at least one useful correspondence φ of $\Pi(r')$, in which r' is selected out of R, to A, which is a subset of $\tilde{E}$. Now, turn constants in atoms in the set A to variables which occur at the same position of $\Pi(r')$ according to the correspondence φ.

STEP4 Replacing predicates

For each pair of atom (α, β) in φ which is a useful correspondence of $\Pi(r')$ to A, if $\Phi(R, \beta)$ contains an atom which consists of the same predicates as α, replace the predicate of β with the predicate of α. Otherwise, let S be a set of atoms in $\Phi(R, \beta)$ provided that none of whose predicates occurs in $\Phi(R, \alpha)$. Replace the predicate of β with the predicate of $\gamma(\in S)$ such that

$$^{\forall}\gamma' \in S, \quad \Phi(R, \gamma') \supseteq \Phi(R, \gamma).$$

STEP5 Generating a rule

Finally, generate a new rule r in which $cons(r)$ consists of the atom which is generalization of the target instance and $prem(r)$ consists of the atoms which are generalizations of the atoms in the set A except the target instance.

3.2 Examples and discussions

In this section, we present the two examples of learning by ANGEL. And we clarify the effectiveness of ANGEL by considering the experimental results.

First, we show a simple example in order to follow the behavior of ANGEL. A set R_2 which consists of seven existing rules defines relations of family. E_1 is an example for the target concept "**grandmother**(s,t)".

$$\begin{aligned} R_2 = \{\ & \texttt{grandfather}(x,z) \leftarrow \texttt{parent}(x,y), \texttt{father}(y,z), && \cdots(r1)\\ & \texttt{uncle}(x,z) \leftarrow \texttt{parent}(x,y), \texttt{brother}(y,z), && \cdots(r2)\\ & \texttt{cousin}(x,y)\\ & \quad \leftarrow \texttt{parent}(x,v), \texttt{parent}(y,w), \texttt{brother}(v,w), && \cdots(r3)\\ & \texttt{parent}(x,y) \leftarrow \texttt{mother}(x,y), && \cdots(r4)\\ & \texttt{parent}(x,y) \leftarrow \texttt{father}(x,y), && \cdots(r5)\\ & \texttt{family}(x,y) \leftarrow \texttt{parent}(x,y), && \cdots(r6)\\ & \texttt{family}(x,y) \leftarrow \texttt{brother}(x,y)\} && \cdots(r7) \end{aligned}$$

$$\begin{aligned} E_1 = \{ & \texttt{grandmother}(\texttt{Peter}, \texttt{Mary}),\\ & \texttt{mother}(\texttt{Paul}, \texttt{Mary}),\\ & \texttt{father}(\texttt{Peter}, \texttt{Paul}),\\ & \texttt{mother}(\texttt{Peter}, \texttt{Lucy}),\\ & \texttt{likes}(\texttt{Paul}, \texttt{Mary}),\\ & \texttt{engineer}(\texttt{Peter}),\\ & \texttt{student}(\texttt{Paul})\} \end{aligned}$$

If E_1 is given, ANGEL starts to extend the example. In this case, since no atom has been deduced, the extension of E_1 is E_1 itself.

In STEP2, candidates of useful subsets of E_1 are found for the rule $r1$ as follows.

$$\begin{aligned} & \{\texttt{grandmother}(\texttt{Peter}, \texttt{Mary}),\\ & \ \texttt{father}(\texttt{Peter}, \texttt{Paul}),\\ & \ \texttt{mother}(\texttt{Paul}, \texttt{Mary})\} && \cdots(s1)\\ & \{\texttt{grandmother}(\texttt{Peter}, \texttt{Mary}),\\ & \ \texttt{father}(\texttt{Peter}, \texttt{Paul}),\\ & \ \texttt{likes}(\texttt{Paul}, \texttt{Mary})\} && \cdots(s2) \end{aligned}$$

In these sets, since the relation

$$[\Pi(r1) : s1] \overset{R_2}{\succ} [\Pi(r1) : s2]$$

holds, the set $s2$ is abandoned. As a result, only $s1$ are adopted as the useful set of atoms. Likewise, $s1$ is adopted for the rule $r2$. And no set of atoms is adopted for other rules $r3 \sim r7$.

Next, the similarity between $\Pi(r1)$ and $\Pi(r2)$ is evaluated. As a result, the rule $r1$ is adopted as a useful rule, because the relation

$$[s1 : \Pi(r1)] \overset{R_2}{\succ} [\ s1 : \Pi(r2)]$$

holds.

In STEP3, the generalization will be accomplished. Now, there have been the following correspondences of $\Pi(r1)$ to $s1$.

$$\begin{aligned} \{ & (\texttt{grandfather}(x,z), \texttt{grandmother}(\texttt{Peter}, \texttt{Mary})),\\ & (\texttt{parent}(x,y), \texttt{father}(\texttt{Peter}, \texttt{Paul})),\\ & (\texttt{father}(y,z), \texttt{mother}(\texttt{Paul}, \texttt{Mary}))\} \end{aligned}$$

Therefore, the set of generalized atoms are obtained as follows.

$$\{\texttt{grandmother}(x,z), \texttt{father}(x,y), \texttt{mother}(y,z)\} \quad \cdots(s1')$$

Next, in STEP4, predicates in $s1'$ are replaced with more applicable one. In this case, predicate **father** in $s1'$ is replaced with predicate **parent**, because predicate **parent** occurs in $\Phi(R_2, \texttt{father}(x,y))$. While predicate **mother** in $s1'$ is not replaced, because predicate **father** never occurs in $\Phi(R_2, \texttt{mother}(y,z))$ and atom $\texttt{mother}(y,z)$ is the only one atom in $\Phi(R_2, \texttt{mother}(y,z))$ except atoms in $\Phi(R_2, \texttt{father}(y,z))$. As a result of the replacement of predicates, a set of atoms are modified as

$$\{\texttt{grandmother}(x,z), \texttt{parent}(x,y), \texttt{mother}(y,z)\}. \quad \cdots(s1'')$$

In STEP5, finally, according to the above set $s1''$, the following new rule is generated and added to R_2.

$$\texttt{grandmother}(x,z) \leftarrow \texttt{parent}(x,y), \texttt{mother}(y,z) \quad \cdots(r8)$$

The rule $r8$ satisfies the requirement for analogical generalization given at Definition 7, and it is just appropriate rule about the target concept. In this case, good learning has been performed, because the rule which is closely similar to the rule for target concept is in the existing knowledge base.

In rule based systems, generally, the lack of rules causes either interruptions or mistakes on inference. ANGEL is useful for such a situation, because it is possible to continue inference by generating new rules from given examples.

Next we show an example of acquiring rules for the system for parsing simple English sentences. The target system is capable of parsing English sentences by means of syntactic rules shown as Figure2. In this system a sentence is treated as a list. For example the sentence "*The sun rises in the east*" is represented as the list,

$$[\texttt{the}, \texttt{sun}, \texttt{rises}, \texttt{in}, \texttt{the}, \texttt{east}]$$

And

```
noun_phrase([the,sun,rises,in,the,east],
            [rises,in,the,east])
```

indicates that [the, sun] is noun phrase. The system examines whether or not a given sentence is grammatically valid by a backward chaining inference by means of the syntax rules.

```
sentence(s,e) ← noun_phrase(s,v1),verb_phrase(v1,e).
sentence(s,e) ← noun_phrase(s,v1),verb_phrase(v1,v2),
                    prepositional_phrase(v2,e).
sentence(s,e) ← present_progressive(s,e).
sentence(s,e) ← present_passive_voice(s,e).
sentence(s,e) ← present_perfect(s,e).
noun_phrase(s,e) ← determiner(s,v1),noun(v1,e).
noun_phrase(s,e) ← noun(s,e).
prepositional_phrase(s,e) ← preposition(s,v1),
                    noun_phrase(v1,e).
verb_phrase(s,e) ← verb(s,e).
verb_phrase(s,e) ← verb(s,v1),noun_phrase(v1,e).
present_progressive(s,e) ← noun_phrase(s,v1),
        present_BE(v1,v2),present_participle(v2,e)
present_progressive(s,e) ← noun_phrase(s,v1),
        present_BE(v1,v2),present_participle(v2,v3),
        noun_phrase(v3,e)
verb(s,e) ← BE(s,e).
verb(s,e) ← main_verb(s,e).
verb(s,e) ← present_verb(s,e).
verb(s,e) ← past_verb(s,e).
BE(s,e) ← present_BE(s,e).
BE(s,e) ← past_BE(s,e).
main_verb(s,e) ← present_main_verb(s,e).
main_verb(s,e) ← past_main_verb(s,e).
present_verb(s,e) ← present_BE(s,e).
past_verb(s,e) ← past_BE(s,e).
present_verb(s,e) ← present_main_verb(s,e).
past_verb(s,e) ← past_main_verb(s,e).
auxiliary_verb(s,e) ← present_auxiliary_verb(s,e).
auxiliary_verb(s,e) ← past_auxiliary_verb(s,e).
participle(s,e) ← present_participle(s,e).
participle(s,e) ← past_participle(s,e).
determiner(s,e) ← THE(s,e).
noun(s,e) ← SUN(s,e).
noun(s,e) ← EAST(s,e).
noun(s,e) ← DOOR(s,e).
noun(s,e) ← HER(s,e).
noun(s,e) ← HE(s,e).
noun(s,e) ← I(s,e).
noun(s,e) ← HOMEWORK(s,e).
present_main_verb(s,e) ← HAVE(s,e).
present_main_verb(s,e) ← RISES(s,e).
present_auxiliary_verb(s,e) ← HAVE(s,e).
present_BE(s,e) ← IS(s,e).
past_participle(s,e) ← CLOSED(s,e).
past_participle(s,e) ← RESPECTED(s,e).
past_participle(s,e) ← FINISHED(s,e).
preposition(s,e) ← IN(s,e).
preposition(s,e) ← BY(s,e).
```

Figure 2: A part of rules in existing knowledge base

As Figure2 indicates, initially, the rule to define syntax about the present passive voice is insufficient. Then we have tried to generate a lacking rule by ANGEL.

For the target concept "present_passive_voice(s,e)", we have given the following example E_2 to ANGEL.

```
E2 = { present_passive_voice([the,door,is,closed],[]),
       THE([the,door,is,closed],[door,is,closed]),
       DOOR([door,is,closed],[is,closed]),
       IS([is,closed],[closed]),
       CLOSED([closed],[])}
```

Firstly, the given example E_2 has been extended to the following set $\tilde{E}_2$.

```
Ẽ2 = { present_passive_voice([the,door,is,closed],[]),
       THE([the,door,is,closed],[door,is,closed]),
       DOOR([door,is,closed],[is,closed]),
       IS([is,closed],[closed]),
       CLOSED([closed],[]),
       noun_phrase([the,door,is,closed], [is,closed]),
       sentence([the,door,is,closed],[closed])}
```

Then, the useful correspondence has been found as follows by using a rule for "present_progressive" as a base rule.

```
{(present_progressive(s,e),
  present_passive_voice([the,door,is,closed],[])),
 (noun_phrase(s,v1),
  noun_phrase([the,door,is,closed],[is,closed])),
 (present_BE(v1,v2), IS([is,closed],[closed])),
 (present_participle(v2,e),CLOSED([closed],[]))}
```

As a result, we have confirmed that ANGEL generates the following one rule successfully.

```
present_passive_voice(s,e) ← noun_phrase(s,v1),
                             present_BE(v1,v2),
                             past_participle(v2,e)  ···(r9)
```

The generated new rule $r9$ is added to the knowledge base.

Again we have given an example sentence "A mouse is caught by a cat." for the same target concept.

In this case, two distinct rules $r10$ and $r11$ are generated by using the identical base rule in the existing knowledge base.

```
present_passive_voice(s,e) ← noun_phrase(s,v1),
                             present_BE(v1,v2),
                             past_participle(v2,v3),
                             prepositional_phrase(v3,e)
                                                    ···(r10)

present_passive_voice(s,e) ← sentence(s,v1),
                             participle(v1,v2),
                             preposition(v2,v3),
                             noun_phrase(v3,e)
                                                    ···(r11)
```

Like the above, ANGEL sometimes generates several rules for one example. It is now important to examine whether each of the generated rules is appropriate. For instance, The rule $r10$ is a suitable rule, whereas the rule $r11$ is obviously strange. The reason for this is none of the rules in the existing knowledge base are really similar to the given example. Since atom noun_phrase(v_3,e) in selected base rule

```
present_progressive(s,e) ← noun_phrase(s,v1),
                           present_BE(v1,v2),
                           present_participle(v2,v3),
                           noun_phrase(v3,e)
```

corresponds to atom prepositional_phrase(v_3, e) in the rule $r10$ and atom noun_phrase(v_3, e) in the rule $r11$ (namely, the given example is regarded as the sentence consisting of some phrases and noun_phrase), the similarity between the base rule and the rule $r11$ are stronger than the one between the base rule and the rule $r10$ in respect of these atoms.

Next, we have supplied a sentence "He was killed by them. " to attempt to generate a rule for another target concept past_passive_voice(s, e). ANGEL could generate a new rule $r12$ by employing a rule $r10$ generated just now.

$$\begin{aligned}\texttt{past\_passive\_voice}(s,e) \leftarrow\ & \texttt{noun\_phrase}(s, v_1),\\ & \texttt{past\_BE}(v_1, v_2),\\ & \texttt{past\_participle}(v_2, v_3),\\ & \texttt{prepositional\_phrase}(v_3, e)\\ & \cdots (r12)\end{aligned}$$

In this case, since an appropriate base rule, which does not exist initially, has occurred in knowledge base, a good rule is generated accurately by selecting it. ANGEL is capable of growing knowledge base gradually by employing rules generated by ANGEL itself as base rules.

Let us discuss the computational complexity of ANGEL. In order to evaluate the similarity between atoms, ANGEL has to compute deductive closures of each of the atoms. And the similarities between atoms in arbitrary correspondences have been estimated to find the most suitable pair of the atoms in the given example and the base rule. Therefore, procedure of ANGEL may be expensive as a whole, although hypothesis space to be considered is small. In fact, as a result of implementing ANGEL on Sun SPARC Station2 with SICStus Prolog, it took a few minutes to generate a English syntax rule.

The approach evaluating similarities between atoms based on their deductive closures is theoretically interesting, but it may not be practical. For the purpose of practical learning, some restrictions on either forms of the background knowledge or the hypothesis language are required like Muggleton's GOLEM[Muggleton 1990]. We think we will have to improve the practicability of ANGEL in the near future.

4 Related works

In this section, we characterize ANGEL from a viewpoint of general machine learning framework.

ANGEL belongs to the category of learning from examples, in the sense that it generates new rules by generalizing given examples. In inductive learning methods, generally, pre-defined generalization rules are used for generalizing examples. ANGEL also uses three kinds of generalization rules corresponding to dropping condition rule, turning constants to variables rule and constructive generalization rule based on logical implications [Michalski 1983], all of them are considered as the primary generalization rules in learning from examples. However, ANGEL differs from the ordinary inductive learning methods in using the existing rules as the bias. That is, ordinary inductive learning uses no existing rules, even if so, it uses them for the constructive induction. On the other hand, ANGEL employs the similarity between the existing rules and the given example in order to drop conditions, so it can reduce the hypothesis space extremely.

ANGEL is related to inductive logic programming (ILP), because it generates rules represented as Horn clauses by induction. ILP is also capable of learning new rules with reference to existing rules. Both Muggleton and Buntine's CIGOL[Muggleton and Buntine 1988] and Wirth's LFP2[Wirth 1989], which are typical examples of ILP system, use operators based on inverting resolution to augment incomplete clausal theories. The difference between these systems and ANGEL is the way of employing existing background knowledge. That is, in both of their systems, background knowledge is not employed as biases at all. In fact, rules can be acquired under no background knowledge. Therefore the interaction between user and system is inevitable in their systems to derive reasonable rules. Whereas, ANGEL employs background knowledge as a bias. A given example is generalized through mapping a structure of a rule in existing knowledge base. It provides a strong restriction for induction and serves to generate a few useful new rules.

ANGEL evaluates a similarity between existing rule and a given example to learn a new rule. Therefore it can also be regarded as a kind of method for learning by analogy. Davies and Russell [1987] have defined, in their paper, reasoning by analogy as the process of inferring that a property Q holds of a particular situation T (called the target) from the fact that T shares a property P with another situation S (called the source) that has property Q. In analogy, it is very important to match between the target and the source. Similarly, in ANGEL, the matching between existing rules and a given example, which is called correspondence in this paper, must be found successfully. Now we compare ANGEL with several methods with respect to the way of matching.

Haraguchi and Arikawa [1986] have formalized the reasoning by analogy on a deduction system. In their method, the domain for reasoning is represented by a set of definite clauses, and the similarity between objects is defined as the identity of predicates. Therefore the matching is performed by pairing the atoms which are described with the same predicate. On the other hand, ANGEL finds a correspondence between atoms based on their similarities, that is, it will not require identity of predicates. And it enables ANGEL to generate completely novel rules.

Recently, Arima [1991] has analyzed analogy from the point of logical relevance. His formulation is based on the idea as follows.

1. The property to be projected from the source to the target must be justified.
2. The similarities, which means the properties shared by both the source and the target, should be formed by the minimum justifications.

Unlike ANGEL, the shared properties must be represented by the same predicates both with the source and with the target.

Gentner [1983] has also developed a method, called Structure Mapping, for the matching between the target and the source. In her method, first an atom is matched with another atom, when both of them are described with the same

predicates, and next, the object in each atom is matched. And the process of the matching is repeated based on newly matched objects. ANGEL is similar to Structure Mapping, because the matching between atoms is achieved based on the matched objects. However, there are the following two differences between them.

1. Although Structure Mapping requires the identity to several kinds of predicates (e.g. *greater*, *cause*, etc.) in order to match between atoms, ANGEL will not require the identity of predicates at all.

2. In Structure Mapping, the similarity between descriptions is defined by the identification of predicates and the number of matched descriptions. On the other hand, in ANGEL, it is defined as the subsumption between deductive closures of atoms based on the logical consideration.

ANGEL is also related to both the explanation-based learning (EBL)[Mitchell *et al.* 1986] and Russell's single-instance generalization (SIG)[Russell 1987], because all of them are capable of learning from one example and background knowledge. However, EBL has to need completeness for background knowledge, so rules produced by EBL are limited to ones which are deducible from background knowledge. In this sense, EBL cannot generate really new rules. SIG requires weak background knowledge, called determinations, in stead of complete one. That is, it can learn rules under comparatively insufficient background knowledge in contrast to EBL. Properly new rules cannot, however, be generated, because it does not deal with non-deductive reasoning.

5 Conclusion

This paper has described an approach to learning from an example by analogical generalization.

The notable features of ANGEL are shown as follows.

1. ANGEL is able to generate a new rule from a given single example by analogical generalization.

2. A similarity between an existing rule and an example can be evaluated a similarity between atoms forming each of them.

3. A similarity between atoms is defined based on the subsumption relation between deductive closures of atoms, and it enables to compute similarities formally.

Through the experiment for the domain of parsing English sentences, we have confirmed that ANGEL is useful for acquiring knowledge on knowledge based systems.

In this paper, from the inductive learning point of view, we have highlighted the method to generate a new rule from a given example. The definition of similarity introduced here is not specific for inductive learning. We plan to apply this idea to other various reasoning paradigms (e.g. ordinary analogical reasoning, deductive reasoning and so on) to improve performance and applicability of them.

This work was supported partly by the Grant-in-Aid for scientific research from the Ministry of Education.

References

[Mitchell 1977] T. M. Mitchell: "Version spaces: a candidate elimination approach to rule learning", Proc. of 5th IJCAI, pp.305–310 (1977).

[Winston 1980] P. H. Winston: "Learning and reasoning by analogy", Comm. of ACM, Vol.23, No.12, pp.689–703 (1980).

[Dietterich and Michalski 1983] T. G. Dietterich and R. S. Michalski: "A comparative review of selected methods for learning from examples", Machine learning, R. S. Michalski, J. G. Carbonell and T. M. Mitchell (Eds.), Morgan Kaufmann, pp.41–81 (1983).

[Michalski 1983] R. S. Michalski: "A theory and methodology of inductive learning", Machine learning, R. S. Michalski, J. G. Carbonell and T. M. Mitchell (Eds.), Morgan Kaufmann, pp.83–134 (1983).

[Gentner 1983] D. Gentner: "Structure-mapping: A theoretical framework for analogy", Cognitive science, Vol. 7, pp.155–170 (1983).

[Haraguchi and Arikawa 1986] M. Haraguchi and S. Arikawa: "A formulation of analogical reasoning and its realization", Journal of JSAI, Vol. 1, No. 1, pp.132–139 (1986) (in Japanese).

[Mitchell *et al.* 1986] T. M. Mitchell, R. M. Keller and S. T. Kedar-Cabelli: "Explanation-based generalization: A unifying view", Machine Learning, Vol.1 No.1, pp.47–80 (1986).

[Davies and Russell 1987] T. R. Davies and S. J. Russell: "A logical approach to reasoning by analogy", Proc. 10th IJCAI, pp.264–270 (1987).

[Russell 1987] S. J. Russell: "Analogy and single-instance generalization", Proc. 4th Int'l Machine Learning Workshop, pp.390–397 (1987).

[Muggleton and Buntine 1988] S. Muggleton and W. Buntine: "Machine invention of first-order predicates by inverting resolution", Proc. of 4th Int'l Machine Learning Workshop, pp.390–397 (1988).

[Wirth 1989] R. Wirth: "Completing logic programs by inverse resolution", Proc. of 4th European Working Session on Learning, pp.239–250 (1989).

[Muggleton 1990] S. Muggleton and C. Feng: "Efficient induction of logic programs", Proc. of 1st Int'l Workshop on Algorithmic Learning Theory, pp.368–381 (1990).

[Arima 1991] J. Arima: "A logical analysis of relevancy in analogy ", Proc. of 5th Annual Conf. of JSAI, pp.235–238 (1991) (in Japanese).

[Mori *et al.* 1991] T. Mori, T. Ohkawa, N. Babaguchi and Y. Tezuka: "Learning based on similarity between rules and examples", IPS Japan research report on artificial intelligence, 74-6, pp.49–57 (1991) (in Japanese).

[Ohkawa *et al.* 1991] T. Ohkawa, T. Mori, N. Babaguchi and Y. Tezuka: "Class directed generalization", Proc. of 3rd SCAI, pp.266–276 (1991).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Logical Structure of Analogy

PRELIMINARY REPORT

Jun ARIMA

Institute for New Generation Computer Technology
21F, Mita Kokusai Bldg., 4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
arima@icot.or.jp

Abstract: This paper treats a general type of analogical reasoning which is described as follows: when two objects, B (the*base*) and T (the *target*), share a property S (the *similarity*), it is conjectured that T satisfies another property P (the *projected property*) which B satisfies as well.

Through a formal analysis of this type of analogy, a logical relation is explored which is necessarily satisfied by the tuple, T, B, S, P, under an axiom, $\mathcal{A}$. Unlike previous studies on analogy, this work does not give any particular assumption a priori to the tuple.

By the analysis, it is shown to be reasonable that analogical reasoning is possible only if a certain form of rule, called the *analogy prime rule*, is a deductive theorem of a given theory, and that, from the rule, together with two particular conjectures, an analogical conclusion is derived. Also, a candidate is shown for a non-deductive inference system which can yield both conjectures.

1 Introduction

When we explain a process of reasoning by analogy, we may say, "An object T is similar to another object B in that T shares a property S with B and B satisfies another property P. Therefore, T also satisfies P". We may express this more formally using the following schema.

$$\frac{S(B) \wedge P(B) \quad S(T)}{P(T)}$$

Here, T will be called the *target*, B the *base*, S the *similarity* between T and B, and P the *projected property*.

The above description of the process of analogy is, however, insufficient. Researchers studying analogy have come to recognize the necessity of revealing some implicit condition which influences the process but does not appear in the above schema. The importance of this has already been discussed enough in [3]. The implicit condition to be satisfied by appropriate analogical factors, T, B, S, and P, can, formally, be characterized only by a given theory (axiom), written as $\mathcal{A}$. The objective of this paper is to explore the particular relation of analogy which T, B, S, P and $\mathcal{A}$ necessarily satisfy.

In the study of analogy, the following have been central problems:

1) what object should be selected as a base w.r.t a target,

2) which property is significant in analogy among properties shared by two objects, and

3) what property is to be projected w.r.t. a certain similarity.

Many significant works have been vigorously conducted on these problems, though they were only partially successful in answering these questions, that is, by giving intuitive and strong assumptions a priori. In many works, a base case was assumed to be given w.r.t. a target case [4, 11, 10]. In almost all works, the important similarity (or similarity measure) was defined a priori independently of what property was projected [20, 6, 10, 7, 5]. In logical works [8, 5], especially in [3], nice logical relations among the analogical factors could be seen, though they, like others, were given without sufficient examinations which would show *why* and *how* their relations were *necessary*.

Unlike previous studies on analogy, this work does not give any particular assumption a priori to the analogical factors. Clarifying the relation between the factors, T, B, S, P and $\mathcal{A}$, will be enough to answer the above three problems once and for all. The relation shown by this paper is a general solution for them and might show how useful a formal treatment is in analyzing analogical behavior.

First, through a logical analysis of analogy, it is shown to be reasonable that, when an analogical inference is done under a theory $\mathcal{A}$, a particular form of rule must be a logical conclusion (a theorem) of $\mathcal{A}$ and that analogical inference is accomplished by two particular types of (generally non-deductive) conjectures. Then, a non-deductive inference is proposed, which is shown to be an

adequate candidate to yield the conclusions of both these conjectures.

2 A Logical Analysis

2.1 Preparations

In this paper, we use standard formal logic and notations, while defining the following. An n-ary *predicate* U is generally expressed by λxQ, where x is a tuple of n object variables, Q is a formula in which no object variables except variables in x occur free. If t is a tuple of n terms, $U(t)$ stands for the result of replacing each occurrence of (elements of) x in Q with (each corresponding element of) t simultaneously. For any formulas A and F, when $A \vdash F$ and $\nvdash F$ (that is, F is not *valid*), we say F is a *genuine* theorem of A and express it simply as $A \mid\!\!-\!\!F$.

We will use a closed formula of first order logic $\mathcal{A}$ for a *theory*, (generally n) terms T for a *target* and (generally n) terms B for a *base*. A property is expressed by a predicate, for instance, a *similarity* and a *projected property* are expressed by predicates, S and P respectively.

2.2 Approach To A Seed of Analogy

We can understand analogical reasoning as follows:

(1) Example-based Information: "An object, x' (corresponding to a base), satisfies both properties S and P $(\exists x'.(S(x') \wedge P(x')))$."

(2) Similarity-based Information: "Another object, x (corresponding to a target), satisfies a shared property S with x' $(S(x))$."

(3) Analogical Conclusion: "The object x would satisfy the other property P $(P(x))$."

Then,

"Analogical reasoning is to reason (3) from $\mathcal{A}$ together with (1)+(2)." (A)

Let this understanding be our starting point of analysis.

As analogy is not, generally, deductive, this starting point may, unfortunately, be expressed only as follows. In the notation of proof theory,

$$\mathcal{A}, \exists x'.(S(x') \wedge P(x')), S(x) \nvdash P(x). \quad (1)$$

As analogy, however, infers $P(x)$ from the premises, it implies that some knowledge is assumed in the premise part of (1). Let the assumed knowledge be $F(x)$, providing that it depends on the x in general. That is,

$$\mathcal{A}, \exists x'.(S(x') \wedge P(x')), S(x), F(x) \vdash P(x). \quad (2)$$

Thus, the essential information newly obtained by analogy is $F(x)$ in the above rather than the explicit projected property P . Making $J(x)$ stand for the conjunction of the example-based information and $F(x)$, the above meta-sentence is transformed equivalently to

$$\mathcal{A} \vdash \forall x.(J(x) \wedge S(x) \supset P(x)), \quad (3)$$

because $\mathcal{A}$ is closed. This implies that a rule must be a theorem of $\mathcal{A}$ and that the rule concludes any object which satisfies $J(x)$ to satisfy P when it satisfies S. Once J is satisfied, (by reason of $(S(x) \supset P(x))$,) the analogical conclusion ("an object satisfies P") can be deduced from the similarity-based information ("the object satisfies S). For this reason, this rule will be called the *analogy prime rule* (it will be specified in more detail later), J will be called the *analogy justification.*

Moreover, it is improbable that the analogy prime rule is a valid formula, because, if so, any pair of predicates can be an analogical pair of a similarity and a projected property independently of $\mathcal{A}$. Thus, the analogical prime rule must be a genuine theorem of $\mathcal{A}$,

$$\mathcal{A} \mid\!\!-\forall x.(J(x) \wedge S(x) \supset P(x)). \quad (4)$$

Consequently, an object T which satisfies S is concluded to satisfy P from an analogy prime rule by analogical reasoning that assumes that T satisfies the analogy justification $(J(T))$. That is, our starting point (A) can be specified from two aspects.

"An analogical conclusion can be obtained from an analogy prime rule together with example-based information and similarity-based information." (B)

"A non-deductive jump by analogy, if it occurs, is to assume that the analogy justification of the prime rule is satisfied." (C)

In the following part of this paper, the analogy justification and non-deductivity will be further explored. Before beginning an abstract discussion, it may be useful to see concrete examples of analogical reasoning. The next section introduces "target" examples of analogical reasoning to be clarified here.

2.3 Examples

Example1: Determination Rule[3]. "Bob's car (C_{Bob}) and Sue's car (C_{Sue}) share the property of being 1982 Mustangs ($Mustang$). We infer that Bob's car is worth about \$3500 just because Sue's car is worth about \$3500. (We could not, however, infer that Bob's car is painted red just because Sue's car is painted red.)"
Example-based Information:

$$Model(C_{Sue}, Mustang) \wedge Value(C_{Sue}, \$3500), \quad (5)$$

Similarity-based Information:

$$Model(C_{Bob}, Mustang), \quad (6)$$

Example2: Brutus and Tacitus [1]. " Brutus feels pain when he is cut or burnt. Also, Tacitus feels pain when he is cut. Therefore, if Tacitus is burnt, he will feel pain."
Example-based Information:

$$(Suffer(Brutus, Cut) \supset FeelPain(Brutus)) \quad (7)$$
$$\wedge(Suffer(Brutus, Burn) \supset FeelPain(Brutus)) \quad (8)$$

Similarity-based Information:

$$Suffer(Tacitus, Cut) \supset FeelPain(Tacitus) \quad (9)$$

Example3: Negligent Student[1]. " When I discovered that one of the newcomers ($Student_T$) to our laboratory was a member of an orchestra club ($Orch$), remembering that another student ($Student_B$) was a member of the same club and he was often negligent of study (***Study***), I guessed that the newcomer would be negligent of study, too."
Example-based Information:

$$Member\_of(Student_B, Orch)$$
$$\wedge Negligent\_of(Student_B, Study) \quad (10)$$

Similarity-based Information:

$$Member\_of(Student_T, Orch) \quad (11)$$

2.4 Logical Analysis: a rule as a seed of analogy

In treating analogy in a formal system, as the information of a base object being S and P is projected into a target object, it is desirable to treat such properties as *objects* so that we can avoid the use of second order language. As an example, the fact that Bob's car is a Mustang is represented by "$Model(C_{Bob}, Mustang)$" rather than simply as "$Mustang(C_{Bob})$". In the remaining part, we rewrite $S(x)$ to $\Sigma(x, S)$ and $P(x)$ to $\Pi(x, P)$. Σ will be called a *similar attribute*, Π will be a *projected attribute*, S as an object will be a *similar attribute value*, and P as an object will be a *projected attribute value*. Then, (4) is rewritten

$$\mathcal{A} \vdash \forall x, s, p.(J(x, s, p) \wedge \Sigma(x, s) \supset \Pi(x, p)), \quad (12)$$

considering the most general case that the analogy justification J depends on all of these factors.

Again, when 3-tuple < object: X, similar attribute value: S, projected attribute value: P > satisfies the analogy justification J, object X is conjectured to satisfy the projected property $\lambda x.\Pi(x, P)$ (analogical conclusion) just because X has the similarity $\lambda x.\Sigma(x, S)$. That is, $J(x, s, p)$ can be considered a condition, where x could be concluded to be p from x being s by analogical reasoning.

Now, recalling that an analogical conclusion is obtained from the analogy prime rule with example-based information and similarity-based information, consider what information can be added by the information in relation to the analogy prime rule.

1) **Example-based Information:** This shows that there exists an object as a base which satisfies a similarity and a projected property ($\exists x'.(\Sigma(x', S) \wedge \Pi(x', P))$). It seems to be adequate that the base, B, satisfying $\Sigma(x', S)$ can also be derived to satisfy $\Pi(x', P)$ from the prime rule, because B can be considered a target which has similarity S. That is, 3-tuple $< B, S, P >$ satisfies the analogy justification. Consequently, from arbitrariness in selection of an object as a base in this information, what is obtained from this information is $\exists x'.\ J(x', S, P)$.

2) **Similarity-based Information:** This shows that an object as a target, T, satisfies the same property S in the above. Just by this fact, an analogical conclusion is obtained, by assuming that the object satisfies J by some conjecture. That is, there exists some attribute value p' and 3-tuple $< T, S, p' >$ satisfies J ($\exists p'.\ J(T, S, p')$).

3) **Analogical Conclusion:** With the above two pieces of information, an analogical conclusion, "T satisfies $\Pi(x, P)$", is obtained from the analogy prime rule. Therefore, such 3-tuple $< T, S, P >$ satisfies J ($J(T, S, P)$).

In the above discussion, T, S, and P are arbitrary. Therefore, the following relation about the analogy justification turns out to be true:

$$\forall x, s, p.(\ \exists x'.J(x', s, p) \wedge \exists p'.J(x, s, p') \supset J(x, s, p)\ ). \quad (13)$$

(13) is able to represent it equivalently as follows:

$$J(x, s, p) = J_{att}(s, p) \wedge J_{obj}(x, s), \quad (14)$$

where both J_{att} and J_{obj} are predicates, that is, each of them has no free variables other than its arguments.

The point shown by this result is that any analogy justification can be represented by a conjunction in which variable x and variable p occur separately in different conjuncts.

By (12) and (14), the analogical prime rule can be defined as follows.

Definition 1 Analogy Prime Rule
A rule is called an analogy prime rule w.r.t. $< \Sigma(x, s); \Pi(x, p) >$, *if it has the following form:*

[1]The author thanks Satoshi Sato (Hokuriku Univ.) for showing this challenging example.

$$\forall x, s, p.(J_{att}(s,p) \wedge J_{obj}(x,s) \wedge \Sigma(x,s) \supset \Pi(x,p)), \quad (15)$$

where J_{att}, J_{obj}, Σ and Π are predicates. (That is, each of $J_{att}(s,p)$, $J_{obj}(x,s)$, $\Sigma(x,s)$ and $\Pi(x,p)$ is a formula in which no variable other than its arguments occurs free.) □

In (15), $J_{att}(s,p)$ will be called the *attribute justification* and $J_{obj}(x,s)$ will be called the *object justification.*

Also, by the above discussion, the following two conjectures can be considered as causes which make analogy non-deductive.

- **Example-based Conjecture (EC):** An object shows a existing concrete combination of a similarity and a projected property. This specializes the prime rule and allows it to be applicable to a similar object. Assuming some generally non-deductive inference system under $\mathcal{A}$, "$\mid\!\sim^{\mathcal{A}}$" (we will propose such a system later),

$$\exists x.(\Sigma(x,S) \wedge \Pi(x,P)) \mid\!\sim^{\mathcal{A}} J_{att}(S,P). \quad (16)$$

- **Similarity-based Conjecture (SC):** Just because an object satisfies S, application of the specialized prime rule to the object is allowed.

$$\Sigma(x,S) \mid\!\sim^{\mathcal{A}} J_{obj}(x,S). \quad (17)$$

In case that the attribution justification ($J_{att}(s,p)$) is a valid formula, example-based information becomes unnecessary in yielding analogical conclusion. Thus, it could, in general, be essential in analogical reasoning to guess $J_{att}(s,p)$ which is not a valid formula. The object justification ($J_{obj}(x,s)$) is, still, important in another sense, because it can be considered to express a *really* significant similarity. It is not an unusual case when a really significant similarity is not observable. Consider a case of Example 2. Having a nervous system will be a sufficient condition for an object to feel pain, thus, whether an object has a nervous system is a significant factor in making a conjecture on feeling pain. In this case, however, we could, without dissection, not obtain a direct evidence which shows that Tacitus and Btutus have nervous systems, while we obtain only a *circumstantial* evidence that the both feel pain when they are cut. Thus, the similarity-based conjecture is to guess such a really significant but implicit similarity, the object justification ($J_{obj}(x,s)$), from an observed similarity $\Sigma(x,s)$.

To summarize, a logical analysis of analogy could draw conclusions as follows.

Analogical reasoning is possible only if a certain *analogical prime rule* is a *genuine* theorem of a given theory and the process of analogical reasoning can be divided into the following 3 steps: 1) the attribute justification part of the rule is satisfied by EC from example-based information. 2) the object justification part of the rule is satisfied by SC from similarity-based information, and, 3) from similarity-based information and the analogy prime rule specialized by the two preceding steps, an analogical conclusion is obtained by deduction.

A question remains unclear, that is, what inference is EC and what SC? Though we cannot identify the mechanism underlying each of the conjectures, we can propose a (generally) non-deductive inference system as their candidates. The next section shows this.

3 Non-deductive Inference for Analogy

This section explores a type of generally non-deductive inference by which a conjecture G is obtained from a given theory $\mathcal{A}$ with additional information K.

Generally speaking, what properties should be satisfied by a, generally, non-deductive inference? It might be desirable that a non-deductive inference satisfies at least the following conditions. First, it should subsume deduction, that is, any deductive theorem is one of its theorems, because any deductive conclusion would be desirable. Secondly, any conclusion obtained by it must be able to be used deductively, that is, from such a conclusion, it should be possible to yield more conclusions using, at least, deduction. And, thirdly, any conclusion obtained must be consistent with given information. We define a class of inference systems which satisfy the above three conditions.

Definition 2 *An inference system under a theory $\mathcal{A}$ (written $\mid\!\sim^{\mathcal{A}}$) is deductively expansible if the following conditions are satisfied. For any set of sentences $\mathcal{A}$ and K and any sentences G and H,*

i) *Subsuming deduction:*

if $\mathcal{A}, K \vdash G$ *then* $K \mid\!\sim^{\mathcal{A}} G$.

ii) *Deductive usefulness:*

if $K \mid\!\sim^{\mathcal{A}} G$ *and* $\mathcal{A}, K, G \vdash H$, *then* $K \mid\!\sim^{\mathcal{A}} H$.

iii) *Consistency:*

if $K \mid\!\sim^{\mathcal{A}} G$ *and* $\mathcal{A} \cup K$ *is consistent, then* $\mathcal{A} \cup K \cup \{G\}$ *is consistent.*

The following inference system is an example of a deductively expansible system.

Definition 3 *G is a conjecture from $\mathcal{A}$ based on K by (atomic) circumstantial reasoning (written $K \mid\sim^{\mathcal{A}}_{*} G$)* [2]. *iff*

i) $\mathcal{A}, K \vdash G$, *or*

ii) $\mathcal{A}, E \vdash G$
if there exists a minimal set of atomic formulas[3] *E s.t. $\mathcal{A}, E \vdash K$, and $\mathcal{A} \cup E$ is consistent if $\mathcal{A} \cup K$ is consistent*[4].

Proposition 1
If $K \mid\sim^{\mathcal{A}} G$ and $K, G \mid\sim^{\mathcal{A}}_{*} H$, then $K \mid\sim^{\mathcal{A}} H$.

Corollary 1 *If $K \mid\sim^{\mathcal{A}}_{*} G$, then $K \mid\sim^{\mathcal{A}} G$.*

Corollary 1 shows that circumstantial reasoning is deductively expansible, and proposition 1 (together with the corollary) shows that inference done by multiple applications of circumstantial reasoning is also deductively expansible.

Circumstantial reasoning ($K \mid\sim^{\mathcal{A}}_{*} G$) implies a very general and useful inference class in that so many types of inference used in AI can be considered as circumstantial reasoning. Deduction and abduction, for example, are obviously circumstantial reasoning. Moreover, if we loosen the condition "atomic formulas" to "clauses", inductive learning from examples is the case where $\mathcal{A}$ is empty in general, K is "examples" and G is inductive knowledge obtained by "learning"[5] [6]

Now, we assume that both EC and SC are circumstantial reasoning, but based on different information. Then, we can see analogical reasoning in more detail.

Let an analogy prime rule w.r.t. $< \Sigma(x,s); \Pi(x,p) >$ be a theorem of $\mathcal{A}$. Then, when example-based information, $\Sigma(B,S) \wedge \Pi(B,P)$, is introduced, by circumstantial reasoning from the prime rule, some justifications are satisfied, that is,

$$\Sigma(B,S) \wedge \Pi(B,P) \mid\sim^{\mathcal{A}}_{*} J_{att}(S,P) \wedge J_{obj}(B,S), \quad (18)$$

which concludes a specialized prime rule,

$$\forall x.(J_{obj}(x,S) \wedge \Sigma(x,S) \supset \Pi(x,P)). \quad (19)$$

Even if similarity-based information $\Sigma(T,S)$ is introduced, to obtain analogical conclusion $\Pi(T,P)$ by circumstantial reasoning, some information apart from the prime rule turns out to be needed in $\mathcal{A}$. And, both EC and SC are generally needed to accomplish analogical reasoning, which implies that multiple application of circumstantial reasoning is necessary. Even in such a case, circumstantial reasoning remains worthwhile (Proposition 1).

4 Classification of Analogy and Examples

Each EC and SC has two cases; a deductive one and a non-deductive one. According to this measure, analogical inference can be divided into 4 types. A typical example is shown in each class and explored.

4.1 deductive EC + deductive SC

Typical reasoning of this type was proposed by T.Davies and S.Russell [3]. They insisted that, to justify an analogical conclusion and to use information of the base case, a type of rule, called a *determination rule*, should be a theorem of a given theory. The rule can be written as follows:

$$\forall s,p.(\ \exists x'.(\Sigma(x',s) \wedge \Pi(x',p)) \supset \forall x.(\Sigma(x,s) \supset \Pi(x,p))\ ) \quad (20)$$

Example 1 (continued). In this example, the following determination rule is assumed to hold under $\mathcal{A}$.

$$\forall s,p.(\ \exists x'.(Model(x',s) \wedge Value(x',p)) \supset \forall x.(Model(x,s) \supset Value(x,p))\ ) \quad (21)$$

This rule is an analogy prime rule, because

$J_{obj}(x,s) = \Sigma(x,s) = Model(x,s)$,
$J_{att}(s,p) = (\exists x.\ Model(x,s) \wedge Value(x,p))$,
$\Pi(x,p) = Value(x,p)$.

Moreover,

EC:
$$Model(C_{Sue}, Mustang) \wedge Value(C_{Sue}, \$3500) \vdash J_{att}(Mustang, \$3500), \quad (22)$$

SC:
$$Model(C_{Bob}, Mustang) \vdash J_{obj}(C_{Bob}, Mustang). \quad (23)$$

This illustrates that reasoning based on determination rules belongs to the "deductive EC + deductive SC" type and that it can also be done by circumstantial reasoning.

[2] Circumstantial reasoning is essentially equivalent to "abduction" + deduction [13, 15]. However, "abduction" has many definitions and various usages in different contexts, so we like to introduce a new term for the type of inference in Definition 3 to avoid confusion.

[3] *Atoms*, that is, formulas which contain only one predicate symbol.

[4] If there exists such a minimal set of atomic formulas E, the case ii) involves the case i) apparently. Thus, the case i) can often be neglected in a usual application, for instance, if K is a *universal* formula which has the form $\forall x.F(x)$, where F is quantifier-free. Note that a clause is universal.

[5] In this case, $G = E$ in Definition 3, which implies that G is a minimal set to explain "example" K. Indeed, such minimality is very common in this field.

[6] Such a unified aspect of various reasoning in AI was pointed out by Koich Furukawa (ICOT) in a private discussion and a similar and more intuitive view can be seen in [5].

4.2 deductive EC + non-deductive SC

This type of analogical reasoning was explored by the author [1]. It was concluded that, once we assumed the following two premises for analogical reasoning, it seemed to be *an inevitable conclusion* that analogical reasoning which infers $P(T)$ from $S(T), S(B)$, and $P(B)$ satisfies *the illustrative criterion.* And if an inference system satisfies the criterion, the system is called an *illustrative analogy.*

Premise 1: "Analogy is done by projecting properties (satisfied by a base) from the base onto a target."

Premise 2: "The target is not a special object."

Premise 2 is also assumed in this paper, it is translated into an arbitrary selection of a target object. Premise 1 was translated as follows: $J(B)$, (where J is the justification in (4) and B stands for a base object) must be a theorem of $\mathcal{A}$, because it is essential in analogical reasoning to project $J(B)$ onto a target object T. That is, the non-deductive part in this reasoning is just SC which conjectures the property of the target object, and EC must be deductive.

Example 2 (continued). By illustrative analogy, a target is conjectured to satisfy properties used in an explanation of why a base satisfies a similarity. In this example, to explain the phenomena of the base case,"Brutus feels pain when he is cut or burnt", the following sentences must be in $\mathcal{A}$.

$$\forall x,i.(\ Nervous\_Sys(x) \wedge Destructive(i) \wedge Suffer(x,i) \supset FeelPain(x)\ ), \quad (24)$$

$$\wedge Nervous\_Sys(Brutus) \quad (25)$$

$$\wedge Destructive(Cut) \wedge Destructive(Burn) \quad (26)$$

From (24), the following follows:

$$\forall x,s,p.(\ Nervous\_Sys(x) \wedge Destructive(s) \wedge Destructive(p) \wedge (Suffer(x,s) \supset FeelPain(x)) \supset (Suffer(x,p) \supset FeelPain(x))\ ), \quad (27)$$

which is an analogy prime rule, that is,

$J_{obj}(x,s) = Nervous\_Sys(x)$,
$J_{att}(s,p) = Destructive(s) \wedge Destructive(p)$,
$\Sigma(x,s) = Suffer(x,s) \supset FeelPain(x)$,
$\Pi(x,p) = Suffer(x,p) \supset FeelPain(x)$.

$J_{att}(Cut, Burn)$ ("Both cut and burn are destructive") is a deductive theorem of $\mathcal{A}$ and a non-deductive conjecture, $J_{obj}(Tacitus, Cut)$ ("Tacitus has a nervous system"), is obtained by circumstantial reasoning from (24) based on the similarity-based information, $Suffer(Tacitus, Cut) \supset FeelPain(Tacitus)$.

4.3 non-deductive EC + deductive SC

As far as the author knows, this type of analogy has never been discussed. Example 3 seems to show this type of analogy.

Example 3 (continued). First, let us consider what we know from example-based information in this case. From the fact that a student ($Student_B$) was a member of the same club ($Orch$) and often neglected study ($Study$), we could find that "the orchestra club keeps its members very busy ($BusyClub(Orch)$)" and that "activities of the club are obstructive to one's study ($Obstructive\_to(Orch, Study)$)". This implies that we knew some causal rule like "If it is a busy club and its activities are obstructive to something, then any member of the club neglects the thing."

$$\forall x,s,p.(\ BusyClub(s) \wedge Obstructive\_to(p,s) \wedge Member\_of(x,s) \supset Negligent\_of(x,p)\ ) \quad (28)$$

Using this rule, we found the above information.

Thus, the above rule is assumed to be a theorem of $\mathcal{A}$. $BusyClub(Orch)$ and $Obstructive\_to(Orch, Study)$ are non-deductive conjectures and it can be obtained by circumstantial reasoning based on the above rule which is just an analogy prime rule, as follows:

$J_{obj}(x,s) = \Sigma(x,s) = Member\_of(x,s)$,
$J_{att}(s,p) = BusyClub(s) \wedge Obstructive\_to(p,s)$,
$\Pi(x,p) = Negligent\_of(x,p)$.

4.4 non-deductive EC + non-deductive SC

As an example of this type, we can take Example 2 again. We might know neither "Brutus has a nervous system" nor "Both cut and burn are destructive", which corresponds to the case that (25) and (26) are not in $\mathcal{A}$ (nor any deductive theorem of $\mathcal{A}$) in the previous Example 2. However, by circumstantial reasoning from (24) based on example-based information ("Brutus feels pain when he is cut or burnt"), "Both cut and burn are destructive" (and "Brutus has a nervous system") can be obtained, and based on similarity-based information ("Tacitus feels pain when he is cut"), "Tacitus has a nervous system", a really significant but implicit similarity, is obtained similarly to the previous example. Consequently, the analogical conclusion ("Tacitus would feel pain when he is burnt") is derived from (27) (or (24)) together with the above conjectures.

5 Conclusion and Remarks

- Through a logical analysis of analogy, it is shown to be reasonable that analogical reasoning is possible only if a certain *analogy prime rule* is a deductive theorem of a given theory. From the rule, together with *an example-based conjecture* and *a similarity-based conjecture*, *the analogical conclusion* is derived. A candidate is shown for a non-deductive inference system which adequately yields both conjectures.

- Results shown here are general and do not depend on particular pragmatic languages like the *purpose* predicate [10] nor on some numeric similarity measure [20]. These results can be applied to any normal deductive data bases (DDB) which consist of logical sentences.

  Application of this analogical reasoning to DDB may be one of the most fruitful. It is, generally speaking, very difficult to build a DDB which involves perfect knowledge about an item. Analogical reasoning will increase the chance of answering queries adequately, even when its deductive operation fails to answer. In a DDB, it is very common to see *inheritance* rules and *transitivity(-like)* rules, which have the form of the analogy prime rule, for instance,

$$\mathtt{Gran\_pa(x,y)} : -\mathtt{Parent(x,z)}, \mathtt{Parent(z,y)}. \quad (29)$$

  This is an analogy prime rule w.r.t. $<$ $\mathtt{Parent(z,y)}; \mathtt{Gran\_pa(x,y)} >$ ($\mathtt{z}$ is a variable for the similar attribute value and $\mathtt{x}$ is a variable for the projected attribute value). Assume that a query "?-$\mathtt{Gran\_pa(x,Tom)}$" is given to a database $\mathcal{A}$ which involves the above rule and the following facts:

$$\mathtt{Parent(Sue,Tom)}. \quad (30)$$
$$\mathtt{Gran\_pa(John,Bob)}. \quad (31)$$
$$\mathtt{Parent(Sue,Bob)}. \quad (32)$$

  The database cannot answer the query deductively, because it does not know who is a parent of **Sue**. If the database uses the proposed type of analogical reasoning, it is able to guess $\mathtt{Gran\_pa(John,Tom)}$ from **Bob**'s case just because **Tom** is similar to **Bob** in that their parents is the same.

  Interestingly, a method which discovers an analogy prime rule from knowledge data-base CYC is explored independently[17]. Such methods make analogical reasoning more common in DDB.

- By the side effect of this analysis, it becomes possible to compare analogy with other reasoning formally which have been studied vigorously in the area of artificial intelligence. *Analogical* reasoning **differs from** other reasoning, *abductive* and *deductive*, in that analogical reasoning actually uses example-based information (the base information). Consider the difference from, this time, abduction in the above database case. Even if the database uses (ordinal) abductive reasoning in the query, it cannot specify an adequate grandparent of **Tom**, the possible answer will be $\mathtt{x}$ s.t. $\mathtt{Gran\_pa(x,Tom)}$, $\mathtt{Parent(x,Sue)}$, $(\exists \mathtt{z}.)(\mathtt{Parent(x,z)}, \mathtt{Parent(z,Tom)})$, or **Sue** assuming $\mathtt{Parent(Sue,Sue)}$, etc [2, 14, 18, 9]. The reason for this failure is that abduction tries to *explain* only the target case.

  Moreover, comparing with enumerative *induction* and *case-based reasoning* (CBR) in which the use of examples are essential similarly to analogical reasoning, analogical reasoning has a salient feature in more strongly depending on a background knowledge (a given theory). Analogy can be seen as a *single instance generalization* as Davies and Russell pointed out [3]. Take an example, Example 3. From the analogy prime rule (28) and example-based information of *an* base case ($Student_B$), some non-deductive inference (ex. circumstantial reasoning) yields a more specified analogy prime rule,

$$\forall x.(\ Member\_of(x, Orch) \supset Negligent\_of(x, Study)\ ), \quad (33)$$

  which is a generalization of the example-based information,

$$Member\_of(Student_B, Orch) \wedge Negligent\_of(Student_B, Study). \quad (34)$$

  We should note that, in the process of this single instance generalization, an analogy prime rule in a background knowledge is used as an intermediary, and it might be considered the reason why analogy seems more plausible than a simple single instance generalization such that it yields (33) just from (34).

  In the research of formal inductive inference [16, 12], a back ground knowledge does not play such an important role. So, plenty of examples are needed until a plausible conclusion is obtained. Concerning CBR [19], though it uses base cases like analogical reasoning and, in order to retrieve their base cases, it uses an *index* which corresponds to the similarity S, the index is assumed to be given in spite of using background knowledge. Intuitively speaking, these methods will be very useful when a background knowledge is rather poor or difficult to formulate, and when the background knowledge is extremely strong or able to be formulated perfectly, deduction will be most useful, on the other

hand, the proposed type of analogy will be useful when rather strong and difficult to formulate.

- An implementation system for this type of analogy has been developed. Given a theory $\mathcal{A}$, a target T and a projected attribute $\Pi(x,p)$ (from a query, "?- $\Pi(T,p)$"), this system finds a base B, a similarity $\Sigma(x,S)$ and a projected property $\Pi(x,P)$ (ie. "$\Pi(T,P)$" is the answer of the query) by the process with backtracking, according to the following steps:

  1) *Find a* **separate rule** $SepR$ *s.t.* $\mathcal{A} \vdash SepR$, *where* $SepR = \Pi(x,p)$:- $G_{att}(s,p), G_{obj}(x,s)$.

  2) *Take a* **similar attribute** $\Sigma(x,s)$ *s.t.* $\Sigma(x,s) \mid\sim^{\mathcal{A}}_{*} G_{obj}(x,s)$.

  3) *Obtain the* **similar attribute value** S *by the side effect of a proof* $\mathcal{A} \vdash \exists s.\Sigma(T,s)$.

  4) *Retrieve a* **base** B *and obtain the* **projected attribute value** P *by the side effect of a proof* $\mathcal{A} \vdash \exists x,p.(\Sigma(x,S) \wedge \Pi(x,p))$.

  Here, a *separate rule* (*w.r.t.* $\Pi(x,p)$) is a Horn clause in which the head is $\Pi(x,p)$, and any variable of x and any variable of p does not appear in the same conjunct in the body. This system guesses successfully for the examples shown here, though each of them is translated into a set of Horn clauses.

  Significant restrictions are needed on the time complexity of this process. Details of this system will be reported elsewhere.

Acknowledgment

I especially wish to thank Satoshi Sato for his frank comments and challenging problems. I am also grateful to Koichi Furukawa, Hideyuki Nakashima, Natsuki Oka, and five anonymous referees for their constructive comments, Makoto Haraguchi and members of ANR-WG, which was supported by ICOT, for discussions on this topic, Katsumi Inoue and Hitoshi Matsubara for discussions on abduction and CBR respectively, and Kazuhiro Fuchi for giving me the opportunity to do this work.

References

[1] Arima, J.: A logical analysis of relevance in analogy, in *Proc. of Workshop on Algorithmic Learning Theory (ALT'91)*, (1991).

[2] Cox P.T. and Pietrzykowski T.: Causes for events: their computation and applications, in: *Proc. of Eighth International Conference on Automated Deduction*, Lecture Notes in Computer Science **230** (Springer-Verlag, Berlin, 1986) pp. 608-621.

[3] Davies, T. and Russell, S.J.: A logical approach to reasoning by analogy, in *IJCAI-87*, pp.264-270 (1987).

[4] Evans, T,G.: A program for the solution of a class of geometric analogy intelligence test questions, in: M.Minsky (Ed.), *Semantic Information Processing* (MIT Press, Cambridge, MA, 1968).

[5] Falkenhainer, B.: A unified approach to explanation and theory formation, in: J.Shrager & P.Langley (Ed.), *Computational Models of Scientific Discovery and Theory Formation*, (Morgan Kaufmann, San Mareo, CA, 1990).

[6] Gentner, D.: Structure-mapping: Theoretical Framework for Analogy, in: *Cognitive Science*, Vol.7, No.2, pp.155-170 (1983).

[7] Greiner, R.: Learning by understanding analogy, *Artificial Intelligence*, Vol. 35, pp.81-125 (1988).

[8] Haraguchi, M. and Arikawa, S: Reasoning by Analogy as a Partial Identity between Models, in *Proc. of Analogical and Inductive Inference (ALL'86)*, Lecture Notes in Computer Science **265**, (Springer-Verlag, Berlin, 1987) pp 61-87.

[9] Inoue, K.: Linear Resolution for Consequence-Finding, in *Artificial Intelligence* (To appear).

[10] Kedar-Cabelli, S.: Purpose-directed analogy, in *the 7th Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Lawrence Erlbaum Associates, pp.150-159 (1985).

[11] Kling, R.E.: A paradigm for reasoning by analogy, *Artificial Intelligence* **2** (1971).

[12] Muggleton, S. and Buntine, W.: Machine Invention of First-Order Predicates by Inverting Resolution, In: *Proc. of 5th International Conference on Machine Learning*, pp 339-352 (1988).

[13] Peirce C.S.: *Elements of Logic*, in: C. Hartshorne and P. Weiss (eds.), *Collected Papers of Charles Sanders Peirce*, Volume 2 (Harvard University Press, Cambridge, MA, 1932).

[14] Poole D., Goebel R. and Aleliunas R.: Theorist: a logical reasoning system for defaults and diagnosis, in: N. Cercone and G. McCalla (eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge* (Springer-Verlag, New York, 1987) 331–352.

[15] Pople, H.E.Jr.: On the mechanization of abductive logic, in: *Proceedings IJCAI-73*, Stanford, CA (1973) 147–152.

[16] Shapiro, E.Y.,: Inductive Inference of Theories From Facts, TR 192, Yale Univ. Computer Science Dept. (1981).

[17] Shen ,W.:Discovering Regularities from Knowledge Bases, *Proc. of Knowledge Discovery in Databases Workshop 1991*, pp 95-107.

[18] Stickel M.E.: Rationale and methods for abductive reasoning in natural-language interpretation, in: R. Studer (ed.), *Natural Language and Logic, Proceedings of the International Scientific Symposium*, Hamburg, Germany, Lecture Notes in Artificial Intelligence **459** (Springer-Verlag, Berlin, 1990) 233–252.

[19] Schank, R.C.: *Dynamic Memory: A Theory of Reminding and Learning in Computers and People* (Cambridge University Press, London. 1982).

[20] Winston, P.H.: Learning Principles from Precedents and exercises, *Artificial Intelligence*, Vol. 19, No. 3 (1982).

Appendix

Proposition 1.
If $K \vdash^{\mathcal{A}} G$ and $K, G \vdash_{*}^{\mathcal{A}} H$, then $K \vdash^{\mathcal{A}} H$.

Proof of Proposition 1.
For any formula G, if $K \vdash^{\mathcal{A}} G$ and $K, G \vdash_{*}^{\mathcal{A}} H$, we write $K \vdash_{m}^{\mathcal{A}} H$.

i) Subsuming deduction:
if $\mathcal{A}, K \vdash H$ then $K \vdash_{m}^{\mathcal{A}} H$.
(proof)
$K \vdash^{\mathcal{A}} K$. (from subsuming deduction of "$\vdash^{\mathcal{A}}$")
$\mathcal{A}, K \vdash H \Rightarrow K \vdash_{*}^{\mathcal{A}} H$. (from Definition 3 i))
Therefore, $K \vdash_{m}^{\mathcal{A}} H$.

ii) Deductive usefulness:
if $K \vdash_{m}^{\mathcal{A}} H$ and $\mathcal{A}, K, H \vdash L$. then $K \vdash_{m}^{\mathcal{A}} L$.
(proof)
$\mathcal{A}, K, H \vdash L \Leftrightarrow \mathcal{A} \vdash K \wedge H \supset L$
For any formula G s.t. $K \vdash^{\mathcal{A}} G$ and $K, G \vdash_{*}^{\mathcal{A}} H$.

case-i) $\mathcal{A}, K, G \vdash H$ (from $K, G \vdash_{*}^{\mathcal{A}} H$)
From the premises, $\mathcal{A}, K, G \vdash L$.
Therefore, $K, G \vdash_{*}^{\mathcal{A}} L$. (from Definition 3 i))

case-ii) otherwise, for some minimal set of atomic formulas E s.t. $\mathcal{A}, E \vdash K \wedge G$,
$\mathcal{A}, E \vdash K \wedge H$. (from $K, G \vdash_{*}^{\mathcal{A}} H$)

Therefore, $\mathcal{A}, E \vdash L$.
Thus, $K, G \vdash_{*}^{\mathcal{A}} L$.

Thus $K, G \vdash_{m}^{\mathcal{A}} L$.

iii) Consistency:
if $K \vdash_{m}^{\mathcal{A}} H$ and $\mathcal{A} \cup K$ is consistent, then $\mathcal{A} \cup K \cup \{H\}$ is consistent.
(proof)
$\mathcal{A} \cup K$ is consistent.
$\Rightarrow$ $\mathcal{A} \cup K \cup \{G\}$ is consistent. (from $K \vdash^{\mathcal{A}} G$)
$\Rightarrow$ $\mathcal{A} \cup E$ is consistent. (from $K, G \vdash_{*}^{\mathcal{A}} H$)
$\Rightarrow$ $\mathcal{A} \cup K \cup \{H\}$. (because $\mathcal{A}, E \vdash K \wedge H$)

Corollary 1.
If $K \vdash_{*}^{\mathcal{A}} G$, then $K \vdash^{\mathcal{A}} G$.

Proof of Corollary 1.
$K \vdash_{*}^{\mathcal{A}} K$ (from subsuming deduction)
If $K \vdash^{\mathcal{A}} K$ and $K, K \vdash_{*}^{\mathcal{A}} G$, then $K \vdash^{\mathcal{A}} G$. (from Proposition 1)
Therefore,
If $K \vdash_{*}^{\mathcal{A}} G$, then $K \vdash^{\mathcal{A}} G$.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

CONSISTENCY-BASED AND ABDUCTIVE DIAGNOSES AS GENERALISED STABLE MODELS

Chris Preist, Kave Eshghi

Hewlett Packard Laboratories, Filton Road,

Bristol, BS12 6QZ, Great Britain

cwp@hplb.hpl.hp.com
ke@hplb.hpl.hp.com

Abstract

If realistic systems are to be successfully modelled and diagnosed using model-based techniques, a more expressive language than classical logic is required. In this paper, we present a definition of diagnosis which allows the use of a nonmonotonic construct, negation as failure, in the modelling language. This definition is based on the generalised stable model semantics of abduction.

Furthermore, we argue that, if negation as failure is permitted in the modelling language, the distinction between abductive and consistency-based diagnosis is no longer clear. Our definition allows both forms of diagnosis to be expressed in a single framework. It also allows a single inference procedure to perform abductive or consistency-based diagnoses, as appropriate.

1 Introduction

Many different definitions of diagnosis have been used in an attempt to formalise and automate the diagnosis process. In the so-called 'logical' approach, two frameworks, namely the *consistency-based* [Reiter 1987] and *abductive* [Cox and Pietrzykowski 1986], have attracted a lot of attention. Typically, the modelling language used in these frameworks is first order logic (or some subset of it). In this paper we present a unified framework for diagnosis which brings together these two styles of diagnosis, as well as providing a non-monotonic modelling language.

We were primarily motivated by the need to incorporate *negation as failure*, the non-monotonic construct in logic programming, into the modelling language. We first show the need for this construct through some examples, and then argue that the incorporation of negation as failure in the modelling language necessitates the inclusion of both consistency-based and abductive diagnosis within the same framework. We then present our unified framework, which allows negation as failure in the modelling language and naturally incorporates both abductive and consistency-based diagnosis. We then show that in the special cases, our approach reduces to pure consistency and pure abductive diagnosis, i.e. it is a generalisation of both styles.

Our work is similar in spirit to the work of Console and Torasso, [1990],[1991], but goes beyond it in many ways. We will compare our approach to that of Console and Torasso in a later section. Our proposed framework is based on the Generalised Stable Model semantics [Kakas and Mancarella 1990a] of generalised logic programs with abduction, strengthening the link between logic programming and diagnosis first explored in [Eshghi 1990].

2 Consistency-based and abductive approaches to diagnosis

In both consistency-based and abductive approaches, a set of axioms SD (called the *system description*) models the system under investigation, and a set of abnormality assumptions $Ab=\{ab_1, ab_2, \ldots ab_n\}$ represents the possible underlying causes of failure. A set of statements, Obs, represents observations of the behaviour of the system which are to be explained.

In the consistency-based approach, a *diagnosis* is a set of abnormality assumptions, Δ, such that

(1) $SD \cup OBS \cup \Delta \cup \{ \neg ab_k \mid ab_k \in Ab - \Delta \}$ is consistent.

The consistency-based approach focuses primarily on a model of the system's correct behaviour. When the abnormality assumptions relate to the failure of the components of the system, it attempts to find a set of normality and abnormality assumptions which can be assigned to the system's components to give a theory consistent with the observations.

In the abductive approach, a diagnosis is a set of abnormality assumptions, Δ, such that

(2) $SD \cup \Delta \vdash OBS$

$SD \cup \Delta$ is consistent.

The abductive approach primarily models the behaviour of a failing system, by using fault models in the system description, SD. The diagnosis process consists of look-

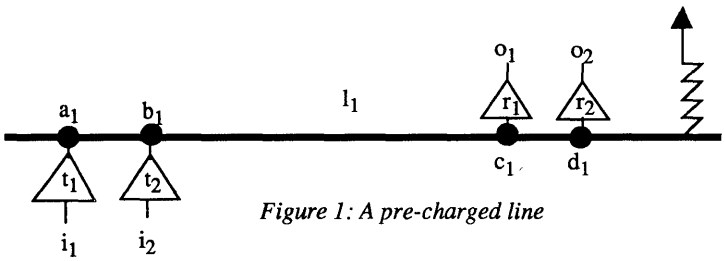


Figure 1: A pre-charged line

ing for a set of abnormality assumptions which, when adopted, will logically predict the observed faulty behaviour given the system description and the context of the observation.

In both approaches, a diagnosis Δ is defined to be *minimal* if there is no other diagnosis, Δ', which is a proper subset of Δ.

3 The Diagnosis Problem

The system description used in model-based diagnosis takes one of two forms. It is either a causal model, or a model consisting of the system's structure and the behaviour of individual components. In general, work on abductive diagnosis has focused on the former, while work on consistency-based diagnosis has focused on the latter.

For the purposes of this paper, we adopt a specification of a diagnosis problem based on those used in [deKleer and Williams 1987] and [Reiter 1987], which uses a component-based approach. However, the results hold equally for a causal model-based approach, and for this reason, we adopt slightly more general language in the definition.

Definition:

A diagnosis problem consists of a triple, <SD, OBS, C> where;

(i) The system description, SD, specifies the behaviour of the system.

(ii) The observation set, OBS, specifies a set of observations of the system as unit clauses.

(iii) C consists of constants, c_i, which represent *causal clusters* within the system.

Causal clusters are groups of causes of abnormal system behaviour which it makes sense to consider together. Each cause, n, within the cluster, c_i, is modelled in SD with two clauses;

$$effects\_of\_cause\_n \leftarrow ab(c_i, n).$$

$$ab(c_i) \leftarrow ab(c_i, n).$$

Furthermore, if so desired, we can define emergent properties of the system which occur when none of the causes in cluster c_i are present, the 'good behaviour model' of this cluster;

$$good\_behaviour\_model \leftarrow \text{not } ab(c_i).$$

In the component-based approach, c_i represents a component, and each cause in cluster c_i represents a possible fault model of the component. Note that the effects of a cause need not be defined deterministically. For example, the 'arbitrary behaviour' mode of a component, proposed in [deKleer and Williams 1989], is consistent with any behaviour of the component, but predicts nothing.

The logical language adopted to represent SD can vary with the definition of diagnosis adopted. In this paper, we focus on two possible languages; classical logic, as adopted by Reiter [1987], and horn clauses with negation as failure, as used in the logic programming community.

4 The need for negation as failure in the system description

The desire to integrate consistency-based and abductive diagnosis was motivated primarily by the need to include negation as failure in our models. The following two examples illustrate this need:

RAM modelling

In order to model the behaviour of a random access memory cell, we needed an axiom that says: the content of a cell at time T is X if X was written to this cell at time T', and no other write operation has been performed between T and T'. The most straightforward way of writing this is as the clause

```
contents(Cell, X, T) ←  written(Cell, X, T'),
                        T'<T,
                        not over-written(Cell,T',T).

over-written(Cell,T',T) ← written(Cell,X,T"),
                          T'<T"<T.
```

This is an instance of the 'frame-problem' being solved through negation-as-failure, as explored in [Shanahan 1989]. If we don't use negation as failure, or some other non-monotonic device, we need to have axioms which allow us to derive ¬over-written(Cell,T',T) for all cells and all time instants, which is very inefficient both in terms of speed of inference and storage required.

Pre-Charged Lines

A common technique used in the computer industry to implement data buses is the pre-charged line. Devices communicate with one another using transmitters and receivers, all connected to a common line whose value floats to 1 when no transmitter is transmitting. (There are n lines for an n-bit wide data bus. Here we concentrate on one line).

Physically, a value of 1 corresponds to high voltage, and a value of 0 to low voltage. In order to give the line its pre-charged value, it is connected to the positive power line by means of a pull-up resistor. Figure 1 gives a schematic of a typical pre-charged line.

To transmit a 0, a transmitter on a line pulls the line to low. Since lines are pre-charged, transmitting a 1 does not involve any action by the transmitter. (Obviously, there is a bus protocol to determine which transmitter, if any, is transmitting at any given time. Here we ignore protocol issues.)

The behaviour of pre-charged lines is best modelled by a default reasoning mechanism. The default value of a line is assumed to be 1 unless it can be proved to be 0. Using negation-as-failure, we could represent this as:

```
received_value(Line,0) ← driven_value(Line,0).
received_value(Line,1) ← not driven_value(Line,0).
driven_value(Line,0)   ← connected(Line,output(X)),
                         trasmits(X,0).
```

The alternative, avoiding the use of negation-as-failure, would be to have an axiom such as:

```
¬driven_value(Line,0) ←
  ∀X(connected(output(X),Line)→ ¬transmits(X,0)).
```

However, in order to prove ∀X(connected(output(X),-Line)→ ¬transmits(X,0)), we would need closure axioms exhaustively enumerating all the transmitters on the line, which would be both cumbersome to write and inefficient to reason with.

Full details of this modelling problem are given in [Eshghi and Preist 1992].

5 Negation As Failure blurs the distinction between abductive and consistency-based diagnosis

Conceptually, the processes behind abductive and consistency-based diagnoses are quite different. In consistency-based diagnosis, one removes normality assumptions until the theory regains consistency. In abductive diagnosis, one adds abnormality assumptions until the specified bad observations are provable in the theory.

However, by moving to a nonmonotonic theory, we can use the same process to perform both styles of diagnosis. We use negation as failure to represent the good behaviour of a cluster as its default behaviour;

```
behaviour ← not ab(c)
```

In a situation where the system is malfunctioning, and in the standard consistency-based approach we would derive an inconsistency by adding normality assumptions, we would get an inconsistency without adding any assumptions. This is because the negation as failure results in clusters defaulting to their 'good' behaviour model. Furthermore, the theory can be restored to consistency by adding abnormality assumptions, as in abduction, rather than by removing normality assumption as in the standard consistency-based approach.

It is exactly because of this effect that an abductive framework can be used to represent both consistency-based and abductive diagnoses. A similar approach to representing a component's good behaviour as its default behaviour was introduced in the context of the Nonmonotonic ATMS, in [Dressler 1990].

If we are to use negation as failure in the system description, as we argued we need to do in many instances, it is necessary to integrate abductive and consistency-based approaches. This is because, in a logic with negation as failure, consistency-based and abductive diagnoses are the dual of each other. By passing through a negation, you pass from a consistency-based problem to an abductive problem, or vice-versa. To see this, let us consider some simple examples;

a) Consistency-Based diagnosis

```
SD:  obs ← not g
     g ← ab(c)
OBS: ¬obs
```

In a consistency-based diagnosis, we attempt to restore consistency by making assumptions so as to *'not-prove'* a certain proposition which contradicts with the integrity constraints. In the case of the above example, we wish to not-prove obs. However, to do this, we must *prove* the negated goal, g. Hence we want an abductive diagnosis of the observation, g.

b) Abductive diagnosis

```
SD:  obs ← not g
     g ← ab(c)
OBS: obs
```

In an abductive diagnosis, we wish to make assumptions so as to *prove* a certain proposition which is required to be true by the integrity constraints. In the above example, we wish to prove obs. However, to do this, we must fail to prove the negated goal, g. Hence, we want a consistency-based diagnosis for the observation ¬g.

Thus a diagnostic problem of one sort may have a diagnostic problem of the other sort embedded in it. So, when the modelling language includes negation as failure, abductive and consistency-based diagnosis cannot

be considered in isolation from each other. It is this that led us to formulate this integration.

6 The Generalised Stable Model Semantics for Abduction

Various semantics have been proposed for abduction, both formally and informally. Originally, an abductive explanation for an observation was informally defined as a set of assumables which, when added to a theory, allowed proof of the observation. This was then formalised to give a metalevel definition of abduction in [Eshghi and Kowalski 1989].

Console *et al.* [1990] have used the completion semantics to give a semantics to abduction in horn clause theories. Recently, they have extended it to cover hierarchical logic programs [Console *et al.* 1991].

The semantics of abduction which we have chosen to use, however, is that provided by Kakas and Mancarella [1990a]. By extending the stable model semantics of logic programs [Gelfond and Lifschitz 1988], they give a semantics for abduction which holds for arbitrary general logic programs with integrity constraints.

Here, we briefly recall their definitions;

Definition 1

An abductive framework is a triple <P,A,IC> where

1) P is a set of clauses of the form $H \leftarrow L_1,..,L_k$ k0 where H is an atom and L_i is a literal.

2) A is a set of predicate symbols, the abducible predicates. The abducibles, Ab, are then all ground atoms with predicate symbols in A.

3) IC, the integrity constraints, is a set of closed formulae.

Hence an abductive framework extends a logic program to include integrity constraints and abducibles. The semantics of this framework is based on the stable model semantics for logic programs;

Definition 2

Let P be a logic program, and M a set of atoms from the Herbrand base. Define P_M to be the set of ground horn clauses formed by taking ground(P), in clausal form, and deleting;

(i) each clause that has a negative literal $\neg l$ in its body, and $l \in M$.

(ii) all negative literals $\neg l$ in the body of clauses, where $l \notin M$.

M is a *stable model* for P if M is the minimal model of P_M.

This definition is extended to give a semantics to abductive frameworks.

Definition 3

Let <P,A,IC> be an abductive framework, and $\Delta \subseteq$ atoms(A) be a set of abducibles. Then the set $M(\Delta)$ of ground atoms is a *generalised stable model* (GSM) for <P,A,IC> iff it is a stable model for the logic program $P \cup \Delta$, it is a model for the integrity constraints IC, and $\Delta = A \cap M(\Delta)$.

The above definition is an extension of that in [Kakas and Mancarella 1990a] to allow abducibles to appear in the head of a clause. As a result of this, the set of abducibles chosen as generators can be smaller than Δ, the set of abducibles true in the generalised stable model.

A unit clause, q, representing an observation, has an abductive explanation with hypothesis set Δ if there exists a generalised stable model, $M(\Delta)$, in which q is true.

Equivalently, we can say that q has an abductive explanation, Δ, within the abductive framework <P,A,IC> if the abductive framework <P,A,IC+q> has a generalised stable model $M(\Delta)$. Having q in the integrity constraints imposes the condition that q must be true in the generalised stable model, and hence must follow from the logic program together with the set of abducibles chosen.

7 Generalised Stable Models and Diagnosis

The generalised stable model semantics for abduction can be applied to diagnosis by mapping a diagnosis problem, <SD, OBS, C>, with multiple observations, onto an abductive framework as follows;

- Represent the system description, SD, as a logic program with integrity constraints, <P,IC>. The integrity constraints will usually contain sentences stating that observation points cannot take multiple values at a given time.
- Let the abducibles represent the causes within the clusters, $\{ab(c_i.n_j) \mid c_i \in C\}$, hence $A = \{ab(X,N)\}$.

Intuitively, given an observation set OBS, represented by a set of unit clauses, we have a choice of how to use it. We either wish to predict it, giving an abductive diagnosis, or make assumptions to restore the theory to consistency, giving a consistency-based diagnosis. By adding OBS to the integrity constraints, only models in which the observations are true, and hence explained by the system description together with selected abducibles, are legal generalised stable models. Hence we get an abductive diagnosis. If, instead, we add OBS to the logic program representing the system description, then a set of assumptions can only be made if they are consistent with the observations; i.e. the observations, system description and assumptions cannot derive anything which violates the integrity constraints. This will give us consistency-based diagnoses. Furthermore,

we can partition OBS into two sets, and predict some observations, OBS_p, while maintaining consistency with others, OBS_c. We do this by placing OBS_p in the integrity constraints, and OBS_c in the logic program.

This allows us to give a definition of unified diagnosis as follows;

Definition 4

Let $<SD,OBS_p,OBS_c,C>$ be a diagnosis problem, where;

SD is a logic program with integrity constraints, <P,IC>.

OBS_p is the set of observations to be predicted by diagnoses.

OBS_c is the set of observations which diagnoses need to be consistent with.

C is the set of causal clusters in the system.

Then;

Δ is a GSM-diagnosis of $<SD,OBS_p,OBS_c,C>$ iff there is a generalised stable model, $M(\Delta)$, of the abductive framework $<P \cup OBS_c,A,IC \cup OBS_p>$.

where A = {ab(C,N)} represents the set of possible root causes of misbehaviour in SD.

To demonstrate this, we consider a simple example from the medical domain, that of *pericardial tamponade*. The heart consists of two parts, the *myocardium* is the muscle which beats, while the *pericardium* is the protective sac which surrounds this muscle. If this sac is pierced, instantaneous pain occurs, which can subside fairly quickly. However, blood slowly flows into the pericardium over a period of time, increasing the pressure on the myocardium. Later, the myocardium will become so compressed that blood does not flow round the arteries, even though the myocardium itself is functioning perfectly.

The model of this phenomenon is given below. For simplicity, we treat time discretely, in units of hours.

```
pulse_ok(T)    ←   normal_cardiac_contraction(T),
                   not heart_compressed(T).

no_pulse(T)    ←   heart_compressed(T).

heart_compressed(T) ←  ab(pericardium,pierced(T')),
                       T' < T - 10.
normal_cardiac_contraction(T) ←
                   not ab(myocardium,failure(T')),
                   T'<T.

bad_ecg(T)     ←   ab(myocardium, failure(T)).
```

We give the pericardium the possible failure cause 'pierced' at a given time, while the myocardium simply suffers a 'failure' of some sort. The latter is consistent with any behaviour of the myocardium, but only predicts a bad ecg trace.

The above clauses form the logic program part of SD.In addition, we need the integrity constraints, IC. These simply state which observations conflict with each other;

```
¬(pulse_ok(T) & no_pulse(T)).
¬(ecg_bad(T) & ecg_good(T)).
```

Assume we have the observation, no_pulse(12). Let us consider the generalised stable models of <P,A,IC>.

If we place the observation in the logic program as a unit clause, any set of abducibles can be assumed as long as they do not violate the integrity constraints - i.e. they must not generate a stable model in which pulse_ok(12) is true. If we assume nothing, the resulting stable model contains pulse_ok(12) as true, resulting in a conflict. There are two possible (minimal) ways to restore consistency. We can assume ab(myocardium,failure(10))[1], and cease to contain normal_cardiac_contraction(12) in the stable model. Alternatively, we assume ab(pericardium,pierced(2))[1], which predicts heart compression at time 12. The resulting stable model will therefore not contain pulse_ok(12), and so be a legitimate generalised stable model of $<P \cup \{no\_pulse(12)\},A,IC>$.

If, instead, we place the observation in the integrity constraints, IC, we are restricted to stable models which contain no_pulse(12). In this case, only by assuming ab(pericardium,pierced(2)) do we generate a stable model which contains no_pulse(12). As this also satisfies IC, it is a legitimate GSM for $<P,A,IC \cup \{no\_pulse(12)\}>$.

Hence, by making a choice of where to place the observation, we can generate either consistency-based or abductive diagnoses. Furthermore, if we have a second observation, ecg_good(12), we can choose to treat it in a different way from the first. Let OBS_p = {no_pulse(12)} and OBS_c = {ecg_good(12)}. In this case, the only (minimal) GSM of $<P \cup OBS_c,A,IC \cup OBS_p>$ is that generated by ab(pericardium, pierced(2)). However, if we swap OBS_p and OBS_c, the only (minimal) GSM is that generated by ab(myocardium, failure(10)).

Note how the model uses negation-as-failure to handle the frame problem. If we used classical negation instead, it would be necessary to have extra clauses to predict not_heart_compressed at all relevant times, resulting in a larger, less understandable, and less efficient model.

8 Abductive and consistency-based diagnosis as special cases

If we restrict our attention to the traditional definitions of diagnosis, we can show that our definition is equivalent to these under certain conditions.

[1] Or, of course, at any other appropriate time instant.

8.1 Abductive Diagnoses as Generalised Stable Models

If all the observations are to be predicted in the abductive sense, and the system description contains only horn clauses, our definition of diagnosis reduces to the standard definition of abduction given in section 1. This is achieved as follows:

Given an abductive diagnosis problem $<SD,OBS_p,C>$, where SD is a horn-clause theory, divide the system description into a set of definite clauses, P, and a set of denials, D. Let A be the set of abducibles.

It is easy to show that abductive diagnoses of SD according to formula (2) correspond to generalised stable models of the framework $<P,A,IC \cup OBS_p>$.

8.2 Consistency-Based Diagnoses as Generalised Stable Models

For a certain class of theories, namely *almost-horn* theories, we show that our definition of diagnosis is equivalent to the traditional definition of consistency-based diagnosis given in [Reiter 1987]. An almost-horn theory is a theory in which negation is used only to represent the negation of certain predicates. In the context of our theorem, these correspond to the abnormality assumptions.

Definition 5

A clause is said to be *almost-Horn with respect to A*, if , when in disjunctive normal form, it contains at most one positive literal with a predicate symbol not in A.

Theorem

Let $<SD, OBS_c,C>$ be a consistency-based diagnosis problem, with SD a theory which is almost-horn with respect to A={ab}.

Then define the logic program with integrity constraints, SD'=<P,IC>, as follows;

Let $a_i \in$ atoms(A), and p, $q_i \notin$ atoms(A).

1. For every clause of the form
$p \leftarrow \neg a_1,\neg a_2...\neg a_k,a_{k+1},..,a_m,q_1,q_2,..,q_n$ in SD, there is a program clause
$p \leftarrow$ not a_1,not a_2...not $a_k,a_{k+1},..,a_m,q1,q_2,..,q_n$ in P.

2. For every clause of the form
$a_1 \vee a_2...\vee a_k \vee \neg a_{k+1} \vee ... \vee \neg a_m \neg q1 \vee \neg q_2 \vee .. \vee \neg q_n$ in SD there is an identical clause in IC.

Then;

D is a consistency-based diagnosis of $<SD, OBS_c,C>$ according to formula (1)

$\Leftrightarrow$ D is a GSM-diagnosis of $<SD', \emptyset, OBS_c,C>$

The proof of this theorem is available in an extended version of this paper, available from the authors.

This theorem shows that, if negation is used only to represent the normality assumptions in the system, ¬ab, then the nonmonotonic definition of diagnosis given by us is equivalent to the monotonic definition given in [Reiter 1987]. However, if negation is used elsewhere in the theory, the two definitions diverge. The classical consistency-based definition requires explicit representation of all negative information. The GSM-diagnosis, however, will make the closed-world assumption, and assume information is false unless it can be proved otherwise.

9 Comparison with Console & Torasso [2]

Console & Torasso have defined a framework for a general abduction problem. This framework allows a spectrum of diagnosis styles to be represented within it, including the pure consistency-based and abductive styles described above.

They divide the observations into two sets. One set, OBS_a, is to be explained by the assumptions, while the other set, OBS_c, must be consistent with the assumptions. They then define two sets;

$\Psi^+ = OBS_a$.

$\Psi^- = \{ \neg f(x) \mid f(y) \in OBS_c, x \neq y \}$

A diagnosis is then a set of abducibles which, when added to the theory, allows prediction of all observations in Ψ^+, and is consistent with the negative literals in Ψ^-.

Our definition is more powerful in several ways.

- It extends the definition of Console and Torasso from horn-clause theories to general logic programs with integrity constraints. This gives a sophisticated and expressive language for modelling, which includes negation as failure.
- The inclusion of the consistency-based observations in the object level, rather than their negations in the integrity constraints, means that these can be used easily during inference. This can reduce the time to find a conflict, by using 'backwards simulation' of components. In some cases, such as the example documented in [van Soest *et al.* 1990] , certain diagnoses cannot be found without access to the observations in this way.
- Within this framework, it is possible to define minimal diagnoses model-theoretically. We will expand on this in section 10.

Placing the consistency-based observations at the object level potentially gives us more efficient inference. However, to do this in the context of joint diagnoses can lead to problems.

It may be possible to conclude that an abductive obser-

vation is true, based on the adding of a consistency-based observation to the theory alone;

SD: obs1 → obs2

OBS_a: obs2

OBS_c: obs1

By adding obs1 to the system description, we can conclude that obs2 is true. Whether this is legitimate depends on how we interpret the consistency-based observations. If we consider them true, but not necessarily explainable, then this is legitimate. This is the case in Reiter's formalisation of diagnosis, and also in the case of the setting factors of Reggia *et al.* [1983]. However, if we consider them not necessarily true, merely not false, then this is unacceptable. In such circumstances, it is necessary to restrict the model so that consistency-based observations do not appear in the body of clauses, or use the approach proposed by Console and Torasso.

10 Minimality

We now focus attention on component-based diagnosis, and consider the problem of minimal diagnoses. We wish to restrict our attention to those diagnoses which contain a minimal number of failing components.

To do this, we introduce minimal generalised stable models;

Definition:

A general stable model, $M(\Delta)$, for an abductive framework,<P,A,IC>, is *minimal* if there is no other GSM, $M(\Delta')$, such that $\Delta' \subset \Delta$.

Hence, a minimal general stable model contains a minimal set of assumptions which allow the consequences of the logic program P to satisfy the integrity constraints, IC. Note that, because abductive frameworks are non-monotonic, this does not imply that any superset of Δ, Φ, will have a GSM, $M(\Phi)$.

If, in our diagnosis framework, we have a 1-1 correspondence between a hypothesised failed component and an abducible being assumed in the abductive framework, then minimal general stable models will correspond to minimal diagnoses. To do this, we must impose two restrictions on the relationship between the frameworks;

(i) There must be no abducible representing the correct behaviour of a component. This must instead be a default behaviour which is used in the absence of abducibles referring to the faulty behaviour of a component.

(ii) It must be illegal to make more than one assumption about a component's behaviour at a time.

Note that the second condition does not force fault modes to be mutually exclusive in real-life, merely that they must be mutually exclusive logically. This can easily be achieved by adding an integrity constraint forbidding a component to have two modes;

$$\text{false} \leftarrow ab(c_i,m_{j1}),\ ab(c_i,m_{j2}),\ m_{j1} \neq m_{j2}.$$

The framework provided by Console and Torasso satisfies the second of these conditions, but not the first. Because they work in a monotonic framework, it is not possible to represent the correct behaviour of a component as the default behaviour; instead, it must be explicitly assumed that a component behaves correctly.

As a result of this, they must specify a semantic minimisation criterion; a diagnosis is minimal if it contains a minimal set of abducibles corresponding to faulty behaviour. We, however, can specify a model theoretic criterion;

A diagnosis, Δ, is *minimal* if its corresponding GSM, $M(\Delta)$, is a minimal GSM.

11 Calculating Diagnoses

By providing a uniform model-theoretic framework for consistency-based, abductive and joint diagnoses, we have also provided a method for a uniform implementation. We simply need an algorithm for generating the minimal generalised stable models of an abductive framework, and we can use this for performing a variety of diagnosis tasks.

Much work has been carried out on the generation of stable models, and several efficient algorithms exist. However, as general stable models are a newer innovation, these results have yet to be fully exploited and extended to the GSM case. Currently, the state of the art in GSM generation is provided by Satoh and Iwayama [1991]. This work, however, has the drawback that it does not produce minimal GSMs.

Traditionally, in the abductive community, top-down algorithms have been used which tend to generate minimal solutions, as they avoid making irrelevant assumptions. (e.g. [Cox and Pietrzykowski 1986] [Kakas and Mancarella 1990b]) However, non-minimal abductive diagnoses are still acceptable in the model-theoretic semantics, and can be generated by the algorithms. Similarly, in the diagnosis community, generation of minimal diagnoses has tended to be a consequence of the algorithm selected (e.g. the ATMS in [deKleer and Williams 1987]) rather than a model-theoretic restriction.

However, Eshghi [1990] proposes an alternative approach. He generates a theory in which minimal diagnoses correspond exactly to the stable models of the theory. This means that non-minimal diagnoses are excluded by the semantics, rather than the algorithm. By extending these results beyond the almost-horn case, we are able to transform an abductive framework into a

logic program. The stable models of this logic program correspond exactly to the minimal generalised stable models of the abductive framework. This means that minimality is brought into the theory as a necessary property of each solution, rather than being a selection criterion between solutions. This work is currently in progress.

As a result of this, a wider variety of literature can be used to select appropriate and efficient algorithms, rather than being restricted to algorithms which have been developed specifically for the task of diagnosis.

12 Conclusions

By moving to a nonmonotonic logical framework, it is possible to bring abductive and consistency-based diagnosis together, and use the same inference method to perform both. We have done this by using generalised stable models to provide the semantics, which provides us with a rich and expressive modelling language. It also gives a link between diagnosis and logic programming, allowing application of theoretical and practical logic programming results to the domain of diagnosis.

Acnowledgements

Thanks to Bruno Bertolino and Enrico Coiera for their assistance.

References

[Console *et al.* 1990] L.Console, D. Theseider Dupre & P.Torasso. *A Completion Semantics for Object-level Abduction.* Proc. AAAI Symposium in Automated Abduction, 1990.

[Console *et al.* 1991] L.Console, D. Theseider Dupre & P.Torasso. *On the relationship between abduction and deduction.* Journal of Logic and Computation, 2(5), Sept. 1991.

[Console and Torasso 1990] L.Console & P.Torasso. *Integrating Models of the Correct Behaviour into Abductive Diagnosis.* Proceedings of the 9th European Conference on Artificial Intelligence, 1990.

[Console and Torasso 1991] L.Console & P.Torasso. *A Spectrum of Logical Definitions of Model-Based Diagnosis.* University of Torino Technical Report, 1991.

[Cox and Pietrzykowski 1986] P.T. Cox & T. Pietrzykowski. *Causes for Events: their Computation and Application.* Proc. 8th conference on Computer Aided Design and Engineering, 1986.

[Davis 1984] R. Davis. *Diagnostic Reasoning based on Structure and Behaviour.* Artificial Intelligence 24:347-410, 1984.

[deKleer *et al.*1990] J. deKleer, A. Mackworth & R Reiter. *Characterizing Diagnoses.* Proceedings of the Eighth National US Conference on Artificial Intelligence, Boston 1990.

[deKleer and Williams 1987] J. deKleer & B. Williams. *Diagnosing Multiple Faults.* Artificial Intelligence 32:97-130, 1987.

[deKleer and Williams 1989] J. deKleer & B. Williams. *Diagnosis with Behavioural Modes.* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit 1989.

[Dressler 1990] O.Dressler. *Computing Diagnoses as Coherent Assumption Sets.* Proceedings of the First International Workshop on Principles of Diagnosis, Menlo Park 1990

[Eshghi 1990] K. Eshghi. *Diagnoses as Stable Models.* Proceedings of the First International Workshop on Principles of Diagnosis, Menlo Park 1990

[Eshghi and Kowalski 1989] K. Eshghi & R. Kowalski. *Abduction compared with Negation as Failure.* Proceedings of the 6th Int. Conf. on Logic Programming, Lisbon 1989, pp234-254.

[Eshghi and Preist 1992] K. Eshghi and C. Preist. *The Cachebus Experiment: Model Based Diagnosis applied to a Real Problem* in Industrial Applications of Knowledge-Based Diagnosis, ed Guida and Stefanini, Elsevier 1992.

[Gelfond and Lifshitz 1988] M. Gelfond & V. Lifshitz. *The Stable Model Semantics for Logic Programming.* Proceedings of the Fifth International Conference on Logic Programming, 1988.

[Kakas and Mancarella 1990a] A. Kakas & P. Mancerella. *Generalised Stable Models: A Semantics for Abduction.* Proceedings of the 9th European Conference on Artificial Intelligence, 1990.

[Kakas and Mancarella 1990b] A. Kakas & P. Mancarella. *On the relation between Truth Maintenance and Abduction.* Proceedings of PRICAI, 1990.

[Reiter 1987] R. Reiter. *A theory of diagnosis from first principles,* Artificial Intelligence Journal 32, 1987

[Reggia *et al.* 1983] J.A. Reggia, D.S. Nau & P.Y. Wang. *Diagnostic Expert Systems based on a Set Covering Model.* Int. J. of Man-Machine Studies 19, p437-460. (1983)

[Satoh and Iwayama 1991] K. Satoh & N. Iwayama. *Computing Abduction by using the TMS.* Proceedings of the Eighth International Conference on Logic Programming, 1991.

[Shanahan 1989] M. Shanahan. *Prediction is Deduction but Explanation is Abduction.* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit 1989.

[vanSoest *et al.* 1990] D.C. van Soest, R.R. Bakker, F. van Raalte & N.J.I. Mars. *Improving effectiveness of model-based diagnosis,* Proc. 10th international workshop on expert systems and their applications, Avignon 1990.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Forward-Chaining Hypothetical Reasoner Based on Upside-Down Meta-Interpretation

Yoshihiko Ohta **Katsumi Inoue**
Institute for New Generation Computer Technology
Mita Kokusai Bldg. 21F, 1-4-28 Mita, Minato-ku, Tokyo 108, Japan
{ohta, inoue}@icot.or.jp

Abstract

A forward-chaining hypothetical reasoner with the assumption-based truth maintenance system (ATMS) has some advantages such as avoiding repeated proofs. However, it may prove subgoals unrelated to proofs of the given goal. To simulate top-down reasoning on bottom-up reasoners, we can apply the upside-down meta-interpretation method to hypothetical reasoning. Unfortunately, when programs include negative clauses, it does not achieve speedups because checking the consistency of solutions by negative clauses should be globally evaluated. This paper describes a new transformation algorithm of programs for efficient forward-chaining hypothetical reasoning. In the transformation algorithm, logical dependencies between a goal and negative clauses are analyzed to find irrelevant negative clauses, so that the forward-chaining hypothetical reasoners based on the upside-down meta-interpretation can restrict consistency checking of negative clauses to those relevant clauses. The transformed program has been evaluated with a logic circuit design problem.

1 Introduction

Hypothetical reasoning [Inoue 88] is a technique for proving the given goal from axioms together with a set of hypotheses that do not contradict with the axioms. Hypothetical reasoning is related to abductive reasoning and default reasoning.

A forward-chaining hypothetical reasoner can be constructed by simply combining a bottom-up reasoner with the assumption-based truth maintenance system (ATMS) [de Kleer 86-1] (for example [Flann *et al.* 87, Junker 88]). We have implemented a forward-chaining hypothetical reasoner [Ohta and Inoue 90], called APRICOT/0, which consists of the RETE-based inference engine [Forgy 82] and the ATMS. With this architecture, we can reduce the total cost of the label computations of the ATMS by giving intermediate justifications to the ATMS at two-input nodes in the RETE-like networks. On the other hand, hypothetical reasoning based on top-down reasoning has been proposed in [Poole *et al.* 87, Poole 91]. Compared with top-down (backward-chaining) hypothetical reasoning, bottom-up (forward-chaining) hypothetical reasoning has the advantage of avoiding duplicate proofs of repeated subgoals and duplicate proofs among different contexts. Bottom-up reasoning, however, has the disadvantage of proving unnecessary subgoals that are unrelated to the proofs of the goal.

To avoid the disadvantage of bottom-up reasoning, Magic Set method [Bancilhon *et al.* 86] and Alexander method [Rohmer *et al.* 86] have been proposed for deductive database systems. Recently, it is shown that Magic Set and Alexander methods are interpreted as specializations of the upside-down meta-interpretation [Bry 90]. The upside-down meta-interpretation has been extended to abduction and deduction with non-Horn clauses in [Stickel 91]. His abduction, however, does not require the consistency of solutions.

Since the consistency requirement is crucial for some applications, we would like to make programs include negative clauses for our hypothetical reasoning. When programs include negative clauses, however, the upside-down meta-interpretation method does not achieve speedups because checking the consistency of solutions by negative clauses should be globally evaluated.

We present a new transformation algorithm of programs for efficient forward-chaining hypothetical reasoning based on the upside-down meta-interpretation. In the transformation algorithm, logical dependencies between a goal and negative clauses are analyzed to find irrelevant negative clauses, so that the forward-chaining hypothetical reasoners based on the upside-down meta-interpretation can restrict consistency checking of negative clauses to those relevant clauses. The transformed program has been evaluated with a logic circuit design problem.

In Section 2, our hypothetical reasoning is defined with the default proofs [Reiter 80]. In Section 3, the outline of the ATMS is sketched. Section 4 shows the basic algorithm for hypothetical reasoning based on the bottom-up reasoner MGTP [Fujita and Hasegawa 91] together with

the ATMS. Section 5 presents two transformation algorithms based on the upside-down meta-interpretation. One is a simple transformation algorithm, the other is the transformation algorithm with the abstracted dependency analysis. We have implemented the hypothetical reasoner and these program transformation systems, and Section 6 shows the result of an experiment for the evaluation of the transformed programs. In Section 7, related works are considered.

2 Problem Definition

In this section, we define our hypothetical reasoning based on a subset of normal default theories [Reiter 80]. A normal default theory (D, W) and a goal G are given as follows:

- W: a set of Horn clauses.

  A *Horn clause* is represented in an implicational form,

  $$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta \tag{1}$$

  or

  $$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \bot. \tag{2}$$

  Here, α_i $(1 \leq i \leq n; n \geq 0)$ and β are atomic formulas, and $\bot$ designates falsity. Function symbols are restricted to 0-ary function symbols. All variables in a clause are assumed to be universally quantified in front of the clause. Each Horn clause has to be *range-restricted*, that is, all variables in the consequent β have to appear in the antecedent $\alpha_1 \wedge \cdots \wedge \alpha_n$. A Horn clause of the form (2) is called a *negative clause.*

- D: a set of normal defaults.

  A *normal default* is an inference rule,

  $$\frac{\alpha : \beta}{\beta}, \tag{3}$$

  where α, called the *prerequisite* of the normal default, is restricted to a conjunction $\alpha_1 \wedge \cdots \wedge \alpha_n$ of atomic formulas and β, called its *consequent*, is restricted to an atomic formula. Function symbols are restricted to 0-ary function symbols. All variables in the consequent β have to appear in the prerequisite α. A normal default with free variables is identified with the set of its ground instances. The normal default can be read as " if α and it is consistent to assume β, then infer β".

- goal G: a conjunction of atomic formulas.

  All variables in G are assumed to be existentially quantified.

Let Δ be the set of all ground instances of the normal defaults of D. A *default proof* [Reiter 80] of G with respect to (D, W) is a sequence $\Delta_0, \cdots, \Delta_k$ of subsets of Δ if and only if

1. $W \cup CONSEQUENTS(\Delta_0) \vdash G$,
2. for $1 \leq i \leq k$,
   $W \cup CONSEQUENTS(\Delta_i) \vdash$
   $PREREQUISITES(\Delta_{i-1})$,
3. $\Delta_k = \emptyset$,
4. $W \cup \bigcup_{i=0}^{k} CONSEQUENTS(\Delta_i)$ is consistent,

where

$$PREREQUISITES(\Delta_{i-1}) \equiv \bigwedge \alpha$$

for $(\alpha : \beta/\beta) \in \Delta_{i-1}$ and

$$CONSEQUENTS(\Delta_i) \equiv \{\beta \mid (\alpha : \beta/\beta) \in \Delta_i\}.$$

A ground instance $G\theta$ of the goal G is an *answer* to G from (D, W) if

$$W \cup \bigcup_{i=0}^{k} CONSEQUENTS(\Delta_i) \models G\theta,$$

where the sequence $\Delta_0, \cdots, \Delta_k$ is a default proof of G with respect to (D, W). If $G\theta$ is an answer to G from (D, W), θ is an *answer substitution* for G from (D, W). A *support* for an answer $G\theta$ from (D, W) is $\bigcup_{i=0}^{k} CONSEQUENTS(\Delta_i)$, where the sequence $\Delta_0, \cdots, \Delta_k$ is a default proof of $G\theta$ with respect to (D, W). For an answer $G\theta$ from (D, W), the *minimal supports* for $G\theta$ from (D, W), written as $MS(G\theta)$, is the set of minimal elements in all supports for $G\theta$ from (D, W). The *solution* to G from (D, W) is the set of all pairs $\langle G\theta, MS(G\theta)\rangle$, where $G\theta$ is an answer to G from (D, W) and $MS(G\theta)$ is the minimal supports for $G\theta$. The task of our hypothetical reasoning is defined to find the solution to a given goal from a given normal default theory.

3 ATMS

The ATMS [de Kleer 86-1] is used as one component of our hypothetical reasoner. The following is the outline of the ATMS.

In the ATMS, a ground atomic formula is called a *datum.* For some datum N, Γ_N designates an *assumption.* The ATMS treats both $\bot$ and Γ_N as special data. The ATMS represents each datum as an *ATMS node*:

$$\langle datum, label, justifications\rangle.$$

Justifications correspond to ground Horn clauses and are incrementally input to the ATMS. Each justification is denoted by:

$$N_1, \cdots, N_n \Rightarrow N,$$

where N_i and N are data. Each datum N_i is called an antecedent, and the datum N is called a consequent. In the slot *justifications*, the ATMS records the set of antecedents of justifications whose consequents correspond to the *datum*.

Let H be a current set of assumptions. An assumption set $E \subseteq H$ is called an *environment*. When we denote an environment by a set of assumptions, each assumption Γ_N is written as N by omitting the letter Γ. Let J be a current set of justifications. An environment E is called *nogood* if $J \cup E$ derives $\bot$. The *label* of the datum N is the set of environments $\{E_1, \cdots, E_j, \cdots, E_m\}$ that satisfies the following four properties [de Kleer 86-1]:

1. N holds in each E_j (soundness),
2. every environment in which N holds is a superset of some E_j (completeness),
3. each E_j is not nogood (consistency),
4. no E_j is a subset of any other (minimality).

If the label of a datum is not empty, the datum is *believed*; otherwise it is not believed. A basic algorithm to compute labels [de Kleer 86-1] is as follows. When a justification is incrementally input to the ATMS, the ATMS updates the labels relevant to the justification in the following procedure.

Step 1: Let L be the current label of the consequent N of the justification and L_i be the current label of the i-th antecedent N_i of the justification. Set $L' = L \cup \{ x \mid x = \bigcup_{i=1}^{n} E_i, \ where \ E_i \in L_i \}$.

Step 2: Let L'' be the set obtained by removing nogoods and subsumed environments from L'. Set the new label of N to L''.

Step 3: Finish this updating if L is equal to the new label.

Step 4: If N is $\bot$, then remove all new nogoods from labels of all data other than $\bot$.

Step 5: Update labels of the consequents of the recorded justifications which contain N as their antecedents.

4 Hypothetical Reasoner with ATMS and MGTP

The MGTP [Fujita and Hasegawa 91] is a model generation theorem prover for checking the unsatisfiability of a first-order theory P. Each clause in P is denoted by:

$$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta_1 \vee \cdots \vee \beta_m,$$

where $\alpha_i (1 \leq i \leq n; n \geq 0)$ and $\beta_j (1 \leq j \leq m; m \geq 0)$ are atomic formulas and all variables in $\beta_1 \vee \cdots \vee \beta_m$ have to appear in $\alpha_1 \wedge \cdots \wedge \alpha_n$. Each clause in P is translated into a KL1 [Ueda and Chikayama 90] clause. Then, model candidates are generated from the set of KL1 clauses. The MGTP works as a bottom-up reasoner on the distributed-memory multiprocessor called Multi-PSI.

As shown in Figure 1, we can construct a hypothetical reasoner by combining the MGTP with the ATMS. The normal default theory (D, W) is translated into a program P,

$$P \equiv \{ \ \alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \textbf{assume}(\beta) \mid (\alpha_1 \wedge \cdots \wedge \alpha_n : \beta / \beta) \in D \ \} \cup W,$$

where **assume** is a metapredicate not appearing anywhere in D and W.

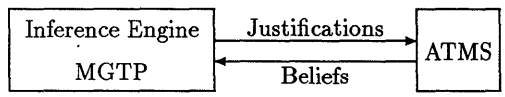


Figure 1: **Forward-Chaining Hypothetical Reasoner with ATMS and MGTP**

```
procedure R(G, P) :
begin
  B_0 := ∅;
  J_0 := { (⇒ β) | (→ β) ∈ P }
       ∪ { (Γ_β ⇒ β) | (→ assume(β)) ∈ P };
  s := 0;
  while J_s ≠ ∅ do
    begin
      s := s + 1;
      B_s := UpdateLabels(J_{s-1}, ATMS);
      J_s := GenerateJustifications(B_s, P, B_{s-1})
    end;
  Solution := ∅;
  for each θ such that Gθ ∈ B_s do
    begin
      L_{Gθ} := GetLabel(Gθ, ATMS);
      Solution := Solution ∪ {⟨Gθ, L_{Gθ}⟩}
    end;
  return Solution
end.
```

Figure 2: **Reasoning Algorithm with ATMS and MGTP**

The reasoning procedure $R(G, P)$ for the MGTP with the ATMS is shown in Figure 2. The reasoning proce-

dure consists of the part for *UpdateLabels* – *GenerateJustifications* cycles and the part for constructing the solution. The *UpdateLabels* – *GenerateJustifications* cycles are repeated while J_s is not empty. The ATMS updates the labels related to a justification set J_{s-1} given by the MGTP. The ATMS returns the set B_s of all the data whose labels are not empty after the ATMS has updated labels with J_{s-1}. The procedure $UpdateLabels(J_{s-1}, ATMS)$ returns a believed data set B_s. The MGTP generates each set J_s of justifications by matching elements of B_s with the antecedent of every clause related to new believed data. The procedure $GenerateJustifications(B_s, P, B_{s-1})$ returns a new justification set J_s. If any element in $(B_s \setminus B_{s-1})$ can match an element of the antecedent of any $(\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow X)$ in P and there exists a ground substitution σ for all α_i such that $\alpha_i\sigma \in B_s$, then J_s is as follows.

- $(\alpha_1\sigma, \cdots, \alpha_n\sigma, \Gamma_{\beta\sigma} \Rightarrow \beta\sigma) \in J_s$ if $X = \mathbf{assume}(\beta)$.
- $(\alpha_1\sigma, \cdots, \alpha_n\sigma \Rightarrow \beta\sigma) \in J_s$ if $X = \beta$.
- $(\alpha_1\sigma, \cdots, \alpha_n\sigma \Rightarrow \bot) \in J_s$ if $X = \bot$.

The procedure $GetLabel(G\theta, ATMS)$ returns the label of $G\theta$ and is used in constructing the solution. Note that the label of $G\theta$ corresponds to the minimal supports for $G\theta$. The hypothetical reasoner with the ATMS and the MGTP can avoid duplicate proofs among different contexts and repeated proofs of subgoals. However, there may be a lot of unnecessary proofs unrelated to the proofs of the goal.

5 Upside-Down Meta-Interpretation

5.1 Simple Transformation Algorithm

Bottom-up reasoning has the disadvantage of proving unnecessarily subgoals that are not related to proofs of the given goal. We introduce a simple transformation of a program P on the basis of the upside-down meta-interpretation for speedups of bottom-up reasoning by incorporating goal information. A bottom-up reasoner interprets a Horn clause

$$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta$$

in such a way that the fact $\beta\sigma$ is derived if facts $\alpha_1\sigma, \cdots, \alpha_n\sigma$ are present for some substitution σ. On the other hand, a top-down reasoner interprets it in such a way that goals $\alpha_1\sigma, \cdots, \alpha_n\sigma$ are derived if a goal $\beta\sigma$ is present, and fact $\beta\sigma$ is derived if both a goal $\beta\sigma$ and facts $\alpha_1\sigma, \cdots, \alpha_n\sigma$ are present. We transform the Horn clause

$$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta$$

into

$$goal(\beta) \rightarrow goal(\alpha_i)$$

for every α_i $(1 \leq i \leq n)$ and

$$goal(\beta) \wedge \alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta,$$

then a bottom-up reasoner can simulate top-down reasoning. Here, *goal* is a metapredicate symbol which does not appear in the original program P. After some facts related to the proofs of the goal have derived with the upside-down meta-interpretation, those facts may derive contradiction with bottom-up interpretation of the original program. Thus, we transform each negative clause

$$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \bot$$

into

$$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \bot$$

and

$$\rightarrow goal(\alpha_i)$$

for every α_i $(1 \leq i \leq n)$. This means that every subgoal related to negative clauses is evaluated.

Note that $(goal(\beta) \rightarrow goal(\alpha_i))$ or $(\rightarrow goal(\alpha_i))$ may not be satisfy the range-restricted condition. We have some techniques which make every clause in transformed programs range-restricted. Here, we take a very simple technique in which only the predicate symbols are used as the arguments of the metapredicate *goal*. When γ is an atomic formula, we denote by $\bar{\gamma}$ the predicate symbol of γ. The algorithm $T1$ as shown in Figure 3 transforms an original program P into the program $\hat{P}$ in which the top-down information is incorporated. The solution to G from $T1(\bar{G}, P)$ is always the same as the solution to G from P because all subgoals related to negative clauses as well as the given goal are evaluated and every label of $goal(\bar{\beta})$ for any atomic formula β is $\{\emptyset\}$.

For example, consider a program,

$$\begin{aligned} P_b = \{ \quad & \rightarrow penguin(a), \\ & penguin(X) \rightarrow bird(X), \\ & bird(X) \rightarrow \mathbf{assume}(fly(X)), \\ & fly(X) \wedge notfly(X) \rightarrow \bot, \\ & penguin(X) \rightarrow notfly(X) \ \}. \end{aligned}$$

By the simple transformation algorithm, we get

$$\begin{aligned} T1(fly, \ P_b) = \\ \{ \quad & goal(penguin) \rightarrow penguin(a), \\ & goal(bird) \wedge penguin(X) \rightarrow bird(X), \\ & goal(bird) \rightarrow goal(penguin), \\ & goal(fly) \wedge bird(X) \rightarrow \mathbf{assume}(fly(X)), \\ & goal(fly) \rightarrow goal(bird), \\ & fly(X) \wedge notfly(X) \rightarrow \bot, \\ & \rightarrow goal(fly), \\ & \rightarrow goal(notfly), \\ & goal(notfly) \wedge penguin(X) \rightarrow notfly(X), \\ & goal(notfly) \rightarrow goal(penguin) \ \} \\ \cup \{ \quad & \rightarrow goal(fly) \ \}. \end{aligned}$$

Next, consider the goal $bird(X)$. Then, the transformed program $T1(bird, P_b)$ is the program

$$T1(bird, P_b) = \{\cdots\} \cup \{ \rightarrow goal(bird) \},$$

where only the last element $(\rightarrow goal(fly))$ of $T1(fly, P_b)$ is replaced with $(\rightarrow goal(bird))$. Even if the goal is $bird(X)$, both $goal(fly)$ and $goal(notfly)$ are evaluated because $\{\cdots\}$ includes $(\rightarrow goal(fly))$ and $(\rightarrow goal(notfly))$ for the negative clause. Then, the computational cost of $R(bird(X), T1(bird, P_b))$ is nearly equal to the cost of $R(fly(X), T1(fly, P_b))$.

procedure $T1(\bar{G}, P)$:
begin
  $\hat{P} := \emptyset$;
  for *each* $(\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow X) \in P$ **do**
  begin
    if $X = \bot$ **then**
      begin
        $\hat{P} := \hat{P} \cup \{\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \bot\}$;
        for $j := 1$ **until** n **do**
          $\hat{P} := \hat{P} \cup \{\rightarrow goal(\bar{\alpha_j})\}$
      end
    else if $X = \mathbf{assume}(\beta)$ **then**
      begin
        $\hat{P} := \hat{P} \cup \{goal(\bar{\beta}) \wedge \alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \mathbf{assume}(\beta)\}$;
        for $j := 1$ **until** n **do**
          $\hat{P} := \hat{P} \cup \{goal(\bar{\beta}) \rightarrow goal(\bar{\alpha_j})\}$
      end
    else if $X = \beta$ **then**
      begin
        $\hat{P} := \hat{P} \cup \{goal(\bar{\beta}) \wedge \alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta\}$;
        for $j := 1$ **until** n **do**
          $\hat{P} := \hat{P} \cup \{goal(\bar{\beta}) \rightarrow goal(\bar{\alpha_j})\}$
      end
    end;
  $\hat{P} := \hat{P} \cup \{\rightarrow goal(\bar{G})\}$;
  return $\hat{P}$
end.

Figure 3: **Simple Transformation Algorithm** $T1$

5.2 Transformation Algorithm with Abstracted Dependency Analysis

In this subsection, we describe a static method to find irrelevant negative clauses to evaluation of the goal. If we can find such irrelevant negative clauses, for every antecedent α_i of each irrelevant clause, we do not need to add $(\rightarrow goal(\alpha_i))$ into the transformed program. We try to find them by analyzing logical dependencies between the goal and each negative clause at the abstracted level. We do not care about any argument in the abstracted dependency analysis.

When γ is an atomic formula, we denote by the proposition $\bar{\gamma}$ the predicate symbol of γ. For each negative clause C, the proposition $false(C)$ is used as the identifier of C. For every $(\alpha \rightarrow \mathbf{assume}(\beta))$, $\bar{\beta}$ is called an *assumable-predicate symbol.* For any environment E, its *abstracted environment* (denoted by $\bar{E}$) is $\{ \Gamma_{\bar{\beta}} \mid \Gamma_\beta \in E\}$. The *abstracted justifications* with respect to P is defined as:

$$\begin{aligned}\bar{J} \equiv\ & \{(\bar{\alpha}_1, \cdots, \bar{\alpha}_n, \Gamma_{\bar{\beta}} \Rightarrow \bar{\beta}) \mid \\ & \qquad (\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \mathbf{assume}(\beta)) \in P\} \\ \cup\ & \{(\bar{\alpha}_1, \cdots, \bar{\alpha}_n \Rightarrow \bar{\beta}) \mid (\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta) \in P\} \\ \cup\ & \{(\bar{\alpha}_1, \cdots, \bar{\alpha}_n \Rightarrow false(C)) \mid \\ & \qquad C = (\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \bot),\ C \in P\}.\end{aligned}$$

Let $\bar{A}$ be the set of propositions appearing in $\bar{J}$. Note that $\bar{A}$ consists of all predicate symbols in P and all $false(C)$ for $C \in P$. For each proposition N in $\bar{A}$, we compute a set of abstracted environments on which N depends. Now, we show an algorithm to compute the set of abstracted environments. This algorithm is obtained by modifying the label-updating algorithm shown in Section 3. The modified points are as follows.

1. Replace **Step 2** with
   Step 2′: Set the new label of N to L'.
2. Remove **Step 4**.

Every proposition in $\bar{A}$ is labeled with the set of abstracted environments obtained by applying the modified algorithm to the abstracted justifications $\bar{J}$. This label is called the *abstracted label* of the proposition. The system to compute the set of abstracted environments for each proposition is called an *abstracted dependency analyzer.* The reasons why we have to modify the label-updating algorithm are as follows. Firstly, in the abstracted justifications, every $\bot$ is replaced with the proposition $false(C)$ for the negative clause C, so that each abstracted label is always consistent. Thus, we do not need **Step 4**. Secondly, each abstracted label may not be minimal because we replace **Step 2** with **Step 2′**. Suppose that every abstracted label is minimal. Then, the theorem we present below may not hold. For example, let

$$\begin{aligned}P_e = \{\ & \rightarrow p(a),\ \rightarrow p(b),\ \rightarrow q(b),\ q(X) \rightarrow t(X), \\ & p(X) \rightarrow \mathbf{assume}(r(X)), \\ & p(X) \rightarrow \mathbf{assume}(s(X)), \\ & r(a) \rightarrow g,\ r(X) \wedge s(X) \rightarrow g, \\ & r(X) \wedge s(X) \wedge t(X) \rightarrow \bot\ \}.\end{aligned}$$

Consider the problem defined with the goal g and P_e. The abstracted label of g is $\{\{r\}, \{r, s\}\}$. The abstracted label of the negative clause is $\{\{r, s\}\}$. The abstracted environment $\{r, s\}$ cannot be omitted for g although the set of minimal elements in the abstracted label of g is $\{\{r\}\}$.

```
procedure T2(Ḡ, P):
begin
  P̂ := ∅;
  J̄ := ∅;
  k := 0;
  for each (α1 ∧ ··· ∧ αn → X) ∈ P do
  begin
    if X = ⊥ then
      begin
        k := k + 1
        P̂ := P̂ ∪ {α1 ∧ ··· ∧ αn → ⊥};
        J̄ := J̄ ∪ {(ᾱ1, ··· , ᾱn ⇒ false(k))};
      end
    else if X = assume(β) then
      begin
        P̂ := P̂ ∪
           {goal(β̄) ∧ α1 ∧ ··· ∧ αn → assume(β)};
        J̄ := J̄ ∪ {(ᾱ1, ··· , ᾱn, Γβ̄ ⇒ β̄)};
        for j := 1 until n do
          P̂ := P̂ ∪ {goal(β̄) → goal(ᾱj)}
      end
    else if X = β then
      begin
        P̂ := P̂ ∪ {goal(β̄) ∧ α1 ∧ ··· ∧ αn → β};
        J̄ := J̄ ∪ {(ᾱ1, ··· , ᾱn ⇒ β̄)};
        for j := 1 until n do
          P̂ := P̂ ∪ {goal(β̄) → goal(ᾱj)}
      end
  end;
  UpdateAbstractedLabels(J̄, ADA);
  LG := GetAbstractedLabel(Ḡ, ADA);
  for i := 1 until k do
    begin
      Li := GetAbstractedLabel(false(i), ADA);
      for each EG ∈ LG do
        for each Ei ∈ Li do
          if Ei ⊆ EG then
            for (ᾱ1, ··· , ᾱn ⇒ false(i)) ∈ J̄ do
              for j := 1 until n do
                P̂ := P̂ ∪ {→ goal(ᾱj)}
    end;
  P̂ := P̂ ∪ {→ goal(Ḡ)};
  return P̂
end.
```

Figure 4: **Transformation Algorithm** $T2$ **with Abstracted Dependency Analysis**

Theorem: Let P be a normal default theory and G a goal, $\bar{J}$ the abstracted justifications with respect to P , $L(\bar{G})$ the abstracted label of $\bar{G}$, $L(false(C))$ the abstracted label of $false(C)$ where $C \in P$. If no element in $L(false(C))$ is a subset of any element in $L(\bar{G})$, then the solution to G from P is equivalent to the solution to G from $P \setminus \{C\}$.

Sketch of the proof: Let C be $(\alpha \rightarrow \bot)$ and P' be $P \setminus \{C\}$. Assume that θ_m is any answer substitution for G from P' and σ_k is any answer substitution for α from P'. Let $MS(\alpha\sigma_k)$ be the minimal supports for $\alpha\sigma_k$ from P' and $MS(G\theta_m)$ be the minimal supports for $G\theta_m$ from P'. Suppose that no element in $L(false(C))$ is a subset of any element in $L(\bar{G})$. From the supposition and similarity between ATMS labels and abstracted labels, no element in $MS(\alpha\sigma_k)$ is a subset of any element in $MS(G\theta_m)$. Therefore, the solution to G from $P' \cup \{C\}$ is the same as the solution to G from P'. ■

On the basis of the theorem, we can omit consistency checking for a negative clause C if the condition of the theorem is satisfied. The transformation algorithm $T2(\bar{G}, P)$ with the abstracted dependency analysis is shown in Figure 4 for the program P and the goal G. In Figure 4, $UpdateAbstractedLabels(\bar{J}, ADA)$ denotes the procedure which computes abstracted labels from abstracted justifications $\bar{J}$ with the abstracted dependency analyzer ADA, and $GetAbstractedLabel(\bar{G}, ADA)$ denotes the procedure which returns the abstracted label of $\bar{G}$ from the abstracted dependency analyzer ADA. The procedure transforms an original program into the program in which the top-down information is incorporated and consistency checking is restricted to those negative clauses relevant to the given goal.

Consider the same example P_b, shown in the previous subsection, in case that the goal is $bird(X)$. The abstracted justifications $\bar{J}_b$ is

$$\{\ (\Rightarrow penguin), (penguin \Rightarrow bird), (bird, \Gamma_{fly} \Rightarrow fly),\ (fly, notfly \Rightarrow false(1)),\ \ (penguin \Rightarrow notfly)\ \}.$$

As the result of the abstracted dependency analysis, the abstracted label of $false(1)$ is $\{\{fly\}\}$ and the abstracted label of $bird$ is $\{\emptyset\}$. Then, no element in the abstracted label of $false(1)$ is a subset of any element in the abstracted label of $bird$, so that we do not need to evaluate this negative clause. As a consequence, we have the transformed program:

$$\begin{aligned} T2(bird,\ P_b) = \\ \{\ & goal(penguin) \rightarrow penguin(a), \\ & goal(bird) \wedge penguin(X) \rightarrow bird(X), \\ & goal(bird) \rightarrow goal(penguin), \\ & goal(fly) \wedge bird(X) \rightarrow \mathbf{assume}(fly(X)), \\ & goal(fly) \rightarrow goal(bird), \\ & fly(X) \wedge notfly(X) \rightarrow \bot, \\ & goal(notfly) \wedge penguin(X) \rightarrow notfly(X), \\ & goal(notfly) \rightarrow goal(penguin)\ \} \\ \cup\ \{\ & \rightarrow goal(bird)\ \}. \end{aligned}$$

Since the transformed program does not include $(\rightarrow goal(fly))$ and $(\rightarrow goal(notfly))$, the reasoner can omit solving both the goal $fly(X)$ and the goal $notfly(X)$.

6 Evaluation with Logic Design Problem

We have taken up the design of logic circuits to calculate the greatest common divisor (GCD) of two integers expressed in 8 bits by using the Euclidean algorithm. The solutions are circuits calculating GCD and satisfying given constraints on area and time [Maruyama *et al.* 88]. The program P_d contains several kinds of knowledge: datapath design, component design, technology mapping, CMOS standard cells and constraints on area and time [Ohta and Inoue 90]. The design problem of calculators for GCD includes design of components such as subtracters and adders.

Table 1 shows the expermental result, on a Pseudo-Multi-PSI system, for the evaluation of the transformed programs. The run time of a program P for a goal G is denoted by $T_{R(G,P)}$. The predicate symbol $\tilde{G}$ of each goal G is *adder* (design of adders), *subtracter* (design of subtracters) or *cGCD* (design of calculators for GCD). The run time $T_{R(G,P_d)}$ of each goal G is equal to the others on the original program P_d.

Table 1: **Run Time of Program**

Goal $\tilde{G}$	$T_{R(G,P_d)}$ [s]	$T_{R(G,P_1)}$ [s]	$T_{R(G,P_2)}$ [s]
adder	10.7	17.5	0.4
subtracter	10.7	17.3	0.6
cGCD	10.7	17.3	16.8

Let P_1 be the simple transformed program of P_d. The experiment on the simple transformation time shows that it takes 6.35 [s] for making P_1 from P_d. However, the run time $T_{R(G,P_1)}$ for each goal G is nearly equal to the others because constraints on area and time of the GCD calculators are represented by negative clauses. Even if we want to design adders or subtracters, the hypothetical reasoner cannot avoid designing GCD calculators for consistency checking.

Let P_2 be the transformed program with the abstracted dependency analysis. The experiment on the transformation time with the abstracted dependency analysis shows that it takes 6.63 [s] for making P_2 from P_d. The transformation time with the abstracted dependency analysis is a little bit longer (0.28 [s]) than the simple transformation time. When $\tilde{G}$ is *adder* or *subtracter*, the run time $T_{R(G,P_2)}$ is much shorter than the run time for the design of GCD calculators. This is because the program can avoid consistency checks for negative clauses representing constraints on area and time of the GCD calculators when the design of adders or the design of subtracters is given as a goal. The result show that each total of the transformation time with abstracted dependency analysis and the run time of the transformed program is shorter than the run time of the original program when the problem does not need the whole of the program.

7 Related Work

The algorithm for first-order Horn-clause abduction with the ATMS is presented in [Ng and Mooney 91]. The system is basically a consumer architecture [de Kleer 86-3] introducing backward-chaining consumers. The algorithm avoids both redundant proofs by introducing the goal-directed backward-chaining consumers and duplicate proofs among different contexts by using the ATMS. Their problem definition is the same as [Stickel 90], whose inputs are a goal and a set of Horn clauses without negative clauses. When there are negative clauses in the program, they briefly suggest that forward-chaining consumer can be used for each negative clause to check the consistency. On the other hand, since we only simulate backward-chaining by the forward-chaining reasoner, we do not require both types of chaining rules. Moreover, we see that when the program includes negative clauses, it is sometimes difficult to represent the clauses as a set of consumers. For example, suppose that the axioms are

$$\{a \rightarrow c,\ b \rightarrow d,\ c \wedge d \rightarrow g,\ c \rightarrow e,\ d \rightarrow f,\ e \wedge f \rightarrow \bot\}$$

and the goal is g. Assume that the set of consumers is

$$\{(c \Leftarrow a),\ (d \Leftarrow b),\ (g \Leftarrow c, d),\\ (e \Leftarrow c),\ (f \Leftarrow d),\ (e, f \Rightarrow \bot)\},$$

where $\Leftarrow$ means a backward-chaining consumer and $\Rightarrow$ means a forward-chaining consumer. Then, we get the solution $\{\langle g, \{\{g\}, \{a, b\}, \{a, d\}, \{c, b\}, \{c, d\}\}\rangle\}$. However, the correct solution is $\{\langle g, \{\{g\}\}\rangle\}$ because $\{a, b\}, \{a, d\}, \{c, b\}$ and $\{c, d\}$ are nogood. To guarantee the consistency when the program includes negative clauses, for every Horn clause, we have to add the corresponding forward-chaining consumer. Such added consumers would cause the same problem as the program that appeared in using the simple transformation algorithm.

In [Stickel 91], deduction and abduction with the upside-down meta-interpretation are proposed. This abduction does not require the consistency of solutions. Furthermore, rules may do duplicate firing in different contexts since it does not use the ATMS. This often causes a problem when it is applied to practical programs where heavy procedures are attached to rules.

Another difference between the frameworks of [Ng and Mooney 91, Stickel 91] and ours is that their

frameworks treat only hypotheses in the form of normal defaults without prerequisites, whereas we allow for normal defaults with prerequisites.

8 Conclusion

We have presented a new transformation algorithm of programs for efficient forward-chaining hypothetical reasoning based on the upside-down meta-interpretation. In the transformation algorithm, logical dependencies between a goal and negative clauses are analyzed at abstracted level to find irrelevant negative clauses, so that consistency checking of negative clauses can be restricted to those relevant clauses. It has been evaluated with a logic circuit design problem on a Pseudo-Multi-PSI system.

We can also apply this abstracted dependency analysis to transformed programs based on Magic Set and Alexander methods. Our dependency analysis with only predicate symbols may be extended to an analysis with predicate symbols and their some arguments.

Acknowledgments

Thanks are due to Mr. Makoto Nakashima of JIPDEC for implementing the ATMS and combining it with the MGTP. We are grateful to Prof. Mitsuru Ishizuka of the University of Tokyo for the helpful discussion. We would also like to thank Dr. Ryuzo Hasegawa and Mr. Miyuki Koshimura for providing us the MGTP, and Dr. Koichi Furukawa for his advise. Finally, we would like to express our appreciation to Dr. Kazuhiro Fuchi, Director of ICOT Research Center, who provided us with the opportunity to conduct this research.

References

[Bancilhon *et al.* 86] F. Bancilhon, D. Maier, Y. Sagiv and J.D. Ullman, Magic Sets and Other Strange Ways to Implement Logic Programs, *Proc. of ACM PODS*, pp.1-15 (1986).

[Bry 90] F. Bry, Query evaluation in recursive databases: bottom-up and top-down reconciled, *Data & Knowledge Engineering*, **5**, pp.289–312 (1990).

[de Kleer 86-1] J. de Kleer, An Assumption-based TMS, *Artificial Intelligence*, **28**, pp.127–162 (1986).

[de Kleer 86-2] J. de Kleer, Extending the ATMS, *Artificial Intelligence*, **28**, pp.163–196 (1986).

[de Kleer 86-3] J. de Kleer, Problem Solving with the ATMS, *Artificial Intelligence*, **28**, pp.197–224 (1986)

[Flann *et al.* 87] N.S. Flann, T.G. Dietterich and D.R. Corpron, Forward Chaining Logic Programming with the ATMS, *Proc. of AAAI-87*, pp.24-29 (1987).

[Forgy 82] C.L. Forgy, Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence*, **19**, pp.17-37 (1982).

[Fujita and Hasegawa 91] H. Fujita and R. Hasegawa, A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm, *Proc. of ICLP '91*, pp.494–500 (1991).

[Inoue 88] K. Inoue, Problem Solving with Hypothetical Reasoning, *Proc. of FGCS '88*, pp.1275-1281 (1988).

[Junker 88] U. Junker, Reasoning in Multiple Contexts, GMD Working Paper No.334 (1988).

[Maruyama *et al.* 88] F. Maruyama, T. Kakuda, Y. Masunaga, Y. Minoda, S. Sawada and N. Kawato, co-LODEX: A Cooperative Expert System for Logic Design, *Proc. of FGCS '88*, pp.1299-1306 (1988).

[Ng and Mooney 91] H.T. Ng and R.J. Mooney, An Efficient First-Order Abduction System Based on the ATMS, Technical Report AI 91-151, The University of Texas at Austin, AI Lab. (1991).

[Ohta and Inoue 90] Y. Ohta and K. Inoue, A Forward-Chaining Multiple-Context Reasoner and Its Application to Logic Design, *Proc. of IEEE TAI*, pp.386–392 (1990).

[Poole *et al.* 87] D. Poole, R. Goebel and R. Aleliunas, Theorist: A logical Reasoning System for Defaults and Diagnosis, N. Cercone and G. McCalla (Eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge*, Springer-Verlag, pp.331–352 (1987).

[Poole 91] D. Poole, Compiling a Default Reasoning System into Prolog, *New Generation Computing*, **9**, pp.3–38 (1991).

[Reiter 80] R. Reiter, A Logic for Default Reasoning, *Artificial Intelligence*, **13**, pp.81–132 (1980).

[Rohmer *et al.* 86] J. Rohmer, R. Lescoeur and J.M. Kerisit, The Alexander Method — A Technique for The Processing of Recursive Axioms in Deductive Databases, *New Generation Computing*, **4**, pp.273–285 (1986).

[Stickel 90] M.E. Stickel, Rationale and Methods for Abductive Reasoning in Natural-Language Interpretation, *Lecture Nodes in Artificial Intelligence*, **459**, Springer-Verlag, pp.233–252 (1990).

[Stickel 91] M.E. Stickel, Upside-Down Meta-Interpretation of the Model Elimination Theorem-Prover Procedure for Deduction and Abduction, ICOT Technical Report TR-664, ICOT (1991).

[Ueda and Chikayama 90] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine, *The Computer Journal*, **33**, 6, pp. 494–500 (1990).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Logic Programming, Abduction and Probability

David Poole
Department of Computer Science,
University of British Columbia,
Vancouver, B.C., Canada V6T 1Z2
poole@cs.ubc.ca
telephone: (604) 822 6254
fax: (604) 822 5485

Abstract

Probabilistic Horn abduction is a simple framework to combine probabilistic and logical reasoning into a coherent practical framework. The numbers can be consistently interpreted probabilistically, and all of the rules can be interpreted logically. The relationship between probabilistic Horn abduction and logic programming is at two levels. At the first level probabilistic Horn abduction is an extension of pure Prolog, that is useful for diagnosis and other evidential reasoning tasks. At another level, current logic programming implementation techniques can be used to efficiently implement probabilistic Horn abduction. This forms the basis of an "anytime" algorithm for estimating arbitrary conditional probabilities. The focus of this paper is on the implementation.

1 Introduction

Probabilistic Horn Abduction [Poole, 1991c; Poole, 1991b; Poole, 1992a] is a framework for logic-based abduction that incorporates probabilities with assumptions. It is being used as a framework for diagnosis [Poole, 1991c] that incorporates both pure Prolog and Bayesian Networks [Pearl, 1988] as special cases [Poole, 1991b]. This paper is about the relationship of probabilistic Horn abduction to logic programming. This simple extension to logic programming provides a wealth of new applications in diagnosis, recognition and evidential reasoning [Poole, 1992a].

This paper also presents a logic-programming solution to the problem in abduction of searching for the "best" diagnoses first. The main features of the approach are:

- We are using Horn clause abduction. The procedures are simple, both conceptually and computationally (for a certain class of problems). We develop a simple extension of SLD resolution to implement our framework.
- The search algorithms form "anytime" algorithms that can give an estimate of the conditional probability at any time. We do not generate the unlikely explanations unless we need to. We have a bound on the probability mass of the remaining explanations which allows us to know the error in our estimates.
- A theory of "partial explanations" is developed. These are partial proofs that can be stored in a priority queue until they need to be further expanded. We show how this is implemented in a Prolog interpreter in Appendix A.

2 Probabilistic Horn abduction

The formulation of abduction used is a simplified form of Theorist [Poole *et al.*, 1987; Poole, 1988] with probabilities associated with the hypotheses. It is simplified in being restricted to definite clauses with simple forms of integrity constraints (similar to that in [Goebel *et al.*, 1986]). This can also be seen as a generalisation of an ATMS [Reiter and de Kleer, 1987] to be non-propositional.

The language is that of pure Prolog (i.e., definite clauses) with special disjoint declarations that specify a set of disjoint hypotheses with associated probabilities. There are some restrictions on the forms of the rules and the probabilistic dependence allowed. The language presented here is that of [Poole, 1992a] rather than that of [Poole, 1991c; Poole, 1991b].

The main design considerations were to make a language the simplest extension to pure Prolog that also included probabilities (not just numbers associated with rules, but numbers that follow the laws of probability, and so can be consistently interpreted as probabilities [Poole, 1992a]). We are also assuming very strong independence assumptions; this is not intended to be a temporary restriction on the language that we want to eventually remove, but as a feature. We can represent any probabilistic information using only independent hypotheses [Poole, 1992a]; if there is any dependence amongst hypotheses, we invent a new hypothesis to explain that dependency.

2.1 The language

Our language uses the Prolog conventions, and has the same definitions of variables, terms and atomic symbols.

Definition 2.1 A **definite clause** is of the form: $a.$ or $a \leftarrow a_1 \wedge \cdots \wedge a_n.$ where a and each a_i are atomic symbols.

Definition 2.2 A **disjoint declaration** is of the form

$$disjoint([h_1 : p_1, \cdots, h_n : p_n]).$$

where the h_i are atoms, and the p_i are real numbers $0 \leq p_i \leq 1$ such that $p_1 + \cdots + p_n = 1$. Any variable appearing in one h_i must appear in all of the h_j (i.e., the h_i share the same variables). The h_i will be referred to as **hypotheses**.

Definition 2.3 A **probabilistic Horn abduction theory** (which will be referred to as a "theory") is a collection of definite clauses and disjoint declarations such that if a ground atom h is an instance of a hypothesis in one disjoint declaration, then it is not an instance of another hypothesis in any of the disjoint declarations.

Given theory T, we define

F_T the **facts**, is the set of definite clauses in T together with the clauses of the form

$$false \leftarrow h_i \wedge h_j$$

where h_i and h_j both appear in the same disjoint declaration in T, and $i \neq j$. Let F'_T be the set of ground instances of elements of F_T.

H_T to be the set of **hypotheses**, the set of h_i such that h_i appears in a disjoint declaration in T. Let H'_T be the set of ground instances of elements of H_T.

P_T is a function $H'_T \mapsto [0, 1]$. $P_T(h'_i) = p_i$ where h'_i is a ground instance of hypothesis h_i, and $h_i : p_i$ is in a disjoint declaration in T.

Where T is understood from context, we omit the subscript.

Definition 2.4 [Poole *et al.*, 1987; Poole, 1987] If g is a closed formula, an **explanation** of g from $\langle F, H\rangle$ is a set D of elements of H' such that

- $F \cup D \models g$ and
- $F \cup D \not\models false$.

The first condition says that D is a sufficient cause for g, and the second says that D is possible.

Definition 2.5 A **minimal explanation** of g is an explanation of g such that no strict subset is an explanation of g.

2.2 Assumptions about the rule base

Probabilistic Horn abduction also contains some assumptions about the rule base. It can be argued that these assumptions are natural, and do not really restrict what can be represented [Poole, 1992a]. Here we list these assumptions, and use them in order to show how the algorithms work.

The first assumption we make is about the relationship between hypotheses and rules:

Assumption 2.6 There are no rules with head unifying with a member of H.

Instead of having a rule implying a hypothesis, we invent a new atom, make the hypothesis imply this atom, and all of the rules imply this atom, and use this atom instead of the hypothesis.

Assumption 2.7 (acyclicity) If F' is the set of ground instances of elements of F, then it is possible to assign a natural number to every ground atom such that for every rule in F' the atoms in the body of the rule are strictly less than the atom in the head.

This assumption is discussed in [Apt and Bezem, 1990].

Assumption 2.8 The rules in F' for a ground non-assumable atom are covering.

That is, if the rules for a in F' are

$$\begin{array}{l} a \leftarrow B_1 \\ a \leftarrow B_2 \\ \vdots \\ a \leftarrow B_m \end{array}$$

if a is true, one of the B_i is true. Thus Clark's completion [Clark, 1978] is valid for every non-assumable. Often we get around this assumption by adding a rule

$$a \leftarrow some\_other\_reason\_for\_a$$

and making "*some_other_reason_for_a*" a hypothesis [Poole, 1992a].

Lemma 2.9 [Console *et al.*, 1991; Poole, 1988] Under assumptions 2.6, 2.7 and 2.8, if $expl(g, T)$ is the set of minimal explanations of g from theory T:

$$g \equiv \bigvee_{e_i \in expl(g,T)} e_i$$

Assumption 2.10 The bodies of the rules in F' for an atom are mutually exclusive.

Given the above rules for a, this means that

$$B_i \wedge B_j \Rightarrow false$$

is true in the domain under consideration for each $i \neq j$. We can make this true by adding extra conditions to the rules to make sure they are disjoint.

Lemma 2.11 *Under assumptions 2.6 and 2.10, minimal explanations of atoms or conjunctions of atoms are mutually inconsistent.*

See [Poole, 1992a] for more justification of these assumptions.

2.3 Probabilities

Associated with each possible hypothesis is a prior probability. We use this prior probability to compute arbitrary probabilities.

The following is a corollary of lemmata 2.9 and 2.11

Lemma 2.12 *Under assumptions 2.6, 2.7, 2.8, 2.10 and 2.13, if $expl(g, T)$ is the set of minimal explanations of conjunction of atoms g from probabilistic Horn abduction theory T:*

$$\begin{aligned} P(g) &= P\left(\bigvee_{e_i \in expl(g,T)} e_i\right) \\ &= \sum_{e_i \in expl(g,T)} P(e_i) \end{aligned}$$

Thus to compute the prior probability of any g we sum the probabilities of the explanations of g.

To compute arbitrary conditional probabilities, we use the definition of conditional probability:

$$P(\alpha|\beta) = \frac{P(\alpha \wedge \beta)}{P(\beta)}$$

Thus to find arbitrary conditional probabilities $P(\alpha|\beta)$, we find $P(\beta)$, which is the sum of the explanations of β, and $P(\alpha \wedge \beta)$ which can be found by explaining α from the explanations of β. Thus arbitrary conditional probabilities can be computed from summing the prior probabilities of explanations.

It remains only to compute the prior probability of an explanation D of g. We assume that logical dependencies impose the only statistical dependencies on the hypotheses. In particular we assume:

Assumption 2.13 Ground instances of hypotheses that are not inconsistent (with F_T) are probabilistically independent. That is, different disjoint declarations define independent hypotheses.

The hypotheses in a minimal explanation are always logically independent. The language has been carefully set up so that the logic does not force any dependencies amongst the hypotheses. If we could prove that some hypotheses implied other hypotheses or their negations, the hypotheses could not be independent. The language is deliberately designed to be too weak to be able to state such logical dependencies between hypotheses.

Under assumption 2.13, if $\{h_1, \cdots, h_n\}$ are part of a minimal explanation, then

$$P(h_1 \wedge \cdots \wedge h_n) = \prod_{1=1}^{n} P(h_i)$$

To compute the prior of the minimal explanation we multiply the priors of the hypotheses. The posterior probability of the explanation is proportional to this.

The following is a corollary of lemmata 2.9 and 2.11

Lemma 2.14 *Under assumptions 2.6, 2.7, 2.8, 2.10 and 2.13, if $expl(g,T)$ is the set of all minimal explanations of g from theory T:*

$$P(g) = P\left(\bigvee_{e_i \in expl(g,T)} e_i\right) = \sum_{e_i \in expl(g,T)} P(e_i)$$

2.4 An example

In this section we show an example that we use later in the paper. It is intended to be as simple as possible to show how the algorithm works.

Suppose we have the rules and hypotheses:

```
rule((a :- b, h)).
rule((a :- q,e)).
rule((q :- h)).
rule((q :- b,e)).
rule((h :- b, f)).
rule((h :- c, e)).
rule((h :- g, b)).
disjoint([b:0.3,c:0.7]).
disjoint([e:0.6,f:0.3,g:0.1]).
```

There are four minimal explanations of a, namely $\{c,e\}$, $\{b,e\}$, $\{f,b\}$ and $\{g,b\}$.

The priors of the explanations are as follows:

$$P(c \wedge e) = 0.7 \times 0.6 = 0.42.$$

Similarly $P(b \wedge e) = 0.18$, $P(f \wedge b) = 0.09$ and $P(g \wedge b) = 0.03$. Thus

$$P(a) = 0.42 + 0.18 + 0.09 + 0.03 = 0.72$$

There are two explanations of $e \wedge a$, namely $\{c,e\}$ and $\{b,e\}$. Thus $P(e \wedge a) = 0.60$. Thus the conditional probability of e given a is $P(e|a) = 0.6/0.72 = 0.833$.

What is important about this example is that all of the probabilistic calculations reduce to finding the probabilities of explanations.

2.5 Tasks

The following tasks are what we expect to implement:

1. Generate the explanations of some goal (conjunction of atoms), in order.
2. Determine the prior probability of some goal. This is implemented by enumerating the explanations of the goal.
3. Determine the posterior probabilities of the explanations of a goal (i.e., the probabilities of the explanations given the goal).
4. Determine the conditional probability of one formula given another. That is, determining $P(\alpha|\beta)$ for any α and β.

All of these will be implemented by enumerating the explanations of a goal, and estimating the probability mass in the explanations that have not been enumerated. It is this problem that we consider for the next few sections, and then return to the problem of the tasks we want to compute.

3 A top-down proof procedure

In this section we show how to carry out a best-first search of the explanations. In order to do this we build a notion of a partial proof that we can add to a priority queue, and restart when necessary.

3.1 SLD-BF resolution

In this section we outline an implementation based on logic programming technology and a branch and bound search.

The implementation keeps a priority queue of sets of hypotheses that could be extended into explanations ("partial explanations"). At any time the set of all the explanations is the set of already generated explanations, plus those explanations that can be generated from the partial explanations in the priority queue.

$Q := \{\langle g \leftarrow g, \{\}\rangle\}$;
$\Pi := \{\}$;
repeat
    choose and remove best $\langle g \leftarrow C, D\rangle$ from Q;
    if $C = true$
      then if $good(D)$ then $\Pi := \Pi \cup \{D\}$ endif
      else Let $C = a \wedge R$
          for each $rule(h \leftarrow B)$ where $mgu(a, h) = \theta$
                $Q := Q \cup \{\langle g \leftarrow B \wedge R, D\rangle\,\theta\}$;
          if $a \in H$ and $good(\{a\} \cup D)$
                then $Q := Q \cup \{\langle g \leftarrow R, \{a\} \cup D\rangle\}$
          endif
    endif
until $Q = \{\}$
where $good(D) \equiv (\forall d_1, d_2 \in D \;\; \nexists \eta \in NG\, \exists \phi \;\; \langle d_1, d_2\rangle = \eta\phi)$
        $\wedge\, (\nexists \pi \in \Pi, \exists \phi \;\; D \subseteq \pi\phi)$

Figure 1: SLD-BF Resolution to find explanations of g in order.

Definition 3.1 a **partial explanation** is a structure

$$\langle g \leftarrow C, D\rangle$$

where g is an atom (or conjunction of atoms), C is a conjunction of atoms and D is a set of hypotheses.

Figure 1 gives an algorithm for finding explanations of q in order of probability (most likely first). At each step we choose an element

$$\langle g \leftarrow C, D\rangle$$

of the priority queue Q with maximum prior probability of D.

We have an explanation when C is the empty conjunction (represented here as *true*). In this case D is added to the set Π of already generated explanations.

Otherwise, suppose C is conjunction $a \wedge R$.

There are two operations that can be carried out. The first is a form of SLD resolution [Lloyd, 1987], where for each rule

$$h \leftarrow b_1 \wedge \cdots \wedge b_n$$

in F, such that h and a have most general unifier θ, we generate the partial explanation

$$\langle g \leftarrow b_1 \wedge \cdots \wedge b_n \wedge R, D\rangle\,\theta$$

and add it to the priority queue.

The second operation is used when $a \in H$. In this case we produce the partial explanation

$$\langle g \leftarrow R, \{a\} \cup D\rangle$$

and add it to Q. We only do this if $\{a\} \cup D$ is consistent, and is not subsumed by another explanation of q. Here we assume the set NG of pairs of hypotheses that appear in the same disjoint declaration (corresponding to *nogoods* in an ATMS [Reiter and de Kleer, 1987]). Unlike in an ATMS this set can be built at compile time from the disjoint declarations.

This procedure will find the explanations in order of likelihood. Its correctness is based on the meaning of a partial explanation

Definition 3.2 *A partial explanation* $\langle g \leftarrow C, D\rangle$ *is* **valid** *with respect to* $\langle F, H\rangle$ *if*

$$F \models D \wedge C \Rightarrow g$$

Lemma 3.3 *Every partial explanation in the queue* Q *is valid with respect to* $\langle F, H\rangle$.

Proof: This is proven by induction on the number of times through the loop.

It is trivially true initially as $q \Rightarrow q$ for any q.

There are two cases where elements are added to Q. In the first case (the "rule" case) we know

$$F \models D \wedge R \wedge a \Rightarrow g$$

by the inductive assumption, and so

$$F \models (D \wedge R \wedge a \Rightarrow g)\theta$$

We also know

$$F \models (B \Rightarrow h)\theta$$

As $a\theta = h\theta$, by a simple resolution step we have

$$F \models (D \wedge R \wedge B \Rightarrow g)\theta.$$

The other case is when $a \in H$. By the induction step

$$F \models D \wedge (a \wedge R) \Rightarrow g$$

and so

$$F \models (D \wedge a) \wedge R \Rightarrow g$$

If D only contains elements of H and a is an element of H then $\{a\} \cup D$ only contains elements of H. □

It is now trivial to show the following:

Corollary 3.4 *Every element of* Π *in figure 1 is an explanation of* q.

Although the correctness of the algorithm does not depend on which element of the queue we choose at any time, the efficiency does. We choose the *best* partial explanation based on the following ordering of partial explanations. Partial explanation $\langle g_1 \leftarrow C_1, D_1\rangle$ is better than $\langle g_2 \leftarrow C_2, D_2\rangle$ if $P(D_1) > P(D_2)$. It is simple to show that "better than" is a partial ordering. When we choose a "best" partial explanation we choose a minimal element of the partial ordering; where there are a number of minimal partial explanations, we can choose any one. When we follow this definition of "best", we enumerate the minimal explanations of q in order of probability.

3.2 Our example

In this section we show how the simple example in Section 2.4 is handled by the best-first proof process.

The following is the sequence of values of Q each time through the loop (where there are a number of minimal explanations, we choose the element that was added

last):

$\{\langle a \leftarrow a, \{\}\rangle\}$
$\{\langle a \leftarrow b \wedge h, \{\}\rangle, \langle a \leftarrow q \wedge e, \{\}\rangle\}$
$\{\langle a \leftarrow q \wedge e, \{\}\rangle, \langle a \leftarrow h, \{b\}\rangle\}$
$\{\langle a \leftarrow h \wedge e, \{\}\rangle, \langle a \leftarrow b \wedge e \wedge e, \{\}\rangle, \langle a \leftarrow h, \{b\}\rangle\}$
$\{\langle a \leftarrow b \wedge f \wedge e, \{\}\rangle, \langle a \leftarrow c \wedge e \wedge e, \{\}\rangle,$
$\langle a \leftarrow g \wedge b \wedge e, \{\}\rangle, \langle a \leftarrow b \wedge e \wedge e, \{\}\rangle, \langle a \leftarrow h, \{b\}\rangle\}$
$\{\langle a \leftarrow c \wedge e \wedge e, \{\}\rangle, \langle a \leftarrow g \wedge b \wedge e, \{\}\rangle,$
$\langle a \leftarrow b \wedge e \wedge e, \{\}\rangle, \langle a \leftarrow f \wedge e, \{b\}\rangle, \langle a \leftarrow h, \{b\}\rangle\}$
$\{\langle a \leftarrow g \wedge b \wedge e, \{\}\rangle, \langle a \leftarrow b \wedge e \wedge e, \{\}\rangle, \langle a \leftarrow e \wedge e, \{c\}\rangle,$
$\langle a \leftarrow f \wedge e, \{b\}\rangle, \langle a \leftarrow h, \{b\}\rangle\}$
$\{\langle a \leftarrow b \wedge e \wedge e, \{\}\rangle, \langle a \leftarrow e \wedge e, \{c\}\rangle, \langle a \leftarrow f \wedge e, \{b\}\rangle,$
$\langle a \leftarrow h, \{b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle\}$
$\{\langle a \leftarrow e \wedge e, \{c\}\rangle, \langle a \leftarrow e \wedge e, \{b\}\rangle, \langle a \leftarrow f \wedge e, \{b\}\rangle,$
$\langle a \leftarrow h, \{b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle\}$
$\{\langle a \leftarrow e, \{e, c\}\rangle, \langle a \leftarrow e \wedge e, \{b\}\rangle, \langle a \leftarrow f \wedge e, \{b\}\rangle,$
$\langle a \leftarrow h, \{b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle\}$
$\{\langle a \leftarrow true, \{e, c\}\rangle, \langle a \leftarrow e \wedge e, \{b\}\rangle, \langle a \leftarrow f \wedge e, \{b\}\rangle,$
$\langle a \leftarrow h, \{b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle\}$

Thus the first, and most likely explanation is $\{e, c\}$.

$\{\langle a \leftarrow e \wedge e, \{b\}\rangle, \langle a \leftarrow f \wedge e, \{b\}\rangle, \langle a \leftarrow h, \{b\}\rangle,$
$\langle a \leftarrow b \wedge e, \{g\}\rangle\}$
$\langle a \leftarrow f \wedge e, \{b\}\rangle, \langle a \leftarrow h, \{b\}\rangle, \langle a \leftarrow e, \{e, b\}\rangle,$
$\{\langle a \leftarrow b \wedge e, \{g\}\rangle\}$
$\{\langle a \leftarrow h, \{b\}\rangle, \langle a \leftarrow e, \{e, b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle,$
$\langle a \leftarrow e, \{f, b\}\rangle\}$
$\{\langle a \leftarrow b \wedge f, \{b\}\rangle, \langle a \leftarrow c \wedge e, \{b\}\rangle, \langle a \leftarrow g \wedge b, \{b\}\rangle,$
$\langle a \leftarrow e, \{e, b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle, \langle a \leftarrow e, \{f, b\}\rangle\}$
$\{\langle a \leftarrow f, \{b\}\rangle, \langle a \leftarrow c \wedge e, \{b\}\rangle, \langle a \leftarrow g \wedge b, \{b\}\rangle,$
$\langle a \leftarrow e, \{e, b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle, \langle a \leftarrow e, \{f, b\}\rangle\}$
$\{\langle a \leftarrow c \wedge e, \{b\}\rangle, \langle a \leftarrow g \wedge b, \{b\}\rangle, \langle a \leftarrow e, \{e, b\}\rangle,$
$\langle a \leftarrow b \wedge e, \{g\}\rangle, \langle a \leftarrow true, \{f, b\}\rangle, \langle a \leftarrow e, \{f, b\}\rangle\}$

Here the algorithm effectively prunes the top partial explanation as $\langle c, b\rangle$ forms a nogood.

$\{\langle a \leftarrow g \wedge b, \{b\}\rangle, \langle a \leftarrow e, \{e, b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle,$
$\langle a \leftarrow true, \{f, b\}\rangle, \langle a \leftarrow e, \{f, b\}\rangle\}$
$\{\langle a \leftarrow e, \{e, b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle, \langle a \leftarrow true, \{f, b\}\rangle,$
$\langle a \leftarrow e, \{f, b\}\rangle, \langle a \leftarrow b, \{g, b\}\rangle\}\}$
$\{\langle a \leftarrow true, \{e, b\}\rangle, \langle a \leftarrow b \wedge e, \{g\}\rangle, \langle a \leftarrow true, \{f, b\}\rangle,$
$\langle a \leftarrow e, \{f, b\}\rangle, \langle a \leftarrow b, \{g, b\}\rangle\}$

We have now found the second most likely explanation, namely $\{e, b\}$.

$\{\langle a \leftarrow b \wedge e, \{g\}\rangle, \langle a \leftarrow true, \{f, b\}\rangle, \langle a \leftarrow e, \{f, b\}\rangle,$
$\langle a \leftarrow b, \{g, b\}\rangle\}$
$\{\langle a \leftarrow true, \{f, b\}\rangle, \langle a \leftarrow e, \{f, b\}\rangle, \langle a \leftarrow e, \{g, b\}\rangle,$
$\langle a \leftarrow b, \{g, b\}\rangle\}$

We have thus found the third explanation $\{f, b\}$.

$\{\langle a \leftarrow e, \{f, b\}\rangle, \langle a \leftarrow e, \{g, b\}\rangle, \langle a \leftarrow b, \{g, b\}\rangle\}$
$\{\langle a \leftarrow e, \{g, b\}\rangle, \langle a \leftarrow b, \{g, b\}\rangle\}$
$\{\langle a \leftarrow b, \{g, b\}\rangle\}$
$\{\langle a \leftarrow true, \{g, b\}\rangle\}$

The fourth explanation is $\{g, b\}$. There are no more partial explanations and the process stops.

4 Discussion

4.1 Probabilities in the queue

We would like to give an estimate for $P(g)$ after having generated only a few of the most likely explanations of g, and get some estimate of our error. This problem reduces to estimating the probability of partial explanations in the queue.

If $\langle g \leftarrow C, D\rangle$ is in the priority queue, then it can possibly be used to generate explanations $D_1, \cdots, D_n$. Each D_i will be of the form $D \cup D'_i$. We can place a bound on the probability mass of all of the D_i, by

$$\begin{aligned} P(D_1 \vee \cdots \vee D_n) &= P(D \wedge (D'_1 \vee \cdots \vee D'_n)) \\ &\leq P(D) \end{aligned}$$

Given this upper bound, we can determine an upper bound for $P(g)$, where $\{e_1, \cdots, e_n\}$ is the set of all minimal explanations of g:

$$\begin{aligned} P(g) &= P(e_1 \vee e_2 \vee \cdots \vee e_n) \\ &= P(e_1) + P(e_2) + \cdots + P(e_n) \\ &= \left(\sum_{e_i \text{ found}} P(e_i)\right) + \left(\sum_{e_j \text{ to be generated}} P(e_j)\right) \end{aligned}$$

We can easily compute the first of these sums, and can put upper and lower bounds on the second. This means that we can put a bound on the range of probabilities of a goal based on finding just some of the explanations of the goal. Suppose we have goal g, and we have generated explanations Π. Let

$$P_\Pi = \sum_{D \in \Pi} P(D)$$

$$P_Q = \sum_{D:\langle g \leftarrow C, D\rangle \in Q} P(D)$$

where Q is the priority queue.

We then have

$$P_\Pi \leq P(g) \leq P_\Pi + P_Q$$

As the computation progresses, the probability mass in the queue P_Q approaches zero[1] and we get a better refinement on the value of $P(g)$. This thus forms the basis of an "anytime" algorithm for Bayesian networks.

4.2 Conditional Probabilities

We can also use the above procedure to compute conditional probabilities. Suppose we are trying to compute the conditional probability $P(\alpha|\beta)$. This can be computed from the definition:

$$P(\alpha|\beta) = \frac{P(\alpha \wedge \beta)}{P(\beta)}$$

We compute the conditional probabilities by enumerating the minimal explanations of $\alpha \wedge \beta$ and β. Note that the minimal explanations of $\alpha \wedge \beta$ are explanations (not

[1] Note that the estimate given above does not always decrease. It is possible that the error estimate increases. [Poole, 1992b] considers cases where convergence can be guaranteed.

necessarily minimal) of β. We can compute the explanations of $\alpha \wedge \beta$, by trying to explain α from the explanations of β. The above procedure can be easily adapted for this task, by making the task to explain $\beta \wedge \alpha$, and making sure we prove β before we prove α, so that we can collect the explanations of β as a we generate them. Let P^β be the sum of the probabilities of the explanations of β enumerated, and let $P^{\alpha\wedge\beta}$ be the sum of the explanations of $\alpha \wedge \beta$ generated.

Thus given our estimates of $P(\alpha \wedge \beta)$ and $P(\beta)$ we have

$$\frac{P^{\alpha\wedge\beta}}{P^\beta + P_Q} \leq P(\alpha|\beta) \leq \frac{P^{\alpha\wedge\beta} + P_Q}{P^\beta}$$

The lower bound is the case where all of the partial descriptions in the queue go towards worlds implying β, but none of these also lead to α. The upper bound is the case where all of the elements of the queue go towards implying α, from the explanations already generated for β.

4.3 Consistency and subsumption checking

One problem that needs to be considered is the problem of what happens when there are free variables in the hypotheses generated. When we generate the hypotheses, there may be some instances of the hypotheses that are inconsistent, and some that are consistent. We know that every instance is inconsistent if the subgoal is subsumed by a nogood. This can be determined by substituting constants for the variables in the the subgoal, and finding if a subset unifies with a nogood.

We cannot prune hypotheses because an instance is inconsistent. However, when computation progresses, we may substitute a value for a variable that makes the partial explanation inconsistent. This problem is similar to the problem of delaying negation-as-failure derivations [Naish, 1986], and of delaying consistency checking in Theorist [Poole, 1991a]. We would like to notice such inconsistencies as soon as possible. In the algorithm of Figure 1 we check for inconsistency each time a partial explanation is taken off the queue. There are cases where we do not have to check this explicitly, for example when we have done a resolution step that did not assign a variable. There is a trade-off between checking consistency and allowing some inconsistent hypotheses on the queue[2]. This trade-off is beyond the scope of this paper.

Note that the assumptions used in building the system imply that there can be no free variables in any explanation of a ground goal (otherwise we have infinitely many disjoint explanations with bounded probability). Thus delaying subgoals eventually grounds all variables.

4.4 Iterative deepening

In many search techniques we often get much better space complexity and asymptotically the same time complexity by using an iterative deepening version of a search procedure [Korf, 1985]. An iterative deepening version of the best-first search procedure is exactly the same as the iterative deepening version of A* with the heuristic function of zero [Korf, 1985]. The algorithm of procedure 1 is given at a level of abstraction which does not preclude iterative deepening.

For our experimental implementations, we have used an interesting variant of iterative deepening. Our queue is only a "virtual queue" and we only physically store partial explanations with probability greater than some threshold. We remember the mass of the whole queue, including the values we have chosen not to store. When the queue is empty, we decrease the threshold. We can estimate the threshold that we need for some given accuracy. This speeds up the computation and requires less space.

4.5 Recomputing subgoals

One of the problems with the above procedure is that it recomputes explanations for the same subgoal. If s is queried as a subgoal many times then we keep finding the same explanations for s. This has more to do with the notion of SLD resolution used than with the use of branch and bound search.

We are currently experimenting with a top-down procedure where we remember computation that we have computed, forming "lemmata". This is similar to the use of memo functions [Sterling and Shapiro, 1986] or Earley deduction [Pereira and Shieber, 1987] in logic programming, but we have to be very careful with the interaction between making lemmata and the branch and bound search, particularly as there may be multiple answers to any query, and just because we ask a query does not mean we want to solve it (we may only want to bound the probability of the answer).

4.6 Bounding the priority queue

Another problem with the above procedure that is not solved by lemmatisation is that the bound on the priority queue can become quite large (i.e., greater than one). Some bottom-up procedures [Poole, 1992b], can have an accurate estimate of the probability mass of the queue (i.e., an accurate bound on how much probability mass could be on the queue based on the information at hand). See [Poole, 1992b] for a description of a bottom-up procedure that can be compared to the top-down procedure in this paper. In [Poole, 1992b] an average case analysis is given on the bottom-up procedure; while this is not an accurate estimate for the top-down procedure, the case where the bottom-up procedure is efficient [Poole, 1992b] is the same case where the top-down procedure works well; that is where there are normality conditions that dominate the probability of each hypothesis (i.e., where all of the probabilities are near one or near zero).

5 Comparison with other systems

There are many other proposals for logic-based abduction schemes (e.g., [Pople, 1973; Cox and Pietrzykowski, 1987; Goebel *et al.*, 1986; Poole, 1987]). These, however, consider that we either find an arbitrary explanation or find all explanations. In practice there are prohibitively many of these. It is also not clear what to do with all of the explanations; there are too many to give to a

[2] We have to check the consistency at some time. This could be as late as just before the explanation is added to Π.

user, and the costs of determining which of the explanations is the "real" explanation (by doing tests [Sattar and Goebel, 1991]) is usually not outweighed by the advantages of finding the real explanation. This is why it is important to take into account probabilities. We then have a principled reason for ignoring many explanations. Probabilities are also the right tool to use when we really are unsure as to whether something is true or not. For evidential reasoning tasks (e.g., diagnosis and recognition) it is not up to us to decide whether some hypothesis is true or not; all we have is probabilities and evidence to work out what is most likely true. Similar considerations motivated the addition of probabilities to consistency-based diagnosis [de Kleer and Williams, 1989].

Perhaps the closest work to that presented here is that of Stickel [Stickel, 1988]. His is an iterative deepening search for the lowest cost explanation. He does not consider probabilities.

6 Using existing logic programming technology

In this section we show how the branch and bound search can be compiled into Prolog. The basic idea is that when we are choosing a partial explanation to explore, we can choose any of those with maximum probability. If we choose the last one when there is more than one, we carry out a depth-first search much like normal Prolog, except when making assumptions. We only add to the priority queue when making assumptions, and let Prolog do the searching when we are not.

6.1 Remaining subgoals

Consider what subgoals remain to be solved when we are trying to solve a goal. Consider the clause:

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_m.$$

Suppose R is the conjunction of subgoals that remain to be solved after h in the proof. If we are using the leftmost reduction of subgoals, then the conjunction of subgoals remaining to be solved after subgoal b_i is

$$b_{i+1} \wedge \cdots \wedge b_m \wedge R$$

The total information of the proof is contained in the partial explanation at the point we are in the proof, i.e., in the remaining subgoals, current hypotheses and the associated answer. The idea we exploit is to make this set of subgoals explicit by adding an extra argument to each atomic symbol that contains all of the remaining subgoals.

6.2 Saving partial proofs

There is enough information within each subgoal to prove the top level goal it was created to solve. When we have a hypothesis that needs to be assumed, the remaining subgoals and the current hypotheses form a partial explanation which we save on the queue. We then fail the current subgoal and look for another solution. If there are no solutions found (i.e., the top level computation fails), we can choose a saved subgoal (according to the order given in section 3.1), and continue the search.

Suppose in our proof we select a possible hypothesis h of cost $P(\{h\})$ with U being the conjunction of goals remaining to be solved, and T the set of currently assumed hypotheses with cost $P(T)$. We only want to consider this as a possible contender for the best solution if $P(\{h\} \cup T)$ is the minimal cost of all proofs being considered. The minimal cost proofs will be other proofs of cost $P(T)$. These can be found by failing the current subgoal. Before we do this we need to add U, with hypotheses $\{h\} \cup T$ to the priority queue. When the proof fails we know there is no proof with the current set of hypotheses; we remove the partial proof with minimal cost from the priority queue, and continue this proof.

We do a branch and bound search over the partial explanations, but when the priorities are equal, we use Prolog's search to prefer the last added. The overhead on the resolution steps is low; we only have to do a couple more simple unifications (a free variable with a term). The main overhead occurs when we reach a hypothesis. Here we store the hypotheses and remaining goals on a priority queue and continue or search by failing the current goal. This is quick (if we implement the priority queue efficiently); the overhead needed to find all proofs is minimal.

Appendix A gives code necessary to run the search procedure.

7 Conclusion

This paper has considered a logic programming approach that uses a mix between depth-first and branch-and-bound search strategies for abduction where we want to consider probabilities, and only want to generate the most likely explanations. The underlying language is a superset of pure Prolog (without negation-as-failure), and the overhead of executing pure Prolog programs is small.

A Prolog interpreter

This appendix gives a brief overview of a meta-interpreter. Hopefully it is enough to be able to build a system. Our implementation contains more bells and whistles, but the core of it is here.

A.1 Prove

$$prove(G, T_0, T_1, C_0, C_1, U)$$

means that G can be proven with current assumptions T_0, resulting in assumptions T_1, where C_i is the probability of T_i, and U is the set of remaining subgoals.

The first rule defining *prove* is a special purpose rule for the case where we have found an explanation; this reports on the answer found.

```
prove(ans(A),T,T,C,C,_) :- !,
   ans(A,T,C).
```

The remaining rules are the real definition, that follow a normal pattern of Prolog meta-interpreters [Sterling and Shapiro, 1986].

```
prove(true,T,T,C,C,_) :- !.
prove((A,B),T0,T2,C0,C2,U) :- !,
```

```
   prove(A,T0,T1,C0,C1,(B,U)),
   prove(B,T1,T2,C1,C2,U).
prove(H,T,T,C,C,_) :-
   hypothesis(H,PH),
   member(H,T),!.
prove(H,T,[H|T],C,C1,U) :-
   hypothesis(H,PH),
   \+ (( member(H1,T), makeground((H,H1)),
         nogood(H,H1) )),
   C1 is C*PH,
   add_to_PQ(process([H|T],C1,U)),
   fail.
prove(G,T0,T1,C0,C1,U) :-
   rul(G,B),
   prove(B,T0,T1,C0,C1,U).
```

A.2 Rule and disjoint declarations

We specify the rules of our theory using the declaration $rule(R)$ where R is the form of a Prolog rule. This asserts the rule produced.

```
rule((H :- B)) :- !,
   assert(rul(H,B)).
rule(H) :-
   assert(rul(H,true)).
```

The disjoint declaration forms *nogoods* and declares probabilities of hypotheses.

```
:- op(  500, xfx, : ).
disjoint([]).
disjoint([H:P|R]) :-
   assert(hypothesis(H,P)),
   make_disjoint(H,R),
   disjoint(R).

make_disjoint(_,[]).
make_disjoint(H,[H2 : _ | R]) :-
   assert(nogood(H,H2)),
   assert(nogood(H2,H)),
   make_disjoint(H,R).
```

A.3 Explaining

To find an explanation for a subgoal G we execute $explain(G)$. This creates a list of solved explanations and the probability mass found (in "done"), and creates an empty priority queue.

```
explain(G) :-
   assert(done([],0)),
   initQ,
   ex((G,ans(G)),[],1),!.
```

$ex(G, D, C)$ tries to prove G with assumptions D such that probability of D is C. If G cannot be proven, a partial proof is taken from the priority queue and restarted. This means that $ex(G, D, C)$ succeeds if there is some proof that succeeds.

```
ex(G,D,C) :-
   prove(G,D,_,C,_,true).
ex(_,_,_) :-
    remove_from_PQ(process(D,C,U)),!,
    ex(U,D,C).
```

We can report the explanations found, the estimates of the prior probability of the hypothesis, etc, by defining $ans(G, D, C)$, which means that we have found an explanation D of G with probability C.

```
ans(G,[],_) :-
   writeln([G,' is a theorem.']),!.
ans(G,D,C) :-
   allgood(D),
   qmass(QM),
   retract(done(Done,DC)),
   DC1 is DC+C,
   assert(done([expl(G,D,C)|Done],DC1)),
   TC is DC1 + QM,
   writeln(['Probability of ',G,
            ' = [',DC1,',',TC,']']),
   Pr1 is C / TC,
   Pr2 is C / DC1,
   writeln(['Explanation: ',D]),
   writeln(['Prior = ',C]),
   writeln(['Posterior = [',Pr1,',',Pr2,']']).
```

more is a way to ask for more answers. It will take the top priority partial proof and continue with it.

```
more :- ex(fail,_,_).
```

A.4 Auxiliary relations used

The following relations were also used. They can be divided into those for managing the priority queue, and those for managing the nogoods.

We assume that there is a global priority queue into which one can put formulae with an associated cost and from which one can extract the least cost formulae. We assume that the priority queue persists over failure of subgoals. It can thus be implemented by asserting into a Prolog database, but cannot be implemented by carrying it around as an extra argument in a meta-interpreter [Sterling and Shapiro, 1986], for example. We would like both insertion and removal from the priority queue to be carried out in $\log n$ time where n is the number of elements of the priority queue. Thus we cannot implement it by having the queue asserted into a Prolog database if the asserting and retracting takes time proportional to the size of the objects asserted or retracted (which it seems to in the implementations we have experimented with).

Four operations are defined:

$$initQ$$

initialises the queue to be the empty queue, with zero queue mass.

$$add\_to\_PQ(process(D, C, U))$$

adds assumption set D, with probability C and remaining subgoals U to the priority queue. Adds C to the queue mass.

$$remove\_from\_PQ(process(D, C, U))$$

if the priority queue is not empty, extracts the element with highest probability (highest value of C) from the priority queue and reduces the queue mass by C. $remove\_from\_PQ$ fails if the priority queue is empty.

$$qmass(M)$$

returns the sum of the probabilities of elements of the queue.

We assume the relation for handling *nogoods*:

$$allgood(L)$$

fails if L has a subset that has been declared nogood.

Acknowledgements

Thanks to Andrew Csinger, Keiji Kanazawa and Michael Horsch for valuable comments on this paper. This research was supported under NSERC grant OGPOO44121, and under Project B5 of the Institute for Robotics and Intelligent Systems.

References

[Apt and Bezem, 1990] K. R. Apt and M. Bezem. Acyclic programs (extended abstract). In *Logic Programming: Proceedings of the Seventh International Conference*, pages 617–633. MIT Press, 1990.

[Clark, 1978] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[Console *et al.*, 1991] L. Console, D. Theseider Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1991.

[Cox and Pietrzykowski, 1987] P. T. Cox and T. Pietrzykowski. General diagnosis by abductive inference. Technical Report CS8701, Computer Science, Technical University of Nove Scotia, Halifax, April 1987.

[de Kleer and Williams, 1989] J. de Kleer and B. C. Williams. Diagnosis with behavioral modes. In *Proc. 11th International Joint Conf. on Artificial Intelligence*, pages 1324–1330, Detroit, August 1989.

[Goebel *et al.*, 1986] R. Goebel, K. Furukawa, and D. Poole. Using definite clauses and integrity constraints as the basis for a theory formation approach to diagnostic reasoning. In E. Shapiro, editor, *Proc. Third International Conference on Logic Programming*, pages 211–222, London, July 1986.

[Korf, 1985] K. E. Korf. Depth-first iterative deepening: an optimal admissable tree search. *Artificial Intelligence*, 27(1):97–109, September 1985.

[Lloyd, 1987] J. W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation Series. Springer-Verlag, Berlin, second edition, 1987.

[Naish, 1986] L. Naish. *Negation and Control in Prolog.* Lecture Notes in Computer Science 238. Springer Verlag, 1986.

[Pearl, 1988] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA, 1988.

[Pereira and Shieber, 1987] F. C. N. Pereira and S. M. Shieber. *Prolog and Natural-Language Analysis.* Center for the Study of Language and Information, 1987.

[Poole *et al.*, 1987] D. Poole, R. Goebel, and R. Aleliunas. Theorist: A logical reasoning system for defaults and diagnosis. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 331–352. Springer-Verlag, New York, NY, 1987.

[Poole, 1987] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1987.

[Poole, 1988] D. Poole. Representing knowledge for logic-based diagnosis. In *International Conference on Fifth Generation Computing Systems*, pages 1282–1290, Tokyo, Japan, November 1988.

[Poole, 1991a] D. Poole. Compiling a default reasoning system into Prolog. *New Generation Computing Journal*, 9(1):3–38, 1991.

[Poole, 1991b] D. Poole. Representing Bayesian networks within probabilistic Horn abduction. In *Proc. Seventh Conf. on Uncertainty in Artificial Intelligence*, pages 271–278, Los Angeles, July 1991.

[Poole, 1991c] D. Poole. Representing diagnostic knowledge for probabilistic Horn abduction. In *Proc. 12th International Joint Conf. on Artificial Intelligence*, pages 1129–1135, Sydney, August 1991.

[Poole, 1992a] D. Poole. Probabilistic Horn abduction and Bayesian networks. Technical Report 92-2, Department of Computer Science, University of British Columbia, January 1992.

[Poole, 1992b] D. Poole. Search for computing posterior probabilities in Bayesian networks. *Proc. Eighth Conf. on Uncertainty in Artificial Intelligence*, submitted, Stanford, California, July 1992.

[Pople, 1973] H. E. Pople, Jr. On the mechanization of abductive logic. In *Proc. 3rd International Joint Conf. on Artificial Intelligence*, pages 147–152, Stanford, August 1973.

[Reiter and de Kleer, 1987] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: preliminary report. In *Proc. 6th National Conference on Artificial Intelligence*, pages 183–188, Seattle, July 1987.

[Sattar and Goebel, 1991] A. Sattar and R. Goebel. Using crucial literals to select better theories. *Computational Intelligence*, 7(1):11–22, February 1991.

[Sterling and Shapiro, 1986] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, MA, 1986.

[Stickel, 1988] M. E. Stickel. A prolog-like inference system for computing minimum-cost abductive explanations in natural language interpretations. Technical Note 451, SRI International, Menlo Park, CA, September 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Abduction in Logic Programming with Equality

P.T. Cox, E. Knill, T. Pietrzykowski

Technical University of Nova Scotia, School of Computer Science
P.O. Box 1000, Halifax, Nova Scotia
Canada B3J 2X4

Abstract

Equality can be added to logic programming by using surface deduction. Surface deduction yields interpretations of unification failures in terms of residual hypotheses needed for unification to succeed. It can therefore be used for abductive reasoning with equality. In surface deduction the input clauses are first transformed to a flat form (involving no nested terms) and symmetrized (if necessary). They are then manipulated by binary resolution, a restricted version of factoring and compression. The theoretical properties of surface deduction, including refutation completeness and weak deductive completeness properties (relative to equality), are established in [Cox *et al.* 1991]. In this paper we show that these properties imply that an enhancement of surface deduction will yield all parsimonious hypotheses when used as an abductive inference engine. The characterization of equational implication for goal clauses given in [Cox *et al.* 1991] is shown to yield a uniquely defined equationally equivalent residuum for every goal clause. The residuum naturally represents the corresponding abductive hypothesis. An example illustrating the use of surface deduction in abductive reasoning is presented.

1 Introduction

In abductive reasoning, the task is to explain a given observation by introducing appropriate hypotheses ([Cox and Pietrzykowski 1987], [Goebel 1990]). Most presentations of abduction do not include reasoning with equality, nor do they allow the introduction of equality assumptions to explain an observation. A notable exception is E. Charniak's work on motivation analysis [Charniak 1988]. Charniak allows the introduction of certain restricted equality assumptions to determine motivations for observed actions. He shows that the introduction of such equality assumptions is required to successfully abduce motivations. In this paper we consider the problem of abductive reasoning with Horn clauses in the presence of equality. We show that surface deduction has the necessary properties for use in an abductive inference system provided that the input theory contains the function substitutivity axioms.

In the presence of equality, an abduction problem consists of a theory $\mathcal{T}$ and a formula O (the *observation*). An *explanation* of $(O, \mathcal{T})$ is a formula E consistent with $\mathcal{T}$ such that E together with $\mathcal{T}$ equationally implies O. We will assume that O and E are existentially quantified conjunctions of facts and that $\mathcal{T}$ is a Horn clause theory.

One way to obtain an explanation E, given an observation O and a theory $\mathcal{T}$, is to deduce $\neg E$ from $\mathcal{T}$ and $\neg O$. Since explanations with less irrelevant information are preferred (the *parsimony principle*), it is sufficient to deduce a clause $\neg E'$ such that $\neg E'$ implies $\neg E$. Intuitively, E' is at least as good an explanation as E (see Section 4). It follows that a deduction system adequate for abductive reasoning should satisfy a weak deductive completeness: If the theory $\mathcal{T}$ implies a non-tautological clause $\neg E$, then we must be able to deduce a clause $\neg E'$ from $\mathcal{T}$ such that $\neg E'$ implies $\neg E$. In the absence of equality, SLD-resolution (see [Lloyd 1984]) satisfies this condition.

The problem of introducing equality to Horn clause logic has been well-studied, see [Hölldobler 1989] for an excellent overview. The simplest approach to this problem involves adding the equality axioms (which are Horn clauses) to the set of input clauses. However, unrestricted use of these axioms results in inefficiency. Furthermore, this approach does not yield any insights into the degree to which the equality axioms are needed. Paramodulation and other term rewriting systems do not explicitly introduce new equality assumptions into derivations and therefore do not satisfy the weak deductive completeness condition. Other approaches, such as the ones in [van Emden and Lloyd 1984] and extended in [Hoddinott and Elcock 1986] using the homogeneous form of clauses, require restricting the form of the input theory. Here, we use the results of [Cox *et al.* 1991] to show that if equality is introduced to Horn clause logic via surface deduction with the function substitutivity axioms, then all preferred explanations for an abduction problem can be obtained. The need for axioms of equality other than function substitutivity is thus eliminated.

In surface deduction, a set of input clauses is first transformed to a flat form and symmetrized. The deduction then proceeds using linear input resolution for Horn clauses (see [Lloyd 1984]) together with a limited use of factoring and a new rule called compression. The additional deduction rules are equivalent to those restricted uses of the reflexivity axiom ($x \doteq x \mathrel{:-}$) which preserve flatness. They are required only at the end of a deduction.

A clause is flat if it has no nested functional expressions, and every variable which appears immediately to the right of an equality symbol ($\doteq$) appears only in such positions. A stronger version of flatness requires that in addition the clause is separated. This means that every variable appears at most once in any given literal and has only one occurrence inside a functional or relational expression. Symmetrization affects only those clauses with equalities in their heads (see Section 3).

The idea of using flattening to add equality to theorem proving is due to [Brand 1975] and is applied to logic programming in [Cox and Pietrzykowski 1986] where surface deduction is defined. Flattening is closely related to narrowing. In narrowing the process of flattening is implicit in the deduction rules. The relationship between the two methods is examined in [Bosco *et al.* 1988]. Separation of terms is implicit in the transformations to the homogeneous forms of [Hoddinott and Elcock 1986]. The symmetrization method used here is similar to the one introduced in [Chan 1986] and does not increase the number of clauses in the theory.

In [Cox *et al.* 1991] it is shown that surface deduction satisfies a weak deductive completeness provided that the input clauses are first transformed to separated form. As an application of this result, equational implication for goal clauses is found to have a simple syntactic characterization analogous to subsumption.

Once an explanation E is obtained by surface deduction, in what form should E be presented? For example if $\neg E$ (the actual clause deduced) is given by

$$\mathrel{:-} x \doteq a, y \doteq b, y \doteq c,$$

then $\mathrel{:-} y \doteq b, y \doteq c$ is equationally equivalent to $\neg E$. Therefore the atom $x \doteq a$ is irrelevant and should be removed. In Section 4 it is shown that the characterization of equational implication for goal clauses given in [Cox *et al.* 1991] implies that for every goal clause C there is a uniquely defined equational residuum RES(C) which cannot be further reduced without weakening the corresponding explanation. The notion of equational residuum is related to that of prime implicates used in switching theory [Kohavi 1978], truth maintenance systems [Reiter and de Kleer 1987] and diagnoses [de Kleer *et al.* 1988]. RES(C) is an equational prime implicate of a flattening of C.

In Section 2 the terminology is established; in Section 3 surface deduction is defined and the completeness results needed for abductive reasoning are given. In Section 4 the formalism of abductive reasoning with surface deduction is discussed; and finally in Section 5 an example is presented of an abductive problem solved by using surface deduction.

2 Preliminaries

Familiarity with logic programming is assumed (see e.g. [Lloyd 1984]). As in [Hölldobler 1990], let $\doteq$ denote the equality predicate symbol. The usual equality symbol $=$ is used exclusively for syntactic equality. If L is an atom and $C = \{M_1, \ldots, M_n\}$ is a set of atoms, then $L \mathrel{:-} C$ denotes the Horn clause $L \vee \neg M_1 \vee \ldots \neg M_n$. In this expression, L is the *head* and C is the *body* of the clause. A clause of the form $\mathrel{:-} C$ is a *goal* clause. The atoms of C are the subgoals of $\mathrel{:-} C$. A clause of the form $L \mathrel{:-}$ is a *fact*. If $C_1, \ldots, C_n$ are sets of atoms and C is the union of the C_i, then $L \mathrel{:-} C_1, \ldots, C_n$ means $L \mathrel{:-} C$. When possible, set notation is omitted for one-element sets.

If OP is an operation which maps clauses to clauses and $\mathcal{A}$ is a set of clauses, then $\mathrm{OP}(\mathcal{A}) = \{\mathrm{OP}(C) \mid C \in \mathcal{A}\}$. Let σ be a substitution. If $x_i\sigma = t_i$ for $i = 1, \ldots, n$ and $x\sigma = x$ for all other variables, then σ is denoted by $\{x_1 \leftarrow t_1, \ldots x_n \leftarrow t_n\}$. A substitution σ is *variable-pure* iff $x\sigma$ is a variable for every variable x.

The expression 'most general unifier' is abbreviated by 'mgu'. An *equality* is an atom of the form $s \doteq t$. Let $\mathcal{E}$ be the set of equality axioms other than $x \doteq x \mathrel{:-}$. If $\mathcal{A}$ and $\mathcal{B}$ are sets of clauses, then $\mathcal{A}$ *satisfies* (or *implies*) $\mathcal{B}$ iff every model of $\mathcal{A}$ is a model of $\mathcal{B}$. $\mathcal{A}$ *equationally* satisfies (or implies) $\mathcal{B}$ iff $\mathcal{A} \cup \mathcal{E} \cup \{x \doteq x \mathrel{:-}\}$ satisfies $\mathcal{B}$. $\mathcal{A}$ and $\mathcal{B}$ are (equationally) *equivalent* iff each (equationally) satisfies the other. $\mathcal{A}$ is equationally inconsistent iff $\mathcal{A}$ equationally implies the empty clause.

3 Surface Deduction

In surface deduction, a refutation of a set of input clauses proceeds by first transforming the input clauses to a flat form and then refuting the result using resolution, factoring and compression. The transformation subsumes the equality axioms other than reflexivity. The rules of factoring and compression subsume reflexivity.

Definition. Let C be a clause and t a term. An occurrence of t on the left-hand side (right-hand side) of an equality $t \doteq s$ ($s \doteq t$) in C is a *root* (*surface*) occurrence of t in C. Every other occurrence of t is an *internal* occurrence of t. The term t is a *root term* of C iff it has a root occurrence in C. *Surface* and *internal terms* are defined analogously.

Definition. A clause C is *flat* iff

(i) every atom of C is of the form $P(x_1,\ldots,x_n)$, $x \doteq f(x_1,\ldots,x_n)$ or $x \doteq y$, and

(ii) no surface variable of C is a root or internal variable of C.

Definition. Let C be a Horn clause. An *elementary flattening* of C is obtained by either

(i) replacing some of the non-surface occurrences of a non-variable term t by a new variable y and adding the equality $y \doteq t$ to the body,

or

(ii) replacing some of the surface occurrences of a root or internal variable x of C by a new variable y and adding the equality $x \doteq y$ to the body.

An elementary flattening of the set of clauses $\mathcal{A}$ is obtained by replacing a clause in $\mathcal{A}$ by an elementary flattening of that clause.

Modifying a clause C by successive elementary flattenings eventually results in a flat clause (a *flattening* of C) which cannot be flattened any further (Theorem 2 of [Cox and Pietrzykowski 1986]).

Definition. Let C be a clause. Then FLAT(C) denotes a (arbitrary but fixed) flattening of C.

For any set of clauses $\mathcal{A}$, FLAT($\mathcal{A}$) is equationally equivalent to $\mathcal{A}$. In [Cox *et al.* 1991] it is shown that for refutation completeness the transformation FLAT subsumes the substitutivity axioms but not transitivity and symmetry.

In order to subsume transitivity and symmetry, we need another transformation.

Definition. Let C be a clause with an equality in its head. Then C is *symmetric* iff C is of the form

$$x \doteq u \text{ :- } x \doteq v, s \doteq v, y \doteq u, y \doteq t, M$$

for some terms s and t and set of atoms M, where x, y, u and v do not occur in M, s or t. The set of clauses $\mathcal{A}$ is *symmetrized* iff every clause C of $\mathcal{A}$ with an equality in its head is symmetric.

Definition. Let C be a Horn clause. If C does not have an equality in its head or if C is symmetric, then the *symmetrization* SYM(C) of C is C. If C is not symmetric and of the form $s \doteq t$:- M, then SYM(C) is given by

$$x \doteq u \text{ :- } x \doteq v, s \doteq v, y \doteq u, y \doteq t, M.$$

Note that if $\mathcal{A}$ is a set of Horn clauses, then SYM($\mathcal{A}$) is equationally equivalent to $\mathcal{A}$, and if $\mathcal{A}$ is flat, then SYM($\mathcal{A}$) is flat. In [Cox *et al.* 1991] it is shown that the transformation SYM subsumes transitivity and symmetry. In order to subsume substitutivity, transitivity and symmetry, the transformations SYM and FLAT are composed.

Flattening and symmetrization followed by SLD-resolution using resolution with $x \doteq x$:- as an additional deduction rule is refutation complete for logic programming with equality. However, weak deductive completeness is not satisfied [Cox *et al.* 1991]. In order to obtain weak deductive completeness an additional transformation is required.

Definition. A *positive (negative) root* occurrence of the term t in the clause C is a root occurrence in the head (body) of C.

Definition. The flat clause C is *separated* in the variable x iff

(i) every literal of C has at most one occurrence of x,

(ii) C has at most one internal occurrence of x, and

(iii) if x has an internal occurrence in C, then x has a negative root occurrence in C.

The clause C is separated iff C is separated in all its variables.

If $\mathcal{A}$ is a set of separated flat Horn clauses, then SYM($\mathcal{A}$) is separated. Separated clauses can be obtained from a given flat clause by using the transformation SEP:

Definition. Let C be a flat clause and x a variable. The clause SEP(C) is the separated flat clause obtained by applying the following transformation to C: For every variable x such that C is not separated in x, replace each internal occurrence of x by a new variable x_i and add the equalities $x \doteq y, x_1 \doteq y, x_2 \doteq y, \ldots$ to the body of C (where y is a new surface variable).

The rules of factoring and compression used in surface deduction are:

(i) *Root factoring.* The clause C' is a root factor of C iff C' is obtained by factoring two equalities of C with the same root variable.

(ii) *Surface factoring.* The clause C' is a surface factor of C iff C' is obtained by factoring two equalities of C with the same surface term.

(iii) *Root compression.* The clause C' is a root compression of C iff C' is obtained by removing an equality $x \doteq t$ from the body of C, where x has only one occurrence in C.

(iv) *Surface compression.* The clause C' is a surface compression of C iff C' is obtained by removing an equality $x \doteq y$ from the body of C, where y has only one occurrence in C.

A *compression* is a root or surface compression. A *compression* of a clause C is a clause C' obtained from C by a sequence of applications of compression rules.

The soundness of root and surface factoring and compression (in the presence of equality) is shown in [Cox and Pietrzykowski 1986]. Observe that binary resolution, surface and root factoring and compression preserve flatness. The relationship between factoring, compression and resolution with the reflexivity axiom is determined by the following result (proved implicitly in [Cox and Pietrzykowski 1986] and explicitly in [Cox *et al.* 1991]; see also [Hoddinott and Elcock 1986]):

Theorem 3.1 *Let $:\!- C$ be a flat goal clause. If $:\!- C'$ is a flat goal clause obtained from $:\!- C$ by a sequence of binary resolutions with $x \doteq x$ $:\!-$, then $:\!- C'$ can be obtained from $:\!- C$ by a sequence of root and surface factorings and compressions.*

Definition. Let $\mathcal{A}$ be a set of flat Horn clauses. The flat goal clause C is *S-deducible* from $\mathcal{A}$ iff C can be obtained from $\mathcal{A}$ by a sequence of binary resolutions, surface and root factorings and compressions. Note that we can assume that the deduction is linear. $\mathcal{A}$ is *S-refutable* iff the empty clause is S-deducible from $\mathcal{A}$.

To state the weak deductive completeness result for flat, separated and symmetrized clauses, we need the transformation defined next.

Definition. Let $:\!- C$ be a flat goal clause. Then $:\!- C$ is *reduced* iff $:\!- C$ has no surface variables and no two equalities of $:\!- C$ have the same right-hand sides. A flat reduced clause REDU($:\!- C$) is obtained from $:\!- C$ by factoring equalities with identical right-hand sides until all right-hand sides are distinct, and by removing all remaining equalities with surface variables by surface compression. Note that for every flat goal clause $:\!- C$, REDU($:\!- C$) is equationally equivalent to $:\!- C$.

Theorem 3.2 [Cox *et al.* 1991] *Let $:\!- C$ be a goal clause and $\mathcal{A}$ a set of Horn clauses which includes the function substitutivity axioms. Then $\mathcal{A}$ equationally implies $:\!- C$ iff there is a flat goal clause $:\!- C'$ such that for some variable-pure substitution σ, $:\!- C'\sigma \subseteq$ REDU(FLAT($:\!- C$)) and $:\!- C'$ is S-deducible from* SYM(SEP(FLAT($\mathcal{A}$))).

As an application of this result, the following theorem is proved in [Cox *et al.* 1991]:

Theorem 3.3 *Let $:\!- A$ and $:\!- B$ be goal clauses. Then $:\!- A$ equationally implies $:\!- B$ iff there is a variable-pure substitution σ such that a compression of* FLAT($:\!- A$)σ *is included in* REDU(FLAT($:\!- B$)).

Definition. Let $:\!- C$ be a goal clause. An *equational residuum* of $:\!- C$ is a minimal subclause of REDU(FLAT($:\!- C$)) which is equationally equivalent to $:\!- C$.

Every equational residuum of $:\!- C$ is equationally equivalent to $:\!- C$. The fact that every subclause of a reduced clause is reduced implies that if $:\!- C'$ is an equational residuum of $:\!- C$, then $:\!- C'$ is reduced. The next theorem shows that the equational residuum is unique.

Theorem 3.4 [Cox *et al.* 1991] *Let $:\!- A'$ and $:\!- B'$ be equational residua of the goal clauses $:\!- A$ and $:\!- B$ respectively. Then $:\!- A$ is equationally equivalent to $:\!- B$ iff $:\!- A'$ is a variant of $:\!- B'$.*

4 Abduction using Surface Deduction

An *existential conjunction of facts* is a conjunction of facts with all its free variables quantified existentially. The abduction problem for Horn clause logic with equality can be stated as follows:

Abduction Problem: An abduction problem is a pair $(\mathcal{A}, O)$, where $\mathcal{A}$ is a *theory* of Horn clauses and O (the *observation*) is an existential conjunction of facts. An *explanation* of the abduction problem $(\mathcal{A}, O)$ is an existential conjunction of facts E consistent with $\mathcal{A}$ such that E and $\mathcal{A}$ equationally imply O.

Let $\neg O$ and $\neg E$ denote the disjunctions of the negations of the constituent facts of O and E respectively. Since E and $\mathcal{A}$ equationally imply O iff $\neg O$ and $\mathcal{A}$ equationally imply $\neg E$, a solution to an abduction problem can be obtained by deducing a clause C from $\mathcal{A}$ and $\neg O$, and negating C to obtain E.

In general, it is desirable for an explanation E of an abductive problem $(\mathcal{A}, O)$ to have certain additional properties (see [Cox and Pietrzykowski 1987]). For example, an explanation E should not contain any facts not required to yield the observation from $\mathcal{A}$ (the parsimony principle). Thus if E and E' are explanations of $(\mathcal{A}, O)$ and E equationally implies E', E' is preferred over E. (Here 'preferred' is to be understood as 'at least as good as'.)

For abduction, a desirable property of a deduction system is that for every explanation E of an abductive problem $(\mathcal{A}, O)$, one can obtain an explanation preferred over E. The weak completeness result of Theorem 3.2 implies that surface deduction with separated clauses and the function substitutivity axioms has this property.

Theorem 4.1 *Let $(\mathcal{A}, O)$ be an abductive problem, where $\mathcal{A}$ contains the function substitutivity axioms. Then for every explanation E of $(\mathcal{A}, O)$, there is an explanation E' preferred over E such*

that $\neg E'$ *is S-deducible from* SYM(SEP(FLAT($\mathcal{A}$))) $\cup$ {SEP(FLAT($\neg O$))}.

Proof. This follows by Theorem 3.2 and the fact that $\neg O$ is a goal clause, so that it does not need to be symmetrized. ∎

Fortunately, it appears that the function substitutivity axioms are rarely needed in abductive problems when using surface deduction with separated clauses.

Flattenings of a clause can be viewed as alternate representations of the clause's term structure and are therefore essentially equivalent. Without loss of generality we restrict our attention to explanations E such that $\neg E$ is flat (*flat* explanations).

If E and E' are explanations of $(\mathcal{A}, O)$ such that E equationally implies E' but is not equationally equivalent to E', then E' is strictly preferred over E. Given an explanation E of $(\mathcal{A}, O)$ there are many equationally equivalent existential conjunctions of facts, all of which are also explanations of $(\mathcal{A}, O)$. The preference criteria introduced so far do not distinguish among equationally equivalent explanations. Using the intuition that a "simpler" explanation should be preferred, we give a stronger definition of preference:

Definition. Let E and E' be flat explanations. Then E' is strictly preferred over E iff either E equationally implies E' but is not equivalent to E', or E is equationally equivalent to E' and E' has fewer atoms.

Given these preference criteria, we have the following theorem which determines the most preferred flat explanation among equationally equivalent ones:

Theorem 4.2 *For any explanation* E*, if* E' *is the negation of the equational residuum of* $\neg E$*, then* E' *is the unique most preferred flat explanation among flat explanations equationally equivalent to* E*.*

Proof. Let $:\!- A$ be a flat clause equationally equivalent to $\neg E$. If $:\!- A$ is not reduced, then REDU($:\!- A$) has fewer atoms than $:\!- A$ and the corresponding explanation is therefore strictly preferred. Assume that $:\!- A$ is reduced. If the equational residuum of $:\!- A$ is not given by $:\!- A$, then the equational residuum of $:\!- A$ has fewer atoms than $:\!- A$, so that the corresponding explanation is strictly preferred. The result now follows by the uniqueness theorem for equational residua, Theorem 3.4. ∎

5 An Application

Examples from the domain of story comprehension and motivation analysis which demonstrate the need for the inclusion of equality in abductive reasoning are given in [Charniak 1988]. Here we give an example from a different domain.

Consider the following (imaginary, but realistic) situation. A researcher X experimentally determines the value of a quantity associated with a physical object (e.g. the mass of an isotope of an element) and sends us the result. We have independently obtained a value for the same quantity (by theory and/or experiment) and our value differs from X's value. We believe our value to be correct and we would like to explain the discrepancy. We do not know the exact means by which X's value was obtained, but we know what kinds of experimental apparatus X might have used. One kind of apparatus (type A) is notorious for a hard-to-control drift in the settings which results in a systematic bias in the readings. Thus we can explain the discrepancy between our and X's values by hypothesizing that X used apparatus of type A with a systematic bias equal to the difference between the two values.

The situation is formalized as follows: Let $TA(x)$ mean that x is an apparatus of type A. Let $Vt(y)$ be the true value of quantity y, $Vm(z,y)$ the value of quantity y measured in experiment z, $A(u)$ the apparatus used in experiment u and $B(x)$ the systematic bias of apparatus x. The quantity measured by X is q, and the experiment performed by X is given the name e. With these definitions, our knowledge $\mathcal{T}$ consists of the clauses

T1: $Vt(q) \doteq 0 :\!-$

T2: $Vm(x_1, x_2) \doteq Vt(x_2) + B(A(x_1)) :\!- TA(A(x_1))$

T3: $x_1 \doteq 0 + x_1 :\!-$

⋮

where knowledge about other types of apparatus and theorems about real numbers other than T3 have been omitted. The observation O is given by

O: $Vm(e, q) \doteq 2 :\!-$

The first task is to obtain a flattening of $\mathcal{T}$ and the negation of the observation:

fT1: $x_1 \doteq 0 :\!- x_1 \doteq Vt(x_2), x_2 \doteq q.$

fT2: $x_4 \doteq x_5 + x_6 :\!- TA(x_3),\ x_6 \doteq B(x_3),\ x_4 \doteq Vm(x_1, x_2),\ x_5 \doteq Vt(x_2),\ x_3 \doteq A(x_1).$

fT3: $x_1 \doteq x_2 + x_1 :\!- x_2 \doteq 0.$

fO: $:\!- x_1 \doteq 2,\ x_1 \doteq Vm(x_2, x_3),\ x_2 \doteq e,\ x_3 \doteq q.$

The clauses fT1 and fO are separated. Separated clauses for fT2 and fT3 are given by

sfT2: $x_4 \doteq x_5 + x_6 :\!- TA(x_3),\ x_6 \doteq B(x_7),\ x_3 \doteq x_8,\ x_7 \doteq x_8,\ x_4 \doteq Vm(x_1, x_2),\ x_5 \doteq Vt(x_{10}),\ x_2 \doteq x_9,\ x_{10} \doteq x_9,\ x_3 \doteq A(x_{11}),\ x_1 \doteq x_{12},\ x_{11} \doteq x_{12}.$

sfT3: $x_1 \doteq x_2 + x_3 :\!-\ x_3 \doteq x_4,\ x_1 \doteq x_4,\ x_2 \doteq 0.$

All clauses of T have equalities in their heads and need to be symmetrized. The fully transformed set of clauses is given by

T1′: $x_3 \doteq x_4 \text{ :- } x_3 \doteq x_5,\ x_1 \doteq x_5,\ x_6 \doteq x_4,\ x_6 \doteq 0,\ x_1 \doteq Vt(x_2),\ x_2 \doteq q.$

T2′: $x_{13} \doteq x_{14} \text{ :- } x_{13} \doteq x_{15},\ x_4 \doteq x_{15},\ x_{16} \doteq x_{14},\ x_{16} \doteq x_5 + x_6,\ TA(x_3),\ x_6 \doteq B(x_7),\ x_3 \doteq x_8,\ x_7 \doteq x_8,\ x_4 \doteq Vm(x_1, x_2),\ x_5 \doteq Vt(x_{10}),\ x_2 \doteq x_9,\ x_{10} \doteq x_9,\ x_3 \doteq A(x_{11}),\ x_1 \doteq x_{12},\ x_{11} \doteq x_{12}.$

T3′: $x_5 \doteq x_6 \text{ :- } x_5 \doteq x_7,\ x_1 \doteq x_7,\ x_8 \doteq x_6,\ x_8 \doteq x_2 + x_3,\ x_3 \doteq x_4,\ x_1 \doteq x_4,\ x_2 \doteq 0.$

O′: $\text{ :- } x_1 \doteq 2,\ x_1 \doteq Vm(x_2, x_3),\ x_2 \doteq e,\ x_3 \doteq q.$

The negation of the desired explanation can now be deduced from O′. In the deduction below, the literals involved in each step are underlined. As is usually the case, the function substitutivity axioms are not needed.

O′ $\text{ :- } \underline{x_1 \doteq 2},\ x_1 \doteq Vm(x_2, x_3),\ x_2 \doteq e,\ x_3 \doteq q.$

res. with T2′ $\text{ :- } \underline{x_1 \doteq x_{18}},\ \underline{x_7 \doteq x_{18}},\ x_{19} \doteq 2,\ x_{19} \doteq x_8 + x_9,\ TA(x_6),\ x_9 \doteq B(x_{10}),\ x_6 \doteq x_{11},\ x_{10} \doteq x_{11},\ \underline{x_7 \doteq Vm(x_4, x_5)},\ x_8 \doteq Vt(x_{13}),\ x_5 \doteq x_{12},\ x_{13} \doteq x_{12},\ x_6 \doteq A(x_{14}),\ x_4 \doteq x_{15},\ x_{14} \doteq x_{15},\ \underline{x_1 \doteq Vm(x_2, x_3)},\ x_2 \doteq e,\ x_3 \doteq q.$

surf. fact. followed by root fact. and compr. $\text{ :- } x_{19} \doteq 2,\ x_{19} \doteq x_8 + x_9,\ TA(x_6),\ x_9 \doteq B(x_{10}),\ x_6 \doteq x_{11},\ x_{10} \doteq x_{11},\ \underline{x_8 \doteq Vt(x_{13})},\ x_3 \doteq x_{12},\ x_{13} \doteq x_{12},\ \underline{x_6 \doteq A(x_{14})},\ x_2 \doteq x_{15},\ x_{14} \doteq x_{15},\ x_2 \doteq e,\ x_3 \doteq q.$

res. with T1′ $\text{ :- } x_{19} \doteq 2,\ x_{19} \doteq x_8 + x_9,\ TA(x_6),\ x_9 \doteq B(x_{10}),\ x_6 \doteq x_{11},\ x_{10} \doteq x_{11},\ x_8 \doteq x_{24},\ x_{20} \doteq x_{24},\ \underline{x_{25} \doteq Vt(x_{13})},\ x_{25} \doteq 0,\ \underline{x_{20} \doteq Vt(x_{21})},\ \underline{x_{21} \doteq q},\ \underline{x_3 \doteq x_{12}},\ \underline{x_{13} \doteq x_{12}},\ x_6 \doteq A(x_{14}),\ x_2 \doteq x_{15},\ x_{14} \doteq x_{15},\ x_2 \doteq e,\ \underline{x_3 \doteq q}.$

surf. fact. and compr. $\text{ :- } x_{19} \doteq 2,\ \underline{x_{19} \doteq x_8 + x_9},\ TA(x_6),\ x_9 \doteq B(x_{10}),\ x_6 \doteq x_{11},\ x_{10} \doteq x_{11},\ x_8 \doteq x_{24},\ x_{20} \doteq x_{24},\ x_{20} \doteq 0,\ x_{20} \doteq Vt(x_3),\ x_6 \doteq A(x_{14}),\ x_2 \doteq x_{15},\ x_{14} \doteq x_{15},\ x_2 \doteq e,\ x_3 \doteq q.$

res. with T3′ $\text{ :- } x_{19} \doteq 2,\ x_{19} \doteq x_{31},\ x_{25} \doteq x_{31},\ \underline{x_{32} \doteq x_8 + x_9},\ \underline{x_{32} \doteq x_{26} + x_{27}},\ x_{27} \doteq x_{28},\ x_{25} \doteq x_{28},\ \underline{x_{26} \doteq 0},\ TA(x_6),\ x_9 \doteq B(x_{10}),\ x_6 \doteq x_{11},\ x_{10} \doteq x_{11},\ \underline{x_8 \doteq x_{24}},\ \underline{x_{20} \doteq x_{24}},\ \underline{x_{20} \doteq 0},\ x_{20} \doteq Vt(x_3),\ x_6 \doteq A(x_{14}),\ x_2 \doteq x_{15},\ x_{14} \doteq x_{15},\ x_2 \doteq e,\ x_3 \doteq q.$

root fact., surf. fact. and compr. $\text{ :- } \underline{x_{19} \doteq 2},\ \underline{x_{19} \doteq x_{31}},\ \underline{x_{25} \doteq x_{31}},\ \underline{x_9 \doteq x_{28}},\ \underline{x_{25} \doteq x_{28}},\ TA(x_6),\ x_9 \doteq B(x_{10}),\ x_6 \doteq x_{11},\ x_{10} \doteq x_{11},\ x_8 \doteq 0,\ x_8 \doteq Vt(x_3),\ x_6 \doteq A(x_{14}),\ x_2 \doteq x_{15},\ x_{14} \doteq x_{15},\ x_2 \doteq e,\ x_3 \doteq q.$

root fact., surf. fact., and compr. $\text{ :- } x_9 \doteq 2,\ TA(x_6),\ x_9 \doteq B(x_{10}),\ x_6 \doteq x_{11},\ x_{10} \doteq x_{11},\ \underline{x_8 \doteq 0},\ x_8 \doteq Vt(x_3),\ x_6 \doteq A(x_{14}),\ x_2 \doteq x_{15},\ x_{14} \doteq x_{15},\ x_2 \doteq e,\ x_3 \doteq q.$

res. with T1′ $\text{ :- } x_9 \doteq 2,\ TA(x_6),\ x_9 \doteq B(x_{10}),\ x_6 \doteq x_{11},\ x_{10} \doteq x_{11},\ \underline{x_8 \doteq x_{21}},\ \underline{x_{17} \doteq x_{21}},\ \underline{x_{22} \doteq 0},\ \underline{x_{17} \doteq Vt(x_{18})},\ \underline{x_{18} \doteq q},\ \underline{x_8 \doteq Vt(x_3)},\ x_6 \doteq A(x_{14}),\ x_2 \doteq x_{15},\ x_{14} \doteq x_{15},\ x_2 \doteq e,\ x_3 \doteq q.$

surf. fact., root fact. and compr. $\text{ :- } x_9 \doteq 2,\ TA(x_6),\ x_9 \doteq B(x_{10}),\ x_6 \doteq x_{11},\ x_{10} \doteq x_{11},\ x_6 \doteq A(x_{14}),\ x_2 \doteq x_{15},\ x_{14} \doteq x_{15},\ x_2 \doteq e,\ x_3 \doteq q.$

reduction to the min. residuum $\text{ :- } x_6 \doteq A(x_2),\ x_2 \doteq e,\ TA(x_6),\ x_9 \doteq B(x_6),\ x_9 \doteq 2.$

The last clause is the negation of the desired explanation. Note how two resolutions with T1′ were used to simulate symmetry.

6 Conclusion

From a theoretical perspective, surface deduction is very appealing in its simplicity. We have seen how (at least in theory) surface deduction can be applied in situations such as abductive reasoning where deduction rather than refutation is the primary goal.

If the equality theory of interest contains function substitutivity, a problem with using surface deduction for abduction is that in general the function substitutivity axioms are still required. Current research indicates that to a large extent, the function substitutivity axioms can be ignored in abductive problems when using surface deduction with symmetrized, separated and flat clauses. We do not know any practical example where this is not the case.

From a practical point of view, one of the frequently recognized problems with flattening the clauses of the input theory is that one loses most of the advantages of unification, particularly if the input theory contains few equalities. One can regain some of these advantages in practice by interpreting the set of equalities in the body of a clause as a directed graph or hypergraph (with arcs from the root variables to the surface terms) which defines the set of possible definitions of the main terms and variables of the clause. Such a directed graph generalizes the usual tree representation of terms. Unification and more generally term rewriting can then be replaced by (hyper)graph rewriting rules. To implement

this idea, the deduction procedures must be substantially enhanced. The types of graph rewriting rules and graph representations needed require further research.

The preference criteria for explanations given in Section 4 are very weak. However, we believe that no matter what preference criteria are used, RES(C) is at least as good an explanation as C. One of the most important problems in abductive reasoning is to determine stronger preference criteria to avoid combinatorial explosion. These issues are discussed in [Poole and Provan 1990].

Many of the results used in this paper can be generalized to arbitrary clauses so that the restriction of abductive reasoning to Horn clause theories can be removed. These generalizations will be the topic of a forthcoming paper.

References

[Baxter 1976] L. D. Baxter. *The Complexity of Unification.* Ph.D. Thesis, University of Waterloo, 1976.

[Bosco *et al.* 1988] P. G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-Resolution. *Theoretical Computer Science* Vol. 59 (1988), pp. 3–23.

[Brand 1975] D. Brand. Proving Theorems with the Modification Method. *SIAM J. Comput.* Vol. 4 (1975), pp. 412–430.

[Chan 1986] K. H. Chan. Equivalent Logic Programs and Symmetric Homogeneous Forms of Logic Programs with Equality. Technical Report 86, Dept. of Computer Science, Univ. of Western Ontario, London, Ont., Canada, 1986.

[Charniak 1988] E. Charniak. Motivation Analysis, Abductive Unification, and Nonmonotonic Equality. *Artificial Intelligence* Vol. 34 (1988), pp. 275–295.

[Colmerauer *et al.* 1982] A. Colmerauer *et al. Prolog II: Reference Manual and Theoretical Model.* Groupe d'Intelligence Artificielle, Faculte des Sciences de Luminy, Marseilles, 1982.

[Cox and Pietrzykowski 1985] P. T. Cox and T. Pietrzykowski. Surface Deduction: a Uniform Mechanism for Logic Programming. In *Proc. Symp. On Logic Programming*, IEEE Press, Washington, 1985. pp. 220–227.

[Cox and Pietrzykowski 1986] P. T. Cox and T. Pietrzykowski. Incorporating Equality into Logic Programming via Surface Deduction. *Ann. Pure Appl. Logic*, Vol. 31 (1986), pp. 177–189.

[Cox and Pietrzykowski 1987] P. T. Cox and T. Pietrzykowski. General Diagnosis by Abductive Inference. In *Proceedings of the Symposium on Logic Programming*, IEEE Press, Washington, 1987. pp. 183–189.

[Cox *et al.* 1991] P.T. Cox, E. Knill, and T. Pietrzykowski. Equality and Abductive Residua for Horn Clauses. Technical Report TR-8-1991, School of Computer Science, Technical University of Nova Scotia, Halifax, NS, Canada, 1991.

[van Emden and Lloyd 1984] M.H. van Emden and J.W. Lloyd. A Logical Reconstruction of Prolog II. In *Proc. 2nd Intl. Conf. on Logic Prog.* Uppsala, 1984. pp. 35–40.

[Goebel 1990] R. Goebel. A Quick Review of Hypothetical Reasoning Based on Abduction. In *AAAI Spring Symposium on Automated Abduction*, Stanford University, 1990. pp. 145–149.

[Hoddinott and Elcock 1986] P. Hoddinott and E.W. Elcock. PROLOG: Subsumption of Equality Axioms by the Homogeneous Form. In *Proceedings of the Symposium on Logic Programming*, 1986. pp. 115–126.

[Hölldobler 1989] S. Hölldobler. *Foundations of Equational Logic Programming.* Lecture Notes in Computer Science 353, Springer Verlag, Berlin, 1989.

[Hölldobler 1990] S. Hölldobler. Conditional Equational Theories and Complete Sets of Transformations. *Theoretical Computer Science*, Vol. 75 (1990), pp. 85–110.

[de Kleer *et al.* 1988] J. de Kleer, A. K. Mackworth, and R. Reiter. Characterizing Diagnoses. In *Proceedings Eighth National Conference on Artificial Intelligence*, 1990. pp. 324–330.

[Kohavi 1978] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, 1978.

[Lloyd 1984] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, Berlin, 1984.

[Paterson and Wegman 1978] M. S. Paterson and M. N. Wegman. Linear Unification. *J. Comput. Syst. Sci.* Vol. 16 (1978), pp. 158–167.

[Poole 1988] D. Poole. A Logical Framework for Default Reasoning. *Artificial Intelligence* Vol. 36 (1988), pp. 27–47.

[Poole and Provan 1990] D. Poole and G. M. Provan. What is an Optimal Diagnosis? In *Conference on Uncertainty in AI*, Boston, 1990.

[Reiter and de Kleer 1987] R. Reiter and J. de Kleer. Foundations of Assumption-Based Truth Maintenance Systems: Preliminary Report. In *Proceedings of the National Conference on Artificial Intelligence*, 1987. pp. 183–188.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Hypothetico-deductive Reasoning

Chris Evans* **and Antonios C. Kakas†**

*Department of Mathematical Studies, Goldsmiths' College, University of London
New Cross, London SE14 6NW, UK. EMAIL: c.evans@gold.lon.ac.uk.

†Department of Computer Science, University of Cyprus, 75 Kallipoleos Street,
Nicosia, Cyprus. EMAIL: kakas@cyearn.earn
(Part of the research for this paper was completed while both authors were at Imperial College, London SW7 2BZ)

Abstract

This paper presents a form of reasoning called "hypothetico-deduction", that can be used to address the problem of multiple explanations which arises in the application of abduction to knowledge assimilation and diagnosis.

In a framework of hypothetico-deductive reasoning the knowledge is split into the theory T and observable relations S which may be tested through experiments. The basic idea behind the reasoning process is to formulate and decide between alternative hypotheses. This is performed through an interaction between the theory and the actual observations. The technique allows this interaction to be user mediated, permitting the acquisition of further information through experimental tests. Abductive explanations which have all their empirical consequences observed are said to be "fully corroborated".

We set up the basic theoretical framework for hypothetico-deductive reasoning and develop a corresponding proof procedure. We demonstrate how hypothetico-deductive reasoning deals with one of the main characteristics of common-sense reasoning, namely incomplete information, through the use of partial corroboration. We study the extension of basic hypothetico-deductive reasoning applied to theories that incorporate default reasoning as captured by negation-as failure (NAF) in Logic Programming. This is applied to the domain of Temporal Reasoning, where NAF is used to formulate default persistence. We show how it can be used successfully to tackle typical problems in this domain.

1 Motivation

Abduction is commonly adopted as an approach to diagnostic reasoning [Reggia & Nau, 1984], [Poole, 1988]. However, there are frequently many possible abductive explanations for a given observation. This is the problem of "multiple explanations". In order to choose between these explanations it becomes necessary to collect more information. Consider the Crime Detection example formalized below (Theory T1).

Suppose we arrive at the scene of the crime and the first observation we make is that someone is dead. We seek an explanation for this on the basis of the theory T1 above. Suppose we accept that there are only three possible causes of death: being strangled, being stabbed, or drinking arsenic (these are technically known as the *abducibles*). Simple abduction starting from the observation "dead" yields precisely these three possible explanations. In order to choose between these multiple explanations, we need to collect more information. For example, if we examined the corpse and discovered that there were marks on the neck, we

Theory T1

strangled → dead	strangled → neck_marks
blood_loss → dead	stabbed → blood_loss
poisoned → dead	drunk_arsenic → poisoned
drunk_arsenic → blue_tongue	

might take this as evidence for the first explanation over the others. Moreover, we know that drinking arsenic also has the consequence of leaving the victim with a blue tongue, so we might like to look for that.

One approach to deciding between multiple explanations is through the performance of *crucial experiments* ([Sattar & Goebel, 1989]): pairs of explanations are examined for contradictory consequences, and an experiment is performed which *refutes* one of them whilst simultaneously *corroborating* the other. With n competing explanations we must thus perform at most $(n-1)$ crucial experiments .

The crucial experiment approach is, however, unable to choose between explanations when they fail to have contradictory consequences or when they have contradictory consequences that are not empirically determinable (e.g. Tychonic and Copernican world systems). In our example, for instance, the explanations "strangled" and "stabbed" are not incompatible. It is possible that the victim was *both* strangled *and* stabbed. As result, there can be no crucial experiment that will decide between the two. However, further evidence might lead us to accept one explanation, whilst tentatively rejecting the other. For example, knowledge that the person exhibits marks on the neck supports the "strangled" hypothesis. In fact we have all the theoretically necessary observations to conclude that the victim was strangled. On the other hand, the "stabbed" hypothesis implies "blood_loss", which if not observed might lead us to favour the "strangled" explanation. Note that later evidence of blood loss would lead us to return to the "stabbed" hypothesis (in addition to "strangled"). From our viewpoint, crucial experiments are the special case of general hypothetico-deductive reasoning when an hypothesis is refuted whilst simultaneously corroborating a second.

The process of hypothetico-deductive reasoning allows the formation and testing of hypotheses within an interactive framework which is applicable to a wide

class of applications and is implementable using existing technology for resolution.

The technique of hypothetico-deductive reasoning has its origin in the Philosophy of Science. It was primarily proposed by opponents of Scientific Induction. Its notable contributors were Karl Popper ([Popper, 1959],[Popper, 1965]), and Carl Hempel [Hempel, 1965]. In its original context, hypothetico-deduction is a method of creating scientific theories by making an hypothesis from which results already obtained could have been deduced and which entails new predictions that can be corroborated or refuted. It is based on the idea that hypotheses cannot be derived from observation, but once formulated can be tested against observation.

The hypothetico-deductive mechanism we formulate, resembles this method in having the two components of hypothesis formation and corroboration. It differs from the accepted usage of the term in philosophy of science by the status of the hypothesis formation component.

In the philosophy of the process of hypothesis formation is equivalent to theory formation: a creative process in which a complete theory is constructed to account for the known observations. By contrast, the method we describe here starts with a fixed generalized theory which is assumed to be complete and correct. The task is to construct some hypotheses which when added to the theory have the known observations as logical consequences. The process is more akin to that used by an engineer when they apply classical mechanics to a particular situation: they don't seek a new physical theory, but rather a set of hypotheses which would explain what they have observed. Since, for us, hypothesis formation can be mechanized, we do not have to tackle the traditional issues of the philosophy of science concerning the basis of theory formation. We thus avoid (like Poole before us [Poole, 1988, p.28]) one of the most difficult problems of science.

This paper is organized as follows. We first describe the reasoning process and present the logical structure of the reasoning mechanism, indicating how it relates to classical deduction and model theory. Abductive and corroborative derivation procedures for implementing the reasoning process are then defined through resolution. We indicate how this reasoning technique relates to current work on abduction and diagnostic reasoning, and suggest some possible extensions. We illustrate the features and applicability of this reasoning method with several examples. We then describe the extension of hypothetico-deduction to apply to theories which include some form of default reasoning, using negation-as-failure as an example. We consider a typical application of defaults in causal reasoning, namely default persistence, and provide several further examples which illustrate this extension.

2 Hypothetico-deductive Framework

Suppose we have a fixed logical theory T about the world. For example, it might be a medical model of the anatomy, or a representation of the connections in an electrical network, or a model of the flow of urban traffic in Madrid. Let us divide the relations in the theory into two categories: empirical and theoretical. How we make this distinction will depend on how we interpret these relations in the domain for the theory. An empirical relation is one which can be (or has been) observed. For example, the blood pressure of a patient, the status of a circuit-breaker (open or closed), or the number of cars passing some point. By contrast, a theoretical relation is in principle not observable. Examples of theoretical relations might be infection with an influenza virus, the occurrence of a short-circuit from the viewpoint of a control centre, or the density of traffic at some point.

Suppose we want an explanation for G on the basis of the theory. By this, what we mean is "what relations (we will call them *hypotheses*) might be true in order to have given rise to G?". The answer to this question could involve either theoretical or empirical relations. In order to be confident that an explanation is the *correct* explanation it is useful to *test it*. Explanations in terms of empirical relations are directly testable. In the simplest case we just consider the other observations we have already made; in more complicated cases, we may need to "go and look" or even perform an "experiment". Explanations in terms of theoretical relations must be tested indirectly, by deducing their empirical consequences, and testing these.

Unfortunately, not all hypotheses that might give rise to the observation G serve as explanations, regardless as to whether they pass any tests. Some are too trivial such as taking G as an explanation for itself. Others we rule out as unsuitably shallow. For example, suppose we sought an explanation for the observation "Jo laughed at the joke"; one possible hypothesis is because "the joke was funny". However, what we really wanted was a deeper explanation: Why was the joke funny? We therefore designate certain types of hypotheses as explanatory (or, more strictly, "abducible").

The problem of explanation, as far as we are concerned in this paper, is the problem of constructing abducible hypotheses which when we add them to T will have G as a logical consequence. Furthermore, explanations must pass (direct or indirect) tests.

The process of constructing hypotheses which have G as a deductive consequence is an example of **hypothesis formation**. It is this stage that corresponds to the "hypothetico-" component of hypothetico-deductive reasoning. The process of testing an explanation is an example of **corroboration**. It is this stage that corresponds to the "deductive" component of hypothetico-deductive reasoning. This is because we use deduction to determine the empirical consequences of a given explanation. The process of hypothetico-deductive reasoning can now be formulated as the construction of an explanation for an observation through interleaving hypothesis formation and corroboration.

3 The Hypothetico-deductive Mechanism

Let us consider the mechanism for hypothetico-deductive reasoning in more detail. To simplify matters we shall require that our theory is composed of rules and no facts. In logical terms, an hypothesis (and thus an explanation) will be a set of ground atomic well-formed formulae.

Suppose we have a (usually causal) theory T, an observation set O, a set of abducible atomic formulae A, and a particular observation G from O which we wish to explain. Let O′ = O-G. In addition we define a set S, the **observables**, containing all the formulae that can occur in O.

There are three components to the reasoning process: hypothesis formation, hypothesis corroboration, and explanation corroboration. In outline, we carry out hypothesis formation on G, and for each component formula in the resultant hypothesis. We repeat this process until all that remains

is a set of abducible relations constituting the explanation. We also carry out hypothesis corroboration at each formation point. Finally we reason forwards from the explanation to perform explanation corroboration.

Hypothesis Formation

From any ground atomic formula F we form an hypothesis for that formula. This is done by determining which rules in T might allow F as a conclusion, and forming an hypothesis from the antecedents of each such rule (after carrying out the relevant substitutions dictated by F). Each hypothesis is thus sufficient to allow the conclusion of F.

Hypothesis Corroboration

An hypothesis for an observation may contain instances of observables defined by S. For each such component we check to see whether it is an observation recorded in O′. If it is a member of O′ then it is corroborated and we can retain it. However, where any component is not corroborated in this fashion, we reject the entire hypothesis.

Explanation Corroboration

An hypothesis H which is composed entirely of instances of abducible predicates defined by A is an **explanatory hypothesis**. To corroborate H, we use T to reason forwards from H as an assumption. Each logical consequence of H which is also an instance of an observable is checked against O′ for corroboration (similar to "hypothesis corroboration"). If it does not occur in O′ then the original hypothesis H is rejected. If all observable consequences are corroborated, then the explanation H is said to be corroborated.

In general, rules may have more than one literal in their antecedent. We must also check the satisfaction of the other literals in a given rule by reasoning backwards until we reach either one of the observations in O′ or one of the other explanatory hypotheses. If neither of these two situations arise, the rule is discarded from the forward reasoning process.

We make a distinction between corroboration *failure*, where an hypothesis or prediction does not occur in the observation set O′, and *refutation*, where the negation of an hypothesis or prediction occurs in O′. Normally the form of O and T means that refutation is impossible (see the next section for details of this form). Later we suggest an extension which allows the possibility of refutation in addition to corroboration failure. In cases where it is natural to apply the closed world assumption to O, these two situations will coincide.

4 The Logical Structure of Hypothetico-deductive Reasoning

Suppose we have a theory T composed of definite Horn clauses and an observation set of ground atomic well-formed formulae O. Let the set of ground atomic formulae which can occur in O be S, the **observables.** Similarly, let us define a set of distinguished ground atomic formulae A, the **abducibles**, in terms of which all explanations must be constructed. An explanation will be a member of the set A. We will assume that the theory T alone does not entail any empirical observation without some other empirical input i.e. there does not exist any formula ϕ such that $\phi \in S$ and $T \vDash \phi$. Consider also a ground atomic formula G (a member of S) for which we seek an explanation.

Given the 4-tuple <T,O,A,S>, a **corroborated explanation** Δ for G, is a set of ground atomic well-formed formulae, which fulfils all of the following criteria:

(1) Each formula in Δ must be a member of A.

(2) $T \cup \Delta \vDash G$

(3) If $T \cup \Delta \vDash \Pi$ and $\Pi \subseteq S$, then $\Pi \subseteq O$

An explanation set Δ which satisfies (1) and (2) but not (3) is said to be **uncorroborated.**

This formulation is easily generalized to explanation for multiple observations by simply replacing G with a conjunction of ground atomic formulae.

We note that since at this stage we have taken our theories to be Horn, a simple extension to hypothetico-deductive reasoning allows us to distinguish between explanation *refutation* when a prediction is inconsistent with observation, and merely the failure of *corroboration* where a prediction is consistent with known observations but not present in them. Such an extension would allow a hypothetico-deductive system to deal with circumstances where our observations cannot ever be complete (where we know our fault-detection system is itself fallible, for instance). We could then discard only those explanations that are refuted, and order the remaining ones according to their degree of corroboration (corresponding to Popper's notion of *versimilitude*, [Popper, 1965]). A later section discusses the extension of hypothetico-deductive reasoning to theories which include negation-as-failure.

This extended version of hypothetico-deductive reasoning is non-monotonic because later information might serve to refute a partially corroborated explanation. To return to our first example for instance, the observation that the victim does not have a blue tongue would lead us to reject the hypothesis that they had drunk arsenic (even if previously this hypothesis had some observational consequences which had been observed).

5 Hypothetico-deductive Proof Procedure

A resolution proof procedure which implements hypothetico-deductive reasoning is formally presented below. Basically we define two types of derivation: abductive derivation and corroboration derivation which are then interleaved to define the proof procedure. Abductive derivation corresponds to the processes of hypothesis formation and corroboration, deriving hypotheses *for* goals. Corroboration derivation corresponds to the process of explanation corroboration, deriving predictions *from* goals. There are two different ways to interleave the abductive and deductive components of the reasoning mechanism. One approach is to derive all the abducible literals in the hypothesis for an observation, *before* any of them are corroborated. The second approach attempts corroboration as soon as an abducible literal is derived, postponing consideration of other (non-abducible) literals in the hypothesis. Here we present a proof procedure based on the second approach.

Definition (*safe selection rule*)

A safe selection rule R is a (partial) function which, given a goal $\leftarrow L_1, \ldots, L_k$ $k \geq 1$ returns an atom L_i, $i=1,\ldots,k$ such that:

either i) L_i is not abducible;
or ii) L_i is ground.

Definition (*Hypothetico-deductive proof procedure*)
An ***abductive derivation*** from $(G_1\ \Delta_1)$ to $(G_n\ \Delta_n)$ via a safe selection rule R is a sequence

$$(G_1\ \Delta_1),\ (G_2\ \Delta_2),\ \ldots,\ (G_n\ \Delta_n)$$

such that for each i>1 G_i has the form $\leftarrow L_1,\ldots,L_k$, $R(G_i)=L_j$ and $(G_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:

A1) If L_j is neither an abducible nor an observable, then $G_{i+1}=C$ and $\Delta_{i+1}=\Delta_i$ where C is the resolvent of some clause in T with G_i on the selected literal L_j;

A2) If L_j is observable, then $G_{i+1}=C$ and $\Delta_{i+1}=\Delta_i$ where C is the resolvent of $C' : \leftarrow L_1',\ldots,L_j',\ldots,L_k'$ with some clause in T on L_j' where $\leftarrow L_1',\ldots,L_{j-1}',L_{j+1}',\ldots,L_k'$ is the resolvent of G_i with some clause (ground assertion) L_j' in O on the selected literal L_j;

A3) If L_j is abducible and $L_j \in \Delta_i$, then $G_{i+1}= \leftarrow L_1,\ldots,L_{j-1},L_{j+1},\ldots,L_k$ and $\Delta_{i+1}=\Delta_i$;

A4) If L_j is abducible and $L_j \notin \Delta$ and there exists a **corroboration derivation** from $(\{L_j\}\ \Delta_i \cup \{L_j\})$ to $(\{\}\ \Delta')$ then $G_{i+1}= \leftarrow L_1,\ldots,L_{j-1},\ L_{j+1},\ldots,L_k$ and $\Delta_{i+1} = \Delta'$.

Step A1) is an SLD-resolution step with the rules of T. In step A2) under the assumption that observables and abducibles are disjoint we need to reason backward from the true observables in the goal to find explanations for them since the definition of an explanation requires that it logically implies G in the theory T alone without the set of observations O. Step A3) handles the case where an abductive hypotheses is required more than once. In step A4) a new abductive hypotheses is required which is added to the current set of hypotheses provided it is corroborated.

A ***corroboration derivation*** from $(F_1\ \Delta_1)$ to $(F_n\ \Delta_n)$ is a sequence

$$(F_1\ \Delta_1),\ (F_2\ \Delta_2)\ \ldots\ (F_n\ \Delta_n) \text{ to } (F_n\ \Delta_n)$$

such that for each i>1 F_i has the form $\{H \leftarrow L_1,\ldots,L_k\} \cup F_i'$ and $(F_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:

C1) If H is not observable then $F_{i+1} = C' \cup F_i'$ where C' is the set of all resolvents of clauses in T with $H \leftarrow L_1,\ldots,L_k$ on the atom H and $\Delta_{i+1}=\Delta_i$;

C2) If H is a ground observable, $H \notin O$ and $L_1,\ldots,L_k$ is not empty then $F_{i+1} = C' \cup F_i'$ where C' is $\leftarrow L_1,\ldots,L_k$ and $\Delta_{i+1}=\Delta_i$; If $H \in O$ then $F_{i+1} = F_i'$ and $\Delta_{i+1}=\Delta_i$.

C3) If H is a non ground observable, $O \not\models \exists x H$ and $L_1,\ldots,L_k$ is not empty then $F_{i+1} = C' \cup F_i'$ where C' is $\leftarrow L_1,\ldots,L_k$ and $\Delta_{i+1}=\Delta_i$;

C4) If H is a non ground observable and L_j is any non observable selected literal from $L_1,\ldots,L_k$ then $F_{i+1} = C' \cup F_i'$ where C' is the set of all resolvents of clauses in $T \cup \Delta_i$ with $H \leftarrow L_1,\ldots,L_k$ on the selected literal L_j and $\Delta_{i+1}=\Delta_i$; If L_j is observable the resolutions are done only with clauses in O.

C5) If H is empty, L_j is any selected literal and L_j is not observable then $F_{i+1} = C' \cup F_i'$ where C' is the set of all resolvents of clauses in $T \cup \Delta_i$ with $\leftarrow L_1,\ldots,L_k$ on the literal L_j and $\square \notin C'$, and $\Delta_{i+1}=\Delta_i$; If L_j is observable the resolutions are done only with clauses in O.

In step C1) we "reason forward" from the conclusion H trying to generate a ground observable at the head. Once this happens if this observable is not "true" steps C2), C3) give the denial of the conditions that imply this observable. Step C4) reasons backward from the conditions either failing or trying to instantiate further the observable head. Step C5) reasons backward from the denials of steps C2), C3) until every possible such backward reasoning branch fails. Note that in the backward reasoning steps observables are resolved from the observations O and not the theory. More importantly notice that we do not reason forward from an observable that is true.

Note that we have included the set of hypotheses Δ_i in the definition of the corroboration derivation although this does not get affected by this part of the procedure. The reason for this is that more efficient extensions of the procedure can be defined by adding extra abducible information in the Δ_i during the corroboration phase e.g.the required absence of some abducible A can be recorded by the addition of a new abducible A^*.

Theorem
Let <T,O,A,S> be a Hypothetico-Deductive framework and G a ground atomic formula. If $(\leftarrow G\ \{\})$ has an adbuctive derivation to $(\square, \Delta)$ then the set Δ is a corroborated explanation for G.

Proof (Sketch)
The soundness of the abductive derivations follows directly from the soundness of SLD resolution for definite Horn theories as every abductive derivation step of this procedure can be mapped into an SLD resolution step. To show that the explanation Δ is corroborated let $A \in S$ be any ground atomic logical consequence of $T \cup \Delta$. Since $T \cup \Delta$ is a definite Horn theory A must belong to its minimal model which can be constructed in terms of the immediate consequence operator T[van Emden & Kowalski, 1976] . Hence there exists a finite integer n such that $A \in T_{T \cup \Delta} \uparrow n$ $(\varnothing)$ and A does not follow from T alone by our assumption on the form of the theory T . The result then follows by induction on the length of the corroboration derivation.

6 Application of Hypothetico-deductive Reasoning

In this section we will illustrate hypothetico-deductive reasoning with some examples. Before this it is worth pointing out that existing abductive diagnosis techniques (e.g. [Poole et al., 1987], [Davis, 1984], [Cox & Pietrzkowski, 1987], [Genesereth, 1984], [Reggia et al., 1983], [Sattar & Goebel, 1989]) can be accommodated within the HD framework. For example in the diagnosis of faults in electrical circuits hypothetico-deductive reasoning exhibits similar behaviour to [Genesereth, 1984], [Sattar & Goebel, 1989].

Problems and domains which are ideally suited to the application of hypothetico-deductive reasoning exhibit two characteristics. Firstly, they have a large number of

possible explanations in comparison to the number of empirical consequences of each of those explanations. Secondly, they have a minimal amount of observational data pertaining to a given explanation so that corroboration failure is maximized.

To illustrate the manner in which general hypothetico-deductive reasoning deals with differing but compatible explanations, let us consider the example of abdominal pain first presented by [Pople, 1985] and axiomatized in [Sattar & Goebel, 1990]. The axioms are reproduced below. To allow the possibility of several diseases occurring simultaneously, the three expressions which capture the fact that the symptoms (nausea, irritation_in_bowel, and heartburn) are incompatible, have been omitted.

Theory T2

abdominal_pain_symp(X) → has_abdominal_pain
problem_is(indigestion) → abdominal_pain_symp(nausea)
problem_is(dysentry) → abdominal_pain_symp(irritation_in_bowel)
problem_is(acidity) → abdominal_pain_symp(heartburn)

Now consider the following observations:

Observations O

has_abdominal_pain
abdominal_pain_symp(nausea)

Abducibles, A = {problem_is(indigestion), problem_is(dysentry), problem_is(acidity)}

Observables, S = {has_abdominal_pain, abdominal_pain_symp(nausea), abdominal_pain_symp(irritation_in_bowel), abdominal_pain_symp(heartburn)}

There are three possible potential explanations for the observation "has_abdominal_pain". Since they are not mutually incompatible (it is possible to have all three diseases, for example), there is no crucial literal which can help us distinguish between the three explanations. There is thus no "best" explanation from this point of view.

From the point of view of hypothetico-deductive reasoning however, one of the explanations stands apart from the others. On the basis of all the currently available evidence "problem_is(indigestion)" is completely corroborated. The two remaining explanations remain possible but uncorroborated; that is to say there is no supplementary evidence in support of them. Experiments might be performed (testing for "abdominal_pain_symp(irritation_in_bowel)", and "abdominal_pain_symp(heartburn)") which could corroborate one or both of the others, which would lead us to extend our explanation. Since physical incompatibilities are rare in common-sense reasoning, hypothetico-deductive reasoning has an advantage in being able to offer a (revisable) "best" explanation based on the currently available evidence, in spite of the absence of possible crucial experiments. It is important to appreciate that it is usually impractical to simply construct the hypotheses by performing abduction on all the observations in O, since in general there may be an extremely large number of them. Moreover, only a few may be relevant to the particular observation for which we seek an explanation.

It might be thought that the checking of *all* the observational consequences of some explanation might be equally impractical: there might be an infinite number of them as well. However, it must be borne in mind that we are only considering the representation of common-sense; we would normally ensure that there are only a small number of observable consequences in which we would be interested. We would define our set of observables, S, accordingly. So, for instance, in the fermentation example below we represent certain critical times (often referred to as "landmarks") at which we might perform observations. Similarly, in the "stolen car" example which we present later, we restrict observables to events that occurred at some specific point in time.

One application area in which incomplete information is intrinsic, is that of temporal reasoning. Reasoning about time is constrained by the fact that factual information is only available concerning the past and the present. By its very nature we must perform temporal diagnosis with no knowledge about the future states of the systems we are trying to model.

As an example of temporal diagnosis which illustrates this characteristic, consider an industrial process involving the fermentation of wine. Suppose we are faced with the task of diagnosing whether the fermentation process has proceeded normally, or that the extremely rare conditions have occurred under which we will produce a vintage wine. To do this we must carry out a test at some time after the wine-making process has begun, such as measuring its pH, its relative density, or its alcohol content. Suppose further that we need to decide on this diagnosis before a certain time, e.g. the bottling-time tomorrow. Let us refer to some property of the mixture which would be observed for vintage wine by the symbol **p1**, and that for ordinary wine as **p2**. These two properties might be entirely compatible: it is perfectly possible for ordinary wine to be produced under conditions which exhibit **p1**(as well as **p2**), but in such a case it is not the fact that the mixture is ordinary wine that *causes* **p1** to be observed. Now suppose we observe **p1** before the bottling time, and suppose there are no further observational consequences for the "vintage wine" hypothesis that are observable before tomorrow. Then the "vintage wine" hypothesis is completely corroborated within the defined time-scale. On the other hand, the "ordinary wine" hypothesis remains at best only partially corroborated. Hypothetico-deductive reasoning would then *prefer* the "vintage wine" hypothesis over the "ordinary wine" one. The temporal dimension illustrates the ability of hypothetico-deductive reasoning to form diagnoses on the basis of incomplete information. Notice that an extension of the time scale would revise the status of the observable relations and perhaps the "vintage wine" hypothesis would become only partially corroborated. The application of hypothetico-deductive reasoning to the temporal domain will be discussed in more detail in the next section as an important special case of the integration of hypothetico-deductive reasoning and default reasoning.

7 Hypothetico-deduction with Default Theories

As we discussed above, the aim of hypothetico-deductive reasoning has been to provide a framework in which we can tackle one of the main characteristics of common sense reasoning, namely incomplete information. More specifically it addresses the fact that

we are often forced to form hypotheses and explanations on the basis of limited information. Another important form of reasoning that deals with the problem of incomplete (or limited) information is default reasoning (see e.g. [Reiter, 1980]). We can then enhance the capability of each framework separately to deal with this problem of missing information by integrating them together into a common framework.

So far we have only considered the application of hypothetico-deduction to classical theories. In this section we study its application to default theories incorporating negation-as-failure (NAF) from Logic Programming. We will then apply this adaptation of hypothetico-deduction to temporal reasoning problems formulated within the event calculus where NAF is used to represent default persistence in time ([Kowalski & Sergot, 1987], [Evans, 1989]).

The approach we adopt is to consider only classical theories to which non-monotonic reasoning mechanisms such as default and hypothetico-deductive reasoning are applied (in contrast to non-monotonic logics). The motivation as before, is to separate representation (classical logic) from reasoning (non-monotonic). Recent formalizations of the semantics of negation-as-failure [Eshghi & Kowalski, 1989], [Kakas & Mancarella, 1990], [Dung, 1991], [Kakas & Mancarella, 1991] have adopted a similar point of view. This approach means that hypothetico-deductive reasoning can be applied to default theories of any system which separates these two components, e.g. circumscription [McCarthy, 1980].

Following this work, we associate to any general logic program, P, (Horn clauses extended with negation-as-failure) a classical theory , P′, as follows. Each negative condition, **not** p, where **not** denotes the negation-as-failure operator, is regarded as a single new positive atom. This can be made explicit by replacing each such negative literal, **not** p, by a syntactic variant, say p*, to give the Horn theory P′. The model-theoretic extension of the new symbol is intended to be the complement of the old one, so that we can omit the **not**. To take a more meaningful example we might replace "**not** alive" with "dead". These new symbols "p*" or "dead" are then defined to be abducible predicates. The above authors show that with this view it is possible to understand (and generalize) the stable model semantics [Gelfond & Lifschitz, 1989] for NAF in logic programming. (Note that this is also the approach taken more generally in [Poole, 1988] for understanding default reasoning through abduction by naming the defaults and considering these as assumptions.)

We can then apply an adapted formulation of hypothetico-deductive reasoning to these classical Horn theories P′ corresponding to general logic programs P. As above we have a 4-tuple <P′,O,A,S> where the set, A, of abducibles has been extended with new abducibles e.g. "p*", "dead", which name the different NAF default assumptions.

Hence given a 4-tuple <P′,O,A,S>, a **corroborated explanation** Δ for an observation G, is a set of ground atomic well-formed formulae, which fulfils all of the following criteria:

(1) Each formula in Δ is a member of A.
Let $\Delta = \Delta_D \cup \Delta_H$ where Δ_D denotes the subset of abducibles corresponding to NAF.

(2) $P' \cup \Delta \models G$

(3) If $P' \cup \Delta \models \Pi$ and $\Pi \subseteq S$, then $\Pi \subseteq O$

(4) There exists a stable model[1] M of $P' \cup \Delta_H \cup O$ such that the negations corresponding to Δ_D hold in M (i.e. are contained in the complement of M).

This is a direct extension of the previous definition of hypothetico-deductive reasoning. The extra condition (4) captures the default reasoning present in the theory P (or P′). This is clearly separated in this condition although it does play an important role in the generation of explanations by rejecting explanations that do not satisfy it. This has the effect of adding extra abducibles in the Δ to make it acceptable. For example in the theory,

G ← p*
p ← q*
q ← a

although {p*} is an explanation for G, this is not accepted until the abducible "a" is added to it which ensures that this default assumption {p*} is valid. In addition condition (4) also ensures that any default assumption (abducible) in Δ is compatible with the observations O. Note that we could have chosen to put together conditions (2) and (4) as "G is true in a stable model of $P \cup \Delta_H$" for generating the explanations Δ, and use condition (4) solely for the purpose of ensuring that Δ_D are compatible with the observations O.

Although at first sight it might seem appropriate to allow default reasoning during the corroboration of an explanation this is not the case as indicated by condition (3). The reason for this is clear: if we allow it then the corroboration process will not be for the explanation Δ alone, but for Δ plus any additional default assumptions made in arriving at the observable test. In other words, we would not want to reject an explanation Δ by failure to corroborate an observation that is a not a consequence of Δ alone but of Δ with some additional default assumptions.

Let us now indicate how the proof procedure for hypothetico-deductive reasoning, defined earlier, needs to be extended to deal with this more general formulation where our theories are general logic programs. The first thing to notice is that, as indicated by condition (3), the corroboration phase of the procedure remains unchanged apart from the fact that it will also be applied whenever a NAF hypothesis, "p*" (or "**not** p"), is added to the explanation. Similarly, the adbuctive derivation phase remains as before with the set of abducibles enlarged to include the NAF default assumptions.

The main extension of the procedure arises from the need to implement the new condition (4). This can be done by adopting the abductive proof procedure developed in [Eshghi & Kowalski, 1989], [Kakas & Mancarella, 1990b], [Kakas & Mancarella, 1990c] for NAF which is an extension of SLDNF. A new type of derivation, called **consistency derivation**, is introduced interleaved with the abductive phase of the procedure whenever a NAF hypothesis, "p*" (or "**not** p"), is required in the explanation. Its purpose is to ensure that "p*" (or "**not** p") is a valid NAF assumption by checking that p does not succeed. This involves reasoning backwards from p in all possible ways and showing that each such branch ends in failure.

During this consistency check for some NAF hypothesis, "p*" (or "**not** p"), it is possible for new

[1] More generally, we can use recent extensions of stable models e.g. preferred extensions or stable theories as defined in [Dung, 1991] and [Kakas & Mancarella, 1991] respectively.

abductive phases to be generated whenever the failure of some consistency branch reduces to showing that some other NAF default assumption e.g. "q*" (or "**not** q") does not hold in the theory $P' \cup \Delta$. To ensure this the procedure starts a new abductive phase to show that q holds where it is possible that new hypotheses may be added in the explanation if this is needed to prove q. Then with this enlarged explanation "q*" (or "**not** q") is not a valid (default) NAF assumption (as q holds) and so the original consistency branch can not succeed. In the example above the abducible "a" in the explanation {p*, a} for G is generated during the consistency check of p* (or **not** p) as described here. More details about this extension of the proof procedure can be found in the references above.

8 Application of HD Reasoning to Temporal Reasoning

As an example of the application of the above extended hypothetico-deductive mechanism, let us consider temporal reasoning with the Event Calculus [Kowalski & Sergot, 1987] where NAF is used to express default persistence in time.

The Event Calculus represents *properties* which hold over intervals of time. They are initiated and terminated by *events* which happen at particular instances of time. NAF is used to conclude that a property is not "clipped" or "broken" over an interval of time, achieving default persistence. Variants of the two main axioms, which define when a property "holds" and when a property is "broken", are given below.

holds-at(p,t2) ← happens-at(e,t1) ∧
initiates(e,p) ∧
t1 < t2 ∧
not broken-during(p,<t1,t2>)

broken-during(p,<t1,t2>) ← happens-at(e,t) ∧
terminates(e,p) ∧
t1 < t ∧
t ≤ t2

The first axiom states that some property p holds at any time after an initiating event, provided it is not (known to be) broken at some time during the intervening time-interval. NAF ensures that we draw the conclusion that it isn't broken if we have no evidence for it: default persistence. The second axiom states that a property is broken during an interval if a terminating event happens at some time within that interval.

Before we can apply HD reasoning to these axioms we must carry out the transformation to eliminate the NAF. A possible renaming of "not broken-during" is "persists":

holds-at(p,t2) ← happens-at(e,t1)∧
initiates(e,p)∧
t1 < t2 ∧
persists(p,<t1,t2>).

Before we present a detailed example of the application of HD, let us briefly consider how the use of a temporal default theory such as the Event Calculus does not modify the process of corroboration (we use the classical version of the theory), although it does modify the process of explanation construction.

Consider an example in which the walls of a house are painted white. Using the Event Calculus, if we wished to explain why the walls were white, we would hypothesize an event of painting them white. In order to corroborate this hypothesis we would look for empirical consequences. One possibility might be that the paint brush has white paint on it. However this prediction involves assuming that the state of "brush-has-white-paint" persisted since the walls were painted; the corroboration is based upon a further (uncorroborated!) hypothesis. Moreover, consider the consequences of observing that the paint brush has *red* paint on it. Does this *refute* the explanation that the walls are white because they were painted white? Obviously not. Under the extended HD scheme we limit default reasoning to be a part of the hypothesis formation component. Corroboration is straightforward classical deduction. This is one of the reasons for having to transform the Event Calculus axioms to eliminate the NAF.

Let us consider a more detailed application of hypothetico-deductive reasoning to a problem formalized in the Event Calculus. We shall take Kautz's "stolen car" problem [Kautz, 1986]. The task is to explain why a car parked in the morning is missing when we look for it in the afternoon. In particular, to explain *when* the car was stolen. Kautz's original motivation was to demonstrate that temporal reasoning which performed chronological minimization (e.g. Shoham's Non-monotonic Logic [Shoham, 1988]) would predict that the car was stolen the instant before it was found to be missing; which was unsatisfactory. From our point of view, the stolen car problem is more correctly viewed as an *explanation* problem in which there are several possible competing explanations, corresponding to the different times that the car might have been stolen.

In the formalism of the Event Calculus we would describe the problem as follows. We know that the car was parked at some particular time, say time "1"; and we know that it was missing at, say, time "4". We also know that stealing initiates the property "missing" and terminates "parked":

initiates(e,missing) ← type(e,steal)

terminates(e,parked) ← type(e,steal)

Our explanatory task is thus to explain the observation "holds-at(missing,4)". We will take the predicates "happens-at", "type" and (since it is a default relation) "persists" to be abducible. Furthermore, let us restrict the abducible "happens-at" events to those which happen between time "1" and "4". Our observables will be instances of the relation "holds-at" which occur at time "4". Using hypothesis formation applied to the rule defining "holds-at" we might hypothesize:

{happens-at(e,2), type(e,steal), persists(missing,<2,4>)}

This states that some stealing event happened at time "2". Notice that we have to include the persistence assumption: if some other event had terminated this "missing" state (such as the returning of the car!), then this particular stealing event would not be the right explanation.

Using a discrete representation of time, there is another explanation corresponding to a stealing event at time "3". Pure abduction is unable to distinguish between these two explanations.

There are two further characteristics of HD to demonstrate. Firstly, note that we have to check the consistency of the default "persists" hypothesis (according to the 4th corroboration requirement). We do this by checking that "~broken-during(missing,<2,4>)" holds in the stable model when we include all our observations; computationally speaking, we must check that "broken-during(missing,<2,4>)" finitely fails.

The second characteristic is corroboration to choose between the two competing explanations. In order to describe this aspect, we must elaborate our example somewhat. Suppose that we had a car alarm fitted and it is not possible to steal the car without setting off the alarm. The hypothesis that the car was stolen at time "2" would lead us to predict "happens-at(alarm,2)" whereas the alternative would predict "happens-at(alarm,3)". We must extend our definition of observables to include "happens-at(alarm, 2)" and "happens-at(alarm, 3)", corresponding, say, to checking with someone near at what time they heard a car alarm start going off. The process of corroboration against observations concerning the alarm events proceeds as in the unextended version of HD reasoning.

Thus the addition of the appropriate observations for the "stolen car" situation allows us to form two explanations, one of which we might reject as uncorroborated and the other of which might be completely corroborated.

The "bloodless" Yale Shooting problem ([Morgenstern & Stein, 1988]) - the explanatory counterpart to the original Yale Shooting prediction problem ([Hanks & McDermott, 1987]) - is of a similar form. In this scenario, a gun is loaded, a period of waiting ensues, and someone is shot with the gun. They are found to be unharmed. The task is to explain how this could be so. Pure abduction produces a number of explanations in terms of unloading events that must have occurred during the period of waiting: one explanation for each different possible time of the event. Hypothetico-deduction allows the possibility of selecting one of the events as preferable on the grounds that it has empirical consequences which were observed.

9 Related and Further Work

Several authors have developed deductive techniques for the generation of hypotheses. In [Cox & Pietrzykowski, 1987] hypotheses are constructed from the terminal nodes of linear resolution proofs. Similarly, [Finger & Genesereth, 1985] perform "deductive synthesis" to provide "solutions to design problems" by "finding a *residue* for a given design goal"; and [Poole et al., 1987] use linear resolution for hypothesis generation implemented in the program THEORIST.

In [Eshghi & Kowalski, 1989], [Kakas & Mancarella, 1990] Horn clause logic programming is extended to include abduction with integrity constraints. The approach taken here, differs by the absence of integrity constraints although the process of checking abductive hypotheses by regarding them as updates, and reasoning forwards to integrity constraints, parallels the process of explanation corroboration we describe. There are two important differences between the application (rather than the technique) of explanation corroboration, and the integrity checking process. Firstly, we reason forwards to observables rather than integrity constraints; and secondly, the set of observables can be "dynamic". That is, we may have not made all the relevant observations: it may be necessary to perform an experiment to determine the outcome of corroboration (e.g. through "Query-the-user" [Sergot, 1983] in the case of an expert system). This approach of interactive acquisition of extra information to help decide between different explanations has been studied in [Kunifuji et al, 1986] in the context of Knowledge Assimilation. However, in some domains of application it may be appropriate to use integrity constraints first for reducing the number of possible explanations before beginning the corroboration of explanations. The mechanisms developed in these papers are directly applicable to the incorporation of integrity checking in the hypothetico-deductive proof procedure defined above.

[Sattar & Goebel, 1989] describes how the THEORIST system can be extended through the notion of performing crucial experiments [Popper, 1965] using "crucial literals" (from [Seki & Takeuchi, 1985]) to decide between competing explanatory hypotheses. As we have mentioned above, this can be understood as special case of explanation corroboration used to decide between multiple incompatible explanations. The relative cost of carrying out the experimental tests for corroborating an explanation over the significance of this particular explanation is another feature that needs to be taken into account when further developing the hypothetico-deductive mechanism. For example, in circuit diagnosis [Davis, 1984] the failures are layered into categories according to their likelihood. De Kleer and Williams in [de Kleer & Williams, 1987] use probability and information theory to propose the next "best" test for localizing the fault in the framework of model based diagnosis. These techniques can be used to make our corroboration more efficient.

Conclusions

We have developed a versatile reasoning mechanism and proof procedure, based on the notion of corroboration, that is applicable to a variety of problems and logic-based systems in artificial intelligence. It combines the explanatory capability of hypothesis formation with the benefits of corroboration through deduction for control and testing. Hypothetico-deductive reasoning tackles the problem of undesired multiple explanations for an observation. It extends the isolated application of deductive and abductive reasoning. We have shown how the basic idea behind the reasoning process is to formulate and decide between alternative hypotheses. This is performed through an interaction between the theory and the actual observations. A suitable proof procedure for the implementation of hypothetico-deduction was presented. We have suggested that this form of reasoning might benefit for the use of a "query-the-user" facility. We have demonstrated how hypothetico-deductive reasoning deals with one of the main characteristics of common-sense reasoning, namely incomplete information, through the use of partial corroboration. Finally we have shown how the semantics of hypothetico-deduction can be extended to deal with default theories, in particular temporal theories such as the Event Calculus which include default persistence through the use of negation-as-failure. We have demonstrated how this extension can be applied to deal with Kautz's "stolen car" problem, and the "bloodless" counterpart to the Yale Shooting Problem.

Acknowledgements

We wish to thank R.A. Kowalski for useful discussions on the subject. Thank you, also, to Kate Moorcock for her common-sense examples. This work has been partially carried out under a long term project at Imperial College sponsored by FUJITSU and partially under Esprit project 2409, (EQUATOR). The EQUATOR partners are Ferranti Computer Systems Ltd. (UK), ERIA (Spain), Imperial College (UK), LABEN (Italy), SYSECA (France), CENA (France), CISE (Italy), PTT (Switzerland), SWIFT (Belgium), UCL (UK), ERITEL (Spain) and EPFL (Switzerland).

References

[Cox & Pietrzykowski, 1986] Cox, P. and Pietrzykowski, T. "Causes for Events: Their Computation and Applications"; Proc. CADE-86, J.Siekmann (ed.), Springer-Verlag, Lecture Notes in Computer Science, 1986, pp.608-621.

[Cox & Pietrzykowski, 1987] Cox, P. and Pietrzykowski, T. "General Diagnosis by Abductive Inference"; Technical Report, CS8701, School of C.S., University of Nova Scotia, 1987

[Davis, 1984] Davis, R. "Diagnostic Reasoning Based on Structure and Behaviour"; AI vol. 24, pp. 347-410, 1984.

[Dung, 1991] Dung P. M., Negation as Hypothesis; An Abductive Foundation for Logic Programming, in *Proc. 8th ICLP*, Paris, 1991.

[van Emden & Kowalski, 1976] van Emden M.H. and Kowalski R.A., The Semantics of Predicate Logic as a Programming Language, *Journal of ACM* 23, a (1976), pp. 733-742.

[Eshghi & Kowalski, 1989] Eshghi, K. and Kowalski, R. "Abduction Compared with Negation by Failure"; Proc. 6th ICLP, 1989.

[Finger & Genesereth, 1985] Finger, J. and Genesereth, M. "RESIDUE: A Deductive Approach to design synthesis"; Technical Report no. STAN-CS-85-1035, Dept. of Computer Science, Stanford University, 1985.

[Gelfond & Lifschitz, 1988] Gelfond, M., and Lifschitz, V. The Stable Model Semantics for Logic Programming; *Proceedings of the Logic Programming Conference*, Seattle, 1988.

[Genesereth, 1984] Genesereth, M. "The Use of Design Descriptions in Automated Diagnosis"; AI vol. 24, pp. 411-436, 1984.

[Hanks & McDermott, 1987] Hanks, S. and McDermott, D., Nonmonotonic Logic and Temporal Projection, in *Artificial Intelligence*, vol. 33, pp.379-412, 1987.

[Hempel, 1965] Hempel, C. "Aspects of Scientific Explanation and Other Essays in the Philosophy of Science"; The Free Press, New York, 1965.

[Kakas & Mancarella, 1990] Kakas, A.C. and Mancarella, P. "Generalized Stable Models: a Semantics for Abduction" In Proc. ECAI-90, 1990.

[Kakas & Mancarella, 1990b] Kakas, A.C. and Mancarella, P. "Database Updates through Abduction" in *Proc.16th International Conference on Very Large Data Bases, VLDB '90* , Brisbane, 1990.

[Kakas & Mancarella, 1990c] Kakas, A.C. and Mancarella, P. "On the relation of Abduction and Truth Maintenance" in *Proc. of the 1st Pacific Rim International Conference on AI, PRICAI-90,* Nagoya, Japan 1990.

[Kakas & Mancarella, 1991] Kakas, A.C. and Mancarella, P. "Stable Theories for Logic Programs", to appear in Proc. of ISLP-91, San Diego, 1991.

[Kautz, 1986] Kautz, H., "The Logic of Persistence" in Proc. AAAI-86, pp. 401, 1986.

[Kowalski & Sergot, 1986] Kowalski, R.A. and Sergot, M., A Logic-Based Calculus of Events, New Generation Computing, vol 4, pp. 267, 1986.

[Kunifuji et al, 1986] Kunifuji, S., Tsurumaki, K. and Furukawa, K., "Considerations os a Hypothesis-based Reasoning System" Journal of Japanese Society for Artificial Intelligence vol. 1 no. 2, pp.228-237, 1986.

[de Kleer & Williams, 1987] de Kleer, J. and Williams, B.C. "Diagnosing Multiple Faults"; Artificial Intelligence vol 32, pp. 97-130, 1987.

[McCarthy, 1980] McCarthy, J., Circumscription: A Form of Non-monotonic Reasoning; in *Artificial Intelligence*, vol. 13, pp.27-39, 1980.

[Morgenstern & Stein, 1988] Morgenstern, L. and Stein, L., "Why Things Go Wrong: A Formal Theory of Causal Reasoning"; Proc. AAAI '88, p.518ff.

[Poole, 1988] Poole, D. "Representing Knowledge for Logic-Based Diagnosis" In Proc.of the FGCS, pp.1282-1290, 1988.

[Poole et al., 1987] Poole, D., Goebel, R., and Aleliunas, R. "Theorist: A Logical Reasoning System for Defaults and Diagnosis"; in *The Knowledge Frontier: Essays in the Representation of Knowledge*, by N.Cercone and G.McCalla (ed.s), Springer-Verlag, New York, 1987, pp.331-352.

[Pople, 1985] Pople, H. "Coming to Grips with the Multiple Diagnosis Problem"; in*The Logic of Discovery and Diagnosis in Medicine*, Schaffner, K. (ed.), University of California Press, 1985.

[Popper, 1959] Popper, K. "The Logic of Scientific Discovery"; Basic Books, New York, 1959.

[Popper, 1965] Popper, K. "Conjectures and Refutations: The Growth of Scientific Knowledge";Harper Torch,New York, 1965.

[Reggia at al., 1983] Reggia, J., Nau, D., and Wang, P. "Diagnostic Expert System Based on a Set Covering Model"; Int. J. Man-machine Studies, vol. 19, pp.437-460, 1983.

[Reggia & Nau, 1984] Reggia, J.A. and Nau, D.S. " An Abductive Non-Monotonic Logic"; in Workshop on Non-Monotonic Reasoning, New Paltz, N.Y., 1984.

[Reiter, 1980] Reiter, R., "A Logic for Default Reasoning" Artificial Intelligence vol. 13, pp. 81-132, 1980.

[Sattar & Goebel, 1989] Sattar, A. and Goebel, R. "Using Crucial Literals to Select Better Theories"; Technical Report, Dept. of CS, University of Alberta, Canada, June 1989.

[Seki & Takeuchi, 1985] Seki, H. and Takeuchi, A. "An Algorithm for Finding a Query which Discriminates Competing Hypotheses"; Technical Report TR-143, Institute for New Generation Computer Technology, Tokyo, Japan, October 1985.

[Sergot, 1983] Sergot, M. "A Query-the-user Facility for Logic Programming"; Integrated Interactive Computer Systems, by P.Degano and E.Sandewell (ed.s), North Holland Press, pp.27-41.

[Shoham, 1988] Shoham, Y., "Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence"; MIT Press, 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

ACYCLIC DISJUNCTIVE LOGIC PROGRAMS WITH ABDUCTIVE PROCEDURE AS PROOF PROCEDURE

Phan Minh Dung

Division of Computer Science
Asian Institute of Technology
GPO Box 2754, Bangkok 10501, Thailand.
E-mail: dung@ait.th

Abstract

We introduce and study a natural subclass of the locally stratified disjunctive logic programs, the class of acyclic disjunctive logic programs which extends the class of acyclic normal logic programs in [AB].

We show that each acyclic disjunctive program P can be transformed into an equivalent normal program N(P) where the equivalence between P and N(P) means that each perfect model of P is a stable model of N(P) and vice versa.

We show that the Eshghi and Kowalski's abductive procedure [EK,Dun] is sound with respect to the stable semantics of N(P). Thus this procedure can be used as a proof procedure for acyclic disjunctive programs.

We give sufficient conditions for the completeness and termination of the abductive procedure.

1. Introduction

Let us consider the following example

Example P: p v q

The semantics of P is defined by its two minimal models {p},{q}.

Let us translate P into N(P): p <- ¬ q
q <- ¬ p

N(P) has two stable models {p},{q}. So P and N(P) are equivalent wrt stable semantics.
//

What can we gain from such translation ??

The gain is indeed significant. While no proof procedure for general disjunctive programs wrt stable semantics has been given so far in the literature, the Eshghi-Kowalski's abductive procedure given in [EK] and studied extensively in [Dun], is such a one for normal logic programs. Hence for those disjunctive programs which can be transformed into an equivalent normal programs, the Eshghi-Kowalski's abductive procedure can be used as a proof procedure for stable semantics.

Acyclic disjunctive programs constitute such a class of programs. Intuitively, an acyclic disjunctive program is a program whose atom dependency graph contains no loop. The class of acyclic disjunctive programs is a natural extension of the class of acyclic normal logic programs in [AB]. Similarly to [AB], we will show that several ways to define the semantics of logic programs, e.g. the predicate completion, perfect model semantics, stable model semantics etc., coincide in the case of acyclic disjunctive programs. The most striking characterization of acyclic disjunctive programs is that each program in this class can be transformed into an equivalent normal logic programs which themself exhibit a remarkable termination behavior as their atom dependency graph does not contain any positive loop. This result suggests immediately that the abductive procedure can be used as a proof procedure for acyclic disjunctive programs.

The paper is organized as follows: In the next paragraph, we define the acyclic disjunctive programs. Then in section 3, we show that each acyclic disjunctive program P can be transformed into an equivalent normal program N(P). In section 4, we show the soundness of the abductive procedure with respect to the stable semantics of N(P). In section 5, we give sufficient condition for the completeness of the abductive procedure.

2. Preliminary

A literal is either an atom or the negation of an atom. A disjunctive clause is a clause of the form A_1 v...v A_n <- $L_1,...,L_m$ where $0<n$, $0\leq m$ and A_i's are atoms and L_j's are literals. If n=1, then a disjunctive clause is called a normal clause. The head and body of a clause C are denoted by head(C) and body(C) respectively. Further, pos(C) denotes the set of atoms occurring positively in the body of C while neg(C) denotes the set of atoms under negation in the body of C. A disjunctive program is a finite set of disjunctive clauses. Similarly, a normal program is a finite

set of normal clauses. The Herbrand base of a program P is denoted by HB_P. As usual, a Herbrand interpretation is considered as a subset of HB_P. The set of all ground instances of clauses of a disjunctive program P is denoted by G_P. If L is a literal then $\neg$ L denotes the complementary of L. If S is a set of literals, then $\neg .S = \{ \neg L \mid L \in S \}$.

A disjunctive program P is locally stratified [Pr1] if it is possible to decompose the Herbrand base of P into disjoint sets, called strata $H_0,H_1,...,H_\alpha,...,H_\tau,...$ where $\alpha < \tau$ and τ is a countable ordinal so that for each ground clause in G_P

$$C_1 \text{ v..v } C_k \text{ <- } A_1,...,A_n,\neg B_1,...,\neg B_m$$

(i) all C_i belong to the same stratum, say H_r.
(ii) all A_i belong to $U\{ H_j \mid j \leq r \}$
(iii) all B_i belong to $U\{ H_j \mid j < r \}$

The intended semantics of a locally stratified disjunctive program is captured by its perfect models [Pr1,Pr2]. A more general approach to semantics of logic programs is the stable model semantics [GL] which coincides with the perfect model semantics in the class of locally stratified programs [GL]. Since the definition of stable model semantics is simpler than that of perfect model semantics, we choose to work with the former in this paper.

Let M be a Herbrand interpretation of P. The Gelfond-Lifschitz transformation of P wrt M is the program $GL(P,M) = \{ head(C) \text{ <- } pos(C) \mid C \in G_P \text{ and } neg(C) \cap M = \phi \}$. M is a stable model of P iff M is a minimal model of GL(P,M) [Pr2,GL].

We introduce now the acyclic disjunctive programs.

Definition

A disjunctive program P is acyclic if it is possible to decompose the Herbrand base of P into disjoint sets, called strata $H_0,H_1,...,H_i,..$ where i is a natural number so that for each ground clause in G_P

$$C_1 \text{ v..v } C_k \text{ <- } A_1,...,A_n,\neg B_1,...,\neg B_m$$

(i) all C_i belong to the same stratum, say H_r.

(ii) all A_i and B_i belong to $U\{ H_j \mid j < r \}$

//

Since acyclic programs are locally stratified, their intended semantics is the perfect model semantics.

3. Transforming Acyclic Disjunctive Programs into Normal Programs

Let us introduce some new notations. Let D be a disjunction of atoms. D is canonical if the atoms in D are pairwise different. For each disjunction D, the canonical form of D, denoted by can(D), is a disjunction containing only distinct atoms in D and is equivalent to D. A disjunction D' is a factor of D with most general unifier (mgu) Θ if D' is can(D) and Θ is the identity substitution or there are two or more unifiable atoms in D with mgu Θ and D' is can(DΘ). For example, the disjunction p(x,a) v p(b,y) has two factors: one is the disjunction itself and the other is p(b,a) with the mgu {b/x,a/y}.

The **normal form of P, written N(P)**, is constructed as follows:

Let $C: A_1 \text{ v...v } A_n \text{ <- } L_1,...,L_m$. *Define*

$$N(C) = \{ A \text{ <- } A_1',...,A_k',L_1\Theta,..,L_m\Theta \mid$$

$$A \text{ v } A_1'\text{v..v } A_k' \text{ is a factor of } A_1\text{v..v } A_n \text{ with mgu } \Theta\}$$

$$N(P) = U\{ N(C) \mid C \in P \}$$

Example P: p(x,a) v p(b,y) <-

N(P): p(x,a) <- $\neg$ p(b,y)
p(b,y) <- $\neg$ p(x,a)
p(b,a) <-

//

It has been showed [DK] that each minimal Herbrand model of a positive disjunctive programs is a model of the Clark's completion of N(P). In this chapter, we are interested in the more general question about the relationship between the stable models of P and N(P).

The following theorem shows the equivalence between P and N(P) for acyclic disjunctive programs.

Theorem 1 Let P be an acyclic disjunctive program P, and M be a Herbrand interpretation of P. Then M is a stable model of P iff M is a stable model of N(P).

Proof "=>" Let $Q = GL(G_P,M)$. Since M is a stable model of P, M is a minimal model of Q. Since M is a minimal model of Q, for each $A \in M$, there is a clause A v A_1 v.. v A_n <- Body in Q such that for each i: $A_i \notin M$ and Body is true in M. Hence, for each $A \in M$, there is a clause A <- Body' in $G_{N(P)}$ such that Body' is true in M. Thus, there exists a clause C' in $GL(G_{N(P)},M)$ such that head(C')=A and body(C') is true in M. Since P is acyclic, $GL(G_{N(P)},M)$ is acyclic, too. It follows, that M is the least Herbrand model of $GL(G_{N(P)},M)$. So M is a stable model of N(P).

"<=" Let M be a stable model of N(P). Since $GL(G_{N(P)},M) = GL(N(GL(G_P,M)),M)$, M is also a stable model of $N(GL(G_P,M))$. Thus M is a minimal model of $GL(G_P,M)$. Hence M is a stable model of P.

//

Corollary Let P be an acyclic disjunctive program, N(P) be its normal form. Then a Herbrand interpretation M is a perfect model of P iff M is a **stable model** of N(P).

//

The following example shows that in general, the above theorem does not hold.

Example Let P: a <- b
b <- a
a v b

N(P): a <- b
b <- a
a <- ¬ b
b <- ¬ a

It is clear that P is not acyclic. It is easy to see that N(P) has no stable model while the unique minimal model of P is {a,b}.
//

Since each locally stratified disjunctive program posseses at least one perfect model [Pr1,Pr2], it is obvious that there exists at least one stable model for N(P). So

Corollary If P is acyclic, then N(P) posseses at least one stable model.

//

The following theorems give important characterizations of the normal form of a acyclic disjunctive program.

Theorem 2 Let P be an acyclic disjunctive program. Then each stable model of N(P) is a Herbrand model of comp(N(P) and vice versa where comp(N(P)) denotes the Clark's predicate completion [Cla,Llo] of N(P).

//

Theorem 3 The three-valued semantics and the two-valued semantics of comp(N(P)) are equivalent in the sense that each three-valued model of comp(N(P)) can be extended into an two-valued **one.**

//

Let L be **a ground literal. We say that** L **holds** with respect to the stable semantics of P, written $P \models_s L$, if L is true in each stable model of P. We say P U {L} is **stable-consistent** if there exists one stable model of P in which L is true.

Summary

Let P be an acyclic disjunctive program, and L be a ground literal.

1) $P \models_s L$ iff $N(P) \models_s L$.

2) P U {L} is stable-consistent iff

N(P) U {L} is stable-consistent iff

comp(N(P)) U {L} is consistent.

//

The question of basic interest to us now is:

(*) "Given an acyclic disjunctive program P and a ground literal L, is P U {L} stable-consistent ?"

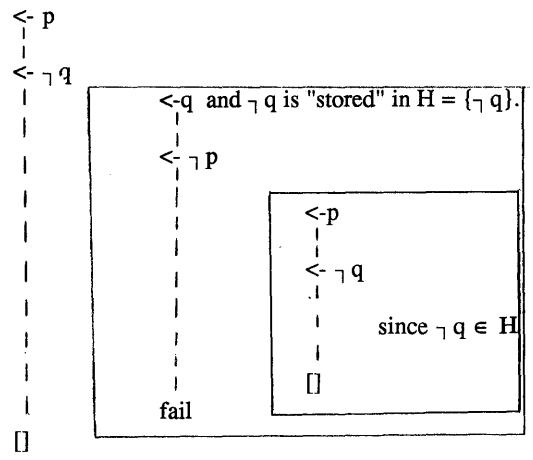
Eshghi and Kowalski have developed an abductive procedure [EK,Dun] which takes as input a query G and a normal program P, and delivers as output a set of ground negative literals H such that P U H U {G} is stable-consistent. From the above obtained results, it is clear that this abductive procedure can be used as a proof procedure for the question (*).

4. The Eshghi and Kowalski's Abductive Procedure

Before presenting the formal definition of the abductive procedure, let us explain the algorithm informally by an example.

Example P: p <- ¬ q
q <- ¬ p

We want to check whether p belongs to some stable model of P, i.e. whether P U {p} is stable-consistent. It is clear that the SLDNF-resolution will not terminate for this goal due to the existence of a negative loop. To avoid getting trapped in this loop, the abductive procedure uses a loop check by "storing" all "encountered" negative literals in a set H. If a selected subgoal belongs to H, then the respected goal is simplified by deleting the selected subgoal from it.



//

Let us recall now the formal definition of the abductive procedure from [EK,Dun].

Let P be a normal logic program.

A **derivation** from (G_1,H_1) to (G_n,H_n) (wrt P) is a sequence

$$(G_1,H_1),(G_2,H_2),...,(G_n,H_n)$$

such that, for each i, $1 \le i < n$, G_i has the form <—l,l' where (without loss of generality) l is selected, and l' is a (possibly empty) collection of atoms, H_i is a set of negative literals, and

ab1) If l is positive

then G_{i+1} = C and $H_{i+1} = H_i$

where C is the resolvent of some clause in P with the clause G_i on the selected literal l.

ab2) If l is negative and $l \in H_i$

then G_{i+1} = <—l' and $H_{i+1} = H_i$

ab3) If l is negative ($l = \neg k$) and $l \notin H_i$ and there is a consistency derivation from $(\{<-k\},H_i \cup \{l\})$ to (ϕ,H')

then G_{i+1} = <—l' and H_{i+1} = H'

An **abductive refutation** is an abductive derivation to a pair ([],H).

A **consistency derivation** from (F_1,H_1) to (F_n,H_n) (wrt P) is a sequence

$$(F_1,H_1), (F_2,H_2),..., (F_n,H_n)$$

such that, for each i, $0 < i \le n$, F_i has the form $\{<-l,l'\} \cup F_i'$, where (without loss of generality) the clause <—l,l' has been selected (to continue the search), l is selected, and

co1) If l is positive

then F_{i+1}= C' U F_i' and $H_{i+1} = H_i$

where C' is the set of all resolvents of clauses in P with the selected clause on the selected literal, and [] ∉ C'.

co2) If l is negative, $l \in H_i$ and l' is not empty

then F_{i+1}= {<—l'} U F_i' and $H_{i+1} = H_i$

co3) If l is negative ($l = \neg k$), $l \notin H_i$

then if there is an abductive derivation from (<—k,H_i) to ([],H')

then $F_{i+1} = F_i'$ and H_{i+1} = H'
else if l' is not empty
then F_{i+1}= {<—l'} U F_i'
and $H_{i+1} = H_i$

A consistency derivation of the goal $(\{G\},\phi)$ is a sequentialization of the search tree of G. This sequentialization is necessary because of the need to accumulate the hypotheses found during this process.

We say that the abductive procedure is **sound** with respect to the stable semantics if whenever there exists a refutation from (<—A,ϕ) to ([],H) for $A \in HB$ then there exists a stable model M such that $A \in M$ and $H \cap M = \phi$.

We say that the abductive procedure is **complete** with respect to the stable semantics if for each ground literal L, if P U {L} is stable-consistent then there exists a refutation for the goal (<-L,ϕ).

Note that in general, the abductive procedure is not sound with respect to the stable semantics, but it is sound with respect to the preferential semantics which is a generalization of stable semantics [Dun]. But since these two semantics coincide for programs N(P) where P is a acyclic disjunctive programs, the soundness with respect to the stable semantics follows directly from the soundness with respect to the preferential semantics.

Theorem 4 (Soundness of the Abductive Procedure)

Let P be an acyclic disjunctive program and (<- A,ϕ),..,([],H) be a refutation with respect to the program N(P). Then there exists a stable model M of P such that A $\in$ M and H $\cap$ M = ϕ.

Proof (Sketch) Let H_0,..,H_i,... be the strata of P. Let P_i consist of those clauses A_1 v ...v A_n <- Bd in G_P such that all A_j belong to H_i. By induction, we can prove that for each i, the stable semantics and preferential semantics [Dun] of P_i coincide. It follows then that the stable and preferential semantics of P coincide. The theorem follows immediately from the fact that the abductive procedure is sound wrt preferential semantics [Dun].
//

Using Abductive Procedure For Skeptical Reasoning

The question of this chapter is:

> "Given a logic program P and a ground literal L, does L hold with respect to the stable semantics of P ?"

The following lemma shows that if the abductive procedure is complete, then it can be used to as a proof procedure for skeptical reasoning.

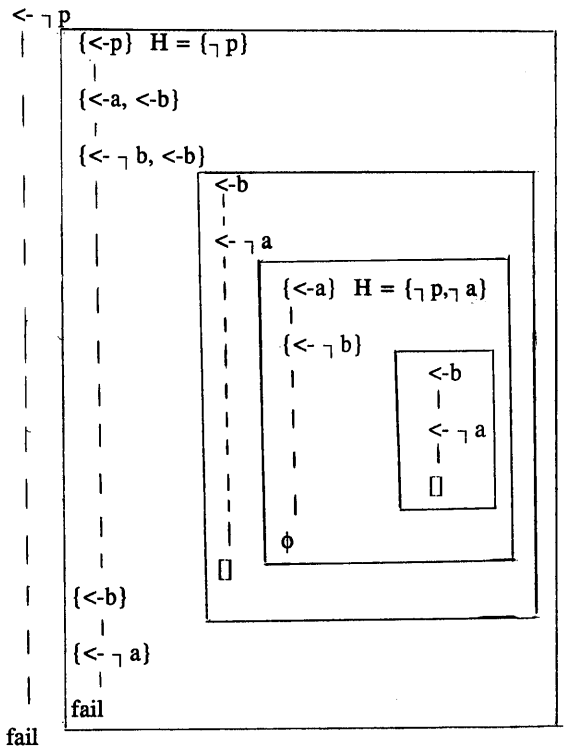
Lemma Let L be a ground literal and assume that the abductive procedure is sound and complete with respect to the stable semantics.

If there exists no refutation for (<- ¬ L,ϕ) then P $\models_s$ L.

If the abductive procedure terminates for ground goals, then it is decidable whether an arbitrary ground literal L holds with respect to the stable semantics.
//

Example
p <- a
p <- b
a <- ¬ b
b <- ¬ a

Since the abductive procedure is complete for this program, the above lemma can be used to check whether p holds wrt stable semantics.



As there is no refutation for (<- ¬ p,ϕ), P $\models_s$ p.
//

The applicability of the abductive procedure as a proof procedure for skeptical reasoning is based on its completeness. In the following paragraph, sufficient conditions for the completeness of abductive procedure are given.

5. Completeness and Termination of the Abductive Procedure

A normal program is said to be **positive acyclic**, written **p-acyclic**, if there is a level mapping |.| assigning each atom A $\in$ HB_P a natural number |A| such that for each clause C in G_P, for each atom A occurring in the head of C and each atom B occurring positively in the body of C, |A| > |B|.

It is not difficult to see that if P is an acyclic disjunctive program then N(P) is always p-acyclic. Note that positive acyclicity is different to local stratifiability, i.e. there exists programs which are p-acyclic and not locally stratified and vice versa.

The atom dependency graph of P is a graph with ground atoms as its nodes such that there exists a positive (resp.

negative) edge from A to B if A occurs in the head, and B occurs positively (resp. negatively) in the body of some clause C in G_P.

An infinite path $(A_1,..,A_n,..)$ of pairwise different atoms in the atom dependency graph of P is said to be a **negative infinite loop** if the path contains infinitely many negative edges. P is said to be **free of infinite negative loop, written INL-free**, if there exists no negative infinite loop in the atom dependency graph of P.

A program P is **allowed** [Llo] if each clause in P satisfies the condition that each variable appearring in the clause appears also in a positive subgoal in the clause body.

Theorem 5 (Completeness of the Abductive Procedure)

Let P be an allowed, p-acyclic, and NIL-free normal program, and L be an arbitrary ground literal. Then the abductive procedure will terminate for the goal $(<-L,\phi)$, and if P U {L} is stable-consistent then there exists a refutation from $(<-L,\phi)$ to ([],H).
//

Let us specify now the class of disjunctive programs such that their normal form N(P) are INL-free. Two disjunctions of atoms A_1 v..v A_n and B_1 v...v B_m are said to be **related** if they have some atom in common. A sequence of disjunctions $D_1,...,D_n,...$ is said to be a **related sequence** if D_i,D_{i+1} are related for each i. A related sequence of disjunctions $D_1,...,D_n,..$ is said to be **prime** if for each i, there exists a common atom A_i in D_i and D_{i+1} such that the sequence $A_1,..,A_n,..$ contains no atom twice. A disjunctive program is said to be **free of prime related sequence (abbreviated as PRS-free)** if no prime related consequence can be built from the instances of the heads of the program clauses of P.

Corollary If P is an allowed, acyclic, PRS-free disjunctive program then the abductive procedure, applied to N(P), is sound and complete wrt perfect model semantic of P.

//

Acknowledgements

First of all, I wish to express my sincere thanks to Robert Kowalski for his generous support as well as for the many spiritful and insightful discussions. The long discussions with Paolo Mancarella, and Tony Kakas on how to find an extension of the abductive procedure which is complete with respect to the preferential semantics, are very helpful. So a lot of thanks to them.

References

[ABW] Apt K., Blair H., Walker A.
'Toward a Theory of Declarative Knowledge'
In Foundations of Deductive Databases & Logic Programming, J. Minker (ed.) 1988

[AB] Apt K., Bezem M.
'Acyclic Programs',
In Proceedings of the ICLP-90, Israel, MIT Press

[Bez] Bezem M.
'Characterizing Termination of logic programs with level mappings',
In Proceedings of the NACLP-89, USA, MIT Press

[Cav] Cavedon L.
'Continuity, consistency and completeness properties of logic programs',
In ICLP 1987, Lisbon, MIT Press

[Cla] Clark,K.L.
'Negation as Failure',
in Logic and Database, Gallaire H., Minker J. (eds), Plenum, New York,1978

[Dun] Dung P.M.
'Negations as hypotheses: an abductive foundation for logic programming'
In Proceedings of Eighth ICLP, 1991, Paris, MIT Press

[DK] Dung P.M., Kanchanasut K.,
'On the generalized predicate completion of non-Horn programs',
In Proc. of NACLP-89, USA, MIT Press

[EK] Eshghi K., Kowalski R.A.
'Abduction Compared with Negation by Failure'
In Proc. of 6th ICLP, 1989

[GL] Gelfond M.,Lifschitz V.
'The stable model semantics for logic programs'
In Proc. of the 5th Int Conf/Sym on Logic Programming, MIT Press, 1988

[HP] Henshen L., Park H.
'Compiling GCWA in indefinite deductive databases',
In Foundation in Deductive Databases and Logic Programming, J. Minker (ed.)

[Kow] Kowalski R.A.
'Logic for problem solving'
Elsvier North Holland, New York, 1979

[Llo] Lloyd J.W.
'Foundations of Logic Programming',
second edition, Springer Verlag, 1987

[Lob] Lobo J.
'Semantics for normal disjunctive logic programs'
PhD thesis, 1991

[SL] Smith B.T., Loveland D.
'A simple near Horn prolog interpreter'
Proc. of the fifth joint conference on logic programming, 1988, USA

[MR] Minker J., Rajasekar A.
'A fixpoint semantics for disjunctive logic programming',
In Journal of Logic programming, to appear

[Pr1] Przymusinski T.C. ,
'On the Declarative Semantics of Deductive Databases and Logic Programs',
In Foundations of Deductive Databases & Logic Programming, J. Minker (ed.) 1988

[Pr2] Przymusinski T.C.
'Extended stable semantics for normal and disjunctive programs',
In Proc. of seventh ICLP, Israel, 1990

[RT] Ross K.A., Topor R.W.
'Inferring negative atoms from dsijunctive databases'
Journal of Automated reasoning, Dec. 1988

[SS] Sakama C., Seki H.
'Possibel model semantics for disjunctive databases'
Preliminary report, ICOT

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Adding Closed World Assumptions to Well Founded Semantics

Luís Moniz Pereira José J. Alferes
Joaquim N. Aparício
AI Centre, Uninova and DCS, U. Nova de Lisboa
2825 Monte da Caparica, Portugal
{lmp,jja,jna}@fct.unl.pt

Abstract

Given a program P we specify an enlargement of its Well Founded Model which gives meaning to the adding of Closed World Assumptions. We do so by proposing the desirable principles of a Closed World Assumption (CWA), and proceed to formally define and apply them to Well Founded Semantics (WFS), in order to obtain a WFS added with CWA, the O-semantics. After an introduction and motivating examples, there follow the presentation of the concepts required to formalize the model structure, the properties it enjoys, and the criteria and procedures which allow the precise characterization of the preferred unique maximal model that gives the intended meaning to the O-Semantics of a program, the O-Model. Some properties are also exhibited that permit a more expedite obtention of the models. Several detailed examples are introduced throughout to illustrate the concepts and their application. Comparison is made with other work, and in the conclusions the novelty of the approach is brought out.

1 Introduction and Motivation

Well Founded Semantics [Van Gelder *et al.*, 1980] has been proposed as a suitable semantics for general logic programs. Its Extended Stable Models (XSM) [Przymusinska and Przymusinski, 1990, Przymusinski, 1990] version has been explored as a framework for formalizing a variety of forms of non-monotonic reasoning [Pereira *et al.*, 1991d, Pereira *et al.*, 1991e] and generalized to deal with contradiction removal and counterfactuals [Pereira *et al.*, 1991a, Pereira *et al.*, 1991b, Pereira *et al.*, 1991c]. The increasing rôle of logic programming extensions as an encompassing framework for these and other AI topics is expounded at length in [Kakas and Mancarella, 1991b], where they argue, and we concur, that WFS is by design overly careful in deciding about the falsity of some atoms, leaving them undefined, and that a suitable form of CWA can be used to safely and undisputably assume false some of the atoms absent from the well founded model of a program. Consider the program P, adapted from [Kakas and Mancarella, 1991a]: $\{a \leftarrow \sim a;\ c \leftarrow \sim a\}$, where $WFM(P) = \{\}$. We argue that the intended meaning of the program may be $\{\sim c\}$, since $\sim a$ may not be true in any model of P, by the first rule, and so, the second rule cannot contradict the assigned meaning. Another way to understand this is that one may safely assume $\sim c$ using a form of CWA on c, since $\sim a$ may not be consistently assumed.

However, when relying on the absence of present evidence about some atom A, we do not always want to assume that $\sim A$ holds, since there may exist consistent assumptions allowing to conclude A. Roughly, we want to define the notion of concluding for the truth of a negative literal $\sim A$ just in case there is no hard nor hypothetical evidence to the contrary, i.e. no consistent set of negative assumptions such that $\sim A$ is untenable.

Consider $P = \{a \leftarrow \sim b;\ b \leftarrow \sim a;\ c \leftarrow a\}$. If we interpret the meaning of this program as its WFM (which is empty), and as we do not have a, a naive CWA could be tempted to derive $\sim c$ based on the assumption $\sim a$. There is however an alternative negative assumption $\sim b$, that if made, defeats the assumption $\sim a$, i.e. the assumption $\sim a$ may not be sustained since it can be defeated by the assumption $\sim b$. We will define later more precisely the notions of sustainability and tenability.

Both programs above have empty well founded

models. We argue that WFS is too careful, and something more can safely be added to the meaning of program, thus reducing the undefinedness of the program, if we are willing to adopt a suitable form of CWA.

We argue that a set $CWA(P)$ of negative literals (assumptions) *added* to a program model $MOD(P)$ by CWA must obey the four principles:

1. $MOD(P) \cup CWA(P) \not\models L$ for any $\sim L \in CWA(P)$. This says that the program model added with the set of assumptions identified by the CWA rule must be **consistent.**
2. There is no other set of assumptions A such that $MOD(P) \cup A \models L$ for some $\sim L \in CWA(P)$. I.e. $CWA(P)$ is **sustainable.**
3. $CWA(P)$ must be **maximal.**
4. $CWA(P)$ must be **unique.**

The paper is organized as follows: in the next section we present some basic definitions. In section 3 we introduce some new definitions, capturing the concepts behind the semantics, accompanied by examples illustrating them. Models are defined and organized into a lattice, and the class of sustainable A-Models is identified. In section 5 we define the O-Semantics of a program P based on the class of maximal sustainable tenable A-Models. A unique model is singled out as the O-Model of P. Afterwards we present some properties of the class of A-Models. Finally, we relate to other semantics and present conclusions.

2 Language

Here we give basic definitions and establish notation ([Monteiro, 1991]). A program is a set of rules of the form: $H \leftarrow B_1, \ldots, B_n, \sim C_1, \ldots, \sim C_m$ $(n \geq 0, m \geq 0)$ or equivalently $H \leftarrow \{B_1, \ldots, B_n\} \cup \sim\{C_1, \ldots, C_m\}$, where $\sim\{A_1, \ldots, A_n\}$ is a shorthand for $\{\sim A_1, \ldots, \sim A_n\}$, and $\sim C$ is short for $\sim\{C1, \ldots, C_m\}$; H,, B_i and C_j are atoms.

The Herbrand Base $B(P)$ of a program P is defined as usual as the set of all ground atoms. An interpretation I of P is denoted by $T \cup \sim F$, where T and F are disjoint subsets of $B(P)$. Atoms in T are said to be *true* in I, atoms in F *false* in I, and atoms in $B(P) - (T \cup F)$ *undefined* in I.

In an interpretation $T \cup \sim F$ a conjunction of literals $\{B_1, \ldots, B_n\} \cup \sim\{C_1, \ldots, C_m\}$ is *true* iff $\{B_1, \ldots, B_n\} \subseteq T$ and $\{C_1, \ldots, C_m\} \subseteq F$, is *false* iff $\{B_1, \ldots, B_n\} \cap F \neq \emptyset$ or $\{C_1, \ldots, C_m\} \cap F \neq \emptyset$, and is *undefined* iff it is neither true nor false.

3 Adding Negative Assumptions to a Program

Here we show how to consistently add negative assumptions to a program P. Informally, it is consistent to add a negative assumption to P if the assumption atom is not among the consequences P after adding the assumption. We also define when a set of negative assumptions is defeated by another, and show how the models of a program, for different sets of negative assumptions added to it, are organized into a lattice.

We begin by defining what it means to add assumptions to a program. This is achieved by substituting *true* for the assumptions, and *false* for their atoms, in the body of all rules.

Definition 3.1 (P+A) The program $P + A$ obtained by adding to a program P a set of negative assumptions $A \subseteq \sim B(P)$ is the result of:

- Deleting all rules $H \leftarrow \{B_1, \ldots, B_n\} \cup \sim C$ from P, such that some $B_i \in A$
- Deleting from the remaining rules all $\sim L \in A$

Definition 3.2 (Assumption Model) An Assumption Model of a program P, or A-Model for short, is a pair $\langle A; M \rangle$ where $A \subseteq \sim B(P)$ and $M = WFM(P + A)$.

Among these models we define the partial order $\leq_a$ in the following way: $\langle A_1; M_1 \rangle \leq_a \langle A_2; M_2 \rangle$ iff $A_1 \subseteq A_2$. On the basis of set union and set intersection among the sets A of negative assumptions, the set of all A-Models becomes organized as a complete lattice.

Having defined assumption models we next consider their consistency. According to the CWA principles above, an assumption $\sim A$ cannot be added to a program P if by doing so A is itself a consequence of P, or some other assumption is contradicted.

Definition 3.3 (Consistent A-Model) An A-Model $\langle A; M \rangle$ is consistent iff $A \cup M$ is an interpretation, i.e. there exists no assumption $\sim L \in A$ such that $L \in M$.

Example 1 Let $P = \{c \leftarrow\sim b;\ b \leftarrow\sim a;\ a \leftarrow\sim a\}$, whose WFM is empty. The A-Model $\langle\{\sim a\};\{a,b,\sim c\}\rangle$ is inconsistent since by adding the assumption $\sim a$ then $a \in WFM(P + \{\sim a\})$. The same happens with all A-Models containing the assumption $\sim a$. The A-Model $\langle\{\sim b,\sim c\};\{c\}\rangle$ is also inconsistent. Thus the only consistent A-Models are $\langle\{\};\{\}\rangle$, $\langle\{\sim b\};\{c\}\rangle$ and $\langle\{\sim c\};\{\}\rangle$. □

Lemma 3.1 *If an A-Model AM is inconsistent then any AM' such that $AM \leq_a AM'$ is inconsistent.*

Proof:[sketch] We prove that for all $\sim a' \in B(P)$, if $\langle A; WFM(P + A)\rangle$ is inconsistent then $\langle A \cup \{\sim a'\}; WFM(P + A \cup \{\sim a'\})\rangle$ is also inconsistent. By definition of consistent A-Model: $\exists \sim b \in A \mid b \in WFM(P+A)$, so it suffices to guarantee that : $b \notin WFM(P + A \cup \{\sim a'\}) \rightarrow a' \in WFM(P + A \cup \{\sim a'\})$.

Consider $b \notin WFM(P + A \cup \{\sim a'\})$. Since $P + A \cup \{\sim a'\}$ only differs from $P + A$ in rules with a' or $\sim a'$, and since b is true in $P + A$, it can be shown a' is also true in $P + A$. As the truth of an atom in the WFM of any program may not rely neither on the truth of itself nor of its complementary, and because the addition of $\sim a'$ to $P + A$ only changes rules with $\sim a'$ or a', the truth value of a' in $P + A \cup \{\sim a'\}$ remains the same, i.e. $a' \in WFM(P + A \cup \{\sim a'\})$. ◇

According to the CWA principles above, an assumption $\sim A$ cannot be sustained if there is some set of consistent assumptions that concludes A. We've already expressed the notion of consistency being used. To capture the notion of sustainability we now formally define how an A-Model can defeat another, and define sustainable A-Models as the nondefeated consistent ones.

Definition 3.4 (Defeating)
A consistent A-Model $\langle A; M\rangle$ is defeated by a consistent $\langle A'; M'\rangle$ iff $\exists \sim a \in A | a \in M'$.

Definition 3.5 (Sustainable A-Models)
An A-Model $\langle A; M\rangle$ is sustainable iff it is consistent and not defeated by any consistent A-Model. Equivalently $\langle\sim S; M\rangle$ is sustainable iff:

$$S \cap \bigcup_{consistent\ \langle A_i;M_i\rangle} M_i = \{\}$$

Example 2 The only sustainable models in example 1 are $\langle\{\};\{\}\rangle$ and $\langle\{\sim b\};\{c\}\rangle$. Note that the consistent A-Model $\langle\{\sim c\};\{\}\rangle$ is defeated by $\langle\{\sim b\};\{c\}\rangle$, i.e. the assumption $\sim c$ is unsustainable since there is a set of consistent assumptions (namely $\{\sim b\}$) that leads to the conclusion c. □

The assumptions part of maximal sustainable A-Models of a program P are maximal sets of consistent Closed World Assumptions that can be safely added to the consequences of P without risking contradiction by other assumptions.

Lemma 3.2 *If an A-Model AM is defeated by another A-Model D, then all A-Models AM' such that $AM \leq_a AM'$ are defeated by D.*

Proof: Similar to the proof of lemma 3.1 above. ◇

Lemma 3.3 *The A-Model $\langle\{\}; WFM(P)\rangle$ is always sustainable.*

Proof: By definition of sustainable. ◇

Theorem 3.4 *The set of all sustainable A-Models is nonempty. On the basis of set union and set intersection among its A sets, the A-Models ordered by $\leq_a$ form a lower semi lattice.*

Proof: Follows directly from the above lemmas. ◇

A program may have several maximal sustainable A-Models.

Example 3 Let
$P = \{c \leftarrow\sim c, \sim b;\ b \leftarrow a;\ a \leftarrow\sim a\}$. Its sustainable A-Models are $\langle\{\};\{\}\rangle$, $\langle\{\sim b\};\{\}\rangle$ and $\langle\{\sim c\};\{\}\rangle$. The last two are maximal sustainable A-Models. We cannot add both $\sim b$ and $\sim c$ to the program to obtain a sustainable A-Model since $\langle\{\sim b,\sim c\};\{c\}\rangle$ is inconsistent. □

4 The O-semantics

This section is concerned with the problem of singling out, among all sustainable A-Models of a program P, one that uniquely determines the meaning of P when the CWA is enforced. This is accomplished by means of a selection criterium that takes a lower semilattice of sustainable A-Models and obtains a subsemilattice of it, by deleting A-Models that in a well defined sense are less preferable, i.e. the untenable ones.

Sustainability of a consistent set of negative assumptions insists that there be no other consistent

set that defeats it (i.e. there is no hypothetical evidence whose consequences contradict the sustained assumptions). Tenability requires that a maximal sustainable set of assumptions be not contradicted by the consequences of adding to it another competing (nondefeating and nondefeated) maximal sustainable set.

The selection process is repeated and ends up with a complete lattice of sustainable A-Models, which defines for every program P its O-Semantics. The meaning of P is then specified by the greatest A-Model of the semantics, its O-Model.

To illustrate the problem of preference among maximal A-Models we give an example.

Example 4
Let $P = \{c \leftarrow\sim c, \sim b;\ b \leftarrow a;\ a \leftarrow\sim a\}$, whose sustainable A-Models are $\langle\{\};\{\}\rangle$, $\langle\{\sim b\};\{\}\rangle$, and $\langle\{\sim c\},\{\}\rangle$. Because we wish to maximize the number of negative assumptions we consider the maximal A-Models, which in this case are the last two. The join of these maximal A-Models, $\langle\{\sim b, \sim c\};\{c\}\rangle$, is per force inconsistent, in this case wrt c. This means that when assuming $\sim c$ there is an additional set of assumptions entailing c, making this A-Model *untenable*. But the same does not apply to $\sim b$. Thus the preferred A-Model is $\langle\{\sim b\},\{\}\rangle$, and the A-Model $\langle\{\sim c\};\{\}\rangle$ is said untenable. The rationale for the preference is grounded on the fact that the inconsistency of the join arises wrt c but not wrt b. □

Definition 4.1 (Candidate Structure)
A Candidate Structure CS of a program P is any subsemilattice of the lower semi lattice of all sustainable A-Models of P.

Definition 4.2 (Untenable A-Models)
Let $\{\langle A_1; M_1\rangle, \ldots, \langle A_n; M_n\rangle\}$ be the set of all maximal A-Models in Candidate Structure CS. Let $J = \langle A_J; M_J\rangle$ be the join of all such A-Models, in the complete lattice of all A-Models . An A-Model $\langle A_i; M_i\rangle$ is untenable wrt CS iff it is maximal in CS and there exists $\sim a \in A_i$ such that $a \in M_J$.

Proposition 4.1 *There exists no untenable A-Model wrt a Candidate Structure with a single maximal element.*

Proof: Since the join coincides with the unique maximal A-Model, which is sustainable by definition of CS, then it cannot be untenable. ◊

The Candidate Structure left after removing all untenable A-Models of a CS, may itself have several maximal elements, some of which might not be maximal A-Models in the initial CS. If the removal of untenable A-Models is performed repeatedly on the retained Candidate Structure, a single maximal element is eventually obtained, albeit the bottom element of all the CSs.

Definition 4.3 (Retained CS) *The Retained Candidate Structure $R(CS)$ of a Candidate Structure CS is:*

- *CS if it has a single maximal A-Model, i.e. CS is a complete lattice.*
- *Otherwise, let Unt be the set of all untenable A-Models wrt CS. Then $R(CS) = R(CS - Unt)$.*

Definition 4.4 (The O-Semantics)
The O-Semantics of a program P is defined by the Retained Candidate Structure of the semilattice of all sustainable A-Models of P.

Let $\langle A; M\rangle$ be its maximal element. The intended meaning of P is $A \cup M$, the O-Model of P.

Theorem 4.1 (Existence of O-Semantics)
The Retained Candidate Structure of the semilattice of all sustainable A-Models is nonempty.

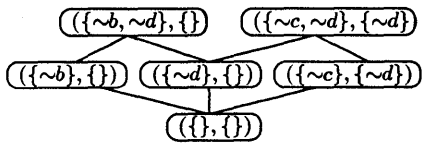
Proof:[sketch] It suffices to guarantee that at each iteration with more than one maximal A-Model at least one is untenable. This is done by contradiction: suppose no maximal A-Model is untenable. Then their join would be the single maximal sustainable one, and so could not be untenable, in the previous and final iteration; accordingly the supposed models cannot be maximal.

When there is a single maximal A-Model then the structure is a complete lattice, since at each iteration only maximal A-Models were removed. This lattice is nonempty since its bottom $\langle\{\};WFM(P)\rangle$ is always sustainable and can never be untenable. ◊

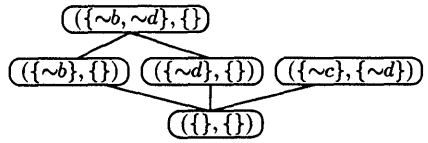
5 Examples

In this section we display some examples and their O-Semantics. Remark that indeed the O-Models obtained express the safe CWAs compatible with the WFMs (which are all $\{\}$).

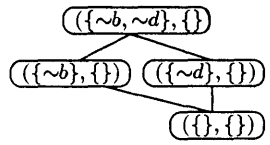
Example 5
Let $P = \{a \leftarrow\sim a;\ b \leftarrow a; c \leftarrow\sim c, \sim b;\ d \leftarrow c\}$ The semilattice of all sustainable A-Models CS is:



The join of its maximal A-Models is $\langle\{\sim b, \sim c, \sim d\}; \{c, \sim d\}\rangle$. Consequently, the maximal A-Model on the right is untenable since it contains $\sim c$ in the assumptions, and c is a consequence of the join. So $R(CS) = R(CS')$ where CS' is:



The join of all maximal elements in CS' is the same as before and the only untenable A-Model is again the maximal one having $\sim c$ in its assumptions. Thus $R(CS) = R(CS'')$ where CS'' is:



So the O-Model is $\{\sim b, \sim d\}$. Note that if P is divided into $P_1 = \{c \leftarrow\sim c, \sim b;\ d \leftarrow c\}$ and $P_2 = \{a \leftarrow\sim a;\ b \leftarrow a\}$, the O-models of P_1 and P_2 both agree on the only common literal $\sim b$. So $\sim b$ rightly belongs to the O-models of P. □

Example 6 Let $P = \{q \leftarrow\sim p;\ p \leftarrow a;\ a \leftarrow\sim b;\ b \leftarrow\sim c;\ c \leftarrow\sim a\}$. Its only consistent A-Models are $\langle\{\};\{\}\rangle$, $\langle\{\sim p\};\{q\}\rangle$ and $\langle\{\sim q\};\{\}\rangle$. As this last one is defeated by the second, the only sustainable ones are the first two. Since only one is maximal, these two A-Models determine the O-Semantics, and the meaning of P is $\{\sim p, q\}$, its O-Model. Note that if the three last rules, forming an *"undefined loop"*, are replaced by another *"undefined loop"* $a \leftarrow\sim a$, the O-model is the same. This is as it should, since the first two rules conclude nothing about a. □

Example 7 Let $P = \{p \leftarrow a, b;\ a \leftarrow\sim b;\ b \leftarrow\sim a\}$. The A-Models with $\sim b$ in their assumptions defeat A-Models with $\sim a$ in their assumptions and vice-versa. Thus the O-Semantics is determined by $\langle\{\};\{\}\rangle$ and $\langle\{\sim p\};\{\}\rangle$, and the meaning of P is $\{\sim p\}$, its O-Model. □

Example 8 Let $P = \{c \leftarrow\sim c, \sim b;\ b \leftarrow\sim c, \sim b;\ b \leftarrow a;\ a \leftarrow\sim a\}$. Its sustainable A-Models are $\langle\{\};\{\}\rangle$, $\langle\{\sim b\};\{\}\rangle$ and $\langle\{\sim c\};\{\}\rangle$. The join of the maximal ones is $\langle\{\sim b, \sim c\};\{b, c\}\rangle$, and so both are untenable. Thus the Retained Candidate Structure has the single element $\langle\{\};\{\}\rangle$ and the meaning of P is $\{\}$ □

6 Properties of Sustainable A-Models

This section explores properties of sustainable A-Models that provide a better understanding of them, and also give hints for their construction without having to previously calculate all A-Models.

We begin with properties that show how our models can be viewed as an extension to Well Founded Semantics (WFS). As mentioned in [Kakas and Mancarella, 1991a], negation in WFS is based on the notion of support, i.e. a literal $\sim L$ only belongs to an Extended Stable Model (XSM) if all the rules for L (if any) have false bodies in the XSM. In contradistinction, we are interested in negations as consistent hypotheses that cannot be defeated. To that end we weaken the necessary (but not sufficient) conditions for a negative literal to belong to a model as explained below. We still want to keep the necessary and sufficient conditions of support for positive literals. More precisely, knowing that XSMs must obey, among others, the following conditions cf. [Monteiro, 1991]:

- If there exists a rule $p \leftarrow B$ in the program such that B is true in model M then p is also true in M (*sufficiency of support for positive literals*).
- If an atom $p \in M$ then there exists a rule $p \leftarrow B$ in the program such that B is true in M (*necessity of support for positive literals*).
- If all rule bodies for p are false in M then $\sim p \in M$ (*sufficiency of support for negative literals*).
- If $\sim p \in M$ then all rules for p have false bodies in M (*necessity of support for negative literals*).

Our consistent A-models, when understood as the union of their pair of elements, assumptions A and $WFM(P+A)$, need not obey the fourth condition. Foregoing it condones making negative assumptions. In our models an atom might be false even if it has a rule whose body is undefined. Thus, only false atoms with an undefined rule body are candidates for having their negation added to the $WFM(P)$.

Proposition 6.1 *Let $\langle A; M\rangle$ be any consistent A-Model of a program P. The interpretation $A \cup M$ obeys the first three conditions above.*

Proof: Here we prove the satisfaction of the first condition. The remaining proofs are along the same lines.

If $\exists p \leftarrow b_1,\ldots,b_n,\sim c_1,\ldots,\sim c_m \in P \mid \{b_1,\ldots,b_n,\sim c_1,\ldots,\sim c_m\} \subseteq A \cup M$ then $b_i \in M$ $(1 \leq i \leq n)$ and $\sim c_j \in M$ or $\sim c_j \in A$ $(1 \leq j \leq m)$. Let $p \leftarrow b_1,\ldots,b_n,\sim c_l,\ldots,\sim c_k (l \geq 1, k \leq m)$ be the rule obtained from an existing one by removing all $\sim c_j \in A$, which is, by definition, a rule of $P+A$. Thus there exists a rule $p \leftarrow B$ in $P+A$ such that $B \subseteq WFM(P+A) = M$. Given that the WFM of any program must obey the first condition above, $p \in WFM(P+A)$. $\diamondsuit$

Next we state properties useful for more directly finding the sustainable A-Models.

Proposition 6.2 *There exists no consistent A-Model $\langle A; M\rangle$ of P with $\sim a \in A$ such that $a \in WFM(P)$.*

Proof: Let $\langle A; M\rangle$ be an A-Model such that $\sim a \in A$ and $a \in WFM(P)$. It is known that the truth of any $a \in WFM(P)$ cannot be supported neither on itself nor on $\sim a$. If $A = \{\sim a\}$ then, after adding $\{\sim a\}$ to the program, the rules supporting the truth of a remain unchanged, i.e. $a \in WFM(P + \{\sim a\})$, and thus $\langle\{\sim a\}; WFM(P + \{\sim a\})\rangle$ is inconsistent. It follows, from lemma 3.1, that all A-Models $\langle A; M\rangle$ such that $\{\sim a\} \subseteq A$ are inconsistent. $\diamondsuit$

Hence, A-Models not obeying the above restriction are not worth considering as sustainable.

Proposition 6.3
If a negative literal $\sim L \in WFM(P)$ then there is no consistent A-Model $\langle A; M\rangle$ of P such that $L \in M$.

Proof:[sketch] We prove that if $L \in M$ for a given A-Model $\langle A; M\rangle$ of P then $\langle A; M\rangle$ is inconsistent. If $L \in M$ there must exist a rule $L \leftarrow B, \sim C$ in P such that $B \cup \sim C \subseteq M \cup A$ and $B \cup \sim C$ is false in $WFM(P)$, i.e. there must exist $L \leftarrow B, \sim C$ in P with at least one body literal true in $M \cup A$ and false in $WFM(P)$. If that literal is an element of $\sim C$, by proposition 6.2, $\langle A; M\rangle$ is inconsistent (its corresponding atom is true in $WFM(P)$ and false in $M \cup A$). If it is an element of B this theorem applies recursively, ending up in a rule with empty body, an atom with no rules or a loop without an interposing $\sim l$. By definition of $WFM(P+A)$ the truth value of literals in these conditions can never be changed. $\diamondsuit$

Theorem 6.1 *If $\sim L \in WFM(P)$ then $\sim L \in M$ in every consistent A-Model $\langle A; M\rangle$ of P.*

Proof: Given proposition 6.3, it suffices to prove that L is not undefined in any consistent A-Model of P. The proof is along the lines of that of the proposition above. $\diamondsuit$

Consequently, all supported negative literals in the $WFM(P)$, which includes those without rules for their atom, belong to every sustainable A-Model.

Lemma 6.2 *Let $WFM(P) = T \cup \sim F$. For any subset S of $\sim F$, $WFM(P) = WFM(P+S)$.*

Proof: This lemma is easily shown using the definition of $P+A$ and the properties of the WFM. $\diamondsuit$

Theorem 6.3 *Let $WFM(P) = T \cup \sim F$ and $\langle A; WFM(P+A)\rangle$ be a consistent A-Model, and let $A' = A \cap \sim F$. Then $WFM(P+A) = WFM(P+(A-A'))$.*

Proof: Let $P' = P + (A - A')$, and $WFM(P) = T \cup \sim F$. By theorem 6.1 $\sim F \subseteq WFM(P')$. So, by lemma 6.2, $WFM(P') = WFM(P' + \sim F) = WFM([P+(A-(A \cap \sim F))] + \sim F)$. By definition of $P+A$ it follows that $(P+A_1)+A_2 = P+(A_1 \cup A_2)$. Thus $WFM(P')$ is:

$$\begin{aligned} & WFM(P + [(A - (A \cap \sim F) \cup \sim F]) \\ = \; & WFM(P + A) \end{aligned}$$

$\diamondsuit$

This theorem shows that sets of assumptions including negative literals of $WFM(P)$ are not worth considering since there exist smaller sets having exactly the same consequences $A \cup M$ and, by proposition 6.3 the larger sets are not defeatable by reason of negative literals from the $WFM(P)$.

Another important hint for calculating the sustainable A-Models is given by lemma 3.1. According to it one should start by calculating A-Models with smaller assumption sets, so that when an inconsistent A-Model is found, by the lemma, sets of assumptions containing it are unworth considering.

Example 9
Let $P = \{p \leftarrow\sim a, \sim b;\ a \leftarrow c, d;\ c \leftarrow\sim c;\ d\}$. The least A-Model is $\langle\{\};\{d, \sim b\}\rangle$ where $\{d, \sim b\} = WFM(P)$. Thus sets of assumptions containing $\sim d$ or $\sim b$ are not worth considering. Take now, for example, the consistent A-Model $\langle\{\sim a\};\{d, \sim b, p\}\rangle$, which we retain. Consider $\langle\{\sim c\};\{c, a, \sim p\}\rangle$; as this A-Model is inconsistent we do not retain it nor consider any other A-Models with assumption sets containing $\sim c$. Now we are left with just two more A-Models worth considering: $\langle\{\sim p\};\{d, \sim b\}\rangle$ which is defeated by $\langle\{\sim a\};\{d, \sim b, p\}\rangle$; and $\langle\{\sim p, \sim a\};\{d, \sim b, p\}\rangle$ which is inconsistent. Thus the only two sustainable A-Models are $\langle\{\};\{d, \sim b\}\rangle$ and $\langle\{\sim a\};\{d, \sim b, p\}\rangle$. In this case, the latter is the single maximal sustainable A-Model, and thus uniquely determines the intended meaning of P to be $A \cup M = \{\sim a, d, \sim b, p\}$. □

7 Relation to other work

Consider the following program ([Van Gelder *et al.*, 1980]):

$$P = \{p \leftarrow q, \sim r, \sim s;\ q \leftarrow r, \sim p;\ r \leftarrow p, \sim q;\ s \leftarrow\sim p, \sim q, \sim r\}$$

In [Przymusinska and Przymusinski, 1990] they argue that the intended semantics of this program should be the interpretation $\{s, \sim p, \sim q, \sim r\}$ due to the mutual circularity of p, q, r. This model is precisely the meaning assigned to the program by the O-Semantics, its O-Model. Note that WFS identifies the (3-valued) empty model as the meaning of the program. This is also the model provided by stable model semantics [Gelfond and Lifschitz, 1988]. The weakly perfect model semantics for this program is undefined as noticed in [Przymusinska and Przymusinski, 1990].

The EWFS [Baral *et al.*, 1990] is also an extension to the WFM based on the notion of GCWA [Minker, 1987]. Roughly EWFS moves closer than the WFM (in the sense of being less undefined) to being the intersection of all minimal Herbrand models of P [Dix, 1991]:

$$EWFM(P) =_{def} WFM(P) + \langle T(WFM(P)), F(WFM(P))\rangle$$

where: $T(I) =_{def} True(I - MIN - MOD(P))$, $F(I) =_{def} False(I - MIN - MOD(P))$ and $I\text{–}MIN\text{–}MOD(P)$ is the collection of all minimal models consistent with the three valued interpretation I.

For the program $P = \{a \leftarrow\sim a\}$ we have:

$$WFM(P) = \{\},$$
$$MIN\text{–}MOD(P) = \{a\} \text{ and } EWFM(P) = \{a\}$$

Note this view identifies the intended meaning of rule $a \leftarrow\sim a$ as the equivalent logic formula $a \leftarrow \neg a$, i.e. a. The O-Model of P is empty.

The difference between the O-Semantics and EWFS may be noticed in the intended meaning of the two rule program: $\{a \leftarrow\sim b;\ b \leftarrow\sim a\}$, which is behind the motivation of the extension EFWS of WFM based on GCWA. EWFS wants to identify $a \vee b$ as the meaning of this program, which also justifies the identification of $a \leftarrow\sim a$ with the fact a. The O-Model is empty.

A similar approach based on the notion of stable negative hypotheses (built upon the notion of consistency) is introduced in [Kakas and Mancarella, 1991b], identifying a stable theory associated with a program P as a *"skeptical"* semantics for P, that always contains the well founded model.

One example showing that their approach is still conservative is: $\{p \leftarrow\sim q;\ q \leftarrow\sim r;\ r \leftarrow\sim p;\ s \leftarrow p\}$. Stable theories identifies the empty set as the meaning of the program; however its O-Model is $\{\sim s\}$, since it is consistent, maximal, sustainable and tenable. Kakas (personnal communication) now also obtains this model, as a result of the investigation mentioned in the conclusions of [Kakas and Mancarella, 1991b].

8 Conclusions

We identify the meaning of a program P as a suitable partial closure of the well founded model of

the program in the sense that it contains the well founded model (and thus always exists). The extension we propose reduces undefinedness (which some authors argue is a desirable property) in the intended meaning of a program P, by an adequate form of CWA based on notions of consistency, sustainability and tenability with regard to alternative negative assumptions. Sustainability of a consistent set of negative assumptions insists that there be no other consistent set that defeats it (i.e. there is no hypothetical evidence whose consequences contradict the sustained assumptions). Tenability requires that a maximal sustainable set of assumptions be not contradicted by the consequences of adding to it another competing (nondefeating and nondefeated) maximal sustainable set.

Acknowledgements

We thank ESPRIT BRA COMPULOG (no. 3012), Instituto Nacional de Investigação Científica, Junta Nacional de Investigação Científica e Tecnológica and Gabinete de Filosofia do Conhecimento for their support. We are indebted to Anthony Kakas and Paolo Mancarella for their previous incursions and intuitions into a similar problem in the setting of their Stable Theories. Luís Monteiro is thanked for helpful discussions.

References

[Baral *et al.*, 1990] C. Baral, J. Lobo, and J. Minker. Generalized well-founded semantics. In M. Stickel, editor, *CAD'90*. Springer-Verlag, 1990.

[Dix, 1991] J. Dix. Classifying semantics of logic programs. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and NonMonotonic Reasoning'91*. MIT Press, 1991.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[Kakas and Mancarella, 1991a] A. C. Kakas and P. Mancarella. Negation as stable hypothesis. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and NonMonotonic Reasoning'91*. MIT Press, 1991.

[Kakas and Mancarella, 1991b] A. C. Kakas and P. Mancarella. Stable theories for logic programs. In Ueda and Saraswat, editors, *International Logic Programming Symposium'91*. MIT Press, 1991.

[Minker, 1987] J. Minker. *On indefinite databases and the closed world assumption*. Readings in Nonmonotonic Reasoning. Morgan Kaufmann, 1987.

[Monteiro, 1991] L. Monteiro. Notes on the semantics of logic programs. Technical report, DI/UNL, 1991.

[Pereira *et al.*, 1991a] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction Removal within Well Founded Semantics. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and NonMonotonic Reasoning'91*. MIT Press, 1991.

[Pereira *et al.*, 1991b] L. M. Pereira, J. J. Alferes, and J. N. Aparício. The extended stable models of contradiction removal semantics. In P. Barahona, L. M. Pereira, and A. Porto, editors, *5th Portuguese AI Conference'91*. Springer-Verlag, 1991.

[Pereira *et al.*, 1991c] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Counterfactual reasoning based on revising assumptions. In Ueda and Saraswat, editors, *International Logic Programming Symposium'91*. MIT Press, 1991.

[Pereira *et al.*, 1991d] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Hypothetical reasoning with well founded semantics. In B. Mayoh, editor, *Scandinavian Conference on AI'91*. IOS Press, 1991.

[Pereira *et al.*, 1991e] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Nonmonotonic reasoning with well founded semantics. In *International Conference on Logic Programming'91*. MIT Press, 1991.

[Przymusinska and Przymusinski, 1990] H. Przymusinska and T. Przymusinski. *Semantic Issues in Deductive Databases and Logic Programs*. Formal Techniques in Artificial Intelligence. North Holland, 1990.

[Przymusinski, 1990] T. Przymusinski. Extended stable semantics for normal and disjunctive programs. In *International Conference on Logic Programming'90*, pages 459–477. MIT Press, 1990.

[Van Gelder *et al.*, 1980] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, pages 81–132, 1980.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Contributions to the Semantics of Open Logic Programs

A. Bossi[1], M. Gabbrielli[2], G. Levi[2] and M.C. Meo[2]

1) Dipartimento di Matematica Pura ed Applicata
Università di Padova
Via Belzoni 7, I-35131 Padova, Italy
mat010@IPDUNIVX.UNIPD.IT

2) Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
{gabbri,levi,meo}@dipisa.di.unipi.it

Abstract

The paper considers *open logic programs* originally introduced in [Bossi and Menegus 1991] as a tool to build an *OR*-compositional semantics of logic programs. We extend the original semantic definitions in the framework of the general approach to the semantics of logic programs described in [Gabbrielli and Levi 1991b]. We first define an *OR-compositional operational semantics* $\mathcal{O}_\Omega(P)$ modeling computed answer substitutions. We consider next the semantic domain of Ω-*interpretations*, which are sets of clauses with a suitable equivalence relation. The fixpoint semantics $\mathcal{F}_\Omega(P)$ given in [Bossi and Menegus 1991] is proved equivalent to the operational semantics, by using an intermediate unfolding semantics. From the model-theoretic viewpoint, an Ω-interpretation is mapped onto a set of Herbrand interpretation, thus leading to a definition of Ω-*model* based on the classical notion of truth. We show that under a suitable partial order, the glb of a set of Ω-models of a program P is an Ω-model of P. Moreover, the glb of all the Ω-models of P is equal to the usual Herbrand model of P while $\mathcal{F}_\Omega(P)$ is a (non-minimal) Ω-model.

1 Introduction

An Ω-*open program* [Bossi and Menegus 1991] P is a program in which the predicate symbols belonging to the set Ω are considered partially defined in P. P can be composed with other programs which may further specify the predicates in Ω. Such a composition is denoted by $\cup_\Omega$. Formally, if $Pred(P) \cap Pred(Q) \subseteq \Omega$ then $P \cup_\Omega Q = P \cup Q$, otherwise $P \cup_\Omega Q$ is not defined ($Pred(P)$ denotes the predicate symbols in P). A typical partially defined program is a program where the intensional definitions are completely known while extensional definitions are only partially known and can be further specified.

Example 1.1 *Let us consider the following program*

$$Q_1 = \{\ anc(X,Y) : -parent(X,Y).\quad anc(X,Z) : -parent(X,Y), anc(Y,Z).\quad parent(isaac, jacob).\quad parent(jacob, benjamin).\ \}$$

New extensional information defining new parent tuples can be added to Q_1 *as follows*

$$Q_2 = \{\ parent(anna, elizabeth).\quad parent(elizabeth, john).\ \}$$

The semantics of open programs must be Ω-compositional w.r.t. program union, i.e. the semantics of $P_1 \cup_\Omega P_2$ must be derivable from the semantics of P_1 and P_2. If Ω contains all the predicates in P, Ω-compositionality is the same as compositionality.

The least Herbrand model semantics, as originally proposed [van Emden and Kowalski 1976] and the computed answer substitution semantics in [Falaschi et al. 1988,Falaschi et al. 1989a], are not compositional w.r.t. program union. For example, in example 1.1, the atom $anc(anna, elizabeth)$ which belongs to the least Herbrand model semantics of $Q_1 \cup Q_2$ cannot be obtained from the least Herbrand model semantics of Q_1 and Q_2 (see also example 2.1).

In this paper we will introduce a semantics for Ω-open programs following the general approach in [Gabbrielli and Levi 1991b] which leads to semantics definitions which characterize the program operational behavior. This approach leads to the introduction of extended interpretations (π-*interpretations*) which are more expressive than Herbrand interpretations. The improved expressive power is obtained by accommodating more syntactic objects in π-interpretations, which are (possibly

infinite) programs. The semantics in terms of π-interpretations can be computed both operationally and as the least fixpoint of suitable continuous immediate consequence operators on π-interpretations. It can also be characterized from the model-theoretic viewpoint, by defining a set of extended models (π-models) which encompass standard Herbrand models. In the specific case of Ω-open programs, extended interpretations are called Ω-interpretations and are sets of *conditional atoms* (i.e. clauses such that all the atoms in the body are open). Each Ω-interpretation represents a set of Herbrand interpretations that could be obtained by composing the open program with a definition for the open predicates. Ω-interpretations of open programs are introduced to obtain a unique representative model, computable as the least fixpoint of a suitable continuous operator, in cases where no such a representative exists in the set of Herbrand models.

The main contribution of this paper is the definition of an OR-compositional (i.e. compositional w.r.t. program union) semantics of logic programs in the style of [Falaschi et al. 1988, Falaschi et al. 1989b]. Other approachs to OR-compositionality can be found in [Lassez and Maher 1984, Mancarella and Pedreschi 1988, Gaifman and Shapiro 1989a, Gaifman and Shapiro 1989b]. An OR-compositional semantics corresponds to an important program equivalence notion, according to which two programs P_1 and P_2 are equivalent iff for any program Q a generic goal G computes the same answers in $P_1 \cup Q$ and $P_2 \cup Q$. An OR-compositional semantics has also some interesting applications. Namely it can be used

- to model logic languages provided with a module-like structure,
- to model incomplete knowledge bases, where new chunks of knowledge can incrementally be assimilated,
- for program transformation
  (the transformed programs must have the same OR-compositional semantics of the original program),
- for semantics-based "modular" program analysis.

The paper is organized as follows. Subsection 1.1 contains notation and useful definitions on the semantics of logic programs. In section 2 we define an *operational semantics* $\mathcal{O}_\Omega(P)$ modeling computed answer substitutions which is *OR-compositional*. Section 3 introduces a suitable *semantic domain* for the $\mathcal{O}_\Omega(P)$ semantics and defines *Ω-interpretations* which are sets of clauses modulo a suitable equivalence relation. In section 4 the *fixpoint semantics* $\mathcal{F}_\Omega(P)$, is proved equivalent to the operational semantics by using an intermediate *unfolding semantics*. Section 5 is concerned with *model theory*. From the model-theoretic viewpoint, an Ω-interpretation is mapped onto a set of Herbrand interpretations, thus leading to a definition of Ω-model based on the classical notion of truth. We show that under a suitable partial order, the glb of a set of Ω-models of a program P is an Ω-model of P. Moreover, the glb of all the Ω-models of P is equal to the usual Herbrand model of P. Moreover, $\mathcal{F}_\Omega(P)$ is a (non-minimal) Ω-model, equivalent to the model-theoretic semantics defined in [Bossi and Menegus 1991] in terms of S_Ω-models. A comparison between Ω-models and the S_Ω-models is made in section 6. Section 7 is devoted to some conclusive remarks. All the proofs of the results given here can be found in [Bossi et al. 1991].

1.1 Preliminaries

The reader is assumed to be familiar with the terminology of and the basic results in the semantics of logic programs [Lloyd 1987,Apt 1988]. Let the signature S consist of a set F of *function symbols*, a finite set P of *predicate symbols*, a denumerable set V of *variable symbols*. All the definitions in the following will assume a given signature S. Let T be the set of terms built on F and V. Variable-free terms are called *ground*. A substitution is a mapping $\vartheta : V \to T$ such that the set $D(\vartheta) = \{X \mid \vartheta(X) \neq X\}$ (*domain* of ϑ) is finite. If $W \subset V$, we denote by $\vartheta_{|W}$ the *restriction* of ϑ to the variables in W, i.e. $\vartheta_{|W}(Y) = Y$ for $Y \notin W$. ε denotes the empty substitution. The *composition* $\vartheta\sigma$ of the substitutions ϑ and σ is defined as the functional composition. A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} = \rho^{-1}\rho = \varepsilon$. The pre-ordering $\leq$ (more general than) on substitutions is such that $\vartheta \leq \sigma$ iff there exists ϑ' such that $\vartheta\vartheta' = \sigma$. The result of the application of the substitution ϑ to a term t is an *instance* of t denoted by $t\vartheta$. We define $t \leq t'$ (t is more general than t') iff there exists ϑ such that $t\vartheta = t'$. A substitution ϑ is *grounding* for t if $t\vartheta$ is ground. The relation $\leq$ is a preorder. $\approx$ denotes the associated equivalence relation (*variance*). A substitution ϑ is a *unifier* of terms t and t' if $t\vartheta \equiv t'\vartheta$. The *most general unifier* of t_1 and t_2 is denoted by $mgu(t_1, t_2)$. All the above definitions can be extended to other syntactic expressions in the obvious way. An atom is an object of the form $p(t_1, \ldots, t_n)$ where $p \in P$, $t_1, \ldots, t_n \in T$.

A *clause* is a formula of the form $H : -L_1, \ldots, L_n$ with $n \geq 0$, where H (the *head*) and $L_1, \ldots, L_n$ (the *body*) are atoms. ": −" and "," denote logic implication and conjunction respectively, and all variables are universally quantified. If the body is empty the clause is a *unit clause*. A *program* is a finite set of clauses. A *goal* is a formula $L_1, \ldots, L_m$, where each L_i is an atom. By $Var(E)$ and $Pred(E)$ we denote respectively the sets of variables and predicates occurring in the expression E. A *Herbrand interpretation* I for a program P is a set of ground atoms. The intersection $M(P)$ of all the Herbrand models of a program P is a model (least Herbrand model). $M(P)$ is also the least fixpoint of a continuous transformation T_P (*immediate consequences operator*) on the complete lattice of Herbrand interpretations. If G is a goal, $G \overset{\vartheta}{\leadsto}_P B_1, \ldots, B_n$ denotes an SLD derivation with fair selection rule of $B_1, \ldots, B_n$ in the program P where ϑ is the composition of the mgu's used in the derivation. $G \overset{\vartheta}{\longmapsto}_P \square$ denotes the refutation of G in the program P with computed answer substitution ϑ. A computed answer substitution is always restricted to the variables occurring in G. The notations $\tilde{t}$, $\tilde{X}$ will be used to denote tuples of terms and variables respectively, while $\tilde{B}$ denotes a (possibly empty) conjunction of atoms.

2 Computed answer substitution semantics for Ω-open programs

The operational semantics is usually given by means of a set of inference rules which specify how derivations are made. From a purely logical point of view the operational semantics is simply defined in terms of successful derivations. However, from a programming language viewpoint, the operational semantics must be concerned with additional information, namely observable properties. A given program in fact may have different semantics depending on which of its properties can be observed. For instance in pure logic programs one can observe successes, finite failure, computed answer substitutions, partial computed answer substitutions or any combination of them. A given choice of the observable induces an equivalence on programs, namely two programs are equivalent iff they are observationally indistinguishable. When the semantics correctly captures the observable, two programs are equivalent if they have the same semantics. When also compositionality is taken into account, for a given observable property we can obtain different semantics (and equivalence relations) depending on which kind of program composition we consider. Indeed, the semantics of logic programs is usually concerned with AND-composition (of atoms in a goal or in a clause body). Consider for example logic programs with computed answer substitutions as observable [Falaschi et al. 1989a]. The operational semantics can be defined as
$\mathcal{O}(P) = \{p(\tilde{X})\theta \mid \tilde{X}$ are distinct var, $p(\tilde{X}) \overset{\vartheta}{\longmapsto}_P \square\}$
where the denotation of a program is a set of non-ground atoms, which can be viewed as a possibly infinite program [Falaschi et al. 1989a]. Since we have syntactic objects in the semantic domain, we need an equivalence relation in order to abstract from irrelevant syntactic differences. If the equivalence is accurate enough the semantics is fully abstract. According to [Gabbrielli and Levi 1991b], Herbrand interpretations are generalized by π-interpretations which are possibly infinite sets of (equivalence classes of) clauses. The operational semantics of a program P is then a π-interpretation I, which has the following property. P and I are observationally equivalent with respect to any goal G. This is the property which allows to state that the semantics does indeed capture the observable behavior [Falaschi et al. 1989a]. The following example shows that when considering OR-composition (i.e. union of sets of clauses), non-ground atoms (or unit clauses) are not sufficient any longer to define a compositional semantics.

Example 2.1 *Let us consider the following programs*

$$P_1 = \{\ q(X) : -p(X).\quad r(X) : -s(X).\quad s(b).\quad p(a).\ \}\qquad P_2 = \{\ p(b).\ \}$$

According to the previous definition of $\mathcal{O}(P)$, $\mathcal{O}(P_1) = \{p(a), q(a), r(b), s(b)\}$ *and* $\mathcal{O}(P_2) = \{p(b)\}$. *Since* $\mathcal{O}(P_1 \cup P_2) = \{p(a), p(b), q(a), q(b), r(b), s(b)\}$, *the semantics of the union of the two programs cannot be obtained from the semantics of the programs.*

In order for a semantics to be compositional, it must contain information in the form of a mapping from sets of atoms to sets of atoms. This is indeed the case of the semantics based on the closure operator [Lassez and Maher 1984] and on the T_P operator [Mancarella and Pedreschi 1988]. If we want a semantics expressed by the program syntax, OR-compositionality can only be obtained by choosing as semantic domain a set of (equivalence classes of) clauses. In example 2.1, for instance, the semantics of P_1 should contain also the clause $q(X) : -p(X)$.

Let us formally give the definition of the program composition we consider.

Definition 2.2 *Let P be a program and Ω be a set of predicate symbols. P is open w.r.t. Ω (or Ω-open) if the information on the predicates in Ω is considered to be partial. Moreover if P, Q are Ω-open programs and $(Pred(Q) \cap Pred(P)) \subseteq \Omega$ then $P \cup_\Omega Q$ is the Ω-open program $P \cup Q$. If $(Pred(Q) \cap Pred(P)) \not\subseteq \Omega$ then $P \cup_\Omega Q$ is not defined.*

Note that when considering an Ω-open program P and an Ω'-open program Q, the composition of P and Q is defined only if $(Pred(Q) \cap Pred(P)) \subseteq (\Omega \cap \Omega')$. Moreover, the composition of P and Q is a Ψ-open program, where $\Psi = \Omega \cup \Omega'$.

The definition of any predicate symbol $p \in \Omega$ in an Ω-open program P can always be extended or refined. For instance in example 1.1 program Q_1 is open w.r.t. the predicate *parent* and this predicate is refined in program Q_2. Therefore, a deduction concerned with a predicate symbol of an Ω-open program P can be either *complete* (when it takes place completely in the program P) or *partial* (when it terminates in P with an atom $p(\tilde{t})$ such that $p \in \Omega$ and $p(\tilde{t})$ does not unify with the head of any clause in P). A partial deduction can be completed by the addition of new clauses. Thus we have an *hypothetic deduction*, conditional on the extension of predicate p.

Let us consider again the program P_1 of example 2.1 and assume $\Omega = \{p\}$. Then, the goal $r(X)$ produces a complete deduction only, computing the answer substitution $\{X/b\}$. The goal $q(X)$ produces a complete deduction, computing the answer substitution $\{X/a\}$ and an hypothetical deduction returning any answer that could be computed by a definition of p external to P_1. The goal $q(b)$ instead has one hypothetical deduction only, conditional on the provability (outside P_1) of $p(b)$. We want to express this hypothetical reasoning, i.e. that $q(b)$ is refutable if $p(b)$ is refutable. Hence we will consider the following operational semantics (recall that by $\tilde{B}$ we denote $B_1, \ldots, B_n$ with $n \geq 0$).

Definition 2.3 *Let Ω be a set of predicate symbols. We define*

$$Id(\Omega) = \{p(\tilde{X}) : -p(\tilde{X}) \mid \ p \in \Omega, \tilde{X} \text{ are distinct variables } \}$$

Definition 2.4 (Ω-compositional computed answer substitutions semantics) *Let P be a program and let $P^* = P \cup Id(\Omega)$. Then we define $\mathcal{O}_\Omega(P) =$*

$$\{A : -\tilde{B}_2 \mid \ p(\tilde{X}) \overset{\vartheta}{\leadsto}_P \tilde{B}_1 \overset{\gamma}{\leadsto}_{P^*} \tilde{B}_2, \ \tilde{X} \text{ distinct variables}, \ A = p(\tilde{X})\vartheta\gamma, \ \{Pred(\tilde{B}_2)\} \subseteq \Omega\}$$

The set of clauses $Id(\Omega)$ in the previous definition is used to delay the evaluation of open atoms. This is a trick which allows to obtain by using a fixed fair selection rule R, all the derivations $p(X_1, \ldots, X_n) \overset{\vartheta}{\leadsto}_P B_1, \ldots, B_n$ which use any selection rule R', for $Pred(B_1, \ldots, B_n) \subseteq \Omega$. Note that the first step of the derivation uses a clause in P (instead than in P^*) because we want $\mathcal{O}_\Omega(P)$ to contain a clause $p(\tilde{X}) : -p(\tilde{X})$ if and only if $p(\tilde{X}) \overset{\varepsilon}{\leadsto}_P p(\tilde{X})$.

Example 2.5 *Let P_1, P_2 be the Ω-open programs of example 2.1 where $\Omega = \{p\}$.*
Then $\mathcal{O}_\Omega(P_2) = \{p(b)\}$ and
$\mathcal{O}_\Omega(P_1) = \{q(X) : -p(X),\ p(a),\ q(a),\ r(b),\ s(b)\}$.
$\mathcal{O}_\Omega$ contains enough information to compute the semantics of compositions. Indeed $\mathcal{O}(P_1 \cup P_2) \subseteq \mathcal{O}_\Omega(P_1 \cup P_2)$ and $\mathcal{O}_\Omega(P_1 \cup P_2) = \mathcal{O}_\Omega(\mathcal{O}_\Omega(P_1) \cup \mathcal{O}_\Omega(P_2))$ (see theorem 2.9).

Example 2.6 *Let $\Omega = \{q, r\}$ and let Q_1, Q_2 be the following programs*

$$Q_1 = \{p(X,Y) : -r(X), q(Y). \quad r(a). \ \} \qquad Q_2 = \{ \ r(b). \ \}$$

Then $\mathcal{O}_\Omega(Q_2) = \{r(b)\}$, $\mathcal{O}_\Omega(Q_1) =$
$\{p(X,Y) : -r(X), q(Y),\ p(a,Y) : -q(Y),\ r(a)\}$ and
$\mathcal{O}_\Omega(Q_1 \cup Q_2) = \mathcal{O}_\Omega(\mathcal{O}_\Omega(Q_1) \cup \mathcal{O}_\Omega(Q_2)) =$
$\{p(X,Y) : -r(X), q(Y),\ p(a,Y) : -q(Y),$
$p(b,Y) : -q(Y),\ r(a), r(b)\}$ (see theorem 2.9).

Note that $\mathcal{O}_\Omega(P)$ is essentially the result of the partial evaluation [Lloyd and Shepherdson 1987] of P, where derivations terminate at open predicates. This operational semantics fully characterizes hypothetic deductions, conditional on the extension of the predicates in Ω. Indeed the semantics of a program P can be viewed as a possibly infinite set of clauses and the partial computed answer substitutions can be obtained by executing the goal in the "program". The equivalence ($\cong_\Omega$) on programs induced by the computed answer substitution observable when considering also programs union, can be formally defined as follows.

Definition 2.7 *Let P_1, P_2 be Ω-open programs. Then $P_1 \cong_\Omega P_2$ if for every goal G and for every program Q s.t. $P_i \cup_\Omega Q$, $i = 1, 2$, is defined, $G \overset{\vartheta}{\longmapsto}_{P_1 \cup_\Omega Q} \square$ iff $G \overset{\vartheta\rho}{\longmapsto}_{P_2 \cup_\Omega Q} \square$ where ρ is a renaming.*

$\mathcal{O}_\Omega$ allows to characterize a notion of answer substitution which enhances the usual one, since also (unresolved) atoms, with predicate symbols in Ω, are considered. Therefore it is able to model computed answer substitutions in an OR compositional way. The following results show that $\mathcal{O}_\Omega(P)$ is compositional w.r.t. $\cup_\Omega$ and therefore it correctly captures

the computed answer substitution observable notion when considering also programs union.

Theorem 2.8 *Let P be an Ω-open program. Then $P \cong_\Omega \mathcal{O}_\Omega(P)$.*

Theorem 2.9 *Let P_1, P_2 be Ω-open programs and let $P_1 \cup_\Omega P_2$ be defined. Then $\mathcal{O}_\Omega(\mathcal{O}_\Omega(P_1) \cup_\Omega \mathcal{O}_\Omega(P_2)) = \mathcal{O}_\Omega(P_1 \cup_\Omega P_2)$.*

Corollary 2.10 *Let P_1, P_2 be Ω-open programs. If $\mathcal{O}_\Omega(P_1) = \mathcal{O}_\Omega(P_2)$ then $P_1 \cong_\Omega P_2$.*

3 Semantic domain for Ω-open programs

In this section we formally define the semantic domain which characterizes the above introduced operational semantics $\mathcal{O}_\Omega$. Since $\mathcal{O}_\Omega$ contains clauses (whose body predicates are all in Ω), we have to accommodate clauses in the interpretations we use. Therefore we will define the notion of Ω-interpretation which extends the usual notion of interpretation since an Ω-interpretation contains conditional atoms. As usual, in the following, Ω is a set of predicates.

Definition 3.1 (Conditional atoms) *An Ω-conditional atom is a clause $A : -B_1, \ldots, B_n$ such that $Pred(B_1, \ldots, B_n) \subseteq \Omega$.*

In order to abstract from the purely syntactical details, we use the following equivalence $\approx$ on conditional atoms.

Definition 3.2 *Let $c_1 = A_1 : -B_1, \ldots, B_n$, $c_2 = A_2 : -D_1, \ldots, D_m$ be clauses. Then $c_1 \leq c_2$ iff $\exists \vartheta$ such that $\exists \{i_1, \ldots, i_n\} \subseteq \{1, \ldots, m\}$ such that $A_1\vartheta = A_2$, $i_h \neq i_k$ for $h \neq k$, and $(B_1\vartheta, \ldots, B_n\vartheta) = (D_{i_1}, \ldots, D_{i_n})$. Moreover we define $c_1 \approx c_2$ iff $c_1 \leq c_2$ and $c_2 \leq c_1$.*

Note that in the previous definition bodies of clauses are considered as multisets (considering sets would give the standard definition of subsumption). Equivalent clauses have the same body (considered as a multiset) up to renaming. Considering sets instead of multisets (subsumption equivalence) is not correct when considering computed answer substitutions. The following is a simple counterexample.

Example 3.3 *Let $c_1 = p(X,Y) : -q(X,Y), q(X,Y)$ and $c_2 = p(X,Y) : -q(X,Y)$. Let $P_1 = \{c_1\}$ and $P_2 = \{c_2\}$ be Ω-open programs where $\Omega = \{q\}$. Obviously, considering bodies of clauses as sets, $c_1 = c_2\epsilon$ where ϵ is the empty renaming. However, $P_1 \not\cong_\Omega P_2$ since by considering $Q = \{q(X,b), q(a,Y)\}$, $p(X,Y) \overset{\vartheta}{\mapsto}_{P_1 \cup Q} \square$ where $\vartheta = \{X/a, Y/b\}$, while the goal $p(X,Y)$ in the program $P_2 \cup Q$ can compute either $\{X/a\}$ or $\{Y/b\}$ only.*

Definition 3.4 *The Ω-conditional base, $\mathcal{C}_\Omega$, is the quotient set of all the Ω-conditional atoms w.r.t. $\approx$.*

In the following we will denote the equivalence class of a conditional atom c by c itself, since all the definitions which use conditional atoms are not dependent on the element chosen to represent an equivalence class. Moreover, any subset of $\mathcal{C}_\Omega$ will be considered implicitly as an Ω-open program. Before giving the formal definition of Ω-interpretation, we need the notion of *u-closed subset* of $\mathcal{C}_\Omega$.

Definition 3.5 *A subset I of $\mathcal{C}_\Omega$ is u-closed iff $\forall H : -B_1, \ldots, B_n \in I$ and $\forall B : -A_1, \ldots, A_m \in I$ such that $\exists \vartheta = mgu(B_i, B)$, for $1 \leq i \leq n$, $(H : -B_1, \ldots, B_{i-1}, A_1, \ldots, A_m, B_{i+1}, \ldots, B_n)\vartheta \in I$. Moreover if $I \subseteq \mathcal{C}_\Omega$, we denote by $\hat{I}$ its u-closure defined as the least (w.r.t. $\subseteq$) $I' \subseteq \mathcal{C}_\Omega$ u-closed such that $I \subseteq I'$.*

Proposition 4.5 will show that the previous notion of u-closure is well defined. A u-closed interpretation I is an interpretation which, if viewed as a program, is closed under unfolding of procedure calls. Interpretations need to be u-closed for the validity of the model theory developed in section 5. Therefore, in order to define Ω-interpretations we will consider u-closed sets of conditional atoms only. Let us now give the formal definition of Ω-interpretation.

Definition 3.6 *An Ω-interpretation I is any subset of $\mathcal{C}_\Omega$ which is u-closed. The set of all the Ω-interpretations is denoted by $\Im$.*

Lemma 3.7 *$(\Im, \subseteq)$ is a complete lattice where the minimal element is $\emptyset$ and $glb(X) = \widehat{\bigcup_{x \in X} x}$ for any $X \subseteq \Im$.*

In the following the operational semantics $\mathcal{O}_\Omega$ will be formally considered as an Ω-interpretation.

4 Fixpoint semantics

In this section we define a fixpoint semantics $\mathcal{F}_\Omega(P)$ which in the next subsection is proved to be equivalent to the previously defined operational semantics $\mathcal{O}_\Omega(P)$. This can be achieved by defining an immediate consequence operator T_P^Ω on the lattice $(\Im, \subseteq)$ of Ω-interpretations. $\mathcal{F}_\Omega(P)$ is the least fixpoint of T_P^Ω.

The immediate consequences operator T_P^Ω is strongly related to the derivation rule used for Ω-open programs and hence to the unfolding rule. Therefore T_P^Ω models the observable properties in an OR compositional way, and may be useful for modular (i.e. OR compositional) bottom-up program analysis.

Definition 4.1 *Let P be an Ω-open program. Then $T_P^\Omega(I) = \widehat{\Gamma_P^\Omega(I)}$ where $\Gamma_P^\Omega(I)$ is the operator defined in [Bossi and Menegus 1991] as follows.*

$$\Gamma_P^\Omega(I) = \{(A : -\tilde{L}_1, \ldots, \tilde{L}_n)\vartheta \in \mathcal{C}_\Omega \mid \exists A : -B_1, \ldots, B_n \in P, \exists B_i' : -\tilde{L}_i \in I \cup Id(\Omega),\ i = 1, \ldots, n,\ m_i \geq 0 \ s.t.\ \vartheta = mgu((B_1, \ldots, B_n), (B_1', \ldots, B_n'))\}$$

Proposition 4.2 *T_P^Ω is continuous in the complete lattice $(\Im, \subseteq)$.*

The notion of ordinal powers for T_P^Ω is defined as usual, namely $T_P^\Omega \uparrow 0 = \emptyset$, $T_P^\Omega \uparrow n+1 = T_P^\Omega(\ T_P^\Omega \uparrow n\ )$ and $T_P^\Omega \uparrow \omega = \bigcup_{n \geq 0}(\ T_P^\Omega \uparrow n\ )$. Since T_P^Ω is continuous on $(\Im, \subseteq)$, well known results of lattice theory allow to prove proposition 4.3 and hence to define the fixpoint semantics as follows.

Proposition 4.3 *$T_P^\Omega \uparrow \omega$ is the least fixpoint of T_P^Ω in the complete lattice $(\Im, \subseteq)$.*

Definition 4.4 *Let P be an Ω-open program. The fixpoint semantics $\mathcal{F}_\Omega(P)$ of P is defined as $\mathcal{F}_\Omega(P) = T_P^\Omega \uparrow \omega$.*

Remark
The original definition of $\Gamma_P^\Omega(I)$ does not require Ω-interpretations to be u-closed subsets of $\mathcal{C}_\Omega$. If we consider an Ω-interpretation as any subset of $\mathcal{C}_\Omega$ and the Γ_P^Ω operator, even if the intermediate results $\Gamma_P^\Omega \uparrow n$ are different, the following proposition 4.5 and theorem 4.6 show that the least fixpoint $\Gamma_P^\Omega \uparrow \omega$ is a u-closed set and it is equal to $\mathcal{F}_\Omega(P)$ (Γ_P^Ω is continuous on $(\wp(\mathcal{C}_\Omega), \subseteq)$). Therefore, when considering the fixpoint semantics we can use the Γ_P^Ω operator. Moreover, proposition 4.5 ensures us that the previous notion of u-closure is well defined.

Proposition 4.5 *Let $I \subseteq \mathcal{C}_\Omega$ and let $\Gamma_I^\Omega(I)$ be defined as in definition 4.7. Then the following hold*

1. *I is u-closed iff $I = \Gamma_I^\Omega(I)$,*
2. *for any program P, $\Gamma_P^\Omega \uparrow \omega$ is u-closed,*
3. *$I' = \Gamma_I^\Omega \uparrow \omega$ is the least (w.r.t. set inclusion) subset of $\mathcal{C}_\Omega$ such that it is u-closed and $I \subseteq I'$.*

Theorem 4.6 *Let P an Ω-open program. $\Gamma_P^\Omega \uparrow \omega = \mathcal{F}_\Omega(P)$.*

4.1 Unfolding semantics and equivalence results

To clarify the relations between the operational and the fixpoint semantics, before proving their equivalence, we introduce the intermediate notion of *unfolding semantics* $\mathcal{U}_\Omega(P)$ [Levi 1988, Levi and Mancarella 1988]. $\mathcal{U}_\Omega(P)$ is obtained as the limit of the unfolding process. Since the unfolding semantics can be expressed top-down in terms of the Γ_P^Ω operator, the unfolding semantics can be proved equal to the standard bottom-up fixpoint semantics. On the other hand, since $\mathcal{U}_\Omega(P)$ and $\mathcal{O}_\Omega(P)$ are based on the same inference rule (applied in parallel and in sequence respectively) $\mathcal{U}_\Omega(P)$ and $\mathcal{O}_\Omega(P)$ can easily be proven equivalent.

Definition 4.7 *Let P and Q be Ω-open programs. Then the unfolding of P w.r.t. Q is defined as*

$$unf_P^\Omega(Q) = \{(A : -\tilde{L}_1, \ldots, \tilde{L}_n)\vartheta \mid \exists A : -B_1, \ldots, B_n \in P, \exists B_i' : -\tilde{L}_i \in I \cup Id(\Omega),\ i = 1, \ldots, n,\ m_i \geq 0 \ s.t.\ \vartheta = mgu((B_1, \ldots, B_n), (B_1', \ldots, B_n'))\}$$

Note that the only difference between $unf_P^\Omega(Q)$ and $\Gamma_P^\Omega(Q)$ is that the second restricts to clauses in $\mathcal{C}_\Omega$ the set resulting from the definition. Therefore if I is an Ω-interpretation (i.e. $I \subseteq \mathcal{C}_\Omega$), $\Gamma_P^\Omega(I) = unf_P^\Omega(I)$ holds. In general, $\Gamma_P^\Omega(I) = \iota_\Omega(unf_P^\Omega(I))$ where $\iota_\Omega(P)$ extracts from a program P an Ω-interpretation.

Definition 4.8 *Let P be an Ω-open program. Then we define*

$$\iota_\Omega(P) = \{c \in P \mid c \in \mathcal{C}_\Omega\}.$$

Definition 4.9 *Let P be an Ω-open program and let $\iota_\Omega(P)$ be as defined in definition 4.8. Then we define the collection of programs*

$$P_0 = P$$
$$P_i = unf_{P_{i-1}}(P)$$

The unfolding semantics $\mathcal{U}_\Omega(P)$ of the program P is defined as

$$\mathcal{U}_\Omega(P) = \bigcup_{i=1,2,\ldots} \iota_\Omega(P_i).$$

The following theorem states the equality of the unfolding and the operational semantics.

Theorem 4.10 *Let P be an Ω-open program. Then $\mathcal{O}_\Omega(P) = \mathcal{U}_\Omega(P)$.*

Note that $\Gamma_P^\Omega \uparrow n+1 = unf_{P_n'}^\Omega(\emptyset)$, where $P_0' = P$ and $P_{i+1}' = unf_P^\Omega(P_i')$. Therefore we have the following theorem.

Theorem 4.11 *Let P be a program. Then $\mathcal{F}_\Omega(P) = \mathcal{U}_\Omega(P)$.*

Corollary 4.12 *Let P be a program. Then $\mathcal{F}_\Omega(P) = \mathcal{O}_\Omega(P)$.*

5 Model Theory

As we have shown, the operational and fixpoint semantics of a program P define an Ω-interpretation I_P, which can be viewed as a syntactic notation for a set of Herbrand interpretations denoted by $\mathcal{H}(I_P)$. Namely, $\mathcal{H}(I_P)$ represents the set of the least Herbrand models of all programs which can be obtained by closing the program I_P with a suitable set of ground atoms defining the open predicates. Our aim is finding a notion of Ω-model such that $\mathcal{O}_\Omega(P)$ (and $\mathcal{F}_\Omega(P)$) are Ω-models and every Herbrand model is an Ω-model. This can be obtained as follows.

Definition 5.1 *Let J be an Ω-interpretation. Then we define*

$$Atom_\Omega(J) = \{p(\tilde{t}) \mid \ p \in \Omega \text{ and } p(\tilde{t}) \text{ is a ground instance of an atom in } J\}.$$

Example 5.2 *Let $\Omega = \{p, q\}$ and $J = \{p(a) : -q(b)\}$. Then $Atom_\Omega(J) = \{p(a), q(b)\}$.*

Definition 5.3 *Let I be an Ω-interpretation for an Ω-open program. Then we define*

$$\mathcal{H}(I) = \{M(I \cup J) \mid J \subseteq Atom_\Omega(I)\}$$

where $M(K)$ denotes the least Herbrand model of K.

Example 5.4 *Let $I = \{p(a) : -q(b)\}$ be an Ω-interpretation. Then*

1) *for $\Omega = \{q\}$*
   $Atom_\Omega(I) = \{q(b)\}$ and
   $\mathcal{H}(I) = \{\emptyset. \{p(a), q(b)\}\}$,

2) *for $\Omega = \{p, q\}$*
   $Atom_\Omega(I) = \{p(a), q(b)\}$ and
   $\mathcal{H}(I) = \{\emptyset, \{p(a)\}, \{p(a), q(b)\}\}$.

Definition 5.5 *Let P be an Ω-open program and I be an Ω-interpretation. I is an Ω-model of P iff $\forall\ J \in \mathcal{H}(I)$, J is a Herbrand model of P.*

Obviously, in general given a Herbrand model M of a program P, $M \cup N$ is not anymore a model of P for an arbitrary set of ground atoms N. Since we want a notion of Ω-model which encompasses the standard notion of Herbrand model, the "closure" of the interpretation I can be performed by adding only ground atoms which unify with atoms already in I. The following example 5.6 shows that if such a condition is not satisfied, a standard Herbrand model would not any more be an Ω-model.

Example 5.6 *Let us consider the Ω-open program $P = \{p(a) : -q(a)\}$ where $\Omega = \{q\}$. Then $\emptyset$ is a (the least) Herbrand model of P. If, by violating the $J \subseteq Atom_\Omega(I)$ condition, $\{q(a)\} \in \mathcal{H}(\emptyset)$, since $\{q(a)\}$ is not a Herbrand model of P, $\emptyset$ would not be an Ω-model of P.*

Example 5.7 *Let us consider the program P_1 where $\Omega = \{p\}$ of the example 2.1. Then*
$\mathcal{O}_\Omega(P_1) = \{q(X) : -p(X),\ p(a),\ q(a),\ r(b),\ s(b)\}$
is an Ω-model of P_1 since
$\mathcal{H}(\mathcal{O}_\Omega(P_1)) = \{H_1, H_2, H_3, \ldots\}$
where, denoting by $[p(X)]$ the set of ground instances of $p(X_0)$,
$H_1 = \{p(a), q(a), r(b), s(b)\}$
$H_2 = \{p(a), p(b), q(a), q(b), r(b), s(b)\}$
$\vdots$
$H_\omega = \{r(b), s(b)\} \cup [p(X)] \cup [q(X)]\}$
and $H_1, H_2, \ldots, H_\omega$ are Herbrand models of P_1.

The following proposition states the mentioned properties of Ω-models.

Proposition 5.8 *Let $P = \{c_1, \ldots, c_n\}$ be an Ω-open program. Then*

1. *every Herbrand model of P is an Ω-model of P,*
2. *$\mathcal{O}_\Omega(P)$ is an Ω-model of P.*

A relevant property of standard Herbrand models states that the intersection of a set of models of a program P is always a model of P. This allows to define the model-theoretic semantics of P as the least Herbrand model obtained by intersecting all the Herbrand models of P. The following example shows that this important property does not hold any more when considering Ω-models with set theoretic operations.

Example 5.9 *Let $\Omega = \{q\}$ and P be the following Ω-open program $P = \{p(b) : -q(b),\ p(X),\ q(a)\}$.*
Then $\mathcal{O}_\Omega(P) = \{p(b) : -q(b),\ p(x),\ q(a)\}$ and
$M(P) = \{q(a)\} \cup \{p(\tilde{t}) \mid \tilde{t}$ is a ground term $\}$.
By proposition 5.8 $\mathcal{O}_\Omega(P)$ and $M(P)$ are Ω-models of P. However $\mathcal{O}_\Omega(P) \cap M(P) = \{q(a)\}$ is not an Ω-model of P.

The Ω-model intersection property does not hold because set theoretic operations do not adequately model the operations on conditional atoms. Namely, the information of an Ω-interpretation I_1 may be contained in I_2 without I_1 being a subset of I_2. In order to define the model-theoretic semantics for Ω-open programs as a unique (least) Ω-model, we then need a partial order $\sqsubseteq$ on Ω-interpretations which

allows to restore the model intersection property. $\sqsubseteq$ should model the meaning of Ω-interpretations, in such a way that $(\Im, \sqsubseteq)$ is a complete lattice and the greatest lower bound of a set of Ω-models is an Ω-model. As we will show in the following, this can be obtained by considering $\sqsubseteq$ as given in definition 5.10. According to the above mentioned property, there exists a least Ω-model. It is worth noting that such a least Ω-model is the standard least Herbrand model (proposition 5.21). Moreover note that, the most expressive Ω-model $\mathcal{O}_\Omega(P)$ is a non-minimal Ω-model. The following definitions extend those given in [Falaschi et al. 1989b] for the non compositional semantics of positive logic programs.

Definition 5.10 *Let I_1, I_2 be Ω-interpretations. We define*

- $I_1 \leq I_2$ *iff* $\forall c_1 \in I_1\ \exists c_2 \in I_2$ *such that* $c_2 \leq c_1$.
- $I_1 \sqsubseteq I_2$ *iff* $(I_1 \leq I_2)$ *and* $(I_2 \leq I_1$ *implies* $I_1 \subseteq I_2)$.

Proposition 5.11 *The relation $\leq$ is a preorder and the relation $\sqsubseteq$ is an ordering.*

Note that if $I_1 \subseteq I_2$, then $I_1 \sqsubseteq I_2$, since $I_1 \subseteq I_2$ implies $I_1 \leq I_2$. The following definitions and propositions will be used to define the model-theoretic semantics.

Definition 5.12 *Let I be an Ω-interpretation. We define $Min'(I) = \{c \in I \mid \forall c' \in I,\ c' \leq c \Rightarrow c' = c\}$ and $Min(I) = \widehat{Min'}(I)$.*

Example 5.13 *We show Min and Min' for the following Ω-interpretations I and J. Let*

$$
\begin{array}{lll}
I = & \{p(x),\ q(b),\ p(a),\ p(a) : -q(b) & \} \\
J = \{ & q(x) : -p(x), r(x) & \\
 & q(b) : -p(b) & \\
 & q(b) : -p(x) & \\
 & r(b) & \}
\end{array}
$$

Then
$Min'(I) = Min(I) = \{p(x),\ q(b)\}$.
$Min'(J) = \{r(b),\ q(x) : -p(x), r(x).\ q(b) : -p(x)\}$,
$Min(J) = J$.

Definition 5.14
Let Λ be a set of Ω-interpretations. We introduce the following notations.

- $\bigtriangledown\Lambda = \bigcup_{I \in \Lambda} I$
- $Min(\Lambda) = Min(\bigtriangledown\Lambda)$
- $\bigsqcup\Lambda = \hat{A}$ *where* $A = Min(\Lambda) \cup \bigtriangledown\{I \in \Lambda \mid Min(\Lambda) \subseteq I\}$)

It is worth noting that $\forall\, I$ Min(I) $\subseteq$ I (recall that I is u-closed) and $Min(\Lambda) = Min(\bigsqcup\Lambda)$.

Proposition 5.15
For any set Λ of Ω-interpretations there exists the least upper bound of Λ , $lub(\Lambda)$. and $lub(\Lambda) = \bigsqcup\Lambda$ holds.

Proposition 5.16 *The set of all the Ω-interpretations $\Im$ with the ordering $\sqsubseteq$ is a complete lattice. $\mathcal{C}_\Omega$ is the top element and $\emptyset$ is the bottom element.*

The model-theoretic construction is possible only if Ω-interpretations can be viewed as representations of Herbrand interpretations. Notice that every Herbrand interpretation is an Ω-interpretation. The following proposition generalizes the standard intersection property of Herbrand models to the case of Ω-models.

Proposition 5.17 *Let* **M** *be a non-empty set of Ω-models of an Ω-open program P. Then $glb(\mathbf{M})$ is an Ω-model of P.*

Corollary 5.18 *The set of all the Ω-models of a program P with the ordering $\sqsubseteq$ is a complete lattice.*

We are now in the position to formally define the model-theoretic semantics.

Definition 5.19 *Let P be a program. Its model-theoretic semantics is the greatest lower bound of the set of its models, i.e.,*
$M_\Omega(P) = glb(\{I \in \Im \mid I$ *is a Ω-model of* $P\}$).

Proposition 5.21 shows that the above defined model-theoretic semantics is the standard least Herbrand model. This fact justifies our choice of the ordering relation.

Proposition 5.20 *For any Ω-model I there exists a standard Herbrand model I' such that $I' \sqsubseteq I$.*

Proposition 5.21 *The least standard Herbrand model is the least Ω-model.*

6 S_Ω-models

We will now consider the relation between Ω-models (definition 5.5) and the S_Ω-models defined in [Bossi and Menegus 1991] on the same set of interpretations. Both the Ω-models and the S_Ω-models are intended to capture specific operational properties, from a model-theoretic point of view. However, S_Ω-models are based on an ad hoc notion of truth (S_Ω-truth) and the least S_Ω-model is exactly $\mathcal{F}_\Omega(P)$.

Conversely, Ω-models are based on the usual notion of truth in a Herbrand interpretation through the function $\mathcal{H}$. Moreover the least Ω-model is the usual least Herbrand model, while $\mathcal{F}_\Omega(P)$ is a non-minimal Ω-model.

Definition 6.1 *[Bossi and Menegus 1991]* (S_Ω-Truth) *Let Ω be a set of predicate symbols and I be an Ω-interpretation. Then*

(a) *An atom A is Ω-true in I iff $A \in I$.*

(b) *A definite clause $A\text{:-}B_1, \ldots, B_m$ is Ω-true in I iff*
$\forall B'_1, \ldots, B'_n$ *such that*
$B'_1 : -\tilde{L}_1, \ldots, B'_n : -\tilde{L}_n \in I \cup Id(\Omega)$
if $\exists \vartheta = mgu((B_1, \ldots, B_n), (B'_1, \ldots, B'_n))$
then $(A : -\tilde{L}_1, \ldots, \tilde{L}_n)\vartheta \in I$.

S_Ω-models are defined in the obvious way.

Proposition 6.2 *Every S_Ω-model is an Ω-model (according to definition 5.5).*

Proposition 6.3 *[Bossi and Menegus 1991] If Λ is a non-empty set of S_Ω-models of an Ω-open program P, then $\bigcap_{M \in \Lambda} M$ is an S_Ω-model of P.*

The previous proposition allows to define the model theoretic semantics $\mathcal{M}_{S_\Omega}(P)$ for a program P in terms of the S_Ω-models as follows.

Definition 6.4 *[Bossi and Menegus 1991] Let P be an Ω-open program and let S be the set of all the S_Ω-models of P. Then $\mathcal{M}_{S_\Omega}(P) = \bigcap_{M \in S} M$.*

Corollary 6.5 *Let Λ be a non-empty set of S_Ω-models of an Ω-open program P. Then $\bigcap_{M \in \Lambda} M$ is an Ω-model of P.*

By definition and by proposition 6.3, $\mathcal{M}_{S_\Omega}(P)$ is the least S_Ω-model in the lattice $(\Im, \subseteq)$ (recall that $\Im$ is the set of all the Ω-interpretations). The following proposition shows that $\mathcal{M}_{S_\Omega}(P)$ is also the least S_Ω-model in the lattice $(\Im, \sqsubseteq)$.

Proposition 6.6 *Let P be a program and let S be the set of all the S_Ω-models of P. Then $\mathcal{M}_{S_\Omega}(P) = glb(S)$ (according to $\sqsubseteq$ ordering).*

The following theorem shows the equivalence of the fixpoint semantics (definition 4.4) and the model-theoretic semantics $\mathcal{M}_{S_\Omega}(P)$.

Theorem 6.7 *[Bossi and Menegus 1991] Let P be an Ω-open program. Then $\mathcal{F}_\Omega(P) = \mathcal{M}_{S_\Omega}(P)$.*

Corollary 6.8 *Let P be an Ω-open program. Then $\mathcal{F}_\Omega(P)$ is an Ω-model of P.*

It is worth noting that, since $\mathcal{O}_\Omega(P) = \mathcal{F}_\Omega(P) = \mathcal{M}_{S_\Omega}(P)$, theorem 2.9 shows that the model-theoretic semantics $\mathcal{M}_{S_\Omega}(P)$ is compositional w.r.t. Ω-union of programs when considering computed answer substitutions as observables. This result was already proved in [Bossi and Menegus 1991] for the $\mathcal{M}_{S_\Omega}(P)$ model. Finally note that, as shown by the following example, T_P^Ω is not monotonic (and therefore it is not continuous) on the complete lattice $(\Im, \sqsubseteq)$. However, proposition 6.10 ensures us that $\mathcal{F}_\Omega(P)$ is still the least fixpoint of T_P^Ω on $(\Im, \sqsubseteq)$.

Example 6.9 *Consider the program*
$P = \{r(b) \;\; p(x) : -q(x)\}$.
Let $\Omega = \emptyset$, $I_1 = \{q(a), q(x)\}$ and $I_2 = \{r(b), q(x)\}$. Then $I_1 \sqsubseteq I_2$ while $T_P^\Omega(I_1) = \{p(x), p(a), r(b)\} \not\sqsubseteq T_P^\Omega(I_2) = \{p(x), r(b)\}$.

Proposition 6.10 *$T_P^\Omega \uparrow \omega$ is the least fixpoint of T_P^Ω on the complete lattice $(\Im, \sqsubseteq)$.*

7 Related work and conclusions

The result of our semantic construction has several similarities with the proof-theoretic semantics defined in [Gaifman and Shapiro 1989a, Gaifman and Shapiro 1989b]. Our construction however is closer to the usual characterization of the semantics of logic programs. Namely we define a top-down operational and bottom-up fixpoint semantics, and, last but not least a model-theoretic semantics which allows us to obtain a declarative characterization of syntactically defined models. The semantics in [Gaifman and Shapiro 1989a] does not characterize computed answer substitutions, while the denotation defined by the fully abstract semantics in [Gaifman and Shapiro 1989b] is not a set of clauses (i.e. a program). The framework of [Gaifman and Shapiro 1989a, Gaifman and Shapiro 1989b] can be useful for defining a program equivalence notion, even if our more declarative (model-theoretic) characterization is even more adequate. Moreover, the presence of an operational or a fixpoint semantics makes our construction useful as a formal basis for program analysis. Another related paper is [Brogi et al. 1991], where Ω-open logic programs are called open theories. Open theories are provided with a model-theoretic semantics which is based on ideas very similar to those underlying our definition 5.3. [Brogi et al. 1991] however does not consider semantic definitions in the style of our $\mathcal{O}_\Omega(P)$ which gives a unique denotation to any open program.

Let us finally remark some interesting properties of the Ω-model $\mathcal{O}_\Omega(P)$.

- By means of a syntactic device, we obtain a unique representation for a possibly infinite set of Herbrand models when a unique representative Herbrand model does not exist. A similar device was used in [Dung and Kanchanasut 1989, Kanchanasut and Stuckey 1990, Gabbrielli et al. 1991] to characterize logic programs with negation.
- Operators, such as $\cup_\Omega$ are quite easy and natural to define on $\mathcal{O}_\Omega(P)$.
- $\mathcal{O}_\Omega(P)$ can be used for modular program analysis [Giacobazzi and Levi 1991] and for studying new equivalences of logic programs, based on computed answer substitutions, which are not considered in [Maher 1988].
- It is strongly related to *abduction* [Eshghi and Kowalski 1989]. If Ω is the set of abducible predicates, the abductive consequences of any goal G can be found by executing G in $\mathcal{O}_\Omega(P)$.
- The delayed evaluation of open predicates which is typical of $\mathcal{O}_\Omega(P)$ can easily be generalized to other logic languages, to achieve compositionality w.r.t the union of programs. In particular this matches quite naturally the semantics of CLP and concurrent constraint programs given in [Gabbrielli and Levi 1990, Gabbrielli and Levi 1991a].

References

[Apt 1988] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[Bossi et al. 1991] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. Contributions to the Semantics of Open Logic Programs. Technical Report TR 17/91, Dipartimento di Informatica, Università di Pisa, 1991.

[Bossi and Menegus 1991] A. Bossi and M. Menegus. Una Semantica Composizionale per Programmi Logici Aperti. In P. Asirelli, editor, *Proc. Sixth Italian Conference on Logic Programming*, pages 95–109, 1991.

[Brogi et al. 1991] A. Brogi, E. Lamma, and P. Mello. Open Logic Theories. In P. Krueger L.-H. Eriksson and P. Shroeder-Heister, editors, *Proc. of the Second Workshop on Extensions to Logic Programming*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 1991.

[Dung and Kanchanasut 1989] Phan Minh Dung and K. Kanchanasut. A Fixpoint Approach to Declarative Semantics of Logic Programs. In E. Lusk and R. Overbeck, editors, *Proc. North American Conf. on Logic Programming'89*, pages 604–625. The MIT Press, Cambridge, Mass., 1989.

[Eshghi and Kowalski 1989] K. Eshghi and R. A. Kowalski. Abduction compared with Negation by Failure. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming*, pages 234–254. The MIT Press, Cambridge, Mass., 1989.

[Falaschi et al. 1988] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A new Declarative Semantics for Logic Languages. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 993–1005. The MIT Press, Cambridge, Mass., 1988.

[Falaschi et al. 1989a] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[Falaschi et al. 1989b] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. Technical Report TR 32/89, Dipartimento di Informatica, Università di Pisa, 1989. To appear in *Information and Computation*.

[Gabbrielli and Levi 1990] M. Gabbrielli and G. Levi. Unfolding and Fixpoint Semantics of Concurrent Constraint Programs. In H. Kirchner and W. Wechler, editors, *Proc. Second Int'l Conf. on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 204–216. Springer-Verlag, Berlin, 1990. Extended version to appear in *Theoretical Computer Science*.

[Gabbrielli and Levi 1991a] M. Gabbrielli and G. Levi. Modeling Answer Constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 238– 252. The MIT Press, Cambridge, Mass., 1991.

[Gabbrielli and Levi 1991b] M. Gabbrielli and G. Levi. On the Semantics of Logic Programs. In J. Leach Albert, B. Monien, and M. Rodriguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium*, volume 510 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, Berlin, 1991.

[Gabbrielli et al. 1991] M. Gabbrielli, G. Levi, and D. Turi. A Two Steps Semantics for Logic Programs with Negation. Technical report, Dipartimento di Informatica, Università di Pisa, 1991.

[Gaifman and Shapiro 1989a] H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 134–142. ACM, 1989.

[Gaifman and Shapiro 1989b] H. Gaifman and E. Shapiro. Proof theory and semantics of logic programs. In *Proc. Fourth IEEE Symp. on Logic In Computer Science*, pages 50–62. IEEE Computer Society Press, 1989.

[Giacobazzi and Levi 1991] R. Giacobazzi and G. Levi. Compositional Abstract Interpretation of Constraint Logic Programs. Technical report, Dipartimento di Informatica, Università di Pisa, 1991.

[Kanchanasut and Stuckey 1990] K. Kanchanasut and P. Stuckey. Eliminating Negation from Normal Logic Programs. In H. Kirchner and W. Wechler, editors, *Proc. Second Int'l Conf. on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 217–231. Springer-Verlag, Berlin, 1990.

[Lassez and Maher 1984] J.-L. Lassez and M. J. Maher. Closures and Fairness in the Semantics of Programming Logic. *Theoretical Computer Science*, 29:167–184, 1984.

[Levi 1988] G. Levi. Models, Unfolding Rules and Fixpoint Semantics. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1649–1665. The MIT Press, Cambridge, Mass., 1988.

[Levi and Mancarella 1988] G. Levi and P. Mancarella. The Unfolding Semantics of Logic Programs. Technical Report TR-13/88. Dipartimento di Informatica, Università di Pisa, 1988.

[Lloyd 1987] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[Lloyd and Shepherdson 1987] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. Technical Report CS-87-09, Computer Science Department, University of Bristol, 1987. Revised version 1989, to appear in *Journal of Logic Programming*.

[Maher 1988] M. J. Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, Ca., 1988.

[Mancarella and Pedreschi 1988] P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1006–1023. The MIT Press, Cambridge, Mass., 1988.

[van Emden and Kowalski 1976] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Generalized Semantics for Constraint Logic Programs *

Roberto Giacobazzi[1], Saumya K. Debray[2], Giorgio Levi[1]

1) Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
{giaco,levi}@di.unipi.it

2) Department of Computer Science
The University of Arizona
Tucson, AZ 85721
debray@cs.arizona.edu

Abstract

We present a simple and powerful generalized algebraic semantics for constraint logic programs that is parameterized with respect to the underlying constraint system. "Generalized semantics" abstract away from standard semantics objects, by focusing on the general properties of any (possibly non-standard) semantics definition. In constraint logic programming, this corresponds to a suitable definition of the constraint system supporting the semantics definition. An algebraic structure is introduced to formalize the constraint system notion, thus making applicable classical mathematical results and both a top-down and bottom-up semantics are considered. Non-standard semantics for CLP can then be formally specified by means of the same techniques used to define standard semantics. Different non-standard semantics for constraint logic languages can be specified in this framework: e.g. abstract interpretation, machine level traces and any computation based on an instance of the constraint system.

1 Introduction

Constraint logic programming (CLP) is a generalization of the pure logic programming paradigm, having similar model-theoretic, fixpoint and operational semantics [Jaffar and Lassez 87]. Since the basic operational step in program execution is a test for solvability of constraints in a given algebraic structure, CLP has in addition an algebraic semantics. CLP is then a general paradigm which may be instantiated on various semantic domains, thus achieving a good expressive power. One relevant feature is the distinction between testing for solvability and computing a solution of a given constraint formula. In the logic programming case, this corresponds to the unification process, which tests for solvability by computing a solution (a set of equations in solved form or *most general unifier*). In CLP, the computation of a solution of a constraint is left to a constraint solver, which does not affect the semantic definition of the language. This allows different computational domains, e.g. real arithmetic, to be considered without requiring complicated encodings of data objects as first order terms. Since the fundamental linguistic aspects of CLP can be separated from the details specific to particular constraint systems, it seems natural to parameterize the semantics of CLP languages with respect to the underlying constraint system [Saraswat *et al.* 91]. We refer to such a semantics as *generalized semantics*. It turns out that generalized semantics provide a powerful tool for dealing with a variety of applications relating to the semantics of CLP programs. For example, by considering a domain of "abstract constraints" instead of the "concrete constraints" that are actually manipulated during program execution, we obtain for free a formal treatment of abstract interpretation of CLP programs: this provides a foundation for dataflow analysis and program manipulation of CLP programs. In this paper we address the problem of defining a generalized semantics for constraint logic programs. This can also be the base to specify non-standard semantics for other logic-based languages (e.g. in [Barbuti *et al.* 92] Prolog control features are expressed in terms of a constraint logic language). The algebraic approach we take to constraint interpretation makes it easy to identify a suitable set of operators, which can be instantiated in different ways to obtain the definition of different

*The work of R. Giacobazzi and G. Levi was supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. under grant n. 9100880.PF69 and by the Esprit Basic Research Action 3012 - Compulog. The work of S. Debray was supported in part by the National Science Foundation under grant number CCR-8901283.

non-standard semantics. An interesting aspect of such a development is that non-standard interpretations such as abstract interpretations can be developed entirely within an algebraic framework: for example, the notion of "abstraction" can be characterized simply via additional axioms that specify which terms are to be considered "equal" under the abstract interpretation, and relationships between different abstract interpretations can be characterized in terms of homomorphisms between the corresponding algebras.

In this paper, two kinds of generalized semantics top-down and bottom-up, are considered. Since computations are always performed in the algebra of constraints, the two approaches represent just two ways to perform possibly non-standard computations. The reader is assumed to be acquainted with the basic notions of *lattice theory* and *sorted algebras*. Full proofs, not included due to space limitations, are present in the full version of this paper.

2 Constraint Algebras

As defined in [Jaffar and Lassez 87], the semantics of constraints is given in terms of an algebraic structure which interprets constraint formulas, while the semantics of a constraint logic program is given in terms of the well known fixpoint, model-theoretic and operational characterizations. In this section we introduce an incremental algebraic specification for constraint systems.

2.1 Term Systems

In the following we introduce the notion of *term system* as an algebra of terms provided with a binary operator which realizes substitutions [Cirulis 88]. We are interested in term systems where every term depends only on a finite number of variables[1]. They represent the first basic definition in the semantics construction.

Definition 2.1 A *term system* $\mathcal{T}$ is an algebraic structure $(\mathcal{T}, S, V)$ where we refer to the elements of $\mathcal{T}$ as *$\mathcal{T}$-terms* (*terms* in short); V is a countable set of *$\mathcal{T}$-variables* (*variables*, for short) in $\mathcal{T}$; S_V is a countable set of binary operations on $\mathcal{T}$, indexed by V; and the following conditions hold, for all $x, y \in V$ and $t, t', t'' \in \mathcal{T}$:

T_1. $s_x(t, x) = t$, *(identity)*

T_2. $s_x(t, y) = y$, if $x \neq y$, *(annihilation)*

T_3. $s_x(t, s_x(y, t')) = s_x(y, t')$ if $x \neq y$, *(renaming)*

T_4. $s_x(t', s_y(t'', t)) = s_y(s_x(t', t''), t)$ if $x \neq y$ and y *ind* t' *(independent composition)*

where a $\mathcal{T}$-term t is *independent* on the $\mathcal{T}$-variable x, denoted as "x *ind* t," if $s_x(t', t) = t$ for every $t' \in \mathcal{T}$. We say that a variable v *occurs in* a term t if $\neg(x \; ind \; t)$. ∎

Intuitively, $s_x(t, t')$ denotes the operation "substitute t for every occurrence of the variable x in t'." For notational convenience, we denote $s_x(t, t')$ as $[t/_x]\, t'$. This notation can be extended to substitutions on multiple variables, by writing $s_{x_1}(t_1, s_{x_2}(t_2, \cdots s_{x_k}(t_k, t) \cdots))$ as $[t_1/_{x_1} \; \ldots \; t_k/_{x_k}]\, t$.

Example 2.1 Let Σ be a denumerable collection of function symbols. We denote by $\tau(\Sigma, V)$ the set of possibly non-ground terms defined on Σ. The standard term system $\mathcal{T}_{(\Sigma,V)} = (\tau(\Sigma, V), \mathsf{Sub}, V)$ is a term system provided that substitutions in Sub perform idempotent substitutions. In this case v *ind* t iff the variable v does not occur in t. □

Let Π be a finite collection of predicate symbols. A $(\mathcal{T}, \Pi)$-atom has the form $p(t_1, ..., t_n)$ where $p \in \Pi$ and $t_i \in \mathcal{T}$, $\forall i = 1, ..., n$. We denote by $W_1 \backslash W_2$ the set W_1 where the elements in W_2 have been removed. The powerset of a set S is denoted by 2^S, and any tuple of syntactic objects (terms, atoms, *etc.*) $o_1, ..., o_n$ is denoted by $\langle o_1, ..., o_n \rangle$.

2.2 An Algebraic Framework

We give now a formal algebraic specification for the language of constraints on a given term system.

Definition 2.2 A *Closed Semiring* [Aho *et al.* 74] is an algebraic structure $(\mathcal{C}, \otimes, \oplus, 1, 0)$, such that: (1) $(\mathcal{C}, \oplus, 0)$ is a (join-)idempotent and commutative monoid; (2) $(\mathcal{C}, \otimes, 1)$ is a (meet-)monoid; (3) 0 is an *annihilator* for $\otimes$; (4) if $a_1, ..., a_n, ..$ is a countable sequence of elements in $\mathcal{C}$, $a_1 \oplus a_2 \oplus ... \oplus a_n \oplus ..$ exists and is unique; (5) associativity, commutativity and idempotence of $\oplus$ apply to infinite as well as finite joins; and (6) $\otimes$ *distributes* over finite and counably infinite joins. ∎

Example 2.2 [[Aho *et al.* 74]]
Let $\mathcal{A}_{\Re} = (\Re^+, +, min, 0, +\infty)$ where $\Re^+$ is the set of non-negative reals including $+\infty$, and $\mathcal{A}_{\Sigma} = (\Sigma^*, \cdot, \cup, \{\epsilon\}, \emptyset)$ where Σ^* is the family of sets of finite-length strings of symbols from the finite alphabet Σ (including the empty string ϵ) and $\cdot$ denotes concatenation. Both $\mathcal{A}_{\Re}$ and $\mathcal{A}_{\Sigma}$ are closed semirings. Notice that in $\mathcal{A}_{\Sigma}$ $\cdot$ is not commutative. □

[1] A more general definition that considers sets of arbitrary cardinalities is given in [Cirulis 88]: for our purposes, it suffices to consider denumerable sets.

Any semantics definition supports the notion of *observable behaviour* for a given program. Modelling answer constraints in constraint logic programming corresponds to consider answer constraints as the observable property for any CLP program. Thus, the notion of solution for a given answer constraint has to be restricted (projected) to the variables of the corresponding query (output variables). Closed semirings are too weak to capture the notion of variable projection. We handle this notion by means of a family of "hiding" operators on the underlying algebra, as in [Saraswat *et al.* 91]. *Cylindric algebras* [Henkin *et al.* 85] provide a suitable framework to enhance our algebraic structures. A cylindric algebra is formed by enhancing a Boolean algebra by means of a family of unary operations called *cylindrifications*. The intuition here is that given a constraint c, the cylindrification operation $\exists_S(c)$ yields the constraint obtained by "projecting out" information about the variables in S from c. They are necessary here because when we solve a goal in a constraint logic program, we are interested only in constraints on the variables that appear in that goal: thus, any additional constraints that may have been imposed on other variables during the course of the computations should be projected away in the representation of the final answer constraint. This is accomplished using cylindrification. Technically, cylindric algebras allow us to make projections on finite sets of variables. However, since our semantic formulation is in terms of infinite unfolding, as discussed later in the paper, it may also be necessary to allow projections on infinite sets. The machinery of cylindric algebras is not quite adequate for this, but the problem can be handled using *polyadic algebras* [Henkin *et al.* 85], which allow possibly countably many cylindrifications.

Diagonal elements [Henkin *et al.* 85] are considered as a way to provide parameter passing [Saraswat *et al.* 91]. In constraint logic programming the equality symbol "=" is assumed in any constraint system to provide term unification. However, cylindric algebras were introduced to provide an algebraic formalization of first-order-logic, actually oriented to first-order-languages without operation symbols; thus ignoring all terms but variables. This framework is not adequate to provide an algebraic semantic framework for constraint logic programs. We extend diagonal elements to deal with generic terms, following the approach in [Cirulis 88]. Diagonal elements represent equations on a given term system $\mathcal{T}$. This approach results in introducing "term-unification" (i.e. equations on terms) as distinguished elements in the algebra.

Definition 2.3 A *cylindric closed semiring* is an algebraic structure $(\mathcal{C}, \otimes, \oplus, 1, 0, \exists_\Delta, d_{t,t'})_{\Delta \subseteq V; t,t' \in \mathcal{T}}$ where $\mathcal{C}$ is a set called the *universe* of semiring, V is a countable set of variables, $\mathcal{T}$ is a term system; $0, 1, d_{t,t'}$ are distinct elements of $\mathcal{C}$, for each $t, t' \in \mathcal{T}$; $\{\exists_\Delta\}_{\Delta \subseteq V}$ are unary operations on $\mathcal{C}$; $\otimes, \oplus$ are binary operations on $\mathcal{C}$; such that the following postulates are satisfied for any $c, c' \in \mathcal{C}$; $\Delta, \Psi \subseteq V$ and $t, t' \in \mathcal{T}$:

S_1. the structure $(\mathcal{C}, \otimes, \oplus, 1, 0)$ is a closed semiring,

C_1. $\exists_\Delta 0 = 0$,

C_2. $c \oplus \exists_\Delta c = \exists_\Delta c$,

C_3. $\exists_\Delta(c \otimes \exists_\Delta c') = \exists_\Delta c \otimes \exists_\Delta c'$,

C_4. $\exists_\Delta \exists_\Psi c = \exists_{(\Delta \cup \Psi)} c$,

C_5. $\exists_\Delta$ distributes over finite and countably infinite joins,

D_1. $d_{t,t} = 1$,

D_2. $d_{t,t'} = \exists_{\{x\}}(d_{t,x} \otimes d_{x,t'})$ where x *ind* t, t',

D_3. $d_{z,[t/x]t'} = \exists_{\{x\}}(d_{z,t'} \otimes d_{x,t})$, where $z \neq x$ and x *ind* t, z *ind* t'.

∎

Notice that Axiom D_3 relates the notion of substitution in the term system $\mathcal{T}$ with diagonal elements of $\mathcal{C}$ (which intuitively correspond to the notion of equality constraints) in the expected way.

The notions of "independence" and "occurrence" of variables extends in the obvious way from terms in $\mathcal{T}$ to constraints in $\mathcal{C}$. Let $\{x_1, ..., x_n\} \subseteq V$, in the following we will denote $\exists_{var(c)/\{x_1,...,x_n\}} c$, i.e. hiding from all the variables in c except $\{x_1, \ldots, x_n\}$, as $\exists(c)_{\{x_1,...,x_n\}}$. We also denote as $d_{\langle t_1,...,t_n\rangle,\langle t'_1,...,t'_n\rangle}$ the element $d_{t_1,t'_1} \otimes ... \otimes d_{t_n,t'_n}$, where $t_1, ..., t_n, t'_1, ..., t'_n \in \mathcal{T}$. Any closed semiring can be extended to a cylindric closed semiring by letting $d_{t,t'} = 1$ for each $t, t' \in \mathcal{T}$ and $\exists_\Delta c = c$ for each $c \in \mathcal{C}$ and $\Delta \subseteq \mu$. Following [Henkin *et al.* 85] we refer to them as *discrete cylindric closed semirings*.

In the general theory of cylindric algebras, the commutative and transitive properties of diagonal elements (i.e. $d_{t,t'} = d_{t',t}$ and $(d_{t,t'} \otimes d_{t',t''}) \oplus d_{t,t''} = d_{t,t''}$) are derived by the axioms. Because of the weakness of cylindric closed semirings, these properties are not derivable from the axioms. However they are not required in proving the semantic results given below. They can be added to provide the theory of equality.

Given a closed semiring, we can induce a partial ordering relation $\sqsubseteq_\oplus$ on $\mathcal{C}$, such that $c_1 \sqsubseteq_\oplus c_2$ iff

$c_1 \oplus c_2 = c_2$. As a consequence, $(\mathcal{C}, \sqsubseteq_\oplus)$ is a complete lattice.

2.3 Constraint Systems

In this section we formalize the notion of *constraint system*, based on the above algebraic framework.

Definition 2.4 A *constraint interpretation structure* is any cylindric closed semiring. Given a constraint interpretation structure $\mathcal{A}$ with universe $\mathcal{C}$, an *$\mathcal{A}$-constraint* (*constraint* for short) is any element in $\mathcal{C}$. ∎

Idempotence, associativity and commutativity are the least set of properties [Barbuti *et al.* 91,Debray and Ramakrishnan 91] which allow $\oplus$ to model the set union operation. $\otimes$ corresponds to the constraint conjunction and plays the important role of collecting the information during the computation. Distributivity allows to represent constraints as possibly infinite joins of finite meets (also called *simple constraints*). Closure on (possibly infinite) countable elements in $\mathcal{C}$ allows to denote infinite joins of constraints.

Example 2.3 Let us assume that $\Pi = \Pi_C \cup \Pi_P$ and $\Pi_C \cap \Pi_P = \emptyset$. We refer to $\Lambda_{(\Sigma,\Pi_C)}$ as the *free algebra of formulas* in the sorted vocabulary (S, Σ, Π_C): where S (sort) is a set of symbols, Σ a specified set of operations with a corresponding signature on S and Π_C a set of predicate symbols with a signature on S; enhanced with the disjunction symbol $\vee$, the conjunction symbol $\wedge$, the existential quantifier $\exists$, the identity symbol $=$, the truth and falsehood symbols T and F and closed under countably infinite disjunctions of formulas in $\Lambda_{(\Sigma,\Pi_C)}$. Equations and possibly existentially quantified $(\mathcal{T}_{(\Sigma,V)}, \Pi_C)$-atoms are called *atomic constraints.*

Let us consider the *solution compact* many sorted algebraic structure $\mathcal{R}_{(\Sigma,\Pi_C)}$ [Jaffar and Lassez 87], defined over the many sorted alphabet (S, Σ, Π_C), consisting of: a collection $D\mathcal{R}$ of non-empty sets denoted $\{D\mathcal{R}_s\}_s$, where $s \in S$; an assignment of a function $D\mathcal{R}_{s_1} \times ... \times D\mathcal{R}_{s_n} \rightarrow D\mathcal{R}_s$ to each n-ary function symbol $f \in \Sigma$, where $(s_1, ..., s_n, s)$ is the signature of f; an assignment of a function $D\mathcal{R}_{s_1} \times ... \times D\mathcal{R}_{s_n} \rightarrow \{true, false\}$ to each n-ary predicate symbol $p \in \Pi_C$, where $(s_1, ..., s_n)$ is the signature of p.

Let us consider a constraint c in $\Lambda_{(\Sigma,\Pi_C)}$. $\mathcal{R} \models c$, iff there exists a mapping ϑ (the solution of the constraint) from each distinct free variable x in c into $D\Re_s$, (free variables in a constraint c are denoted $FV(c)$) where s is the sort associated with x, and $c\vartheta$ is $\mathcal{R}$-equivalent to T ($\mathcal{R} \models c\vartheta$).

Let $c_1 = \bigvee_{i \in I_1} c'_i$ and $c_2 = \bigvee_{i \in I_2} c''_i$ denote possibly infinite disjunctions of conjunctions of atomic constraints c'_i and c''_i, where i ranges over I_1 and I_2 being sets of possibly infinite indexes. The equivalence relation $\approx_\mathcal{R}$ on $\Lambda_{(\Sigma,\Pi_C)}$ is defined as follows

$$c_1 \approx_\mathcal{R} c_2 \text{ iff } \bigcup_{i \in I_1} \{\vartheta \mid \mathcal{R} \models c'_i\vartheta\} = \bigcup_{i \in I_2} \{\vartheta \mid \mathcal{R} \models c''_i\vartheta\}.$$

$\approx_\mathcal{R}$ is a congruence relation on the one sorted algebra $(\Lambda_{(\Sigma,\Pi_C)}, \wedge, \vee, T, F, \exists_X, t = t')_{X \subseteq V; t,t' \in \mathcal{T}_{(\Sigma,V)}}$. The standard constraint interpretation structure is then given by the quotient algebra, denoted as $\mathcal{A}_{st} = (\Lambda_{(\Sigma,\Pi_C)}, \wedge, \vee, T, F, \exists_X, t = t')_{X \subseteq V; t,t' \in \mathcal{T}_{(\Sigma,V)}}/_{\approx_\mathcal{R}}$. It is trivially a meet-idempotent and commutative cylindric closed semiring. □

Example 2.4 [$CLP(\mathcal{H})$] Let us consider the following signature associated with the usual Herbrand universe definition, $\Sigma = \{a, b, ..., f, g, ...\}$. Atomic constraints are one sorted equations on the term system $\mathcal{T}_{(\Sigma,V)}$. The corresponding Herbrand interpretation structure $\mathcal{A}_\mathcal{H}$, is the quotient algebra $(\mathcal{C}_\mathcal{H}, \wedge, \vee, T, F, \exists_X, t = t')_{X \subseteq V; t,t' \in \mathcal{T}_{(\Sigma,V)}}/_{\approx_{EQ}}$, modulo $\approx_{EQ}$, where $\mathcal{C}_\mathcal{H} = \left\{ t = t' \,\middle|\, t, t' \in \mathcal{T}_{(\Sigma,V)} \right\}$ and $\approx_{EQ}$ is the equivalence relation induced by the algebraic structure interpreting diagonal elements as unification [Jaffar and Lassez 87]. It is straightforward to prove that this corresponds to the pure logic programming case. □

To relate constraint interpretation structures, we follow the approach to "static semantics correctness" in [Barbuti and Martelli 83]. Correctness of non-standard semantics specifications can be handled in an algebraic way through the notion of morphism. However, the algebraic notion of morphism can be made less restrictive by assuming that the carriers of the involved algebras be partially ordered sets. We introduce a weaker notion of morphism, capturing the approximation possibly induced by abstract interpretations or any approximate semantics defined in the framework.

Definition 2.5 Let A_Σ and B_Σ be (many sorted) algebraic structures over the sorted alphabet (S, Σ). Let us assume that for each $s \in S$, $(DB_s, \preceq_{DB_s})$ is a partially ordered set. A *weak morphism* $\sigma : A \rightarrow B$ is a family of functions $\sigma_s : DA_s \rightarrow DB_s$, for $s \in S$, such that: $\sigma_s(f_A) \preceq_{DB_s} f_B$, for each constant symbol $f :\rightarrow s$ in Σ and $\sigma_s(f_A(a_1, ..., a_n)) \preceq_{DB_s} f_B(\sigma_{s_1}(a_1), ..., \sigma_{s_n}(a_n))$, for each operation symbol $f : s_1 ... s_n \rightarrow s$ in Σ. ∎

Definition 2.6 Let $\mathcal{A}$ be a constraint interpretation structure. A *constraint interpretation morphism* is a weak morphism ε from $(\Lambda_{(\Sigma,\Pi_C)}, \wedge, \vee, T, F, \exists_X, t = t')_{X \subseteq V; t,t' \in \mathcal{T}_{(\Sigma,V)}}$ in $\mathcal{A}$. ∎

Example 2.5 The *standard constraint interpretation morphism* ε_{st} is a morphism which associates with any formula in $\Lambda_{(\Sigma,\Pi_C)}$, the corresponding equivalence class modulo $\approx_{\mathcal{R}}$. □

In general, a constraint system is an interpretation (in a closed semiring) for constraint formulas.

Definition 2.7 A *constraint system* is a pair $\Gamma = \langle \mathcal{A}, \varepsilon \rangle$ where $\mathcal{A}$ is a constraint interpretation structure and ε is a corresponding constraint interpretation morphism. ∎

Similar algebraic structures for the definition of constraint systems have been introduced in [Saraswat *et al.* 91] to specify the semantics of the more complex class of *concurrent constraint* languages characterized by the *ask/tell* paradigm.

Constraint systems are specified as *systems of partial information* in the style of Scott's information systems [Scott 82], (*simple constraint systems*), which are tuples $(\mathcal{C}, \Delta, \vdash)$, where $\mathcal{C}$ is an non-empty set of "primitive" constraints and $\vdash \subseteq 2^{\mathcal{C}} \times \mathcal{C}$ is an *entailment relation* such that $\forall u, v \in 2^{\mathcal{C}}$: (1) $u \vdash \Delta$, (2) $u \vdash X$ whenever $X \in u$ and (3) if $v \vdash Y$ for all $Y \in u$ and $u \vdash X$, then $v \vdash X$. The relation $\vdash$ can be extended on $2^{\mathcal{C}} \times 2^{\mathcal{C}}$ as follows: $\forall u, v \in 2^{\mathcal{C}}$, $u \vdash v$ iff $u \vdash X$ for every $X \in v$.

Composition of constraints is defined in terms of set-union, which is a well known commutative and idempotent operator. Hiding and parameter passing are handled by cylindrification (only finite-variable cylindrifications are allowed) and diagonal elements. The difference is then in the underlying algebraic structure: while information systems provide an elegant framework to develop the (standard) semantics for concurrent constraint languages, we are interested in more appropriate algebraic structures to generalize standard semantics results on CLP. In our case, the constraint system is parametric with respect to a given term system. This introduces a more structured approach (two steps) to non-standard constraint system definition (e.g. abstract interpretation). As for the basic algebraic structure, the choice of *closed semirings* results more natural in the context of the present paper. We are interested in possibly non-commutative/idempotent compositions (meets) of constraints (see $\mathcal{A}_\Sigma$ in Example 2.2). Moreover (see *Prop* in Section 4.1 below) standard logical and arithmetic operators (e.g. $\vee$, $\wedge$, T and F) can be specified more naturally as an instance of a closed semiring instead of as an instance of an information system. Nevertheless, it is easy to associate an information system with any $\otimes$-commutative and idempotent closed semiring. Let $(\mathcal{C}, \otimes, \oplus, 1, 0)$ be a $\otimes$-commutative and idempotent closed semiring. The corresponding information system $(\mathcal{C}, \Delta, \vdash)$ is defined as follows $\Delta = 0$ and $\forall u, v \in 2^{\mathcal{C}}$: $u \vdash v$ iff $v \sqsubseteq_{\oplus} u$.

The key difference is in the semantics definition. In [Saraswat *et al.* 91] the semantics of constraint languages is specified as closures on the constraint system, thus amalgamating the semantics construction and data-objects. We follow the standard approach (see section below) in generalizing the standard operational and fixpoint semantics characterizations, already known in logic programming. A more structured approach to the generalization process can be obtained by separating the domain of constraints with the various techniques to construct models (e.g. fixpoints of continuous transformations) for constraint logic programs. The independence of the semantics constructions from the underying constraint systems focuses the generalization process on the constraint system definition, thus simplifying the specification of non-standard semantics.

Generalized constraint logic programs are defined in the usual way. Let $\mathcal{A}$ be a constraint interpretation structure on the term system $\mathcal{T}$. An *$\mathcal{A}$-clause* is a formula of the form $H \ :- \ c \Box B_1, ..., B_n$ with $n \geq 0$ where H (the *head*) and $B_1, ..., B_n$ (the *body*) are $(\mathcal{T}, \Pi_P)$-atoms, c is an $\mathcal{A}$-constraint and $:-$ and "," denote logic implication and conjunction respectively. An *$\mathcal{A}$-goal* is a formula $c \Box B_1, ..., B_n$, where c is constraint and each B_i is $(\mathcal{T}, \Pi_P)$-atom. A *(generalized) constraint logic program*, also called $\mathcal{A}$-program is a finite set of $\mathcal{A}$-clauses.

3 Generalized Semantics

The mechanism introduced in [Falaschi *et al.* 89] to model computed answer substitutions is generalized in CLP, by allowing constrained atoms into the base of interpretations [Gabbrielli and Levi 91]. Each constrained atom $p(\bar{x}) \ :- \ c$, in fact, represents the set of instances $p(\bar{x})\vartheta$, where ϑ is a solution of the constraint c.

Definition 3.1 Let $\mathcal{A}$ be an interpretation structure. A *constrained atom* has the form $p(\bar{x}) \ :- \ c$ where c is an $\mathcal{A}$-constraint, $p(\bar{x})$ is a $(\mathcal{T}, \Pi_P)$-atom and $FV(c) = \bar{x}$. ∎

Definition 3.2 Let $\mathcal{A}$ be an interpretation structure and $\Im$ be the corresponding set of constrained

atoms. We define a partial order $\preceq$ on $\Im$ such that $p(\bar{x}_1) :- c_1 \preceq p(\bar{x}_2) :- c_2$ iff there exists $\bar{x}'$ such that $\exists_{\{\bar{x}_1\}}(d_{\bar{x}',\bar{x}_1} \otimes c_1) \sqsubseteq_{\oplus} c_2$. ∎

The equivalence relation induced by the partial order $\preceq$ is denoted by $\sim$. The *$\mathcal{A}$-base of interpretations* $\mathcal{B}$, is $\Im/_{\sim}$.

Definition 3.3 $\Xi \subseteq 2^{\mathcal{B}}$ is the collection of sets of constrained atoms I such that $I \in \Xi$ iff $I \uplus \emptyset = \emptyset \uplus I = I$, where $\uplus$ is defined as: $\lambda I_1, I_2. \left\{ p(\bar{x}) :- \sum_j \tilde{c}_j \;\middle|\; \begin{array}{l} p(\bar{x}_j) :- c_j \in I_1 \cup I_2, \\ \tilde{c}_j = \exists_{\bar{x}_j}(d_{\bar{x},\bar{x}_j} \otimes c_j) \\ \text{and } \bar{x} \; ind \; c_j \end{array} \right\}$. ∎

An *$\mathcal{A}$-interpretation* is any element of Ξ. $\uplus$ is strongly related to $\oplus$. As usual we define $I_1 \sqsubseteq_{\uplus} I_2$ iff $I_1 \uplus I_2 = I_2$ such that $(\Xi, \sqsubseteq_{\uplus})$ is a complete lattice. Each interpretation always consists of a finite set of constrained atoms, containing at most one constrained atom for each program predicate symbol: $p(\bar{x}) :- \sum_{j \in W} c_j \in I$, where for each $j \in W$, c_j represents the set of admissible (i.e. computable in the program) solutions for the predicate symbol p, on the variables $\bar{x}$. As a consequence infinite joins of constraints are allowed in constrained atoms. This is well defined by the closure of $\mathcal{C}$. In the following we will often omit $\mathcal{A}$ in specifying programs, goals, *etc.*

3.1 Operational Semantics

Let $\Gamma = \langle \mathcal{A}, \varepsilon \rangle$ be a constraint system and P be an $\mathcal{A}$-program. Define $\rightsquigarrow_P \subseteq$ *$\mathcal{A}$-Goals* $\times$ *$\mathcal{A}$-Goals* (an *$\mathcal{A}$-derivation step*) to be the smallest relation such that $G^{\mathcal{A}} \rightsquigarrow_P G'^{\mathcal{A}}$ iff $G^{\mathcal{A}} = c_0 \Box p_1(\bar{t}_1), ..., p_n(\bar{t}_n)$; there exist n (renamed apart) versions of clauses in P: $p_i(\bar{x}_i) :- c_i \Box \bar{G}_i$, $i = 1..n$; $G'^{\mathcal{A}} = c_0 \otimes \tilde{c}_1 \otimes ... \otimes \tilde{c}_n \Box \bar{G}_1, ..., \bar{G}_n$, where for each $i = 1..n$, $\tilde{c}_i = d_{\bar{x}_i,\bar{t}_i} \otimes c_i$.

An $\mathcal{A}$-derivation from an $\mathcal{A}$-goal $G^{\mathcal{A}}$ is a finite or infinite sequence of (different) $\mathcal{A}$-goals such that every $\mathcal{A}$-goal is obtained from the previous one by means of a single $\mathcal{A}$-derivation step. A successful derivation is a finite sequence whose last element has an empty body. The operational semantics is then defined in terms of the successful computations specified by the transitive closure of the transition relation on $\mathcal{A}$-goals:

$$\mathcal{O}^{\Gamma}(P) = \left\{ p(\bar{x}) :- \sum \exists(c)_{\bar{x}} \;\middle|\; 1 \Box p(\bar{x}) \rightsquigarrow_P^* c \Box \right\}.$$

Goal dependent semantics is defined in terms of a function $\mathcal{G}$ that yields the computed answer constraint for any $\mathcal{A}$-goal, such that

$$\mathcal{G}(G^{\mathcal{A}}) = \exists(c)_{var(G^{\mathcal{A}})} \text{ iff } G^{\mathcal{A}} \rightsquigarrow_P^* c \Box .$$

Theorem 3.1 *Let $G^{\mathcal{A}} = c_0 \Box p_1(\bar{t}_1), ..., p_n(\bar{t}_n)$ be a goal. $\mathcal{G}(G^{\mathcal{A}}) = c$ iff there exist $p_i(\bar{x}_i) :- c_i \in \mathcal{O}^{\Gamma}(P)$, for $i = 1, ..., n$ and $c = \exists(c_0 \otimes d_{\bar{x}_1,\bar{t}_1} \otimes c_i ... \otimes d_{\bar{x}_n,\bar{t}_n} \otimes c_n)_{var(G^{\mathcal{A}})}$.*

3.2 Fixpoint Semantics

In this section we define a fixpoint semantics which is proved to be equivalent to the operational semantics.

Definition 3.4 Let P be an $\mathcal{A}$-program, the mapping $T_P^{\mathcal{A}} : \Xi \to \Xi$, is defined as follows $T_P^{\mathcal{A}}(I) = \biguplus_{C \in P} T_C^{\mathcal{A}}(I)$ where if $C :\ p(t) :- c \Box p_1(\bar{t}_1), ..., p_n(\bar{t}_n)$ then

$$T_C^{\mathcal{A}}(I) = \left\{ p(\bar{x}) : -\exists(\tilde{c})_{\bar{x}} \;\middle|\; \begin{array}{l} \text{for each } i = 1...n : \\ p_i(\bar{x}_i) : -c_i \in I \\ c'_i = d_{\bar{x}_i,\bar{t}_i} \otimes c_i \\ \tilde{c} = d_{\bar{x},\bar{t}} \otimes c \otimes c'_1 .. \otimes .. c'_n \\ \bar{x} \; ind \; c, c'_1, ..., c'_n \end{array} \right\}$$

∎

$T_P^{\mathcal{A}}$ is a continuous function on the complete lattice $(\Xi, \sqsubseteq_{\uplus})$. Let $lfp(f)$ denote the least fixpoint of a function f and $\mathcal{F}^{\Gamma}(P) = lfp(T_P^{\mathcal{A}}) = T_P^{\mathcal{A}} \uparrow \omega$. The following result states the equivalence between the operational and the fixpoint semantics, for any constraint system Γ.

Theorem 3.2 *Let P be a program and Γ a constraint system. Then $\mathcal{F}^{\Gamma}(P) = \mathcal{O}^{\Gamma}(P)$.*

4 Abstract Interpretation of CLP

The definition of an abstract constraint system is performed in two steps: *term abstraction* and *constraint abstraction.* In the first step new syntactic objects are introduced to represent sets of concrete terms. In the second one, constraints on the abstracted term system are abstracted. Since the complete lattice of interpretations is induced by the closed semiring structure, any abstract interpretation will correspond to a suitable definition of a constraint system associated with a particular application.

Definition 4.1 Given a constraint system $\Gamma = \langle \mathcal{A}, \varepsilon \rangle$, a constraint system $\Gamma' = \langle \mathcal{A}', \varepsilon' \rangle$, is correct with respect to Γ iff there exists a weak algebraic morphism α_c ($\alpha_c : \mathcal{A} \to \mathcal{A}'$) which is a monotonic mapping of $(\mathcal{C}, \sqsubseteq_{\oplus})$ into $(\mathcal{C}', \sqsubseteq_{\oplus'})$. ∎

Notice that, since α_c is monotonic, it behaves as an algebraic morphism with respect to the $\oplus$ operator. Termination has been guaranteed by requiring that all chains be finite.

Definition 4.2 A constraint interpretation structure $\mathcal{A}$ is *Noetherian* iff $(\mathcal{C}, \sqsubseteq_{\oplus})$ does not contain any infinite chain. A constraint system $\langle \mathcal{A}, \varepsilon \rangle$ is Noetherian iff $\mathcal{A}$ is Noetherian. ∎

Given a Noetherian constraint system Γ, it is easy to prove that $(\Xi, \sqsubseteq_{\uplus})$ is Noetherian. An *abstract constraint system* is a Noetherian constraint system $\Gamma^\sharp$ which is correct with respect to the standard one Γ_{st}. It is straightforward to show that in any abstract constraint system $\langle \mathcal{A}^\sharp, \varepsilon^\sharp \rangle$, $\varepsilon^\sharp = \alpha_c \circ \varepsilon_{st}$. Moreover, by weakness and monotonicity, the composition of two monotonic weak morphisms is still a monotonic weak morphism. Let $\Gamma^\sharp$ be a correct abstract constraint system. The mapping $\alpha : \Xi \to \Xi^\sharp$ such that $\alpha(I) = \left\{ p(x) \ :- \ \alpha_c(c) \,\middle|\, p(x) \ :- \ c \in I \right\}$ is continuous. Abstract interpretations for constraint logic programs correspond to the definition of an abstract constraint system together with a program evaluation strategy. The first defines what an abstract computation is, while the second one deals with a specific evaluation strategy to collect abstract information. Top-down abstract interpretations correspond to the abstraction of the operational semantics. Bottom-up evaluations instead allow to compute a finite abstract approximation of the fixpoint semantics associated with a given constraint logic program. Goal-independence is an attractive feature of bottom-up evaluations. Global program analysis, especially useful in type inference, can then be specified as a bottom-up evaluation in a suitable constraint system.

Proposition 4.1 *Given a program P and an abstract constraint system $\Gamma^\sharp = \langle \mathcal{A}^\sharp, \varepsilon^\sharp \rangle$, there exists a finite positive k such that $\mathcal{F}^{\Gamma^\sharp}(P) = T_P^{\mathcal{A}^\sharp} \uparrow k$.*

The correctness of the analysis is reduced to the correctness of the constraint system.

Theorem 4.2 *Let P and $\Gamma^\sharp$ be a program and an abstract constraint system respectively. Then $\alpha(\mathcal{O}^\Gamma(P)) \sqsubseteq^\sharp_{\uplus} \mathcal{O}^{\Gamma^\sharp}(P)$ and $\alpha(\mathcal{F}^\Gamma(P)) \sqsubseteq^\sharp_{\uplus} \mathcal{F}^{\Gamma^\sharp}(P)$.*

Example 4.1 The closed semiring $\mathcal{A}_{\Re}$ developed in Example 2.2 can be used to define a simple complexity analysis tool for constraint logic programs on reals, $CLP(\Re)$ [Jaffar and Lassez 87]. Let $|.|_\tau : \tau(\Sigma, V) \to \mathcal{N}$ be a mapping associating a "weight" with any term, where $\mathcal{N}$ is the set of natural numbers. Let us consider a morphism ε such that for each constraint c: $t_1 < t_2$, $\varepsilon(c) = |t_1|_\tau + |t_2|_\tau$.

The interpretation structure $(\mathcal{N}, +, min, 0, +\infty)$, where cylindrifications are defined as in the discrete case and diagonal elements are natural numbers $d_{t_1,t_2} = |t_1|_\tau + |t_2|_\tau$, is trivially Noetherian. A *lower-bound* complexity analysis can be performed returning a lower bound estimation of the costs in arithmetic computations, as in the following example for a simple integration routine:

$$
\begin{array}{l}
\mathsf{int}(A, B, x) \ :- \ 0 < B - A \leq \epsilon, \\
\qquad x = (B - A) * f(A + (B - A)/2) \square \ \mathsf{E}(\epsilon). \\
\mathsf{int}(A, B, x) \ :- \ B - A > \epsilon, \\
\qquad M = A + (B - A)/2, \ x = x_1 + x_2 \\
\qquad \square \ \mathsf{int}(A, M, x_1), \mathsf{int}(M, B, x_2), \mathsf{E}(\epsilon). \\
\mathsf{E}(x) \ :- \ x = 1/n \square \mathsf{N}(n). \\
\mathsf{N}(n') \ :- \ n' = n + 1 \square \mathsf{N}(n). \\
\mathsf{N}(n') \ :- \ n' = 1 \square.
\end{array}
$$

Let, for instance, c_+, c_* and c_f be the costs of addition, multiplication/division and f respectively. Variables and constants have a zero cost. Thus, denoting Γ_τ such constraint system:

$$
\mathcal{F}^{\Gamma_\tau}(P) = \left\{ \begin{array}{lll} int(A, B, x) & :- & 4c_+ + 3c_* + c_f, \\ E(n') & :- & c_*, \\ N(n') & :- & 0 \end{array} \right\}
$$

□

A space of approximate constraints can be specified by defining an auto-weak morphism ρ which is an *upper closure operator* (i.e. an idempotent, monotonic and extensive operator) on $(\mathcal{C}, \sqsubseteq_{\oplus})$. As shown in [Cousot and Cousot 79] the approximation process essentially consists in partitioning the space of constraints so that no distinction is made between equivalent constraints, all approximated by a representant of their equivalence class. The equivalence relation is induced by an upper closure operator ρ: $c_1 \equiv_\rho c_2$ iff $\rho(c_1) = \rho(c_2)$. In [Cousot and Cousot 79] different equivalent methods for specifying abstract domains (i.e. upper closure operators) are presented. However, there are standard techniques in algebraic specifications that allow the definition of abstract constraint systems. For example, cylindrifications can be interpreted as abstractions on the algebra of constraints.

Proposition 4.3 *Let $\Delta \subseteq V$; $\exists_\Delta$ is an auto-weak morphism and upper closure operator on $(\mathcal{C}, \sqsubseteq_{\oplus})$.*

Existential quantification is then a way to define abstract domains. The space of approximate constraints can also be specified by adding axioms to

the underlying constraint system $\mathcal{A}$. These additional axioms extend the meaning of the diagonal elements $d_{t,t'}$ of the algebra, in effect specifying which objects are to be considered "equivalent" from the perspective of the analysis. This is illustrated by the following example:

Example 4.2 Consider the logic program P

$$\begin{array}{l} \mathsf{p}(0). \\ \mathsf{p}(s(x)) \ :- \ \mathsf{q}(x). \\ \mathsf{q}(s(x)) \ :- \ \mathsf{p}(x). \end{array}$$

and a simple type (parity) analysis for P. Interpreting P as a constraint logic program on the Herbrand constraint system $\mathcal{A}_{\mathcal{H}}$, the type analysis can be specified by extending the axioms specifying the constraint system with the additional axiom: $s(s(x)) = x$. The resulting constraint system, denoted by $\mathcal{A}^{ex}_{\mathcal{H}}$, is trivially Noetherian. The semantics of P in $\mathcal{A}_{\mathcal{H}}$ is $\{p(x) \ :- \ x = 0 \ \bigvee_{n\geq 1} x = s^{2n}(0); \ q(x) \ :- \ \bigvee_{n\geq 1} x = s^{2n-1}(0)\}$; whereas the interpretation in $\mathcal{A}^{ex}_{\mathcal{H}}$ returns $\{p(x) \ :- \ x = 0; \ q(x) \ :- \ x = s(0)\}$. The meaning of P in $\mathcal{A}^{ex}_{\mathcal{H}}$ captures the type of the predicate p and q, computing even and odd numbers respectively. □

A very useful analysis on the relationships among variables of a program can be specified in our framework [Cousot and Halbwachs 78]. The automatic derivation technique in [Verschaetse and De Schreye 91] for linear size relations among variables in logic programs can be suitably specified as a constraint computation. A constraint system of *affine relationships* (i.e. linear equalities of the form $c_0 = c_1X_1 + ... + c_nX_n$) can be defined by specifying intersection, disjunction and cylindrification (restriction) as given in [Verschaetse and De Schreye 91]. Generalizations considering linear inequalities, as proposed in [Cousot and Halbwachs 78], can still be defined in our framework, thus making explicit the strong connection between automatic detection of linear relationships among variables and $CLP(\Re)$ computations. Applications of this analysis are: compile time overflow, mutual exclusion, constraint propagation, termination *etc.* [Jørgensen *et al.* 91].

4.1 Generalized Rigidity Analysis

There exists a wide class of abstract interpretation techniques for the analysis of *ground dependences* (also named *covering*) of pure logic programs [Barbuti *et al.* 91,Cortesi *et al.* 91]. In this section we extend the ground dependence notion by means of the notion of *rigidity.*

A norm is a function weighting terms. Let us recall some basic concepts about norms. For a more accurate treatment on this subject see [Bossi *et al.* 90].

Definition 4.3 Let $\mathcal{T}$ ba a term system. A norm on $\mathcal{T}$ is a function $|..|_\varsigma : \mathcal{T} \rightarrow \mathcal{N}$, mapping any term $t \in \mathcal{T}$ into a natural number. ∎

Example 4.3 The following weighting map is a norm on the Herbrand term system: $|t|_{size} = 0$ if t is a variable or $t = []$, $|t|_{size} = 1 + |tail|_{size}$ if $t = [h|tail]$. □

In order to extend the notion of groundness and ground dependences [Barbuti *et al.* 91, Cortesi *et al.* 91] to deal with a more refined one, able to take into account only the relevant subterms of a given (possibly non-ground) term t, we address the notion of rigidity as introduced in [Bossi *et al.* 90].

Definition 4.4 Let $|..|_\varsigma$ be a norm on the term system $\mathcal{T}$. A term $t \in \mathcal{T}$ is rigid with respect to $|..|_\varsigma$ iff for any substitution of variables σ: $|\sigma t|_\varsigma = |t|_\varsigma$. ∎

The rigidity of terms turns out to be important in simplifying termination proofs. If a term is rigid, its weight will not be modified by further substitutions. Rigidity is then strongly related to groundness. Any ground term can not change its weight by instantiation, thus it is always rigid. This notion allows to identify those subterms which are relevant for the analysis purposes. Notice that given a norm $|..|_\varsigma$, and a non-rigid term $t \in \mathcal{T}$, there must exist some variable in t whose instantiation affects the weight of t. In the Herbrand case, results in [Bossi *et al.* 90] allow to restrict our attention to a particular class of norms: *semilinear norms* on Herbrand.

Definition 4.5 A norm on $\mathcal{T}_{(\Sigma,V)}$ is *semilinear* iff it may be defined according to the following structure: $|t|_\varsigma = 0$ if t is a variable; $|t|_\varsigma = c_0 + |t_{i_1}|_\varsigma + ... + |t_{i_m}|_\varsigma$ if $t = f(t_1, ..., t_n)$, where $c_0 \geq 0$ and $\{i_1, ..., i_m\} \subseteq \{1, ..., n\}$. ∎

Note that the position of the subterms which allow the principal term to change its weight by instantiation depends on the outermost term constructor only (i.e. f). These subterms are then relevant from the analysis viewpoint. All the non-relevant subterms are discarded by the analysis. Semilinear norms allow to reduce the rigidity notion to a syntactical property of terms. Let

$$Vrel_\varsigma(t) = \left\{ \ v \in V \ \middle| \ \exists \ \sigma \ such \ that \ |\sigma t|_\varsigma \neq |t|_\varsigma \ \right\}.$$

As shown in [Bossi *et al.* 90], given a semilinear norm $|..|_\varsigma$, a term $t \in \mathcal{T}_{(\Sigma,V)}$ is rigid iff $Vrel_\varsigma(t) = \emptyset$. The notion of semilinear norms can be generalized to

arbitrary term systems in a straightforward way, as follows: given a term system $\mathcal{T}$, we define a function $w : \mathcal{T} \longrightarrow \mathcal{N}$; for each $t \in \mathcal{T}$, an associated finite set of functions $F_t : t \longrightarrow \mathcal{T}$; and an associative and commutative function $\bowtie$: $\mathcal{N} \times \mathcal{N} \longrightarrow \mathcal{N}$.

Intuitively, for any term t, the value of $w(t)$ is the "initial weight" of the term t, the set of functions F_t correspond to the set of selectors for the "relevant" subterms, and $\bowtie$ indicates how the sizes of the subterms of a term are to be combined. Then, *generalized semilinear norms* can be defined as follows:

$$|t| = w(t) + \bowtie_{f \in F_t} |f(t)|.$$

Example 4.4 The "usual" notion of semilinear norms for Herbrand constraint systems can now be generalized as follows, let $c_0 \in \mathcal{N}$: $w(t) = 0$ if t is a variable, c_0 otherwise; if t is a variable then $F_t = \emptyset$; otherwise F_t consists of selectors for the relevant positions of t; $\bowtie$ is summation.

The "depth norm", which could not be expressed as a semilinear norm in the development of [Bossi *et al.* 90], can be defined as follows: $w(t) = 0$ if t is a variable, 1 otherwise; if t is a variable then $F_t = \emptyset$; otherwise if $t = f(t_1, \ldots, t_n)$ then $F_t = \{f_i | 1 \leq i \leq n\}$, where $f_i(t) = t_i$, i.e. f_i is the selector for the subterm at the i^{th} position; and $\bowtie$ is max. □

Let us consider the set $C(V)$ of finite conjunctions of variables in V (the empty conjunction is denoted ϵ) and a term abstraction map $\alpha_{\mathcal{T}} : \mathcal{T} \to C(V)$ such that, given a semilinear norm $|..|_\zeta$ and $t \in \mathcal{T}$, $\alpha_{\mathcal{T}}(t) = \left\{ x_1 \wedge ... \wedge x_m \;\middle|\; Vrel_\zeta(t) = \{x_1, ..., x_m\} \right\}$. Let $\mathcal{T}_\zeta$ be the corresponding abstract term system where substitutions are performed as usual. Marriott and Søndergaard have proposed an elegant domain, named *Prop*, further studied in [Cortesi *et al.* 91], to represent ground dependences among arguments in atoms. In [Codognet and Filè 91] an interesting application is introduced. *Prop* is formalized as a constraint system, and both groundness and definiteness analysis are specified by executing programs in CLP(*Bool*). The corresponding constraint system does not allow disjunctions of varables, without fully exploiting the expressive power of *Prop*. The general notion of ground dependence corresponding with any *Prop* formula (including disjunctions) cannot be specified.

Let $\mathcal{A}_\zeta = (Prop_\zeta, \vee, \wedge, T, F, \exists_X, t \leftrightarrow t')_{X \subseteq V; t,t' \in \mathcal{T}_\zeta}$ be the algebra of possibly existentially quantified formulas defined on the term system $\mathcal{T}_\zeta$; including the set of connectives $\vee$, $\wedge$, $\leftrightarrow$. Intuitively, the formula $x \wedge y \wedge z \leftrightarrow w \wedge v$ represents an equation $t = t'$ where $Vrel_\zeta(t) = \{x, y, z\}$ and $Vrel_\zeta(t') = \{w, v\}$; $x \wedge y$ represents a term whose rigidity depends upon variables x and y; while $x \vee y$ represents a set of terms whose rigidity depends upon variables x or y. Local variables are hidden by existential quantification, projecting away non-global variables in the computation [Codognet and Filè 91].

Let *Bool* be a boolean algebraic structure; $c \approx_{Bool} c'$ iff $Bool \models c \leftrightarrow c'$. It is easy to prove that $\mathcal{A}_\zeta / \approx_{Bool}$ is an abstract constraint system.

Example 4.5 Let us consider the semilinear norm "*size*" and the following constraint logic program on the Herbrand constraint system

$$\begin{array}{lll} \mathsf{append}(x_1, x_2, x_3) & :- & x_1 = [] \wedge x_2 = x_3. \\ \mathsf{append}(x_1, x_2, x_3) & :- & x_1 = [h|y] \wedge x_3 = [h|z] \\ & & \square \mathsf{append}(y, x_2, z). \end{array}$$

The corresponding abstract model is: $\{append(x_1, x_2, x_3) :- x_1 \leftrightarrow \epsilon \wedge x_2 \leftrightarrow x_3\}$, generalizing the standard ground behavior (where $Vrel(t) = var(t)$: and the abstract model is $\{append(x_1, x_2, x_3) :- x_3 \leftrightarrow x_1 \wedge x_2\}$) vs. size-rigidity behavior: "the second argument list-size can change iff the third argument does". □

5 Machine-level Traces

In this section, we consider an example non-standard semantics for constraint logic programs, that of machine-level traces (for a discussion of similar non-standard semantics in a denotational context, see [Stoy 77]). Such a semantics is essential, for example, if we wish to reason formally about the correctness of a compiler (e.g. see [Hanus 88]) or the behavior of a debugger or profiler. In this section, we show how the semantics described in earlier sections may be instantiated to describe such low-level behaviors. Instead of constrained atoms where each atom is associated with a constraint, this semantics will associate each atom with a set of machine states (equivalently, instruction sequences) that may be generated on an execution of that atom.

The code generated by a compiler for a constraint language must necessarily depend on both the constraint system and the target machine under consideration. Suppose that each "primitive" constraint $op(t_1, \ldots, t_n)$ in the language under consideration corresponds to (an instance of) a (virtual) machine instruction $\mathsf{op}(t_1, \ldots, t_n)$.[2] For example, correspond-

[2] In an actual implementation, each such virtual machine instruction may, of course, "macro-expand" to a sequence of lower-level machine instructions.

ing to a constraint '$X = Y + 5$' in the language under consideration, we might have a virtual machine instruction '$\mathsf{eq}(X, Y + 5)$'. Each such machine instruction defines a transformation on machine states, representing the changes that are performed to the heap, stack, registers, *etc.* of the machine by the execution of that instruction (e.g., see [Hanus 88] for a discussion of the WAM along these lines). In other words, let $\mathcal{S}$ be the set of all possible states of the machine under consideration, then an instruction I denotes a function $I : \mathcal{S} \longrightarrow \mathcal{S} \cup \{\mathsf{fail}\}$, where fail denotes a state where execution has failed.

Given a set S, let S^∞ denote the set of finite and infinite sequences of S. Intuitively, with each execution we want to associate a set of finite and infinite sequences of machine states, that might be generated by an OR-parallel interpreter. Thus, we want the universe of our algebra to be $2^{\mathcal{S}^\infty}$, the set of sets of finite and infinite sequences of machine states. One subtlety, however, is that instructions may "fail" at runtime because some constraints may be unsatisfiable. To model this, it is necessary to handle failure explicitly, since "forward" execution cannot continue on failure. To deal with this, we define the notion of concatenation of sequences of machine states as follows: given any two sequences s_1 and s_2 of states in $\mathcal{S} \cup \{\mathsf{fail}\}$, their concatenation $s_1 \odot s_2$ is given by $s_1 \odot s_2$ = **if** s_1 **contains** fail **then** s_1 **else** $concat(s_1, s_2)$, where $concat(s_1, s_2)$ denotes the "usual" notion of concatenation of finite and countably infinite sequences. Thus, the cylindric closed semiring in this case is $(\mathcal{C}, \otimes, \oplus, 1, 0, \exists_\Delta, d_{t,t'})_{\Delta \subseteq V; t,t' \in \mathcal{T}}$ where: $\mathcal{C} = 2^{(\mathcal{S} \cup \{\mathsf{fail}\})^\infty}$ is the set of finite and infinite sequences of machine states; for any $S_1, S_2 \in \mathcal{C}$, $S_1 \otimes S_2 = \{s_1 \odot s_2 \mid s_1 \in S_1, s_2 \in S_2\}$; $\oplus = \cup$; $1 = \{\varepsilon\}$, where ε is the empty sequence; $0 = \emptyset$; $\exists_\Delta$ corresponds to the function that, given any machine state S, yields the machine state obtained by discarding all information about the variables in Δ; and for any $t, t' \in \mathcal{T}$, $d_{t,t'}$ corresponds to the function that, given any machine state S, yields the machine state resulting from constraining t and t' to be equal, and fail if this is not possible.

A simple variation on this semantics is one where failed execution sequences are discarded silently. To obtain such a semantics, it suffices to redefine the operation $\oplus$ as follows:
$S_1 \oplus S_2 = \{\, s \mid s \in S_1 \cup S_2 \ \wedge \text{fail is not in } s \,\}$.

6 Related Work

A related framework is considered in [Codognet and Filè 91] where an algebraic definition of constraint systems is given. Program analysis based on abstract interpretation techniques are considered, like *groundness analysis* and *definiteness analysis* for CLP programs. Only $\otimes$-composition is considered. The notion of "computation system" is introduced but it is neither formalized as a specific algebraic structure nor extended with the join-operator. In particular, because of the underlying semantics construction, mainly based on a generalization of the top-down SLD semantics, a loop-checker consisting in a "tabled"-interpreter is introduced. The use of tabled interpreters allows to keep separate the notion of abstraction from the finiteness required by any static analysis. As a consequence, static analysis can be performed by "running" the program in the standard CLP interpreter with tabulation. In our framework, no tabulation is considered. This makes the semantics construction more general. Finiteness is a specific property of the constraint system (expressed in terms of $\oplus$-chains), thus allowing to specify non-standard computations as standard CLP computations over an appropriate non-standard constraint system. Both the traditional top-down and bottom-up semantics can then be specified in the standard way thus allowing the definition of goal-independent static analysis as an abstract fixpoint computation, without loop-checking. If the constraint system is not Noetherian, a *widening/narrowing* technique [Cousot and Cousot 91] can be applied in the fixpoint computation to get a finite approximation of the T_P^A fixpoint.

In a related paper, Marriott and Søndergaard consider abstract interpretation of CLP. A meta-language is defined to specify, in a denotational style, the semantics of logic languages. Abstract interpretation is performed by abstracting such a semantics [Marriott and Søndergaard 90]. In this framework, both standard and non-standard semantics are viewed as an instance of the meta language specification.

Acknowledgment

The stimulating discussions with Maurizio Gabbrielli, Michael Maher and Nino Salibra are gratefully acknowledged.

References

[Aho *et al.* 74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley Publishing Company, 1974.

[Barbuti *et al.* 92] R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog Control. In *Proc. Nineteenth Annual ACM Symp. on Principles of Programming Languages*, pages 95–104, 1992.

[Barbuti *et al.* 91] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. Technical Report TR 12/91, Dipartimento di Informatica, Università di Pisa, 1991. To appear in *ACM Transactions on Programming Languages and Systems.*

[Barbuti and Martelli 83] R. Barbuti and A. Martelli. A Structured Approach to Semantics Correctness. *"Science of Computer Programming"*, 3:279–311, 1983.

[Bossi *et al.* 90] A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91*, volume 494 of *Lecture Notes in Computer Science*, pages 153–180. Springer-Verlag, Berlin, 1991.

[Cirulis 88] J. Cirulis. An Algebraization of First Order Logic with Terms. *Colloquia Mathematica Societatis Jànos Bolyai*, 54, 1991.

[Codognet and Filè 91] P. Codognet and G. Filè. Computations, Abstractions and Constraints. Technical Report 13, Dipartimento di Matematica Pura e Applicata, Università di Padova, Italy, 1991.

[Cortesi *et al.* 91] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formulas as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press,1991.

[Cousot and Cousot 77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[Cousot and Halbwachs 78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proc. Fifth ACM Symp. Principles of Programming Languages*, pages 84–96, 1978.

[Cousot and Cousot 79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.

[Cousot and Cousot 91] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. *Preliminary draft*, ICLP'91 Pre-conference workshop, Paris, 1991.

[Debray and Ramakrishnan 91] S. Debray and R. Ramakrishnan. Generalized Horn Clause Programs. Technical report, Dept. of Computer Science, The University of Arizona, 1991.

[Falaschi *et al.* 89] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[Gabbrielli and Levi 91] M. Gabbrielli and G. Levi. Modeling Answer Constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 238–252. The MIT Press, Cambridge, Mass., 1991.

[Hanus 88] M. Hanus. Formal Specification of a Prolog Compiler. In P. Deransart, B. Lorho, and J. Maluszyński, editors, *Proc. International Workshop on Programming Languages Implementation and Logic Programming*, volume 348 of *Lecture Notes in Computer Science*, pages 273–282. Springer-Verlag, Berlin, 1988.

[Henkin *et al.* 85] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras. Part I and II.* North-Holland, Amsterdam, 1971. (Second edition 1985)

[Jaffar and Lassez 87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

[Jørgensen *et al.* 91] N. Jørgensen, K. Marriott, and S. Michaylov. Some Global Compile-Time Optimizations for CLP($\Re$). Technical report, Department of Computer Science, Monash University, 1991.

[Kemp and Ringwood 90] R. Kemp and G. Ringwood. An Algebraic Framework for the Abstract Interpretation of Logic Programs. In S. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 506–520. The MIT Press, Cambridge, Mass., 1990.

[Marriott and Søndergaard 90] K. Marriott and H. Søndergaard. Analysis of Constraint Logic Programs. In S. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 531–547. The MIT Press, Cambridge, Mass., 1990.

[Saraswat *et al.* 91] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundation of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, 1991.

[Scott 82] D. Scott. Domains for Denotational Semantics. In *Proc. ICALP*, volume 140 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1982.

[Stoy 77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Verschaetse and De Schreye 91] K. Verschaetse and D. De Schreye. Automatic Derivation of Linear Size Relations. Technical report, Dept. of Computer Science, K.U. Leuven, 1991.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Extended Well-Founded Semantics for Paraconsistent Logic Programs

Chiaki Sakama

ASTEM Research Institute
17 Chudoji Minami-machi, Shimogyo, Kyoto 600 Japan
sakama@astem.or.jp

Abstract

This paper presents a declarative semantics of logic programs which possibly contain inconsistent information. We introduce a multi-valued interpretation of logic programs and present the extended well-founded semantics for paraconsistent logic programs. In this setting, a meaningful information is still available in the presence of an inconsistent information in a program and any fact which is affected by an inconsistent information is distinguished from the others. The well-founded semantics is also extended to disjunctive paraconsistent logic programs.

1 Introduction

Recent studies have greatly enriched an expressive power of logic programming as a tool for knowledge representation. Handling classical negation as well as negation by failure in a program is one of such extension. An *extended logic program*, which is introduced by Gelfond and Lifschitz [GL90], distinguishes two types of negation and enables us to deal with explicit negation as well as default negation in a program. An extended logic program is, however, possibly inconsistent in general, since it contains negative heads as well as positive ones in program clauses. Practically, an inconsistency is likely to happen when we build a large scale of knowledge base in such a logic program. A knowledge base may contain local inconsistencies that would make a program contradictory and yet it may have a natural intended global meaning. However, in an inconsistent program, the answer set semantics proposed in [GL90] implies every formula from the program. This is also the case for most of the traditional logics in which a piece of inconsistent information might spoil the rest of the whole knowledge base.

To avoid such a situation, the so-called *paraconsistent logics* have been developed which are not destructive in the presence of an inconsistent information [Co74]. From the point of view of logic programming, a possibly inconsistent logic program is called a *paraconsistent logic program*. Blair and Subrahmanian [BS87] have firstly developed a fixpoint semantics of such programs by using Belnap's four-valued logic [Be75]. Recent studies such as [KL89, Fi89, Fi91] have also developed a logic for possibly inconsistent logic programs and provided a framework for reasoning with inconsistency. However, from the point of view of logic programming, negation in these approaches is classical in its nature and the treatment of default negation as well as classical one in paraconsistent logic programming is still left open.

In this paper, we present a framework for paraconsistent logic programming in which classical and default negation are distinguished. The rest of this paper is organized as follows. In section 2, we first present an application of Ginsberg's lattice-valued logic to logic programming and provide a declarative semantics of paraconsistent logic programs by extending the well-founded semantics of general logic programs. Then we show how the extended well-founded semantics isolates an inconsistent information and distinguishes meaningful information from others in a program. In section 3, the well-founded semantics is also extended to paraconsistent disjunctive logic programs.

2 Well-Founded Semantics for Paraconsistent Logic Programs

2.1 Multi-valued Logic

To present the semantics of possibly inconsistent logic programs, multi-valued logics are often used instead of the traditional two-valued logic. Among them, Belnap's four-valued logic [Be75] is well-known and several researchers have employed this logic to give the semantics of paraconsistent logic programs [BS87, KL89, Fi89, Fi91]. In Belnap's logic, truth values consist of $\{\mathbf{t}, \mathbf{f}, \top, \bot\}$ in which each element respectively denotes *true, false, contradictory*, and *undefined*. Each element makes a complete lattice under a partial ordering defined over these truth values (figure 1).

To represent nonmonotonic aspect of logic programming, however, we need extra truth values which represent default assumption. Such a logic is firstly introduced by Ginsberg [Gi86] in the context of bilattice for

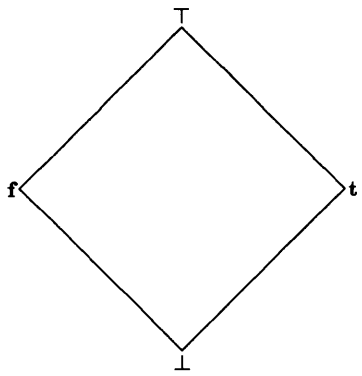


Figure 1. Four-valued logic

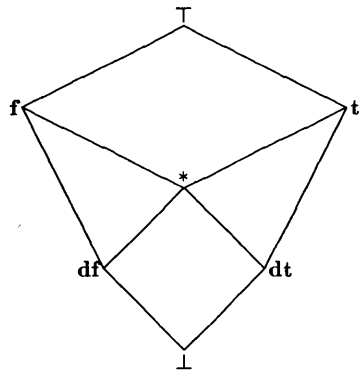


Figure 2. The logic **VII**

default logic. We use this logic to give the semantics of paraconsistent logic programs.[1]

The set $\mathbf{VII} = \{\mathbf{t}, \mathbf{f}, \mathbf{dt}, \mathbf{df}, *, \top, \bot\}$ is the space of truth values in our *seven-valued* logic. Here, additional elements **dt**, **df**, and *, are read as *true by default*, *false by default*, and *don't-care by default*, respectively. In **VII**, each element makes a complete lattice under the ordering $\preceq$ such that: $\forall \mathbf{x} \in \mathbf{VII}, \mathbf{x} \preceq \mathbf{x}$ and $\bot \preceq \mathbf{x} \preceq \top$; and for $\mathbf{x} \in \{\mathbf{t}, \mathbf{f}\}, \mathbf{dx} \preceq * \preceq \mathbf{x}$ (figure 2).

A *program* is a (possibly infinite) set of clauses of the form:

$$A \leftarrow B_1 \wedge ... \wedge B_m \wedge notC_1 \wedge ... \wedge notC_n$$

where $m, n \geq 0$, each A, $B_i (1 \leq i \leq m)$ and $C_j (1 \leq j \leq n)$ are literals and all the variables are assumed to be universally quantified at the front of the clause. In a program, two types of negation are distinguished; hereafter, $\neg$ denotes a monotonic classical negation, while *not* denotes a nonmonotonic default negation. A *ground clause* (resp. *program*) is a clause (resp. program) in which every variable is instantiated by the elements of the Herbrand universe of a program. Also, such an instantiation is called *Herbrand instantiation* of a clause (resp. program).

An *interpretation* I of a program is a function such that $I : H_B \rightarrow \mathbf{VII}$ where H_B is the Herbrand base of the program. (Throughout of this paper, H_B denotes the Herbrand base of a program.)

A *formula* is defined as usual; (i) any literal L or $\neg L$ is a formula, (ii) for any literal L, $notL$ and $not\neg L$ are formulas, and (iii) for any formula F and G, $\forall F$, $\exists F$, $F \vee G$, $F \wedge G$ and $F \leftarrow G$ are all formulas. A formula is *closed* if it contains no free variable. Satisfaction of a formula is also defined as follows.

[1][KL89] has also suggested the extensibility of their logic for handling defaults by using Ginsberg's lattice-valued logic.

Definition 2.1 Let P be a program and I be its interpretation. Suppose $I \models F$ denotes that I *satisfies* a formula F, then:

1. For any atom $A \in H_B$,
   (a) $I \models A$ if $\mathbf{t} \preceq I(A)$,
   (b) $I \models \neg A$ if $\mathbf{f} \preceq I(A)$,
   (c) $I \models notA$ if $\mathbf{df} \preceq I(A) \preceq *$,
   (d) $I \models not\neg A$ if $\mathbf{dt} \preceq I(A) \preceq *$.
2. For any closed formula $\exists F$ (resp. $\forall F$), $I \models \exists F$ (resp. $I \models \forall F$) if $I \models F'$ for some (resp. every) Herbrand instantiation F' of F.
3. For closed formulas F and G,
   (a) $I \models F \vee G$ if $I \models F$ or $I \models G$,
   (b) $I \models F \wedge G$ if $I \models F$ and $I \models G$,
   (c) $I \models F \leftarrow G$ if $I \models F$ or $I \not\models G$. □

The ordering $\preceq$ on truth values is also defined between interpretations. For interpretations I_1 and I_2, $I_1 \preceq I_2$ iff $\forall A \in H_B, I_1(A) \preceq I_2(A)$. An interpretation I is called *minimal*, if there is no interpretation J such that $J \neq I$ and $J \preceq I$. An interpretation I is also called *least*, if $I \preceq J$ for every interpretation J.

An interpretation I is called a *model* of a program if every clause in a program is satisfied in I. Note that in our logic, the notion of *model* is also defined for an inconsistent set of formulas. For example, a program $\{p, \neg p\}$ has a model I such that $I(p) = \top$. Especially, an interpretation I of a program is called *consistent* if for every atom A in H_B, $I(A) \neq \top$. A program is called *consistent* if it has a consistent model.

2.2 Extended Well-Founded Semantics

The *well-founded semantics* is known as one of the most powerful semantics which is defined for every general logic program [VRS88, Pr89]. The well-founded semantics has also extended to programs with classical negation in [Pr90], however, it is not well-defined for inconsistent programs in which inconsistent models are all thrown away. In this section, we reformulate the well-founded semantics for possibly inconsistent logic programs.

To compute the well-founded model, we first present an interpretation of a program by a pair of sets of ground literals.

Definition 2.2 For a program P, a pair of sets of ground literals $I =< \sigma; \delta >$ presents an interpretation of P in which each literal in I is interpreted as follows:

For a positive literal L,

(i) if L (resp. $\neg L$) is in σ, L is *true* (resp. *false*) in I;

(ii) else if L (resp. $\neg L$) is in δ, L is *false by default* (resp. *true by default*) in I;

(iii) otherwise, neither L nor $\neg L$ is in σ nor δ, L is *undefined.*

Especially, if both L and $\neg L$ are in σ (resp. δ), L is *contradictory* (resp. *don't-care by default*) in I. □

Intuitively, σ presents proven facts while δ presents default facts, and an interpretation of a fact is defined by the *least upper bound* of its truth values in the pair.

Now we extend the constructive definition of the well-founded semantics for general logic programs [Pr89] to paraconsistent logic programs.

Definition 2.3 Let P be a program and $I =< \sigma; \delta >$ be an interpretation of P. For sets T and F of ground literals, the mapping Φ_I and Ψ_I are defined as follows:

$\Phi_I(T) = \{A \mid$ there is a ground clause $A \leftarrow B_1 \wedge ... \wedge B_m \wedge notC_1 \wedge ... \wedge notC_n$ from P s.t. $\forall B_i\ (1 \leq i \leq m)$ $B_i \in \sigma \cup T$ and $\forall C_j\ (1 \leq j \leq n)\ C_j \in \delta\}$,

$\Psi_I(F) = \{A \mid$ for every ground clause $A \leftarrow B_1 \wedge ... \wedge B_m \wedge notC_1 \wedge ... \wedge notC_n$ from P, either $\exists B_i\ (1 \leq i \leq m)$ s.t. $B_i \in \delta \cup F$ or $\exists C_j\ (1 \leq j \leq n)$ s.t. $C_j \in \sigma\}$. □

Definition 2.4 Let I be an interpretation. Then,

$T_I \uparrow 0 = \emptyset$ and $F_I \downarrow 0 = H_B \cup \neg H_B$ (where $\neg H_B = \{\neg A \mid A \in H_B\}$);

$T_I \uparrow n+1 = \Phi_I(T_I \uparrow n)$ and $F_I \downarrow n+1 = \Psi_I(F_I \downarrow n)$;

$T_I = \bigcup_{n<\omega} T_I \uparrow n$ and $F_I = \bigcap_{n<\omega} F_I \downarrow n$. □

As in [Pr89], T_I and F_I are the least fixpoints of the monotonic operators Φ_I and Ψ_I, respectively.

Definition 2.5 For every interpretation I, an operator Θ is defined by:

$\Theta(I) = I \cup < T_I; F_I >$;

$I \uparrow 0 =< \emptyset; \emptyset >$;

$I \uparrow n+1 = \Theta(I \uparrow n)$;

$M_P = \bigcup_{n<\omega} I \uparrow n$. □

Lemma 2.1 M_P is the least fixpoint of the monotonic operator Θ and also a model of P. □

By definition, M_P is uniquely defined for *every* paraconsistent logic program. We call such an M_P the *extended well-founded model* of a program and the meaning of a program represented by such a model is called the *extended well-founded semantics* of a program.

Note that the original fixpoint definition of the well-founded semantics in [Pr89] is three-valued and defined for general logic programs, while our extended well-founded semantics is seven-valued and defined for extended logic programs. Compared with the three-valued well-founded semantics, the extended well-founded semantics handles positive and negative literals symmetrically during the computation of the fixpoint. Further, the extended well-founded model is the least fixpoint of a program under the ordering $\preceq$, while the three-valued well-founded model is the least fixpoint with respect to the ordering $\mathbf{f} < \perp < \mathbf{t}$, which is basically different from $\preceq$.[2]

Example 2.1 (barber's paradox) Consider the following program:

$$shave(b, X) \leftarrow not\ shave(X, X)$$

Then $shave(b, b)$ is undefined under the three-valued well-founded semantics, while $M_P =< \emptyset; \{\neg shave(b, b)\} >$ then $shave(b, b)$ is true by default under the extended well-founded semantics. In another words, the extended well-founded semantics assumes the fact 'the barber shaves himself' without conflicting the sentence in the program. □

Also it should be noted that the extended well-founded model is the least fixpoint of a program, but *not* necessarily the least model of the program in general.

Example 2.2 Let $P = \{\neg p \leftarrow not\ p,\ \neg q \leftarrow \neg p,\ q \leftarrow\}$. Then $M_P =< \{\neg p, q, \neg q\}; \{p\} >$ and the truth value of each predicate is $\{p \rightarrow \mathbf{f}, q \rightarrow \top\}$. While, the least model assigns truth values such as $\{p \rightarrow \perp, q \rightarrow \mathbf{t}\}$. □

In fact, the above least model is not the fixpoint of the program. In this sense, our extended well-founded semantics is different from the least fixpoint model semantics of [BS87] (even for a program without nonmonotonic negation). The difference is due to the fact that in their least fixpoint model semantics each fact which cannot be proved in a program is assumed to be undefined, while it possibly has a default value under the extended well-

[2] This point is also remarked in [Pr89, Pr90]. In terms of the *bilattice* valued logic [Gi86, Fi91], the ordering $<$ is called a *truth ordering*, while the ordering $\preceq$ is called a *knowledge ordering.*

founded semantics. The above example also suggests the fact that for a consistent program P, M_P is not always consistent.

The extended well-founded semantics is also different from Fitting's bilattice-valued semantics [Fi89, Fi91].

Example 2.3 Let $P = \{p \leftarrow q,\ p \leftarrow \neg q,\ q \leftarrow\}$. Then, as is pointed out in [Su90], p is unexpectedly contradictory under Fitting's semantics, while $M_P = <\{p, q\}; \{\neg p, \neg q\}>$ then both p and q are true under the extended well-founded semantics. □

Now we examine the behavior of the extended well-founded semantics more carefully in the presence of an inconsistent information.

Example 2.4 Let P be the following program:

$innocent \leftarrow \neg guilty$

$\neg guilty \leftarrow charged \wedge not\ guilty$

$charged \leftarrow$

Then M_P is $< \{charged, innocent, \neg guilty\}; \{guilty, \neg innocent, \neg charged\} >$. Then the truth values of *charged* and *innocent* are true, while *guilty* is false. □

In the above example, when we consider the program $P' = P \cup \{\neg innocent \leftarrow\}$, the truth value of *innocent* turns contradictory, while truth values of *charged* and *guilty* are unchanged. That is, a meaningful information is still available from the inconsistent program.

On the other hand, when we consider the program $P'' = P \cup \{\neg charged \leftarrow,\ man \leftarrow\}$, the truth value of *charged* is now contradictory, while *man*, *innocent* are true and *guilty* is false. Carefully observing this result, however, the truth of *innocent* is now less credible than the truth of *man*, since *innocent* is derived from the fact $\neg guilty$ which is now supported by the inconsistent fact *charged* in the program.

Such a situation also happens in Blair and Subrahmanian's fixpoint semantics [BS87], in which a truth fact is not distinguished even if it is supported by an inconsistent fact in a program. In the next section, we refine the extended well-founded semantics to distinguish such suspicious truth facts from others.

2.3 Reasoning with Inconsistency

When a program contains an inconsistent information, it is important to detect a fact affected by such an information and distinguish it from other meaningful information in a program. In this section, we present such skeptical reasoning under the extended well-founded semantics.

First we introduce one additional notation. For a program P and each literal L from H_B, L^Γ is called a *suffixed literal* where Γ is a collection of sets of ground literals (possibly preceded by *not*). Informally speaking, each element in Γ presents a set of facts which are used to derive L in P (it is defined more precisely below). An interpretation of such a suffixed literal L^Γ is supposed to be the same with the interpretation of L.

Definition 2.6 Let P be a program and $I = <\sigma; \delta>$ be an interpretation in which σ (resp. δ) is a set of suffixed literals (resp. a set of ground literals). For a set T (resp. F) of suffixed literals (resp. ground literals), the mapping Φ^s_I and Ψ^s_I are defined as follows:

$\Phi^s_I(T) = \{A^\Gamma \mid$ there are k ground clauses $A \leftarrow B_{l1} \wedge ... \wedge B_{lm} \wedge notC_{l1} \wedge ... \wedge notC_{ln}$ $(1 \le l \le k)$ from P s.t. $\forall B_{li}$ $(1 \le i \le m)$ $B^{\Gamma_{li}}_{li} \in \sigma \cup T$ and $\forall C_{lj}$ $(1 \le j \le n)$ $C_{lj} \in \delta$ and $\Gamma = \bigcup_l \{\{B_{l1}, .., B_{lm}, notC_{l1}, .., notC_{ln}\} \cup \gamma_{l1} \cup .. \cup \gamma_{lm} \mid \gamma_{li} \in \Gamma_{li}\}$.

$\Psi^s_I(F) = \{A \mid$ for every ground clause $A \leftarrow B_1 \wedge ... \wedge B_m \wedge notC_1 \wedge ... \wedge notC_n$ from P, either $\exists B_i$ $(1 \le i \le m)$ s.t. $B_i \in \delta \cup F$ or $\exists C_j$ $(1 \le j \le n)$ s.t. $C^{\Gamma_j}_j \in \sigma\}$. □

The least fixpoint M^s_P of a program is similarly defined by using the mapping Φ^s_I and Ψ^s_I instead of Φ_I and Ψ_I, respectively in the previous section. Clearly, M^s_P is also a model of P and we call such M^s_P the *suspicious well-founded model.*

Example 2.5 Let $P = \{p \leftarrow q \wedge not\ r,\ p \leftarrow \neg r,\ q \leftarrow s,\ \neg r \leftarrow,\ s \leftarrow\}$. Then, $M^s_P = <\{p^{\{\{q,s,not\ r\},\{\neg r\}\}}, q^{\{\{s\}\}}, \neg r^{\{\emptyset\}}, s^{\{\emptyset\}}\}; \{\neg p, \neg q, r, \neg s\}>$. □

Definition 2.7 Let P be a program and M^s_P be its suspicious well-founded model. For a suffixed literal L^Γ in M^s_P, if every set in Γ contains a literal L' or $\neg L'$ such that L' is contradictory in M^s_P, L is called *suspicious.* □

We consider a proven fact to be suspicious if every proof of the fact includes an inconsistent information. In another words, if there is at least one proof of a fact which contains no inconsistent information, we do not consider such a fact to be suspicious. A proven fact which is not suspicious is called *sure.*

Note that we do not consider any fact derived from true and false by default information to be suspicious, since such a don't-care information just presents that both positive and negative facts are failed to prove in a program and does not present any inconsistency by itself.

The following lemma presents that a fact which is derived using a suspicious fact is also suspicious.

Lemma 2.2 Let P be a program and L^Γ be a suffixed literal in M^s_P. If each set in Γ contains a suspicious fact, then the truth value of L is also suspicious.

Proof Suppose that each set γ in Γ contains a suspicious fact A. Then A has its own derivation histories Γ' such that each γ' in Γ' contains a literal which is contradictory in M^s_P. By definition, $\gamma' \subseteq \gamma$ then γ also contains

the contradictory literal. □

Now reasoning under the *suspicious well-founded semantics* is defined as follows.

Definition 2.8 Let P be a program and M_P^s be its suspicious well-founded model. Then, for each atom A such that A^Γ (resp. $\neg A^\Gamma$) is in M_P^s, A is called *true with suspect* (resp. *false with suspect*) if A (resp. $\neg A$) is suspicious and $\neg A$ (resp. A) is not sure in M_P^s.

On the contrary, if A (resp. $\neg A$) is suspicious but $\neg A$ (resp. A) is sure in M_P^s, then A is false (resp. true) in M_P^s without suspect. □

Especially, if A is both true and false with suspect, A is contradictory with suspect.

Example 2.6 Let P be the following program:

$innocent \leftarrow \neg guilty$
$\neg guilty \leftarrow charged \wedge not\ guilty$
$charged \leftarrow$
$\neg charged \leftarrow$
$man \leftarrow$

where M_P^s is $< \{charged^{\{\emptyset\}}, \neg charged^{\{\emptyset\}}, man^{\{\emptyset\}}, innocent^{\{\{\neg guilty, charged, not\ guilty\}\}}, \neg guilty^{\{\{charged, not\ guilty\}\}}\}; \{guilty, \neg innocent, \neg man\} >$. Then, *man* is true, *charged* is contradictory, while *innocent* and *guilty* are true with suspect and false with suspect, respectively. □

In the above example, if a new fact *guilty* is added to P, this fact now holds for sure then *guilty* becomes true without suspect.

2.4 Related Work

Alternative approaches to paraconsistent logic programming based upon the stable model semantics [GL88] are recently proposed in [PR91, GS92a]. These approaches have improved the result of [GL90] in the sense that stable models are well-defined in inconsistent programs. However, these semantics still inherit the problem of the stable model semantics and there exists a program which has no stable model and yet it contains a meaningful information. For example, a program $\{p \leftarrow,\ q \leftarrow not\ q\}$ has no stable model, while it has an (extended) well-founded model in which p is true. Wagner [Wa91] has also introduced a logic for possibly inconsistent logic programs with two kinds of negation. His logic is paraconsistent and not destructive in the presence of an inconsistent information, but it is still restricted and different from our lattice valued logic.

Several studies have also been done from the standpoint of contradiction removal in extended logic programs. Kowalski and Sadri [KS90] have extended the answer set semantics of [GL90] in an inconsistent program by giving higher priorities to negative conclusions in a program. This solution is rather ad-hoc and also easily simulated in our framework by giving higher priorities to negative facts in a program. Another approaches such as [PAA91] and [DR91] consider removing contradiction brought about by default assumptions. For instance, consider a program $\{p \leftarrow not\ q,\ \neg p \leftarrow r,\ r\}$. This program has an inconsistent well-founded model, however, it often seems legal to prefer the fact $\neg p$ to p, since p is derived by the default assumption *not* q, while its negative counterpart $\neg p$ is derived by the proven fact r. Then they present program transformations for taking back such a default assumption to generate a consistent well-founded model. In our framework, such a distinction is also achieved as follows. Consider a suspicious well-founded model of the program $< \{p^{\{\{not\ q\}\}}, \neg p^{\{\{r\}\}}, r^{\{\emptyset\}}\}; \{q, \neg q, \neg r\} >$ where a fact p has a default fact in its derivation history while $\neg p$ does not, then we can prefer the fact $\neg p$ as a more reliable one. These approaches [PAA91, DR91] further discuss contradiction removal in the context of belief revision or abductive framework, but from the point of view of paraconsistent logic programming, they provide no solution for an inconsistent program such as $\{p, \neg p, q\}$. Another approaches in this direction are [In91, GS92b] in which the meaning of an inconsistent program is assumed to be a collection of maximally consistent subsets of the program.

3 Extension to Disjunctive Programs

The semantics of logic programs is recently extended to disjunctive logic programs which contain incomplete information in a program. The well-founded semantics is also extended to disjunctive logic programs by several authors [Ro89, BLM90, Pr90]. In paraconsistent logic programming, [Su90] has also extended the fixpoint semantics of [BS87] to paraconsistent disjunctive logic programs. In this section, we present the extended well-founded semantics for paraconsistent disjunctive logic programs.

A *disjunctive program* is a (possibly infinite) set of the clauses of the form:

$$A_1 \vee ... \vee A_l \leftarrow B_1 \wedge ... \wedge B_m \wedge notC_1 \wedge ... \wedge notC_n$$

where $l > 0$, $m, n \geq 0$, each A_i, B_j and C_k are literals and all the variables are assumed to be universally quantified at the front of the clause. The notion of a ground clause (program) is also defined in the same way as in the previous section. Hereafter, we use the term *normal program* to distinguish a program which contains no disjunctive clause.

As in [Sa89], we consider the meaning of a disjunctive program by a set of its *split programs*.

Definition 3.1 Let P be a disjunctive program and

G be a ground clause from P of the form:

$A_1 \vee ... \vee A_l \leftarrow B_1 \wedge ... \wedge B_m \wedge notC_1 \wedge ... \wedge notC_n$ $(l \geq 2)$

Then G is *split* into $2^l - 1$ sets of clauses $G_1, .., G_{2^l-1}$ such that for each non-empty subset S_i of $\{A_1, .., A_l\}$;

$G_i = \{A_j \leftarrow B_1 \wedge ... \wedge B_m \wedge notC_1 \wedge ... \wedge notC_n \mid A_j \in S_i\}$.

A *split program* of P is a ground normal program which is obtained from P by replacing each disjunctive clause G with its split clauses G_i. □

Example 3.1 Let $P = \{p \vee \neg q \leftarrow not\ r,\ s \leftarrow p,\ s \leftarrow \neg q\}$. Then there are three split programs of P;

$P_1 = \{p \leftarrow not\ r,\ s \leftarrow p,\ s \leftarrow \neg q\}$,
$P_2 = \{\neg q \leftarrow not\ r,\ s \leftarrow p,\ s \leftarrow \neg q\}$,
$P_3 = \{p \leftarrow not\ r,\ \neg q \leftarrow not\ r,\ s \leftarrow p,\ s \leftarrow \neg q\}$.
□

Intuitively, each split program presents a possible world of the original program in which each disjunction is interpreted in either exclusive or inclusive way. The following lemma holds from the definition.

Lemma 3.1 Let P be a disjunctive program and P_i be its split program. If I is a model of P_i, I is also a model of P. □

The extended well-founded models of a disjunctive program are defined by those of its split programs.

Definition 3.2 Let P be a disjunctive program. Then M_P is called the *extended well-founded model* of P if M_P is the extended well-founded model of some split program of P. □

Clearly, the above definition reduces to the extended well-founded model of a normal program in the absence of disjunctive clauses in a program.

A disjunctive program has multiple extended well-founded models in general and each atom possibly has different truth value in each model. In classical two-valued logic programming, a ground atom is usually assumed to be true (resp. false) if it is true (resp. false) in every minimal model of a program. In our multi-valued setting, we define an interpretation of an atom under the extended well-founded semantics as follows.

Definition 3.3 Let P be a disjunctive program, $M_P^1, .., M_P^n$ be its extended well-founded models and $M_P^i(A)(i = 1, .., n)$ be the truth value of an atom A in M_P^i. Then an atom A in P has a truth value μ under the *extended well-founded semantics* if $M_P^1(A) = ... = M_P^n(A) = \mu$. □

Example 3.2 For the program P in example 3.1, there are three extended well-founded models such that $M_P^1 =< \{p, s\}; \{\neg p, q, \neg q, r, \neg r, \neg s\} >$, $M_P^2 =< \{\neg q, s\}; \{p, \neg p, q, r, \neg r, \neg s\} >$ and $M_P^3 =< \{p, \neg q, s\}; \{\neg p, q, r, \neg r, \neg s\} >$. Then s is true and r is don't-care by default in P under the extended well-founded semantics, while truth values of p and q are not uniquely determined. □

When a program has inconsistent models as well as consistent ones, however, it seems natural to prefer consistent models and consider truth values in such models.

Example 3.3 Let $P = \{p \leftarrow,\ \neg p \vee q \leftarrow\}$. Then the extended well-founded models of P are $M_P^1 =< \{p, \neg p\}; \{q, \neg q\} >$, $M_P^2 =< \{p, q\}; \{\neg p, \neg q\} >$ and $M_P^3 =< \{p, \neg p, q\}; \{\neg q\} >$ where only M_P^2 is consistent. □

In the above example, a rational reasoner seems to prefer the consistent model M_P^2 to M_P^1 and M_P^3, and interprets both p and q to be true. The extended well-founded semantics for such a reasoner is defined bellow.

Definition 3.4 Let P be a disjunctive program such that $M_P^1, .., M_P^n$ $(n \neq 0)$ are its *consistent* extended well-founded models. Then an atom A in P has a truth value μ under the *rational extended well-founded semantics* if $M_P^1(A) = ... = M_P^n(A) = \mu$. □

Lemma 3.2 Let P be a disjunctive program such that it has at least one consistent extended well-founded model. If an atom A has a truth value μ under the extended well-founded semantics, then A has also the truth value μ under the rational extended well-founded semantics, but not vice versa. □

The suspicious well-founded semantics presented in section 2.3 is also extensible to disjunctive programs in a similar way.

4 Concluding Remarks

In this paper, we have presented the extended well-founded semantics for paraconsistent logic programs. Under the extended well-founded semantics, a contradictory information is localized and a meaningful information is still available in an inconsistent program. Moreover, a suspicious fact which is affected by an inconsistent information can be distinguished from others by the skeptical well-founded reasoning. The extended well-founded semantics proposed in this paper is a natural extension of the three-valued well-founded semantics and it is well-defined for every possibly inconsistent extended logic program. Compared with other paraconsistent logics, it can treat both classical and default negation in a uniform way and also simply be extended to disjunctive paraconsistent logic programs.

This paper has centered on a declarative semantics

of paraconsistent logic programs, but a proof procedure of the extended well-founded semantics is achieved in a straightforward way as an extension of the SLS-procedure [Pr89]. That is, each fact which is true/false in a program have a successful SLS-derivation in a program, while a default fact in a program has a failed derivation. A fact which is inconsistent in a program has a successful derivation from its positive and negative goals. The proof procedure for the suspicious well-founded semantics is also achieved by checking consistency of each literal appearing in a successful derivation. These procedures are sound and complete with respect to the extended well-founded semantics and also computationally feasible.

Acknowledgments I would like to thank V. S. Subrahmanian and John Grant for useful correspondence on the subject of this paper.

References

[Be75] Belnap, N. D., A Useful Four-Valued Logic, in *Modern Uses of Multiple-Valued Logic*, J. M. Dunn and G. Epstein (eds.), Reidel Publishing, 8-37, 1975.

[BLM90] Baral, C., Lobo, J. and Minker, J., Generalized Disjunctive Well-Founded Semantics for Logic Programs, CS-TR-2436, Univ. of Maryland, 1990.

[BS87] Blair, H. A. and Subrahmanian, V. S., Paraconsistent Logic Programming, *Proc. Conf. on Foundations of Software Technology and Theoretical Computer Science (LNCS 287)*, 340-360, 1987.

[Co74] Costa, N. C. A. da, On the Theory of Inconsistent Formal Systems, *Notre Dame J. of Formal Logic 15*, 497-510, 1974.

[DR91] Dung, P. M. and Ruamviboonsuk, P., Well-Founded Reasoning with Classical Negation, *Proc. 1st Int. Workshop on Logic Programming and Nonmonotonic Reasoning*, 120-132, 1991.

[Fi89] Fitting, M., Negation as Refutation, *Proc. 4th Annual Symp. on Logic in Computer Science*, 63-69, 1989.

[Fi91] Fitting, M., Bilattices and the Semantics of Logic Programming, *J. of Logic Programming 11*, 91-116, 1991.

[Gi86] Ginsberg, M. L., Multivalued Logics, *Proc. of AAAI'86*, 243-247, 1986.

[GL88] Gelfond, M. and Lifschitz, V., The Stable Model Semantics for Logic Programming, *Proc. 5th Int. Conf. on Logic Programming*, 1070-1080, 1988.

[GL90] Gelfond, M. and Lifschitz, V., Logic Programs with Classical Negation, *Proc. 7th Int. Conf. on Logic Programming*, 579-597, 1990.

[GS92a] Grant, J. and Subrahmanian, V. S., Reasoning in Inconsistent Knowledge Bases, draft manuscript, 1992.

[GS92b] Grant, J. and Subrahmanian, V. S., The Optimistic and Cautious Semantics for Inconsistent Knowledge Bases, draft manuscript, 1992.

[In91] Inoue, K., Extended Logic Programs with Default Assumptions, *Proc. 8th Int. Conf. on Logic Programming*, 490-504, 1991.

[KL89] Kifer, M. and Lozinskii, E. L., RI: A Logic for Reasoning with Inconsistency, *Proc. 4th Annual Symp. on Logic in Computer Science*, 253-262, 1989.

[KS90] Kowalski, R. A. and Sadri, F., Logic Programs with Exception, *Proc. 7th Int. Conf. on Logic Programming*, 598-613, 1990.

[PAA91] Pereira, L. M., Alferes, J. J. and Aparicio, N., Contradiction Removal within Well-Founded Semantics, *Proc. 1st Int. Workshop on Logic Programming and Nonmonotonic Reasoning*, 105-119, 1991.

[Pr89] Przymusinski, T. C., Every Logic Program has a Natural Stratification and an Iterated Least Fixed Point Model, *Proc. 8th ACM Symp. on Principle of Database Systems*, 11-21, 1989.

[Pr90] Przymusinski, T. C., Extended Stable Semantics for Normal and Disjunctive Logic Programs, *Proc. 7th Int. Conf. on Logic Programming*, 459-477, 1990.

[PR91] Pimentel, S. G. and Rodi, W. L., Belief Revision and Paraconsistency in a Logic Programming Framework, *Proc. 1st Int. Workshop on Logic Programming and Nonmonotonic Reasoning*, 228-242, 1991.

[Ro89] Ross, K., The Well-Founded Semantics for Disjunctive Logic Programs, *Proc. 1st Int. Conf. on Deductive and Object Oriented Databases*, 352-369, 1989.

[Sa89] Sakama, C., Possible Model Semantics for Disjunctive Databases, *Proc. 1st Int. Conf. on Deductive and Object Oriented Databases*, 337-351, 1989.

[Su90] Subrahmanian, V. S., Paraconsistent Disjunctive Deductive Databases, *Proc. 20th Int. Symp. on Multiple-valued Logic*, 339-345, 1990.

[Su90] Subrahmanian, V. S., Y-Logic: A Framework for Reasoning about Chameleonic Programs with Inconsistent Completions, *Fundamenta Informaticae XIII*, 465-483, 1990.

[VRS88] Van Gelder, A., Ross, K. and Schlipf, J. S., Unfounded Sets and Well-Founded Semantics for General Logic Programs, *Proc. 7th ACM Symp. on Principle of Database Systems*, 221-230, 1988.

[Wa91] Wagner, G., A Database Needs Two kinds of Negation, *Proc. 3rd Symp. on Mathematical Fundamentals of Database and Knowledge Base Systems (LNCS 495)*, 357-371, 1991.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Formalizing Database Evolution in the Situation Calculus

Raymond Reiter
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4
and
The Canadian Institute for Advanced Research
email: reiter@ai.toronto.edu

Abstract

We continue our exploration of a theory of database updates (Reiter [21, 23]) based upon the situation calculus. The basic idea is to take seriously the fact that databases evolve in time, so that updatable relations should be endowed with an explicit state argument representing the current database state. Database transactions are treated as functions whose effect is to map the current database state into a successor state. The formalism is identical to that arising in the artificial intelligence planning literature and indeed, borrows shamelessly from those ideas.

Within this setting, we consider several topics, specifically:

1. A logic programming implementation of query evaluation.
2. The treatment of database views.
3. State constraints and the ramification problem.
4. The evaluation of historical queries.
5. An approach to indeterminate transactions.

1 Introduction

Elsewhere (Reiter [21, 23]), we have described how one may represent databases and their update transactions within the situation calculus (McCarthy [13]). The basic idea is to take seriously the fact that databases evolve in time, so that updatable relations should be endowed with an explicit state argument representing the current database state. Database transactions are treated as functions, and the effect of a transaction is to map the current database state into a successor state. The resulting formalism becomes identical to theories of planning in the AI literature (See, for example, (Reiter [18])).

Following a review of some of the requisite basic concepts and results, we consider several topics in this paper:

1. We sketch a logic programming implementation of the axioms defining a database under updates. While we give no proof of its correctness, we observe that under suitable assumptions, Clark completion axioms (Clark [3]) should yield such a proof.
2. We show how our approach can accommodate database views.
3. The so-called *ramification problem*, as defined in the AI planning literature, arises in specifying database updates. Roughly speaking, this is the problem of incorporating, in the axiom defining an update transaction, the indirect effects of the update as given by arbitrary state constraints. We discuss this problem in the database setting, and characterize its solution in terms of inductive entailments of the database.
4. An historical query is one that references previous database states. We sketch an approach to such queries which reduces their evaluation to evaluation in the initial database state, together with conventional list processing techniques on the list of those update transactions leading to the current database state.
5. The database axiomatization of this paper addresses only *determinate* transactions; roughly speaking, in the presence of complete information about the current database state, such a transaction determines a unique successor state. By appealing to some ideas of Haas ([7]) and Schubert ([24]), we indicate how to axiomatize indeterminate database transactions.

2 Preliminaries

This section reviews some of the basic concepts and results of (Reiter [23, 21, 19]) which provide the necessary background for presenting the material of this paper.

These include a motivating example, a precise specification of the axioms used to formalize update transactions and databases, an induction axiom suitable for proving properties of database states, and a discussion of query evaluation.

2.1 The Basic Approach: An Example

In (Reiter [23]), the idea of representing databases and their update transactions within the situation calculus was illustrated with an example education domain, which we repeat here.

Relations

The database involves the following three relations:

1. $enrolled(st, course, s)$: Student st is enrolled in course $course$ when the database is in state s.
2. $grade(st, course, grade, s)$: The grade of student st in course $course$ is $grade$ when the database is in state s.
3. $prerequ(pre, course)$: pre is a prerequisite course for course $course$. Notice that this relation is state independent, so is not expected to change during the evolution of the database.

Initial Database State

We assume given some first order specification of what is true of the initial state S_0 of the database. These will be arbitrary first order sentences, the only restriction being that those predicates which mention a state, mention only the initial state S_0. Examples of information which might be true in the initial state are:

$$enrolled(Sue, C100, S_0) \vee enrolled(Sue, C200, S_0),$$

$$(\exists c) enrolled(Bill, c, S_0),$$

$$(\forall p).prerequ(p, P300) \equiv p = P100 \vee p = M100,$$

$$(\forall p) \neg prerequ(p, C100),$$

$$(\forall c).enrolled(Bill, c, S_0) \equiv c = M100 \vee c = C100 \vee c = P200,$$

$$enrolled(Mary, C100, S_0),$$

$$\neg enrolled(John, M200, S_0), \ldots$$

$$grade(Sue, P300, 75, S_0), \quad grade(Bill, M200, 70, S_0), \ldots$$

$$prerequ(M200, M100), \quad \neg prerequ(M100, C100), \ldots$$

Database Transactions

Update transactions will be denoted by function symbols, and will be treated in exactly the same way as actions are in the situation calculus. For our example, there will be three transactions:

1. $register(st, course)$: Register student st in course $course$.
2. $change(st, course, grade)$: Change the current grade of student st in course $course$ to $grade$.
3. $drop(st, course)$: Student st drops course $course$.

Transaction Preconditions

Normally, transactions have preconditions which must be satisfied by the current database state before the transaction can be "executed". In our example, we shall require that a student can register in a course iff she has obtained a grade of at least 50 in all prerequisites for the course:

$$Poss(register(st, c), s) \equiv \{(\forall p).prerequ(p, c) \supset (\exists g).grade(st, p, g, s) \wedge g \geq 50\}.^1$$

It is possible to change a student's grade iff he has a grade which is different than the new grade:

$$Poss(change(st, c, g), s) \equiv (\exists g').grade(st, c, g', s) \wedge g' \neq g.$$

A student may drop a course iff the student is currently enrolled in that course:

$$Poss(drop(st, c), s) \equiv enrolled(st, c, s).$$

Update Specifications

These are the central axioms in our formalization of update transactions. They specify the effects of all transactions on all updatable database relations. As usual, all lower case roman letters are variables which are implicitly universally quantified. In particular, notice that these axioms quantify over transactions. In what follows, $do(a, s)$ denotes that database state resulting from performing the update transaction a when the database is in state s.

$$Poss(a, s) \supset [enrolled(st, c, do(a, s)) \equiv a = register(st, c) \vee enrolled(st, c, s) \wedge a \neq drop(st, c)],$$

$$Poss(a, s) \supset [grade(st, c, g, do(a, s)) \equiv a = change(st, c, g) \vee grade(st, c, g, s) \wedge (\forall g') a \neq change(st, c, g')].$$

2.2 An Axiomatization of Updates

The example education domain illustrates the general principles behind our approach to the specification of

[1] In the sequel, lower case roman letters will denote variables. All formulas are understood to be implicitly universally quantified with respect to their free variables whenever explicit quantifiers are not indicated.

database update transactions. In this section we precisely characterize a class of databases and updates of which the above example will be an instance.

Unique Names Axioms for Transactions

For distinct transaction names T and T',

$$T(\vec{x}) \neq T'(\vec{y}).$$

Identical transactions have identical arguments:

$$T(x_1, ..., x_n) = T(y_1, ..., y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n$$

for each function symbol T denoting a transaction.

Unique Names Axioms for States

$$(\forall a, s) S_0 \neq do(a, s),$$
$$(\forall a, s, a', s').do(a, s) = do(a', s') \supset a = a' \wedge s = s'.$$

Definition: The Simple Formulas

The *simple* formulas are defined to be the smallest set such that:

1. $F(\vec{t}, s)$ and $F(\vec{t}, S_0)$ are simple whenever F is an updatable database relation, the $\vec{t}$ are terms, and s is a variable of sort *state*. [2]
2. Any equality atom is simple.
3. Any other atom with predicate symbol other than *Poss* is simple.
4. If S_1 and S_2 are simple, so are $\neg S_1$, $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \supset S_2$, $S_1 \equiv S_2$.
5. If S is simple, so are $(\exists x)S$ and $(\forall x)S$ whenever x is an individual variable not of sort *state*.

In short, the simple formulas are those first order formulas whose updatable database relations do not mention the function symbol *do*, and which do not quantify over variables of sort *state*.

Definition: Transaction Precondition Axiom

A transaction precondition axiom is a formula of the form

$$(\forall \vec{x}, s).Poss(T(x_1, \cdots, x_n), s) \equiv \Pi_T,$$

where T is an n-ary transaction function, and Π_T is a simple formula whose free variables are among $x_1, \cdots, x_n, s$.

Definition: Successor State Axiom

A successor state axiom for an $(n+1)$-ary updatable database relation F is a sentence of the form

$$(\forall a, s).Poss(a, s) \supset$$
$$(\forall x_1, \ldots, x_n).F(x_1, \ldots, x_n, do(a, s)) \equiv \Phi_F$$

where, for notational convenience, we assume that F's last argument is of sort *state*, and where Φ_F is a simple formula, all of whose free variables are among $a, s, x_1, \ldots, x_n$.

2.3 An Induction Axiom

There is a close analogy between the situation calculus and the theory of the natural numbers; simply identify S_0 with the natural number 0, and $do(Add1, s)$ with the successor of the natural number s. In effect, an axiomatization in the situation calculus is a theory in which each "natural number" s has arbitrarily many successors.[3] Just as an induction axiom is necessary to prove anything interesting about the natural numbers, so also is induction required to prove general properties of states. This section is devoted to formulating an induction axiom suitable for this task.

We begin by defining an ordering relation $<$ on states. The intended interpretation of $s < s'$ is that state s' is reachable from state s by some sequence of transactions, each action of which is possible in that state resulting from executing the transactions preceeding it in the sequence. Hence, $<$ should be the smallest binary relation on states such that:

1. $\sigma < do(a, \sigma)$ whenever transaction a is possible in state σ, and
2. $\sigma < do(a, \sigma')$ whenever transaction a is possible in state σ' and $\sigma < \sigma'$.

This can be achieved with a second order sentence, as follows:

Definitions: $s < s'$, $s \leq s'$

$$\begin{aligned}(\forall s, s').s < s' \equiv \\ (\forall P).\{[(\forall a, s_1).Poss(a, s_1) \supset P(s_1, do(a, s_1))] \wedge \\ [(\forall a, s_1, s_2).Poss(a, s_2) \wedge P(s_1, s_2) \supset \\ P(s_1, do(a, s_2))]\} \\ \supset P(s, s').\end{aligned} \tag{1}$$

$$(\forall s, s')s \leq s' \equiv s < s' \vee s = s'. \tag{2}$$

Reiter [20] shows how these axioms entail the following induction axiom suitable for proving properties of states s when $S_0 \leq s$:

$$\begin{aligned}(\forall W).\{W(S_0) \wedge \\ [(\forall a, s).Poss(a, s) \wedge S_0 \leq s \wedge W(s) \supset W(do(a, s))]\} \\ \supset (\forall s).S_0 \leq s \supset W(s).\end{aligned} \tag{3}$$

This is our analogue of the standard second order induction axiom for Peano arithmetic.

[2] For notational convenience, we assume that the last argument of an updatable database relation is always the (only) argument of sort *state*.

[3] There could even be infinitely many successors whenever an action is parameterized by a real number, as for example *move(block, location)*.

Reiter [23, 20] provides an approach to database integrity constraints in which the concept of a database satisfying its constraints is defined in terms of inductive entailment from the database, using this and other axioms of induction for the situation calculus. In this paper, we shall find other uses for induction in connection with database view definitions (Section 4), the so-called *ramification problem* (Section 5), and historical queries (Section 6).

2.4 Databases Defined

In the sequel, unless otherwise indicated, we shall only consider background database axiomatizations $\mathcal{D}$ of the form:

$$\mathcal{D} = \textit{less-axioms} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{tp} \cup \mathcal{D}_{uns} \cup \mathcal{D}_{unt} \cup \mathcal{D}_{S_0}$$

where

- *less-axioms* are the axioms (1), (2) for $<$ and $\leq$.
- $\mathcal{D}_{ss}$ is a set of successor state axioms, one for each updatable database relation.
- $\mathcal{D}_{tp}$ is a set of transaction precondition axioms, one for each database transaction.
- $\mathcal{D}_{uns}$ is the set of unique names axioms for states.
- $\mathcal{D}_{unt}$ is the set of unique names axioms for transactions.
- $\mathcal{D}_{S_0}$ is a set of first order sentences with the property that S_0 is the only term of sort *state* mentioned by the database updatable relations of a sentence of $\mathcal{D}_{S_0}$. See Section 2.1 for an example $\mathcal{D}_{S_0}$. Thus, no updatable database relation of a formula of $\mathcal{D}_{S_0}$ mentions a variable of sort *state* or the function symbol *do*. $\mathcal{D}_{S_0}$ will play the role of the initial database (i.e. the one we start off with, before any transactions have been "executed").

2.5 Querying a Database

Notice that in the above account of database evolution, all updates are *virtual*; the database is never physically changed. To query the database resulting from some sequence of transactions, it is necessary to refer to this sequence in the query. For example, to determine if John is enrolled in any courses after the transaction sequence

$$drop(John, C100), register(Mary, C100)$$

has been 'executed', we must determine whether

$$Database \models (\exists c).enrolled(John, c, do(register(Mary, C100), do(drop(John, C100), S_0))).$$

Querying an evolving database is precisely the *temporal projection problem* in AI planning [8].[4]

Definition: A Regression Operator $\mathcal{R}$

Let W be first order formula. Then $\mathcal{R}[W]$ is that formula obtained from W by replacing each atom $F(\vec{t}, do(\alpha, \sigma))$ mentioned by W by $\Phi_F(\vec{t}, \alpha, \sigma)$ where F's successor state axiom is

$$(\forall a, s).Poss(a, s) \supset (\forall \vec{x}).F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s).$$

All other atoms of W not of this form remain the same.

The use of the regression operator $\mathcal{R}$ is a classical plan synthesis technique (Waldinger [25]). See also (Pednault [16, 17]). Regression corresponds to the operation of *unfolding* in logic programming. For the class of databases of this paper, Reiter [23, 19] provides a sound and complete query evaluator based on regression. In this paper, we shall have a different use for regression, in connection with defining database views (Section 4).

3 Updates in the Logic Programming Context

It seems that our approach to database updates can be implemented in a fairly straightforward way as a logic program, thereby directly complementing the logic programming perspective on databases (Minker [15]). For example, the axiomatization of the education example of Section 2.1 has the following representation as clauses:

Successor State Axiom Translation:

$enrolled(st, c, do(register(st, c), s))$
$\quad \leftarrow Poss(register(st, c), s).$
$enrolled(st, c, do(a, s))$
$\quad \leftarrow a \neq drop(st, c), enrolled(st, c, s), Poss(a, s).$
$grade(st, c, g, do(change(st, c, g), s))$
$\quad \leftarrow Poss(change(st, c, g), s).$
$grade(st, c, g, do(a, s))$
$\quad \leftarrow a \neq change(st, c, g'), grade(st, c, g, s), Poss(a, s).$[5]

Transaction Precondition Axiom Translation:

$Poss(register(st, c), s) \leftarrow not\ P(st, c, s).$
$Q(st, p, s) \leftarrow grade(st, p, g, s), g \geq 50.$[6]
$Poss(change(st, c, g), s) \leftarrow grade(st, c, g', s), g \neq g'.$
$Poss(drop(st, c), s) \leftarrow enrolled(st, c, s).$

[4] This property of our axiomatization makes the resulting approach quite different than Kowalski's situation calculus formalization of updates [9], in which each database update is accompanied by the addition of an atomic formula to the theory axiomatizing the database.

[5] This translation is problematic because it invokes negation-as-failure on a non-ground atom. The intention is that whenever a is bound to a term whose function symbol is *change*, the call should fail. This can be realized procedurally by retaining the clause sequence as shown, and simply deleting the inequality $a \neq change(st, c, g')$.

With a suitable clausal form for $\mathcal{D}_{S_0}$, it would then be possible to evaluate queries against updated databases, for example

$$\leftarrow enrolled(John, C200, do(register(Mary, C100), do(drop(John, C100), S_0))).$$

Presumably, all of this can be made to work under suitable conditions. The remaining problem is to characterize what these conditions are, and to prove correctness of such an implementation with respect to the logical specification of this paper. In this connection, notice that the equivalences in the successor state and transaction precondition axioms are reminiscent of Clark's [3] completion semantics for logic programs, and our unique names axioms for states and transactions provide part of the equality theory required for Clark's semantics (Lloyd [12], pp.79, 109).

4 Views

In our setting, a *view* is an updatable database relation $V(\vec{x}, s)$ defined in terms of so-called *base* predicates:

$$(\forall \vec{x}, s).V(\vec{x}, s) \equiv \mathcal{B}(\vec{x}, s), \tag{4}$$

where $\mathcal{B}$ is a simple formula with free variables among $\vec{x}$ and s, and which mentions only base predicates.[7] Unfortunately, sentences like (4) pose a problem for us because they are precluded by their syntax from the databases considered in this paper. However, we can accommodate nonrecursive views by representing them as follows:

$$(\forall \vec{x}).V(\vec{x}, S_0) \equiv \mathcal{B}(\vec{x}, S_0), \tag{5}$$

$$(\forall a, s).Poss(a, s) \supset (\forall \vec{x}).V(\vec{x}, do(a, s)) \equiv \mathcal{R}[\mathcal{B}(\vec{x}, do(a, s))].^{8} \tag{6}$$

Sentence (5) is a perfectly good candidate for inclusion in $\mathcal{D}_{S_0}$, while (6) has the syntactic form of a successor state axiom and hence may be included in $\mathcal{D}_{ss}$.

This representation of views requires some formal justification, which the following theorem provides:

Theorem 1 *Suppose $V(\vec{x}, s)$ is an updatable database relation, and that $\mathcal{B}(\vec{x}, s)$ is a simple formula which does not mention V and whose free variables are among $\vec{x}, s$. Suppose further that $\mathcal{D}_{ss}$ contains the successor state axiom (6) for V, and that $\mathcal{D}_{S_0}$ contains the initial state axiom (5). Then,*

$$\mathcal{D} \cup \{3\} \models (\forall s).S_0 \leq s \supset (\forall \vec{x}).V(\vec{x}, s) \equiv \mathcal{B}(\vec{x}, s).$$

Theorem 1 informs us that from the initial state and successor state axioms (5) and (6) we can inductively derive the view definition

$$(\forall s).S_0 \leq s \supset (\forall \vec{x}).V(\vec{x}, s) \equiv \mathcal{B}(\vec{x}, s).$$

This is not quite the same as the view definition (4) with which we began this discussion, but it is close enough. It guarantees that in any database state reachable from the initial state S_0, the view definition (4) will be true. We take this as sufficient justification for representing views within our framework by the axioms (5) and (6).

5 State Constraints and the Ramification Problem

Recall that our definition of a database (Section 2.4) does not admit state-dependent axioms, except those of $\mathcal{D}_{S_0}$ referring only to the initial state S_0. For example, we are prevented from including in a database a statement requiring that any student enrolled in $C200$ must also be enrolled in $C100$.

$$(\forall s, st).S_0 \leq s \wedge enrolled(st, C200, s) \supset enrolled(st, C100, s). \tag{7}$$

In a sense, such a state-dependent constraint should be redundant, since the successor state axioms, because they are equivalences, uniquely determine all future evolutions of the database given the initial database state S_0. The information conveyed in axioms like (7) must already be embodied in $\mathcal{D}_{S_0}$ together with the successor state and transaction precondition axioms. We have already seen hints of this observation. Reiter [20] proposes that dynamic integrity constraints should be viewed as inductive entailments of the database, and gives several examples of such derivations. Moreover, Theorem 1 shows that the view definition

$$(\forall s).S_0 \leq s \supset (\forall \vec{x}).V(\vec{x}, s) \equiv \mathcal{B}(\vec{x}, s).$$

is an inductive entailment of the database containing the initial state axiom (5) and the successor state axiom (6).

These considerations suggest that a *state constraint* can be broadly conceived as any sentence of the form

$$(\forall s_1, \ldots, s_n).S_0 \leq s_i \wedge s_i \leq s_j \wedge \cdots \supset W(s_1, \ldots, s_n),$$

and that a database is said to *satisfy* this constraint iff the database inductively entails it.[9]

[6] We have here invoked some of the program transformation rules of (Lloyd [12], p.113) to convert the non-clausal formula

$$\{(\forall p).prerequ(p, c) \supset (\exists g).grade(st, c, g, s) \wedge g \geq 50\} \supset Poss(register(st, c), s)$$

to a Prolog executable form. P and Q are new predicate symbols.

[7] We do not consider recursive views. Views may also be defined in terms of other, already defined views, but everything eventually "bottoms out" in base predicates, so we only consider this case.

[8] Notice that since we are not considering recursive views (i.e., $\mathcal{B}$ does not mention V), the formula $\mathcal{R}[\mathcal{B}(\vec{x}, do(a, s))]$ is well defined.

[9] See Section 2.3 for a brief discussion of inductively proving properties of states in the situation calculus.

The fact that state constraints like (7) must be inductive entailments of a database does not of itself dispense with the problem of how to deal with such constraints in defining the database. For in order that a state constraint be an inductive entailment, the successor state axioms must be so chosen as to guarantee this entailment. For example, the original successor state axiom for *enroll* (Section 2.1) was:

$$Poss(a,s) \supset \{enrolled(st,c,do(a,s)) \equiv \\ a = register(st,c) \vee \\ enrolled(st,c,s) \wedge a \neq drop(st,c)\}. \quad (8)$$

As one would expect, this does not inductively entail (7). To accommodate the state constraint (7), this successor state axiom must be changed to:

$$Poss(a,s) \supset \{enrolled(st,c,do(a,s)) \equiv \\ a = register(st,c) \wedge [c = C200 \supset enrolled(st,C100,s)] \\ \vee \\ enrolled(st,c,s) \wedge a \neq drop(st,c) \wedge \\ [c = C200 \supset a \neq drop(st,C100)]\}. \quad (9)$$

It is now simple to prove that, provided $\mathcal{D}_{S_0}$ contains the unique names axiom $C100 \neq C200$ and the initial instance of (7),

$$enrolled(st,C200,S_0) \supset enrolled(st,C100,S_0),$$

then (7) is an inductive entailment of the database.

The example illustrates the subtleties involved in getting the successor state axioms to reflect the intent of a state constraint. These difficulties are a manifestation of the so-called *ramification problem* in artificial intelligence planning domains (Finger [4]). Transactions might have ramifications, or *indirect effects*. For the example at hand, the transaction of registering a student in $C200$ has the direct effect of causing the student to be enrolled in $C200$, and the indirect effect of causing her to be enrolled in $C100$ (if she is not already enrolled in $C100$). The modification (9) of (8) was designed to capture this indirect effect. In our setting, the ramification problem is this: Given a static state constraint like (7), how can the indirect effects implicit in the state constraint be embodied in the successor state axioms so as to guarantee that the constraint will be an inductive entailment of the database? A variety of circumscriptive proposals for addressing the ramification problem have been proposed in the artificial intelligence literature, notably by Baker [1], Baker and Ginsberg [2], Ginsberg and Smith [5], Lifschitz [10] and Lin and Shoham [11]. Our formulation of the problem in terms of inductive entailments of the database seems to be new. For the databases of this paper, Fanghzen Lin[10] appears to have a solution to this problem.

[10]Personal communication.

6 Historical Queries

Using the relations $<$ and $\leq$ on states, as defined in Section 2.3, it is possible to pose *historical* queries to a database. First, some notation.

Notation: $do([a_1,\ldots,n],s)$

Let $a_1,\ldots,a_n$ be transactions. Define

$$do([\,],s) = s,$$

and for $n = 1,2,\ldots$

$$do([a_1,\ldots,a_n],s) = do(a_n, do([a_1,\ldots,a_{n-1},s)).$$

$do([a_1,\ldots,a_n],s)$ is a compact notation for the state term $do(a_n, do(a_{n-1},\ldots do(a_1,s)\ldots))$ which denotes that state resulting from performing the transaction a_1, followed by $a_2,\ldots$, followed by a_n, beginning in state s.

Now, suppose $\mathbf{T}$ is the transaction sequence leading to the current database state (i.e., the current database state is $do(\mathbf{T},S_0)$). The following asks whether the database was ever in a state in which John was simultaneously enrolled in both $C100$ and $M100$?

$$(\exists s).S_0 \leq s \wedge s \leq do(\mathbf{T},S_0) \wedge \\ enrolled(John,C100,s) \wedge enrolled(John,M100,s). \quad (10)$$

Has Sue always worked in department 13?

$$(\forall s).S_0 \leq s \wedge s \leq do(\mathbf{T},S_0) \supset emp(Sue,13,s). \quad (11)$$

The rest of this section sketches an approach to answering historical queries of this kind. The approach is of interest because it reduces the evaluation of such queries to evaluations in the initial database state, together with conventional list processing techniques on the list of those transactions leading to the current database state.

Begin by considering two new predicates, *last* and *mem-diff*. The intended interpretation of $last(s,a)$ is that the transaction a is the last transaction of the sequence s. For example,

$$last(do([drop(Mary,C100), register(John,C100)],S_0), \\ register(John,C100))$$

is true, while

$$last(do([drop(Mary,C100), drop(John,C100)],S_0), \\ register(John,C100))$$

is false, assuming unique names axioms for transactions. The following two axioms are sufficient for our purposes:

$$\neg(last(S_0,a).$$

$$last(do(a,s),a') \equiv a = a'.$$

The intended interpretation of $mem\text{-}diff(a, s, s')$ is that transaction a is a member of the "list difference" of s and s', where state s' is a "sublist" of s. For example,

$$mem\text{-}diff(drop(Mary, C100),\\ do([register(John, C100), drop(Bill, C100),\\ drop(Mary, C100), drop(John, M100)], S_0),\\ do([register(John, C100)], S_0))$$

is true, whereas

$$mem\text{-}diff(register(Mary, C100),\\ do([register(John, C100), drop(Bill, C100),\\ drop(Mary, C100), drop(John, M100)], S_0),\\ do([register(John, C100)], S_0))$$

is false (assuming unique names axioms for transactions). The following axioms will be sufficient for our needs:

$$\neg mem\text{-}diff(a, s, s).$$

$$s \leq s' \supset mem\text{-}diff(a, do(a, s'), s).$$

$$mem\text{-}diff(a, s, s') \supset mem\text{-}diff(a, do(a', s), s').$$

$$mem\text{-}diff(a, do(a', s), s') \wedge a \neq a' \supset mem\text{-}diff(a, s, s').$$

We begin by showing how to answer query (11). Suppose, for the sake of the example, that the successor state axiom for emp is:

$$Poss(a, s) \supset emp(p, d, do(a, s)) \equiv a = hire(p, d) \vee\\ emp(p, d, s) \wedge a \neq fire(p) \wedge a \neq quit(p).$$

Using this, and the sentences for *last* and *mem-diff* together with the induction axiom (3), it is possible to prove:

$$S_0 \leq s \supset emp(p, d, s) \equiv emp(p, d, S_0) \wedge\\ \neg mem\text{-}diff(fire(p), s, S_0) \wedge \neg mem\text{-}diff(quit(p), s, S_0) \vee\\ (\exists s').S_0 \leq s' \leq s \wedge last(s', hire(p, d)) \wedge\\ \neg mem\text{-}diff(fire(p), s, s') \wedge \neg mem\text{-}diff(quit(p), s, s').$$

Using this and the (reasonable) assumption that the transaction sequence $\mathbf{T}$ is legal,[11] it is simple to prove that the query (11) is equivalent to:

$$\left\{\begin{array}{l} emp(Sue, 13, S_0) \wedge\\ \neg mem\text{-}diff(fire(Sue), do(\mathbf{T}, S_0), S_0) \wedge\\ \neg mem\text{-}diff(quit(Sue), do(\mathbf{T}, S_0), S_0) \end{array}\right\}$$

$$\vee$$

$$\left\{\begin{array}{l} (\exists s').S_0 \leq s' \leq do(\mathbf{T}, S_0) \wedge\\ last(s', hire(Sue, 13)) \wedge\\ \neg mem\text{-}diff(fire(Sue), do(\mathbf{T}, S_0), s') \wedge\\ \neg mem\text{-}diff(quit(Sue), do(\mathbf{T}, S_0), s'). \end{array}\right\}$$

[11] Intuitively, $\mathbf{T}$ is legal iff each transaction of $\mathbf{T}$ satisfies its preconditions (see Section 2.1) in that state resulting from performing all the transactions preceeding it in the sequence, beginning with state S_0. See (Reiter [19]) for details, and a procedure for verifying the legality of a transaction sequence.

This form of the original query is of interest because it reduces query evaluation to evaluation in the initial database state, together with simple *list processing* on the list $\mathbf{T}$ of those transactions leading to the current database state. We can verify that Sue has always been employed in department 13 in one of two ways:

1. Verify that she was initially employed in department 13, and that neither $fire(Sue)$ nor $quit(Sue)$ are members of list $\mathbf{T}$.
2. Verify that $\mathbf{T}$ has a sublist ending with $hire(Sue, 13)$, and that neither $fire(Sue)$ nor $quit(Sue)$ are members of the list difference of $\mathbf{T}$ and this sublist.[12]

We now consider evaluating the first query (10) in the same list processing spirit. We shall assume that (8) is the successor state axiom for *enrolled*. Using the above sentences for *last* and *mem-diff*, together with (8) and the induction axiom (3), it is possible to prove:

$$S_0 \leq s \supset enrolled(st, c, s)) \equiv\\ enrolled(st, c, S_0) \wedge \neg mem\text{-}diff(drop(st, c), s, S_0) \vee\\ (\exists s').S_0 \leq s' \leq s \wedge last(s', register(st, c)) \wedge\\ \neg mem\text{-}diff(drop(st, c), s, s').$$

Then, on the assumption that the transaction sequence $\mathbf{T}$ is legal, it is simple to prove that the query (10) is equivalent to:

$$(\exists s).S_0 \leq s \leq do(\mathbf{T}, S_0) \wedge$$

$$[$$

$$\left\{\begin{array}{l} enrolled(John, C100, S_0) \wedge\\ enrolled(John, M100, S_0) \wedge\\ \neg mem\text{-}diff(drop(John, C100), s, S_0) \wedge\\ \neg mem\text{-}diff(drop(John, M100), s, S_0) \end{array}\right\}$$

$$\vee$$

$$\left\{\begin{array}{l} enrolled(John, C100, S_0) \wedge\\ \neg mem\text{-}diff(drop(John, C100), s, S_0) \wedge\\ (\exists s').S_0 \leq s' \leq s \wedge\\ last(s', register(John, M100)) \wedge\\ \neg mem\text{-}diff(drop(John, M100), s, s') \end{array}\right\}$$

$$\vee$$

$$\left\{\begin{array}{l} enrolled(John, M100, S_0) \wedge\\ (\exists s'').S_0 \leq s'' \leq s \wedge\\ last(s'', register(John, C100)) \wedge\\ \neg mem\text{-}diff(drop(John, C100), s, s'') \end{array}\right\}$$

$$\vee$$

$$\left\{\begin{array}{l} (\exists s', s'').S_0 \leq s' \leq s \wedge S_0 \leq s'' \leq s \wedge\\ last(s', register(John, M100)) \wedge\\ last(s'', register(John, C100)) \wedge\\ \neg mem\text{-}diff(drop(John, M100), s, s') \wedge\\ \neg mem\text{-}diff(drop(John, C100), s, s'') \end{array}\right\}$$

$$]$$

[12] The correctness of this simple-minded list processing procedure relies on some assumptions, notable suitable unique names axioms.

Despite its apparent complexity, this sentence also has a simple list processing reading; we can verify that *John* is simultaneously enrolled in $C100$ and $M100$ in some previous database state as follows. Find a sublist (loosely denoted by s) of $\mathbf{T}$ such that one of the following four conditions holds:

1. *John* was initially enrolled in both $C100$ and $M100$ and neither $drop(John, C100)$ nor $drop(John, M100)$ are members of list s.
2. *John* was initially enrolled in $C100$, $drop(John, C100)$ is not a member of list s, s has a sublist s' ending with $register(John, M100)$ and $drop(John, M100)$ is not a member of the list difference of s and s'.
3. *John* was initially enrolled in $M100$, $drop(John, M100)$ is not a member of list s, s has a sublist s' ending with $register(John, C100)$ and $drop(John, C100)$ is not a member of the list difference of s and s'.
4. There are two sublists s' and s'' of s, s' ends with $register(John, M100)$, s'' ends with $register(John, C100)$, $drop(John, M100)$ is not a member of the list difference of s and s', and $drop(John, C100)$ is not a member of the list difference of s and s''.

We can even pose queries about the future, for example, is it possible for the database ever to be in a state in which *John* is enrolled in both $C100$ and $C200$?

$$(\exists s).S_0 \leq s \wedge enrolled(John, C100, s) \wedge enrolled(John, C200, s).$$

Answering queries of this form is precisely the problem of plan synthesis in AI (Green [6]). For the class of databases of this paper, Reiter [22, 18] shows how regression provides a sound and complete evaluator for such queries.

7 Indeterminate Transactions

A limitation of our formalism is that it requires all transactions to be *determinate*, by which we mean that in the presence of complete information about the initial database state a transaction completely determines the resulting state.

One way to extend the theory to include indeterminate transactions is by appealing to a simple idea due to Haas [7], as elaborated by Schubert [24]. As an example, consider the indeterminate transaction *drop-a-student*(c), meaning that some student – we don't know whom – is to be dropped from course c. Notice that we cannot now have a successor state axiom of the form

$$Poss(a, s) \supset \{enrolled(st, c, do(a, s)) \equiv \Phi(st, c, a, s)\}.$$

To see why, consider the following instance of this axiom:

$$Poss(drop\text{-}a\text{-}student(C100), S_0) \supset \{enrolled(John, C100, do(drop\text{-}a\text{-}student(C100), S_0)) \equiv \Phi(John, C100, drop\text{-}a\text{-}student(C100), S_0)\}.$$

Suppose Σ_0 is a complete description of the initial database state, and suppose moreover, that

$$\Sigma_0 \models Poss(drop\text{-}a\text{-}student(C100), S_0) \wedge enrolled(John, C100, S_0).$$

By the completeness assumption,

$$\Sigma_0 \models \pm\Phi(John, C100, drop\text{-}a\text{-}student(C100), S_0),$$

in which case

$$\Sigma_0 \models \pm enrolled(John, C100, do(drop\text{-}a\text{-}student(C100), S_0)).$$

In other words, we would know whether *John* was the student dropped from $C100$, violating the intention of the *drop-a-student* transaction.

Despite the inadequacies of the axiomatization of Section 2.2 (specifically the failure of successor state axioms for specifying indeterminate transactions), we can represent this setting with something like the following axioms:

$$(\exists st) enrolled(st, c, s) \supset Poss(drop\text{-}a\text{-}student(c), s).$$

$$enrolled(st, c, s) \supset Poss(drop(st, c), s).$$

$$Poss(a, s) \supset \{a = drop(st, c) \supset \neg enrolled(st, c, do(a, s))\}.$$

$$Poss(a, s) \supset \{a = drop\text{-}a\text{-}student(c) \supset (\exists! st) enrolled(st, c, s) \wedge \neg enrolled(st, c, do(a, s))\}.$$[13]

$$Poss(a, s) \supset \{\neg enrolled(st, c, s) \wedge enrolled(st, c, do(a, s)) \supset a = register(st, c)\}.$$

$$Poss(a, s) \supset \{enrolled(st, c, s) \wedge \neg enrolled(st, c, do(a, s)) \supset a = drop(st, c) \vee a = drop\text{-}a\text{-}student(c)\}.$$

The last two formulas are examples of what Schubert [24] calls *explanation closure axioms*. For the example at hand, the last axiom provides an exhaustive enumeration of those transactions (namely $drop(st, c)$ and *drop-a-student*(c)) which could possibly explain how it came to be that st is enrolled in c in the current state s and is not enrolled in c in the successor state. Similarly, the second last axiom explains how a student could come to be enrolled in a course in which she was not enrolled previous to the transaction.[14] The feasibility of

[13] $(\exists! st)$ denotes the existence of a *unique* st.

[14] It is these explanation closure axioms which provide a succinct alternative to the frame axioms (McCarthy and Hayes [14]) which would normally be required to represent dynamically changing worlds like databases (Reiter [23]).

such an approach relies on a closure assumption, namely that we, as database designers, can provide a finite exhaustive enumeration of such explaining transactions.[15] In the "real" world, such a closure assumption is problematic. The state of the world has changed so that a student is no longer enrolled in a course. What can explain this? The school burned down? The student was kidnapped? The teacher was beamed to Andromeda by extraterrestrials? Fortunately, in the database setting, such open-ended possible explaining events are precluded by the database designer, by virtue of her initial choice of some closed set of transactions with which to model the application at hand; no events outside this closed set (school burned down, student kidnapped, etc.) can be considered in defining the evolution of the database. This initial choice of a closed set of transactions having been made, explanation closure axioms provide a natural representation of this closure assumption.

By appealing to explanation closure axioms, we can now specify indeterminate transactions. The price we pay is the loss of the simple regression-based query evaluator of (Reiter [23, 21]); we no longer have a simple sound and complete query evaluator. Of course, conventional first order theorem-proving does provide a query evaluator for such an axiomatization. For example, the following are entailments of the above axioms, together with unique names axioms for transactions and for $John$ and $Mary$:

$$enrolled(John, C100, S_0) \wedge enrolled(Mary, C100, S_0)$$
$$\supset$$
$$enrolled(John, C100, do(drop(Mary, C100), S_0)) \wedge$$
$$\neg enrolled(Mary, C100, do(drop(Mary, C100), S_0)).$$

$$\{(\forall st).enrolled(st, C100, S_0) \equiv st = John\} \supset$$
$$(\forall st)\neg enrolled(st, C100,$$
$$do(drop\text{-}a\text{-}student\ (C100), S_0)).$$

$$\{(\forall st).enrolled(st, C100, S_0) \equiv$$
$$st = John \vee st = Mary\}$$
$$\supset$$
$$enrolled(John, C100, do(drop\text{-}a\text{-}student(C100), S_0)) \oplus$$
$$enrolled(Mary, C100, do(drop\text{-}a\text{-}student(C100), S_0)).$$

Notice that the induction axiom (3) of Section 2.3 does not depend on any assumptions about the underlying database. In particular, it does not depend on successor state axioms. It follows that we can continue to use induction to prove properties of database states and integrity constraints in the more generalized setting of indeterminate transactions. The fundamental perspective on integrity constraints of (Reiter [20]) – namely that they are inductive entailments of the database – remains the same.

[15] This assumption is already implicit in our successor state axioms of Section 2.2

Acknowledgements

Many of my colleagues provided important conceptual and technical advice. My thanks to Leo Bertossi, Alex Borgida, Craig Boutilier, Charles Elkan, Michael Gelfond, Gösta Grahne, Russ Greiner, Joe Halpern, Hector Levesque, Vladimir Lifschitz, Fangzhen Lin, Wiktor Marek, John McCarthy, Alberto Mendelzon, John Mylopoulos, Javier Pinto, Len Schubert, Yoav Shoham and Marianne Winslett. Funding for this work was provided by the National Science and Engineering Research Council of Canada, and by the Institute for Robotics and Intelligent Systems.

References

[1] A. Baker. A simple solution to the Yale shooting problem. In R. Brachman, H.J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 11–20. Morgan Kaufmann Publishers, Inc., 1989.

[2] A. Baker and M. Ginsberg. Temporal projection and explanation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 906–911, Detroit, MI, 1989.

[3] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 292–322. Plenum Press, New York, 1978.

[4] J. Finger. *Exploiting Constraints in Design Synthesis.* PhD thesis, Stanford University, Stanford, CA, 1986.

[5] M.L. Ginsberg and D.E. Smith. Reasoning about actions I: A possible worlds approach. *Artificial Intelligence*, 35:165–195, 1988.

[6] C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 183–205. American Elsevier, New York, 1969.

[7] A. R. Haas. The case for domain-specific frame axioms. In F. M. Brown, editor, *The frame problem in artificial intelligence. Proceedings of the 1987 workshop*, pages 343–348, Los Altos, California, 1987. Morgan Kaufmann Publishers, Inc.

[8] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *Proceedings of the National Conference on Artificial Intelligence*, pages 328–333, 1986.

[9] R. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 12:121–146, 1992.

[10] V. Lifschitz. Toward a metatheory of action. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 376–386, Los Altos, CA, 1991. Morgan Kaufmann Publishers, Inc.

[11] F. Lin and Y. Shoham. Provably correct theories of action. In *Proceedings of the National Conference on Artificial Intelligence*, 1991.

[12] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, second edition, 1987.

[13] J. McCarthy. Programs with common sense. In M. Minsky, editor, *Semantic Information Processing*, pages 403–418. The MIT Press, Cambridge, MA, 1968.

[14] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969.

[15] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1988.

[16] E.P.D. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4:356–372, 1988.

[17] E.P.D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In R.J. Brachman, H. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332. Morgan Kaufmann Publishers, Inc., 1989.

[18] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[19] R. Reiter. The projection problem in the situation calculus: A soundness and completeness result, with an application to database updates. 1992. submitted for publication.

[20] R. Reiter. Proving properties of states in the situation calculus. 1992. submitted for publication.

[21] R. Reiter. On specifying database updates. Technical report, Department of Computer Science, University of Toronto, in preparation.

[22] R. Reiter. A simple solution to the frame problem (sometimes). Technical report, Department of Computer Science, University of Toronto, in preparation.

[23] R. Reiter. On formalizing database updates: preliminary report. In *Proc. 3rd International Conference on Extending Database Technology*, Vienna, March 23 - 27, 1992. to appear.

[24] L.K. Schubert. Monotonic solution of the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In H.E. Kyberg, R.P. Loui, and G.N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, 1990.

[25] R. Waldinger. Achieving several goals simultaneously. In E. Elcock and D. Michie, editors, *Machine Intelligence 8*, pages 94–136. Ellis Horwood, Edinburgh, Scotland, 1977.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Learning Missing Clauses by Inverse Resolution

Peter Idestam-Almquist*

Department of Computer and Systems Sciences
Stockholm University
Electrum 230, 164 40 Kista, Sweden
pi@dsv.su.se

Abstract

The incomplete theory problem has been of large interest both in explanation based learning and more recently in inductive logic programming. The problem is studied in the context of Horn clause logic, and it is assumed that there is only one clause missing for each positive example given. Previous methods have used either top down or bottom up induction. Both these induction strategies include some undesired restriction on the hypothesis space for the missing clause. To overcome these limitations a method where the different induction strategies are completely integrated is presented. The method involves a novel approach to inverse resolution by using resolution, and it implies some extensions to the framework of inverse resolution which makes it possible to uniquely determine the most specific result of an inverse resolution step.

1 Introduction

Completion of incomplete theories has been of large interest in machine learning, particularly in the area of explanation based learning, for which a complete theory is crucial [Mitchell *et al.* 1986, DeJong and Mooney 1986]. Research on augmenting an incomplete domain has been reported in [Hall 1988, Wirth 1988, Ali 1989]. A new framework for inductive learning was invented by inverting resolution [Muggelton and Buntine 1988]. Papers considering augmentation of incomplete theories in this framework are [Wirth 1989, Rouveirol and Puget 1990, Rouveirol 1990].

We only consider Horn clause logic, which is a subset of first order logic, and we follow the notation in logic programming [Lloyd 1987]. The incomplete theory problem can then be formulated as follows. Let P be a definite program (an incomplete theory) and E a definite program clause which should but does not follow from P ($P \not\models E$). Then find a definite program clause H such that:

(a) $P \cup \{E\} \not\models \overline{H}$

(b) $P \cup \{H\} \models E$

H is an ***inductive conclusion*** according to [Genesereth and Nilsson 1987].

Let $E=(A \leftarrow B_1,...,B_n)$. Then by ***top down induction*** we mean any reasoning procedure, to infer an inductive conclusion, that starts from A. By ***bottom up induction*** we mean any inductive reasoning procedure that starts from $B_1,...,B_n$.

Most previous methods use either top down [Hall 1988, Wirth 1988, Ali 1989] or bottom up induction [Sammut and Banerji 1986, Muggelton and Buntine 1988, Rouveirol and Puget 1990]. Both these induction strategies have some undesired restrictions on the hypothesis space of H. In [Wirth 1989] a method that combines top down and bottom up induction is presented, while in this paper a method where they are completely integrated will be described. In the previous methods there are also other undesired restrictions, namely that the input clause E must be fully instantiated [Hall 1988, Wirth 1988, Wirth 1989, Ali 1989, Sammut and Banerji 1986] or a unit clause [Muggelton and Buntine 1988]. Our method works for full Horn clause logic.

Logical entailment is used as a definition of generality. Let E and F be two expressions. Then E is ***more general*** than F, if and only if E logically entails F ($E \models F$). We also say that F is ***more specific*** than E.

In the examples, predicate symbols are denoted by p, q, r, s, t and u. Variables (universally quantified) are denoted by x, y, z and w. Constants are denoted by a, b and c. Skolem functions are denoted by k.

In section 2 the inductive framework of inverse resolution is given. In section 3 some extensions to this framework, which make it possible to determine the most specific inverse resolvent, are described. In section 4 a new

* This research was supported by NUTEK, the Swedish National Board for Industrial and Technical Development.

inverse resolution method is presented, and finally in section 5 related work and contributions is discussed.

2 The Framework of Inverse Resolution

The inductive framework of inverse resolution was first presented in [Muggelton and Buntine 1988]. First, as a background, resolution will be described. Then inverse resolution will be definied, and some problems considering inverse reslution will be pointed out.

2.1 Resolution

A ***substitution*** is a finite set of the form $\{v_1/t_1,...,v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i, and the variables $v_1,...,v_n$ are distinct. Each element v_i/t_i is called a ***binding*** for v_i. A substitution is applied by simultaneously replacing each occurence of the variable v_i, in an expression, by the term t_i.

An ***expression*** is either a term, a literal, a clause or a set of clauses. (A fixed ordering of literals in clauses and a fixed ordering of clauses in sets of clauses are assumed.)

Let E be an expression and V be the set of variables occurring in E. A ***renaming substitution*** for E is a substitution $\{x_1/y_1,...,x_n/y_n\}$ such that $y_1,...,y_n$ are distinct variables and $(V-\{x_1,...,x_n\})\cap\{y_1,...,y_n\}=\varnothing$.

Let E and F be expressions. Then E is a ***variant*** of F if there exists a renaming substitution θ such that $E=F\theta$.

A ***unifier*** for two terms or literals t_1 and t_2 is a substitution θ such that $t_1\theta=t_2\theta$.

A unifier θ for t_1 and t_2 is called a ***most general unifier*** (mgu) for t_1 and t_2, if for each unifier θ' of t_1 and t_2 there exists a substitution θ'' such that $\theta'=\theta\theta''$.

Let C and D be two clauses which have no variables in common. Then the clause R is ***resolved*** from C and D, denoted $(C;D) \vdash_R R$, if the following conditions hold:
(a) A is a literal in C and B is a literal in D.
(b) θ is an mgu of A and $\overline{B}$.
(c) R is the clause $((C-\{A\})\cup(D-\{B\}))\theta$.
The clause R is called a ***resolvent*** of C and D.

Since C and D have no variables in common, the mgu θ can uniquely be divided into two disjunctive parts θ_A and θ_B such that $\theta=\theta_A\cup\theta_B$ and $A\theta_A=\overline{B}\theta_B$. Consequently condition (c) can be rewritten as:
(c) R is the clause $(C-\{A\})\theta_A\cup(D-\{B\})\theta_B$, where $\theta=\theta_A\cup\theta_B$ and $A\theta_A=\overline{B}\theta_B$.

Let R_0 be a definite program clause and P a definite program. A ***linear derivation*** from R_0 and P consists of a sequence $R_0,R_1,...$ of definite program clauses and a sequence $C_1,C_2,...$ of variants of definite program clauses in P such that each R_{i+1} is resolved from C_{i+1} and R_i. A linear derivation of R_k from R_0 and P is denoted:
$(R_0;C_1) \vdash_R (R_1;C_2) \vdash_R ... \vdash_R R_k$ or for short
$(R_0;P) \vdash_R^* R_k$.

2.2 Inverse Resolution

A ***place*** within an expression is denoted by an n-tuple and defined recursively as follows. The term, literal or clause at place $<a_1>$ within $f(t_1,...,t_n)$ or $\{t_1,...,t_n\}$ is t_{a_1}. The term or literal at place $<a_1,...,a_m>$ (m>1) within $f(t_1,...,t_n)$ or $\{t_1,...,t_n\}$ is the term or literal at place $<a_2,...,a_m>$ in t_{a_1}.

Let E be an expression. Then for each substitution θ there exists a unique inverse substitution θ^{-1} such that $E\theta\theta^{-1}=E$. Whereas the substitution θ maps variables in E to terms, the inverse substitution θ^{-1} maps terms in $E\theta$ to variables. An ***inverse substitution*** is a finite set of the form $\{(t_1,\{p_{1,1},...,p_{1,m_1}\})/v_1,...,(t_n,\{p_{n,1},...,p_{n,m_n}\})/v_n\}$ where each v_i is a variable distinct from the variables in E, each t_i is a term distinct from v_i, the variables $v_1,...,v_n$ are distinct, each $p_{i,j}$ is a place at which t_i is found within E and the places $p_{1,1},...,p_{n,m_n}$ are distinct. An inverse substitution is applied by replacing all t_i at places $\{p_{i,1},...,p_{i,m_i}\}$ in the expression E by v_i.

Example: If the following inverse substitution $\{(a,\{<1,1,2>,<1,2,1,1>,<2,2,1>\})/x\}$ is applied on the expression $\{(p(a,a)\leftarrow p(f(a)),(q(a)\leftarrow r(a))\}$, the expression $\{(p(a,x)\leftarrow p(f(x)),(q(a)\leftarrow r(x))\}$ is obtained.

Let R, C and D be three clauses. If R can be resolved from C and D, then D can be inverse resolved from R and C. The clause D is ***inverse resolved*** from R and C, denoted $(R;C) \vdash_{IR} D$, if the following conditions hold:
(a) A is a literal in C.
(b) θ_A is a substitution whose variables are variables that occur in A.
(c) $(C-\{A\})\theta_A$ is a subset of R.
(d) Γ is a subset of $(C-\{A\})\theta_A$.
(e) θ_B^{-1} is an inverse substitution whose terms are terms that occur in A.
(f) D is the clause $((R-\Gamma)\cup\{\overline{A}\}\theta_A)\theta_B^{-1}$.

The clause D is called an ***inverse resolvent*** of R and C.

Given R and C there are four sources of indeterminacy for D, namely: A, θ_A, Γ and θ_B^{-1}.

If A is a positive literal then D is ***forwardly inverse resolved***, and if A is a negative literal then D is ***backwardly inverse resolved.***

Example: Suppose we have R=(s(a,z)←q(a),r(b)), C=(p(a,x)←q(a),r(x)) and D=(s(y,z)←p(y,b)). The clause D can be forwardly inversed resolved from R and C, (R;C) $\vdash_{IR}$ D, if A=p(a,x), θ_A={x/b}, Γ=(←q(a),r(b)) and θ_B^{-1}={(a,{<1,1>,<2,1>})/y}. The clause C can be backwardly inverse resolved from R and D, (R;D) $\vdash_{IR}$ C, if A=¬p(y,b), θ_A={y/a}, Γ=(s(a,z)←) and θ_B^{-1}= {(b,{<1,2>,<3,1>})/x}. (It is assumed that the positive literal is first in the ordering of literals in a clause.)

Let D_0 be a definite program clause and P a definite program. An ***inverse linear derivation*** from D_0 and P consists of a sequence $D_0,D_1,...$ of definite program clauses and a sequence $C_1,C_2,...$ of variants of definite program clauses in P such that each D_{i+1} is inverse resolved from C_{i+1} and D_i. An inverse linear derivation of D_k from D_0 and P is denoted:
$(D_0;C_1) \vdash_{IR} (D_1;C_2) \vdash_{IR} ... \vdash_{IR} D_k$ or for short
$(D_0;P) \vdash_{IR}^* D_k$.

A ***backward inverse linear derivation*** is an inverse linear derivation where each D_i is backwardly inverse resolved, and a ***forward inverse linear derivation*** is an inverse linear derivation where each D_i is forwardly inverse resolved.

2.3 Some Problems

Consider the definition of inverse resolved. The substitution θ_A can be divided into two disjunctive parts, θ_{A1} including the variables that occur both in A and (C−{A}), and θ_{A2} including the variables that only occur in A ($\theta_A=\theta_{A1}\cup\theta_{A2}$). Then, to determine an inverse resolvent D, we have to choose A, θ_{A1}, θ_{A2}, Γ and θ_B^{-1}. Only in some special cases there are more than one alternative for A and θ_{A1}.

Example: Let R=(p←q(a),r(b)) and C=(p←q(x),r(x)). Then we have either A=¬q(x) and θ_{A1}={x/b}, or A=¬r(x) and θ_{A1}={x/a}.

For Γ and θ_B^{-1} there are limited numbers of alternatives, but for θ_{A2} there is not. The terms in θ_{A2} can be any possible terms. Consequently, it is hard to choose θ_{A2}. Unfortunately there are examples when the choice of θ_{A2} is crucial.

Example: Let R=(r←q), C_1=(p(x)←q), C_2=(s←p(a)) and D=(r←s). Then there is a linear derivation of R from D and $\{C_1,C_2\}$:
$(D;C_2) \vdash_R ((r←p(a));C_1) \vdash_R (r←q)$.
Consequently, there is an inverse linear derivation of D from R and $\{C_1,C_2\}$:
$(R;C_1) \vdash_{IR} ((r←p(a));C_2) \vdash_R (r←s)$.
In the first inverse resolution step θ_{A2} is chosen as {x/a}. With any other choice of θ_{A2} the inverse linear derivation of D would not have been possible.

If R, C, A and θ_{A1} are given, then it is desirable that a unique most specific inverse resolvent can be determined. Unfortunately, in Horn clause logic, it is not possible due to the substitution θ_{A2}.

Example: Let R=(r←q) and C=(p(x)←q). If we seek the most specific clause D such that (R;C) $\vdash_{IR}$ D, then we let $\Gamma=\varnothing$ and $\theta_B^{-1}=\varnothing$ but what should θ_{A2} be? If we let $\theta_{A2}=\varnothing$, the clause D_1=(r←p(x),q) is obtained. For example the clauses D_2=(r←p(a),q) and D_3=(r←p(b),q) are more specific than D_1, but neither D_2 nor D_3 is more specific than the other. Consequently, there is no unique most specific inverse resolvent.

3 Extended Inverse Resolution

Our inverse resolution method (see section 4) implies some extensions to the framework of inverse resolution. After these extensions the choices of θ_{A2}, Γ and θ_B^{-1} in inverse linear derivations can be postponed, and the most specific inverse resolvent can be determined.

3.1 Existentially Quantified Variables

To postpone the choice of θ_{A2}, existentially quantified variables will temporarily be introduced. Any sentence, in which the existentially quantified variables are replaced by Skolem functions, is equal to the original sentence with respect to satisfiability [Genesereth and Nilsson 1987]. Therefore the existentially quantified variables will be represented by Skolem functions. As a consequence of the introduction of existentially quantified variables (Skolem functions), some additional types of substitutions are needed.

A ***Skolem function*** is a term $f(x_1,...,x_n)$ where f is a new function symbol and $x_1,...,x_n$ are the variables associated with the enclosing universal quantifiers.

A ***Skolem substitution*** is a finite set of the form $\{v_1/k_1,...,v_n/k_n\}$, where each v_i is a variable, each k_i is a Skolem function, and the variables $v_1,...,v_n$ are distinct.

An ***inverse Skolem substitution*** is a finite set of the form $\{k_1/v_1,...,k_n/v_n\}$, where each k_i is a Skolem function, each v_i is a new variable, and the Skolem functions $k_1,...,k_n$ are distinct.

Let $\sigma=\{x_1/k_1,...,x_n/k_n\}$ be a Skolem substitution and $\sigma^{-1}=\{k_1/y_1,...,k_n/y_n\}$ an inverse Skolem substitution such that the Skolem functions in σ and σ^{-1} are exactly the same. Then the ***composition*** $\sigma\sigma^{-1}$ of σ and σ^{-1} is a renaming substitution $\{x_1/y_1,...,x_n/y_n\}$ for any expression E.

An ***existential substitution*** is a finite set of the form $\{k_1/t_1,...,k_n/t_n\}$, where each k_i is a Skolem function (existentially quantified variable), each t_i is a term (possibly a Skolem function) distinct from k_i, and the Skolem functions $k_1,...,k_n$ are distinct. While a substitution or a Skolem substitution corresponds to a specialization an existential substitution corresponds to a generalization.

As an inverse substitution, an inverse existential substitution is specified with respect to an expression E. An ***inverse existential substitution*** is a finite set of the form $\{(t_1,\{p_{1,1},...,p_{1,m_1}\}/k_1,...,(t_n,\{p_{n,1},...,p_{n,m_n}\})/k_n\}$ where each k_i is a Skolem function distinct from the Skolem functions in E, each t_i is a term distinct from k_i, the Skolem functions $k_1,...,k_n$ are distinct, each $p_{i,j}$ is a place at which t_i is found within E and the places $p_{1,1},...,p_{n,m_n}$ are distinct. An inverse existential substitution is applied by replacing all t_i at places $\{p_{i,1},...,p_{i,m_i}\}$ in E by k_i.

Let $\sigma=\{v_1/k_1,...,v_n/k_n\}$ be a Skolem substitution and $\eta=\{k_1/t_1,...,k_n/t_n\}$ an existential substitution such that the Skolem functions in σ and η are exactly the same. Then the ***composition*** $\sigma\eta$ of σ and η is the substitution $\{v_1/t_1,...,v_n/t_n\}$. In this way Skolem substitutions and existential substitutions can be used to postpone the choice of θ_{A2}.

3.2 Most Specific Inverse Resolution

To postpone the choice of Γ, the notion of ***optional literals*** will be used. A clause $\{B_1,...,B_k,B_{k+1},...,B_n\}$, in which the literals $\{B_{k+1},...,B_n\}$ are optional, is denoted $C[c]=\{B_1,...,B_k,[B_{k+1},...,B_n]\}$ where $C=\{B_1,...,B_k\}$ and $c=\{B_{k+1},...,B_n\}$. Consequently, if $c=\varnothing$ then $C[c]=C$.

Example: Let $R=(p\leftarrow q,r,s)$ and $C=(t\leftarrow q,r,s)$ be two clauses. Then $(R;C) \vdash_{IR} D$, where $D=(p\leftarrow t,q,r,s)-\Gamma$ and $\Gamma\subseteq\{\neg q,\neg r,\neg s\}$. All these alternatives for D can be described in a compact way by using optional literals. Thus, $D[d]=(p\leftarrow t,[q,r,s])$.

The definition of inverse resolved can now be modified in such a way that the choices of θ_{A2}, Γ and θ_B^{-1} are postponed. The clause D is ***most specific inverse resolved*** from R[r] (which may include Skolem functions) and C, denoted $(R[r];C) \vdash\downarrow_{IR} D$, if the following conditions hold:

(a) A is a literal in C.
(b) θ_{A1} is a substitution whose variables are variables that occur both in A and $(C-\{A\})$.
(c) η^{-1} is an inverse existential substitution whose terms are terms that occur in $(C-\{A\})$.
(d) $(C-\{A\})\theta_{A1}\eta^{-1}$ is a subset of R[r].
(e) σ is a Skolem substitution whose variables are all the variables that only occur in A.
(f) D[d] is the clause $D=((R-\Gamma)\cup\{\overline{A}\}\theta_{A1}\sigma)$, $d=r\cup\Gamma$, where $\Gamma=(C-\{A\})\theta_{A1}\eta^{-1}$.

The clause $D\cup d$ is called a ***most specific inverse resolvent*** of R and C.

Given R and C, there are only two sources of indeterminacy, namely: A and θ_{A1}. Consequently, given R, C, A and θ_{A1} there is a unique most specific inverse resolvent $D\cup d$.

Example: Let $R=(r\leftarrow q)$ and $C=(p(x)\leftarrow q)$. Then the unique most specific inverse resolvent of R and C is the clause $D\cup d=(r\leftarrow p(k),q)$ where k is a Skolem functions (representing an existentially quantified variable). This is true, since $\forall x(r\leftarrow p(x),q) \models (r\leftarrow p(t),q)$, and $(r\leftarrow p(t),q) \models \exists x(r\leftarrow p(x),q)$ for any term t.

Let $D_0[d_0]$ a be definite program clause and P a definite program. A ***most specific inverse linear derivation*** from $D_0[d_0]$ and P consists of a sequence $D_0[d_0],D_1[d_1],...$ of definite program clauses and a sequence $C_1,C_2,...$ of variants of definite program clauses in P such that each $D_{i+1}[d_{i+1}]$ is most specific inverse resolved from C_{i+1} and $D_i[d_i]$. A most specific inverse linear derivation of $D_k[d_k]$ from $D_0[d_0]$ and P is denoted:
$(D_0[d_0];C_1) \vdash\downarrow_{IR} (D_1[d_1];C_2) \vdash\downarrow_{IR} ... \vdash\downarrow_{IR} D_k[d_k]$
or for short $(D_0[d_0];P) \vdash\downarrow_{IR}^* D_k[d_k]$.

Each result of an inverse linear derivation can be obtained from the result of some most specific inverse linear derivation, if we apply an inverse substitution, an existential substitution, and drop a subset of the optional literals.

Example: Suppose we have the following clauses $R=(r\leftarrow q)$, $C_1=(p(x)\leftarrow q)$, $C_2=(s\leftarrow p(a))$, $C_3=(t(b)\leftarrow p(b))$, $D=(r\leftarrow s,t(x),p(c))$ and $D'[d']=(r\leftarrow s,t(b),[p(k),q])$. Then
$(R;\{C_1,C_2,C_3\}) -_{IR}* D$ and
$(R;\{C_1,C_2,C_3\}) \vdash\downarrow_{IR}* D'[d']$.
The clause D can be obtained from D'[d'] by application of the inverse substitution $\{(b,\{<3,1>\})/x\}$ and the existential substitution $\{k/c\}$, and by dropping the optional literal q.
The most specific inverse linear derivation of D'[d'] looks as follows:
$(R;C_1) \vdash\downarrow_{IR} ((r\leftarrow p(k),[q]);C_2) \vdash\downarrow_{IR}$
$((r\leftarrow s,[p(k),q]);C_3) \vdash\downarrow_{IR} (r\leftarrow s,t(b),[p(k),q])$.
That η^{-1}, in the two last steps are $\{(a,<1,1>)/k\}$ and $\{(b,<1,1>)/k\}$, and that k then can be replaced by a third term c, may seem inconsistent, but it is not. Consider the corresponding inverse linear derivation of D from R and $\{C_1,C_2,C_3\}$:
$((r\leftarrow q);C_1) \vdash_{IR} ((r\leftarrow q,p(c));C_1) \vdash_{IR}$
$((r\leftarrow q,p(b),p(c));C_1) \vdash_{IR} ((r\leftarrow p(a),p(b),p(c));C_2) \vdash_{IR}$
$((r\leftarrow s,p(b),p(c)),C_3) \vdash_{IR} (r\leftarrow s,t(x),p(c))$.
Note that since k has been used as three different terms (a, b and c) in the most specific inverse linear derivation, three inverse resolution steps are needed to compensate for the step where k is introduced. Note also that $\theta_{A2}=\{x/c\}$ in the first, $\theta_{A2}=\{x/b\}$ in the second and $\theta_{A2}=\{x/a\}$ in the third inverse resolution step. To choose exactly those substitutions is hard, but in a most specific inverse linear derivation it is not necessary.

3.3 Truncation Generalization

A clause C_1 ***$\theta\eta$-subsumes*** a clause C_2 if there exists a substitution θ and an existential substitution η such that $C_1\theta\subseteq C_2\eta$. If C_1 $\theta\eta$-subsumes C_2 then $C_1 \models C_2$.

To perform a ***$\theta\eta$-truncation*** is to apply some arbitrary existential substitution η, apply some arbitrary inverse substitution θ^{-1}, and drop some arbitrary literals. The generalizarion technique $\theta\eta$-truncation corresponds to $\theta\eta$-subsumption.

Let P be a definite program (an incomplete theory) and E a definite program clause which should but does not follow from P ($P \not\models E$), let $\mathbb{D}$ be the set of definite program clauses D such that $(E;P) \vdash_{IR}* D$, and let $\mathbb{H}$ be the set of definite program clauses H such that $P\cup\{H\} \models E$. Since resolution is not complete [Rob65] $\mathbb{D}$ is a subset of $\mathbb{H}$ ($\mathbb{D} \subseteq \mathbb{H}$). In particular each definite program clause D' that $\theta\eta$-subsumes some clause D, where $D \in \mathbb{D}$, will be in $\mathbb{H}$. This is true since $P\cup\{D\} \models E$, and $D' \models D$, gives us $P\cup\{D'\} \models E$. Consequently, we can perform any $\theta\eta$-truncation on the result D of a most specific inverse linear derivation and still have an inductive conclusion.

4 The Method

In this section a method, which in an easy way realizes inverse linear derivations, will be described. Instead of performing an inverse linear derivation from the example clause E, a variant of ordinary resolution derivation is performed from the complement $\overline{E}$ of E.

4.1 Complement

A ***definite program clause complement set*** (dpcc-set) is set of clauses containing exactly one unit goal and a number of unit clauses.

Let C be a definite program clause $(A\leftarrow B_1,...,B_n)$, σ_S^{-1} an inverse Skolem substitution including all Skolem functions in C, and σ_S a Skolem substitution including all the universally quantified variables in C. Then the ***complement*** $\overline{C}$ of C is the definite program clause complement set $\{(\leftarrow A),(B_1\leftarrow),...,(B_n\leftarrow)\}\sigma_S^{-1}\sigma_C$. Let S be a dpcc-set $\{(\leftarrow A),(B_1\leftarrow),...,(B_n\leftarrow)\}$, σ_C^{-1} an inverse Skolem substitution including all Skolem functions in S, and σ_S a Skolem substitution including all the universally quantified variables in S. Then the complement $\overline{S}$ of S is the definite program clause $(A\leftarrow B_1,...,B_n)\sigma_C^{-1}\sigma_S$. Thus, the complement of a dpcc-set is a definite program clause and vice versa.

Example: Let C be the clause $(p(a,x)\leftarrow q(k,x,y))$. Then the complement $\overline{C}$ of C is the definite program clause complement set $\{(\leftarrow p(a,k_x)),(q(x_k,k_x,k_y)\leftarrow)\}$, which is obtained by application of the inverse Skolem substitution $\{k/x_k\}$ and the Skolem substitution $\{x/k_x,y/k_y\}$ on the set of clauses $\{(\leftarrow p(a,x)),(q(k,x,y)\leftarrow)\}$. The complement C' of $\overline{C}$ is the definite program clause $(p(a,x')\leftarrow q(k',x',y'))$, which is obtained by application of the inverse Skolem substitution $\{k_x/x',k_y/y'\}$ and the Skolem substitution

$\{x_k/k'\}$ on the clause $(p(a,k_x)\leftarrow q(x_k,k_x,k_y))$. The clause C' is a variant of C, since C'=Cθ where θ is the renaming substitution {x/x',y/y'}.

4.2 Clause Set Resolution

The notion of ***optional clauses*** will be used a similar same way as optional literals. A set of clauses $\{C_1,\ldots,C_k,C_{k+1},\ldots,C_n\}$, in which the clauses $\{C_{k+1},\ldots,C_n\}$ are optional, is denoted $S[s]=\{C_1,\ldots,C_k,[C_{k+1},\ldots,C_n]\}$ where $S=\{C_1,\ldots,C_k\}$ and $s=\{C_{k+1},\ldots,C_n\}$. Consequently, if $s=\emptyset$ then $S[s]=S$.

An ***elementary clause set*** Σ is a set of clauses containing at most one clause, that is $\Sigma=\emptyset$ or $\Sigma=\{C\}$ where C is a clause.

Let $S_i[s_i]$ be a clause set and Σ an elementary clause set. Then $S_{i+1}[s_{i+1}]$ is ***clause set resolved*** from $S_i[s_i]$ and Σ, denoted $(S_i[s_i];\Sigma) \vdash_{CSR} S_{i+1}[s_{i+1}]$, if the following conditions hold:

(a) C' is a variant of a clause C in $S_i[s_i]\cup\Sigma$.
(b) D is a clause in $S_i[s_i]$.
(c) R is a resolvent of C' and D.
(d) Δ is the elementary clause set of unit clauses in {C,D}.
(e) $S_{i+1}[s_{i+1}]$ is the clause set $S_{i+1}=(S_i-\{C,D\})\cup\{R\}$, $s_{i+1}=s_i\cup\Delta$.

If D is a definite goal then R will also be a definite goal, and we say that $S_{i+1}[s_{i+1}]$ is ***backwardly clause set resolved*** from $S_i[s_i]$ and Σ. If both C and D are definite program clauses then R will also be a definite program clause, and we say that $S_{i+1}[s_{i+1}]$ is ***forwardly clause set resolved*** from $S_i[s_i]$ and Σ.

Let $S_0[s_0]$ be a clause set and P a definite program. A ***clause set derivation*** from $S_0[s_0]$ and P consists of a sequence $S_0[s_0],S_1[s_1],\ldots$ of clause sets, and a sequence $\Sigma_1,\Sigma_2,\ldots$ of elementary clause sets, such that each Σ_i is a subset of P and each clause set $S_{i+1}[s_{i+1}]$ is clause set resolved from $S_i[s_i]$ and Σ_{i+1}. A clause set derivation of $S_k[s_k]$ from $S_0[s_0]$ and P is denoted:
$(S_0[s_0];\Sigma_1) \vdash_{CSR} (S_1[s_1];\Sigma_2) \vdash_{CSR} \ldots \vdash_{CSR} S_k[s_k]$ or
$(S_0[s_0];P) \vdash_{CSR}^* S_k[s_k]$.

A ***backward clause set derivation*** is a clause set derivation where each $S_i[s_i]$ is backwardly clause set resolved, and a ***forward clause set derivation*** is a clause set derivation where each $S_i[s_i]$ is forwardly clause set resolved.

Example: Let $S_0=\{(\leftarrow p(k)),(q(k)\leftarrow),(r(k)\leftarrow)\}$ and $C=(p(x)\leftarrow r(x),s(x))$. Then we have the following backward clause set derivation:
$(S_0;\{C\}) \vdash_{CSR} (\{(\leftarrow r(k),s(k)),(q(k)\leftarrow),(r(k)\leftarrow)\};\emptyset) \vdash_{CSR} \{(\leftarrow s(k)),(q(k)\leftarrow),[(r(k)\leftarrow)]\}$.

4.3 The Algorithm

Let P be a definite program and E a definite program clause which should but does not follow from P ($P \not\models E$). Our algorithm to produce an inductive conclusion H looks as follows.

Completion of Refutation Proof Algorithm:

1. Compute the complement $\overline{E}$ of E, which is a dpcc-set.
2. Perform a clause set derivation from P and $\overline{E}$ of a dpcc-set $\overline{H'[h']}$.
3. Compute the complement H'[h'] of $\overline{H'[h']}$, which is a definite program clause.
4. Perform a θη-truncation of H'[h'] to obtain H.

The generalization performed in steps 1-3, is called a ***reformulation generalization***, which in fact is equivalent to performing a most specific inverse linear derivation.

Reconsider the definition of most specific inverse linear resolved in section 2. Let $\{A_1,\ldots,A_m\}=C-\{A\}$ and $\{B_1,\ldots,B_n\}=R-\{A_1,\ldots,A_m\}\theta_{A1}\eta^{-1}$. Then the clause $D[d]=\{B_1,\ldots,B_n\}\cup\{\overline{A}\}\theta_{A1}\sigma_{S2}\cup[\{A_1,\ldots,A_m\}\theta_{A1}\eta^{-1}]$ is most specific inverse resolved from $R=\{A_1,\ldots,A_m\}\theta_{A1}\eta^{-1}\cup\{B_1,\ldots,B_n\}$ and $C=\{A\}\cup\{A_1,\ldots,A_m\}$.

The corresponding reformulation generalization looks as follows:

1. The complement $\overline{R}$ of R is the dpcc-set $(\{\{\overline{A}_1\},\ldots,\{\overline{A}_m\}\}\theta_{A1}\eta^{-1}\cup\{\{\overline{B}_1\},\ldots,\{\overline{B}_n\}\})\sigma_{S1}^{-1}\sigma_R$ where σ_{S1}^{-1} is an inverse Skolem substitution including all Skolem functions in R and σ_R is a Skolem substitution including all universally quantified variables in R.
2. The following clause set derivation is performed: $(\overline{R};\{C\}) \vdash_{CSR}^* \overline{D[d]}$ where $\overline{D}=(\{\{\overline{B}_1\},\ldots,\{\overline{B}_n\}\}\cup\{\{A\}\theta_{A1}\}$ and $\overline{d}=[\{\{\overline{A}_1\},\ldots,\{\overline{A}_m\}\}\theta_{A1}\eta^{-1}])\sigma_{S1}^{-1}\sigma_R$.
3. The complement D[d] of $\overline{D[d]}$ is the definite program clause $(\{B_1,\ldots,B_n\}\cup\{\overline{A}\}\theta_{A1}\sigma_{S2}\cup[\{A_1,\ldots,A_m\}\theta_{A1}\eta^{-1}])\theta$ where σ_{S2} is a Skolem substitution including all universally quantified variables in $\overline{D[d]}\sigma_{S1}$ and θ is the renaming substitution $\theta=\sigma_{S1}^{-1}\sigma_R\sigma_R^{-1}\sigma_{S1}$.

Consequently, a most specific inverse linear derivation
$(D_0[d_0];P) \vdash_{\downarrow IR}^* D_k[d_k]$
is equivalent to the clause set derivation
$(S_0[s_0];P) \vdash_{CSR}^* S_k[s_k]$,
where $S_0[s_0] = \overline{D_0[d_0]}$, $D_k[d_k] = \overline{S_k[s_k]}\theta$ and θ is a renaming substitution.

Example: Let $R=(r(a,z)\leftarrow q(z))$, $C_1=(p(x,y)\leftarrow q(y))$, $C_2=(s(w,y)\leftarrow p(b,y))$, $D_1=(r(x,z)\leftarrow s(c,z))$, $D_2=(r(x,z)\leftarrow)$ and $D[d]=(r(a,z)\leftarrow s(k_w,z),[p(k_x,z),q(z)])$ Then
$(R;\{C_1,C_2\}) \vdash_{\downarrow IR}^* D[d]$, and
$(R;\{C_1,C_2\}) \vdash_{IR}^* D_1$.
Although D_2 is not inverse linear derivable, it is still an inductive conclusion, since $\{C_1,C_2,D_2\} \models R$.
With our algorithm D'[d'], D_1' and D_2', which are equal to D[d], D_1 and D_2 up to variable renaming, are constructed in the following way:
1. The complement $\overline{R}$ of R is the dpcc-set
$\{(\leftarrow r(a,k_z)),(q(k_z)\leftarrow)\}$.
2. The following clause set derivation is performed:
$(\overline{R};\{C_1\}) \vdash_{CSR}$
$(\{(\leftarrow r(a,k_z)),(p(x,k_y)\leftarrow),[(q(k_z)\leftarrow)]\};\{C_2\}) \vdash_{CSR} \overline{D[d]}$,
where
$\overline{D[d]}=\{(\leftarrow r(a,k_z)),(s(w,k_z)\leftarrow),[(p(x,k_y)\leftarrow),(q(k_z)\leftarrow)]\}$.
3. The complement D'[d'] of $\overline{D[d]}$ is the definite program clause
$(r(a,z')\leftarrow s(k_w,z'),[p(k_x,z'),q(z')])$.
4. By application of the inverse substitution $\{(a,\{<1,1>\})/x\}$ and the existential substitution $\{k_w/c\}$ and by dropping the optional literals, $D_1'=(r(x,z')\leftarrow s(c,z'))$ is obtained.
If the last negative literal in D_1' also is dropped then $D_2'=(r(x,z')\leftarrow)$ is obtained.

Steps 2 and 4 in the completion of refutation proof algorithm are indeterministic. The use of a preference bias can make them deterministic. Such a preference bias must specify which clause set is the most preferable result of the clause set derivation (reformulation bias), and which generalization should be done in the $\theta\eta$-truncation (truncation bias).

The algorithm is implemented in a system, called CRP1, in which a depth first search is used to find the best dpcc-set $\overline{H'[h']}$ according to some given preference bias.

4.4 Integrating Top down and Bottom up Induction

Backward inverse linear derivations correspond to top down induction, and forward inverse linear derivations correspond to bottom up induction. In our method, backward clause set derivations correspond to top down induction, and forward clause set derivations correspond to bottom up induction. Each step in a clause set derivation can be either backwardly or forwardly clause set resolved. Consequently, in our method (and in the system CRP1) top down and bottom up induction are completely integrated.

Example: Let $E=(p\leftarrow q,t,u)$ and $P=\{(p\leftarrow q,r),(s\leftarrow t,u)\}$. Then the inductive conclusion $H_1=(r\leftarrow t,u)$ is inferable by top down induction (backward inverse linear derivation), but not by bottom up induction (forward inverse linear derivation). The inductive conclusion $H_2=(p\leftarrow q,s)$ is inferable by bottom up induction, but not by top down induction. The inductive conclusion $H_3=(r\leftarrow s)$ can only be inferred by a method that combines top down and bottom up induction. With our algorithm the clause H_3 is constructed as follows:
1. The complement $\overline{E}$ of E is the dpcc-set
$\{(\leftarrow p),(q\leftarrow),(t\leftarrow),(u\leftarrow)\}$.
2. The following clause set derivation is performed:
$(\overline{E};\{(p\leftarrow q,r)\}) \vdash_{CSR}$
$(\{(\leftarrow q,r),(q\leftarrow),(t\leftarrow),(u\leftarrow)\};\varnothing) \vdash_{CSR}$
$(\{(\leftarrow r),(t\leftarrow),(u\leftarrow),[(q\leftarrow)]\};\{(s\leftarrow t,u)\}) \vdash_{CSR}$
$(\{(\leftarrow r),(s\leftarrow u),(u\leftarrow),[(t\leftarrow),(q\leftarrow)]\};\varnothing) \vdash_{CSR} \overline{H_3[h_3]}$
where $\overline{H_3[h_3]}=\{(\leftarrow r),(s\leftarrow),[(u\leftarrow),(t\leftarrow),(q\leftarrow)]\}$
3. The complement $H_3[h_3]$ of $\overline{H_3[h_3]}$ is the definite program clause $(r\leftarrow s,[u,t,q])$.

4. By dropping the optional literals $H_3=(r\leftarrow s)$ is obtained.
The first two steps in the clause set derivation are backwardly clause set resolved (top down induction) and the last two steps are forwardly clause set resolved (bottom up induction).

5 Concluding Remarks

Some extensions to the inverse resolution framework and a new inverse resolution method have been presented. This method subsumes the previous methods based on inverse resolution and completely integrates top down and bottom up induction.

Reconsider the definition of inverse resolved in section 2. If we let A be a positive literal, $\theta_{A2}=\varnothing$ and $\Gamma=(C-\{A\})\theta_A$ then it is a definition of the absorption operator [Muggelton and Buntine 1988]. If we let A be a positive literal, $\theta_{A2}=\varnothing$, $\Gamma=\varnothing$ and $\theta_B{}^{-1}=\varnothing$ then it is a definition of elementary saturation [Rouveirol and Puget 1990]. The saturation operator [Rouveirol and Puget 1990] is equal to an exhaustive forward inverse linear derivation, in which each step is restricted according to elementary saturation. If we let A be a positive literal, $\theta_{A2}=\varnothing$, $\Gamma=(C-\{A\})\theta_A$ and $\theta_B{}^{-1}=\varnothing$ then it is a definition of the learning procedure called generalize in [Banerji 1991]. If we let A be a negative literal, $\theta_{A2}=\varnothing$ and $\Gamma=(C-\{A\})\theta_A$ then it is a definition of the identification operator [Muggelton and Buntine 1988].

Since our method performs inverse linear derivations without any restrictions on A, θ_{A2}, Γ or $\theta_B{}^{-1}$, all the methods mentioned above can be seen as special cases of our method.

Our notion of optional literals is the same as in [Rouveirol and Puget 1990]. Our $\theta\eta$-truncation is similar to the truncation generalization in [Rouveirol and Puget 1990] and the truncation operator in [Muggelton and Buntine 1988], which both correspond to θ-subsumption.

Wirth [Wirth 1989] and Rouveirol [Rouveirol 1991] have both pointed out the advantages of combining top down and bottom up induction. In [Wirth 1989], a system called LFP2, which uses both top down and bottom up induction is presented. However, the different induction strategies are separated into different parts of the system. The first part (top down) is based on completion of partial proof trees, while the second part (bottom up) is based on operators performing inverse resolution. The second part uses the result from the first part, and different types of bias are used in the different parts. Our method has the major advantage that the two different induction strategies are completely integrated, which not only eliminates the restrictions that they imply when separated, but also makes possible the use of an overall preference bias.

The main contributions of this research are:

1. A complete integration of top down and bottom up induction.
2. Introduction of existentially quantified variables, which makes it possible to uniquely determine the most specific inverse resolvent.
3. A method to perform inverse resolution for full Horn clause logic by using resolution.

References

[Ali 1989] K. M. Ali, "Augmenting Domain Theory for Explanation Based Generalization" in *Proceedings of the 6th International Workshop on Machine Learning*, Morgan Kaufmann, 1989.

[Banerji 1991] R. B. Banerji, "Learning Theoretical Terms" in *Proceedings of International Workshop on Inductive Logic Programming*, 1991.

[DeJong and Mooney 1986] G. DeJong and Mooney, "Explanation-Based Learning: An Alternative View" in *Machine Learning* 1: 145-176, 1986.

[Genesereth and Nilsson 1987] Nilsson and Genesereth, *Logic Foundations of Artificial Intelligence*, Morgan Kaufmann, 1987.

[Hall 1988] R. J. Hall, "Learning by Failing to Explain: Using Partial Explanations to Learn in Incomplete or Intractable Domains" in *Machine Learning* 3: 45-77, 1988.

[Lloyd 1987] J. W. Lloyd, *Foundations of Logic Programming* (second edition), Springer-Verlag, 1987.

[Mitchell *et al.* 1986] T. M. Mitchell, S. Kedar-Cabelli and R. Keller, "Explanation-Based Generalization: A Unifying View" in *Machine Learning* 1: 47-80, 1986.

[Muggleton and Buntine 1988] S. Muggleton and W. Buntine, "Machine Invention of First-order Predicates by Inverting Resolution" in *Proceedings of the 5th International Conference on Machine Learning*, Morgan Kaufmann, 1988.

[Robinson 1965] J. Robinson, "A Machine-oriented Logic Based on the Resolution Principle" in *Journal of ACM* 12(1), 1965.

[Rouveirol 1990] C. Rouveirol, "Saturation: Postponing Choices when Inverting Resolution" in *Proceedings of the 9th European Conference on Artificial Intelligence*, Pitman, 1990.

[Rouveirol 1991] Céline Rouveirol, "ITOU: Induction of First Order Theories" in *Proceedings of International Workshop on Inductive Logic Programming*, 1991.

[Rouveirol and Puget 1990] C. Rouveirol and J. F. Puget, "Beyond Inversion of Resolution" in *Proceedings of the 7th International Conference on Machine Learning*, Morgan Kaufmann, 1990.

[Sammut and Banerji 1986] C. Sammut and R. Banerji, "Learning concepts by asking questions" in Michalski, Carbonell and Mitchell (eds), *Machine Learning: an artificial intelligence approach* volume 2, Morgan Kaufmann, 1986.

[Wirth 1988] R. Wirth, "Learning by Failure to Prove" in *Proceedings of the 3rd European Working Session on Learning*, Pitman, 1988.

[Wirth 1989] R. Wirth, "Completing Logic Programs by Inverse of Resolution" in *Proceedings of the 4th European Working Session on Learning*, Pitman, 1989.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Machine Discovery from Amino Acid Sequences by Decision Trees over Regular Patterns

Setsuo Arikawa† Satoru Kuhara‡ Satoru Miyano† Yasuhito Mukouchi††
Ayumi Shinohara† Takeshi Shinohara‡‡

† Research Institute of Fundamental Information Science, Kyushu University 33, Fukuoka 812, Japan.
‡ Graduate School of Genetic Resources Technology, Kyushu University 46, Fukuoka 812, Japan.
†† Department of Information Systems, Kyushu University 39, Kasuga 816, Japan.
‡‡ Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka 820, Japan.

Abstract

This paper describes a machine learning system that discovered a "negative motif", in transmembrane domain identification from amino acid sequences, and reports its experiments on protein data using PIR database. We introduce a decision tree whose nodes are labeled with regular patterns. As a hypothesis, the system produces such a decision tree for a small number of randomly chosen positive and negative examples from PIR. Experiments show that our system finds reasonable hypotheses very successfully. As a theoretical foundation, we show that the class of languages defined by decision trees of depth at most d over k-variable regular patterns is polynomial-time learnable in the sense of probably approximately correct (PAC) learning for any fixed d, $k \geq 0$.

1 Introduction

Hydrophobic transmembrane domains can be identified by a very simple decision tree over regular patterns. This result was discovered by the machine learning system we developed. The system takes some training sequences of positive and negative examples, and produces a hypothesis explaining them. When a small number of positive and negative examples of transmembrane domains were given as input, our system found a small decision tree over regular patterns as a hypothesis. Although the hypothesis is made from just 10 positive and 10 negative examples, it can explain all data in PIR database [PIR] with high accuracy more than 90%. The hypothesis exhibits that "two consecutive polar amino acids" (Arg, Lys, His, Asp, Glu, Gln, Asn) are *not* included in the transmembrane domains. This indicates that significant motifs are not in the inside of the transmembrane domains but in the *outside*. We call such motifs "negative motifs."

This paper describes a machine learning system together with a background theory that discovered such negative motifs, and reports its experiments on knowledge acquisition from amino acid sequences that reveal the importance of negative data. Traditional approaches to motif-searching are to find subsequences common to functional domains by various alignment techniques. Hence the eyes are focused only on positive examples, and negative examples are mostly ignored. Our approach by decision trees over regular patterns provides new direction and method for discovering motifs.

A regular pattern [Shinohara 1982, Shinohara 1983] is an expression $w_0x_1w_1x_2\cdots x_nw_n$ that defines the sequences containing $w_0, w_1, ..., w_n$ in this order, where each w_i is a sequence of symbols and x_j varies over arbitrary sequences. Regular patterns have been used to describe some features of amino acid sequences in PROSITE database [Bairoch 1991] and DNA sequences [Arikawa *et al.* 1992, Gusev and Chuzhanova 1990]. Our view to these sequences is through such regular patterns. A decision tree over regular patterns is a tree which describes a decision procedure for determining the class of a given sequence. Each node is labeled with either a class name (1 or 0) or a regular pattern. At a node with a regular pattern, the decision tree tests if the sequence matches the pattern or not. Starting from the root toward a leaf, the decision procedure makes a test at each node and goes down by choosing the left or right branch according to the test result. The reached leaf answers the class name of the sequence. Such decision trees are produced as hypotheses by our machine learning system. Since the system searches a decision tree of smaller size, regular patterns on the resulting decision tree exhibit motifs which play a significant role in classification. Hence, compared with neural network approaches [Holly and Karplus 1989, Wu *et al.*], our system shows important motifs in a hypothesis more explicitly.

Email addresses:
arikawa@rifis.sci.kyushu-u.ac.jp
kuhara@grt.kyushu-u.ac.jp
miyano@rifis.sci.kyushu-u.ac.jp
mukouchi@rifis.sci.kyushu-u.ac.jp
ayumi@rifis.sci.kyushu-u.ac.jp
shino@donald.ai.kyutech.ac.jp

We employ the idea of ID3 algorithm [Quinlan 1986, Utgoff 1989] for constructing a decision tree since it is sufficiently fast and experiments show that small enough trees are usually obtained. We also devise a new method for constructing a decision tree over regular patterns using another evaluation function. Given two sets of positive and negative examples, our machine learning system finds appropriate regular patterns as node attributes dynamically during the construction of the decision tree. Hence, unlike ID3, we need not assume any concrete knowledge about attributes and can avoid struggles from defining the attributes of a decision tree beforehand. Our system makes a decision tree just from a small number of training sequences, which we also guarantee with the PAC learning theory [Valiant 1984] in some sense. Therefore it may cope with a diversity of classification problems for proteins and DNA sequences.

We made an experiment on raw sequences from twenty symbols of amino acid residues. The system discovered a small decision tree just from 20 sequences with more than 85% accuracy that show if a sequence contains neither E nor D (both are polar amino acids) then it is very likely to be a transmembrane domain.

A hydropathy plot [Engelman *et al.* 1986, Kyte and Doolittle 1982, Rao and Argos 1986] has been used generally to predict transmembrane domains from primary sequences. With this knowledge, we first transform twenty amino acids to three categories (*, +, -) according to the hydropathy index of Kyte and Doolittle [1982]. From randomly chosen 10 positive and 10 negative training examples, our system has successfully produced some small decision trees over regular patterns which are shown to achieve very high accuracy. The regular patterns appearing in these decision trees indicate that two consecutive polar amino acid residues are important negative motifs for transmembrane domains. From the view point of Artificial Intelligence, it is quite interesting that the polar amino acid residues D and E were found by our machine learning system without any knowledge on the hydropathy index.

After knowing the importance of negative motifs, we examined decision trees with a single node with regular patterns $x_1\text{-}x_2\text{-}\cdots\text{-}x_n$ for $n \geq 3$. The best is the pattern $x_1\text{-}x_2\text{-}x_3\text{-}x_4\text{-}x_5\text{-}x_6$ that gives the sequences containing at least five polar amino acids. The result is very acceptable. The accuracy is 95.4% for positive and 95.0% for negative examples although it has been believed to be difficult to define transmembrane domains as a simple expression when the view point was focussed on positive examples.

2 Decision Trees over Regular Patterns

Let Σ be a finite alphabet and $X = \{x, y, z, x_1, x_2, \cdots\}$ be a set of variables. We assume that Σ and X are disjoint. A *pattern* is an element of $(\Sigma \cup X)^+$, the set of all nonempty strings over $\Sigma \cup X$. For a pattern π, the language $L(\pi)$ is the set of strings obtained by substituting each variable in π for a string in Σ^*. We say that a pattern π is *regular* if each variable occurs at most once in π. For example, $xaybza$ is a regular pattern, but xx is not. Obviously, regular patterns define regular languages, but not vice versa. In this paper we consider only regular patterns. A regular pattern containing at most k variables is called a *k-variable regular pattern.*

A *decision tree over regular patterns* is a binary tree such that the leaves are labeled with 0 or 1 and each internal node is labeled with a regular pattern (see Figure 1). For an internal node v, we denote the left and right children of v by left(v) and right(v), respectively. We denote by $\pi(v)$ the regular pattern assigned to the internal node v. For a leaf u, value(u) denotes the value 0 or 1 assigned to u. The depth of a tree T, denoted by depth(T), is the length of the longest path from the root to a leaf.

For a decision tree T over regular patterns, we define a function $f_T : \Sigma^* \to \{0, 1\}$ as follows. For a string w in Σ^*, we determine a path from the root to a leaf and define the value $f_T(w)$ by the following algorithm:

begin /* Input: $w \in \Sigma^*$ */
  $v \leftarrow$ root;
  while v is not a leaf **do**
    if $w \in L(\pi(v))$ **then** $v \leftarrow$right(v)
      else $v \leftarrow$left(v);
  $f_T(w) \leftarrow$ value(v)
end

For a decision tree T over regular patterns, we define $L(T) = \{w \in \Sigma^* \mid f_T(w) = 1\}$. It is easy to see that $L(T)$ is also a regular language. But the converse is not true. Let $L = \{a^{2n} \mid n \geq 1\}$. It is straightforward to show that there is no decision tree T over regular patterns with $L = L(T)$. The same holds for the language $\{a^{2n}b \mid n \geq 1\}$.

3 Constructing Decision Trees

This section gives two kinds of algorithms for constructing decision trees over regular patterns that are used in our machine learning system.

The first algorithm employs the idea of ID3 algorithm [Quinlan 1986] in the construction of decision trees. The ID3 algorithm assumes data together with explicit attributes in advance. On the other hand, our approach assumes a space of regular patterns which are simply generated by given positive and negative examples. No

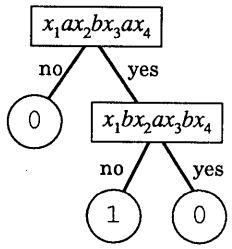


Figure 1: Decision tree over regular patterns defining a language $\{a^m b^n a^l \mid m, n, l \geq 1\}$ over $\Sigma = \{a, b\}$

extra knowledge about data is required. Although the space may be large and contain meaningless attributes, our algorithm finds appropriate regular patterns from this space dynamically during the construction of a decision tree in a feasible amount of time. This is a point which is very suited for our empirical research.

Let P and N be finite sets of strings with $P \cap N = \emptyset$. Using P and N, we deal with regular patterns of the form $w_0 x_1 w_1 x_2 \cdots x_k w_k$ such that $w_0, ..., w_k$ are substrings of some strings in $P \cup N$. Let $\Pi(P, N)$ be some family of such regular patterns made from P and N. The family $\Pi(P, N)$ is appropriately given and used as a space of attributes.

For a regular pattern $\pi \in \Pi(P, N)$, the cost $E(\pi, P, N)$ is the one defined in [Quinlan 1986] by

$$E(\pi, P, N) = \frac{p_1 + n_1}{|P| + |N|} I(p_1, n_1) + \frac{p_0 + n_0}{|P| + |N|} I(p_0, n_0),$$

where p_1 (resp. n_1) is the number of positive examples in P (resp. negative examples in N) that match π, i.e., $p_1 = |P \cap L(\pi)|$, $n_1 = |N \cap L(\pi)|$, and p_0 (resp. n_0) is the number of positive examples in P (resp. negative examples in N) that do not match π, i.e., $p_0 = |P \cap \overline{L(\pi)}|$, $n_0 = |N \cap \overline{L(\pi)}|$, $\overline{L(\pi)} = \Sigma^* - L(\pi)$, and

$$I(x, y) = \begin{cases} 0 & (\text{if } x = 0 \text{ or } y = 0) \\ -\frac{x}{x+y} \log \frac{x}{x+y} - \frac{y}{x+y} \log \frac{y}{x+y} & (\text{otherwise}). \end{cases}$$

The first algorithm DT1(P, N) (Algorithm 1) sketches our decision tree algorithm for $\Pi(P, N)$, where CREATE(π, T_0, T_1) returns a new tree with a root labeled with π whose left and right subtrees are T_0 and T_1, respectively.

The second algorithm uses a different evaluation function. For a decision tree T over regular patterns, let nodes(T) be the number of nodes in T, and $\mathcal{T}(T)$ be the set of trees constructed by replacing a leaf v of T by the tree of Fig. 2 (a) or Fig. 2 (b) for some pattern π.

The score function $Score(T, P, N)$ balances the information gains in classification and is defined as

$$Score(T, P, N) = \frac{|P \cap L(T)|}{|P|} \cdot \frac{|N \cap \overline{L(T)}|}{|N|}.$$

The second algorithm DT2$(P, N, MaxNode)$ (Algorithm 2) checks all leaves at each phase of a node generation using the evaluation function $Score(T, P, N)$.

Algorithm 2 is slower than Algorithm 1 since all leaves are checked at each phase of a node generation. However,

```
function DT1 ( P, N : sets of strings ): node;
 begin
   if N = ∅ then
     return( CREATE("1", null, null) )
   else if P = ∅ then
     return( CREATE("0", null, null) )
   else begin
     Find a shortest pattern π in Π(P, N)
     that minimizes E(π, P, N);
     P1 ← P ∩ L(π);   P0 ← P − P1;
     N1 ← N ∩ L(π);   N0 ← N − N1;
     return(CREATE(π,DT1(P0, N0),DT1(P1, N1)))
     end
 end
```

Algorithm 1

Figure 2: A leaf is replaced by (a) or (b) for some pattern π.

```
function DT2( P, N: sets of strings,
              MaxNode: int      ) : tree;
 begin
    if N = ∅ then
       return( CREATE("1", null, null) )
    else if P = ∅ then
       return( CREATE("0", null, null) )
    else begin
       T ←CREATE("1", null, null);
       while ( nodes(T) < MaxNode
          and Score(T, P, N) < 1 ) do
          begin
             find T_max ∈ 𝒯(T)
               that maximizes Score(T_max, P, N);
             T ← T_max
          end
    end
    return ( T )
 end
```

Algorithm 2

Algorithm 2 constructs decision trees which are finely tuned when the size of decision trees is large. Moreover, it is noise-tolerant, i.e., it allows conflicts between positive and negative training examples. If the size of $\Pi(P, N)$ is polynomial with respect to the size of P and N, then these algorithms run in polynomial time.

4 Transmembrane Domain Identification

The problem of transmembrane domain identification is one of the most important protein classification problems and some methods and experiments have been reported. For example, Hartman et al. [1989] proposed a method using the hydropathy index for amino acid residues in [Kyte and Doolittle 1982]. The reported success rate is about 75%. Most approaches deal with positive examples, i.e., sequences corresponding to transmembrane domains, and try to find properties common to them.

The sequence in Figure 3 is an amino acid sequence of a membrane protein. There is a tendency to assume that a membrane protein contains several transmembrane domains each of which consists of $20 \sim 30$ amino acid residues. Therefore, if a sequence corresponding to a transmembrane domain is found in an amino acid sequence, it is very likely that the protein is a membrane protein.

Our idea for transmembrane domain identification is to use decision trees over regular patterns for classification. Algorithm 1 and 2 introduced in Section 3 are used to find good decision trees from positive and negative training examples. In order to avoid combinatorial explosion, we restrict the space of attributes to the regular patterns of the form

$$x\alpha y,$$

where x and y are variables and α is a substring taken from given examples.

In our experiments, a *positive example* is a sequence which is already known to be a transmembrane domain. A *negative example* is a sequence of length around 30 cut out from the parts other than transmembrane domains. The length 30 is simply due to the reasonable length of a transmembrane domain. From PIR database our machine learning system chooses randomly two small sets P and N of positive and negative training examples, respectively. Then, at each trial, by using Algorithm 1 or Algorithm 2, the system tries to construct a small decision tree over regular patterns which classifies P and N exactly.

We have evaluated the performance ratio of a produced decision tree in the following way. As the total space of positive examples, we use the set POS of all transmembrane domain sequences (689 sequences) from PIR database. The total space NEG of negative examples consists of 19256 negative examples randomly chosen from *all* proteins from PIR. The success rate of a decision tree for positive examples is the percentage of the positive examples from POS recognized as positive (class 1). The success rate for negative examples is counted as the percentage of the negative examples from NEG recognized as negative (class 0).

Figure 5 (a) is one of the smallest decision trees discovered by our system just from 10 positive and 10 negative raw sequences that achieve good accuracy. The performance ratio is $(84.8\%, 89.6\%)$ for all data in POS and NEG, respectively. This decision tree suggests that if a sequence of length around 30 contains neither `D` not `E` then it is very likely to be a part of transmembrane domain.

The alphabet of amino acid sequences consists of twenty symbols. It has been shown that the use of the hydropathy index for amino acids is very successful [Arikawa *et al.* 1992, Hartmann *et al.* 1989]. According to the hydropathy index of [Kyte and Doolittle 1982], we transform these twenty symbols to three symbols as shown in Table 1. This transformation reduces the size of a search space drastically small while less information is, fortunately, lost in classification.

Then by this transformation table, the sequence in Figure 3 becomes the sequence in Figure 4.

Figure 5 (b), (c) show two of the best decision trees over regular patterns that our machine learning system found from 10 positive and 10 negative training examples. The decision tree (b) recognizes 91.4% of positive examples and 94.8% of negative examples. Even the decision tree of (c) can recognize 92.6% of the positive examples and 91.6% of the negative examples. The negative motif "`--`" which indicates consecutive polar amino acid residues plays a key role in classification. This may have a close relation to the signal-anchor structure that consists of two parts, the hydrophobic part of a membrane-spanning sequence and the charged residues around the hydrophobic part [Lipp *et al.* 1989, Von Heijine 1988].

The decision tree (a) also shows the importance of a cluster of polar amino acids in transmembrane domain identification although our machine learning system assumed no knowledge about the hydropathy.

We examined how the performance of our machine learning system changes with respect to the number of training examples. The training examples are chosen randomly ten times in each case and a point of the graph of Figure 6 is the average of these ten results for each case. Figure 6 shows the results. We may observe the following facts:

1. The hydropathy index of Kyte and Doolittle [Kyte and Doolittle 1982] is very useful. When indexed sequences are used, the system can produce from 40 positive and 40 negative examples a decision tree with only several nodes whose accuracy is more

```
MDVVNQLVAGGQFRVVKE(PLGFVKVLQWVFAIFAFATCGSY)TGELRLSVECANKTESALNIEVEFEYPFRLHQVYFDA
PSCVKGGTTKIFLVGDYSSSAE(FFVTVAVFAFLYSMGALATYIFL)QNKYRENNK(GPMMDFLATAVFAFMWLVSSSAW
A)KGLSDVKMATDPENIIKEMPMCRQTGNTCKELRDPVTS(GLNTSVVFGFLNLVLWVGNLWFVF)KETGWAAPFMRAPP
GAPEKQPAPGDAYGDAGYGQGPGGYGPQDSYGPQGGYQPDYGQPASGGGGYGPQGDYGQQGYGQQGAPTSFSNQM
```

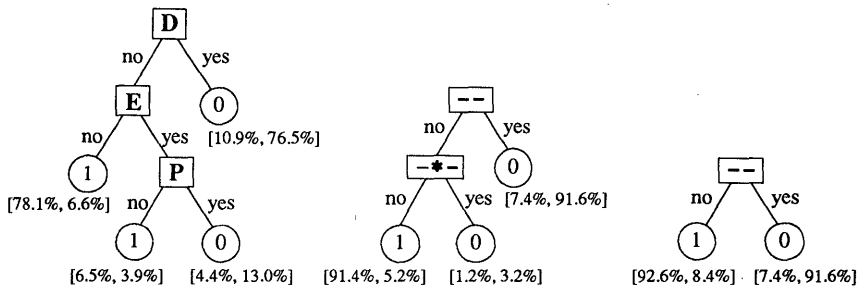
Figure 3: An amino acid sequence which contains four transmembrane domains shown by the parenthesized parts.

Amino Acids	Hydropathy Index		New Symbol
A M C F L V I	$1.8 \sim 4.5$	$\rightarrow$	*
P Y W S T G	$-1.6 \sim -0.4$	$\rightarrow$	+
R K D E N Q H	$-4.5 \sim -3.2$	$\rightarrow$	-

Table 1: Transformation rules

```
*-**--*****-*-**--(+****-**-+**************)++-*-***-**--+-***-*-*-*-+**-*--***-*
++**-++++-*****-+++++-(**********************)---+-----(++**-*****************+**+
)-+**-*-***+-+--**--****--++-+*--*--++++(+*-++******-******+-*****)--++******-***
+**---++++-***-****-+++++++--++++-+++-+-++-++++++++++-+-++--+++--+******--*
```

Figure 4: The sequence obtained by the transformation



(a) (84.8%, 89.6%) (b) (91.4%, 94.8%) (c) (92.6%, 91.6%)

Figure 5: The node label, for example, -- is an abbreviation of x_1--x_2 that tests if a given sequence contains the sequence --. The leaf label 1 (resp. 0) is the class name of transmembrane domains (resp. non-transmembrane domains). The total space consists of 689 positive examples and 19256 negative examples. Each of the decision trees (a)-(c) is constructed from 10 positive and 10 negative training sequences. The pair $[p\%, n\%]$ attached to a leaf shows that $p\%$ of positive examples and $n\%$ of negative examples have reached to the leaf. The pair $(p\%, n\%)$ means that $p\%$ of 689 positive (resp. $n\%$ of 19256 negative) examples are recognized as transmembrane domains (resp. non-transmembrane domains).

than 90% for the total space in average. On the other hand, for raw sequences the accuracy is not so good but both accuracies approach to the same line as the number of training examples increases.

2. The number of nodes of a decision tree is reasonably small. But when the number of training examples is larger, the number of nodes in a decision tree becomes larger while the accuracy is not improved very much. There may arise the problem of overfitting.

A new discovery obtained from these decision trees is that the motif "--" drastically rejects positive examples. After knowing the negative motif "--", we have examined the decision trees with a single node with the patterns of the form

$$x_1\text{-}x_2\text{-}\cdots\text{-}x_n$$

for $n \geq 3$. The best is the pattern containing "-" five times. The result is quite acceptable as shown in Table 2.

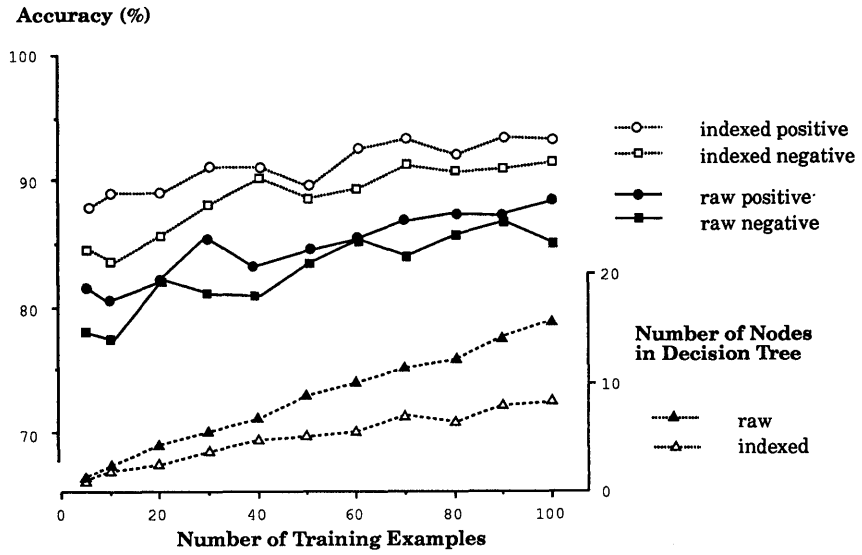


Figure 6: Relations between the number of training examples, accuracy and the number of nodes in a decision tree

Pattern	*POS* (689)	*NEG* (19256)
x_1-x_2-x_3-x_4-x_5-x_6	657 (95.4 %)	18296 (95.0%)

Table 2: Result for x_1-x_2-x_3-x_4-x_5-x_6

With these decision trees over regular patterns, we have developed a transmembrane domain predictor that reads an amino acid sequence of a protein as an input and predicts symbol by symbol whether each location of a symbol is in a transmembrane domain or not. Experiments on all protein sequences in PIR show that the success rate is 85% $\sim$ 90%.

5 PAC-Learnable Class

This section provides a theoretical foundation on the classes of sets classified by decision trees over regular patterns from the point of algorithmic learning theory [Valiant 1984].

For integers $k, d \geq 0$, we consider a decision tree T over k-variable regular patterns whose depth is at most d. We denote by DTRP(d, k) the class of languages defined by decision trees over k-variable regular patterns with depth at most d.

Theorem 1 DTRP(d, k) is polynomial-time learnable for all $d, k \geq 0$.

We need some terminology for the above theorem. When we are concerned with learning, we call a subset of Σ^* a *concept*. A *concept class* $\mathcal{C}$ is a nonempty collection of concepts. For a concept $c \in \mathcal{C}$, a pair $\langle x, c(x)\rangle$ is called an example of c for $x \in \Sigma^*$,where $c(x) = 1$ ($c(x) = 0$) if x is in c (is not in c). For an alphabet Σ and an integer $n \geq 0$, $\Sigma^{\leq n}$ denotes the set $\{x \in \Sigma^* \mid |x| \leq n\}$.

A concept class $\mathcal{C}$ is said to be *polynomial-time learnable* [Blumer *et al.* 1989, Natarajan 1989, Valiant 1984] if there is an algorithm $\mathcal{A}$ which satisfies (1) and (2).

(1) $\mathcal{A}$ takes a sequence of examples as an input and runs in polynomial-time with respect to the length of input.

(2) There exists a polynomial $p(\cdot,\cdot,\cdot)$ such that for any integer $n \geq 0$, any concept $c \in \mathcal{C}$, any real number ε, δ $(0 < \varepsilon, \delta < 1)$, and any probability distribution P on $\Sigma^{\leq n}$, if $\mathcal{A}$ takes $p(n, \frac{1}{\varepsilon}, \frac{1}{\delta})$ examples which are generated randomly according to P, then $\mathcal{A}$ outputs, with probability at least $1 - \delta$, a representation of a hypothesis h with $P(c \oplus h) < \varepsilon$.

Theorem 2 [Blumer *et al.* 1989, Natarajan 1989] A concept class $\mathcal{C}$ is polynomial-time learnable if the following conditions hold.

1. $\mathcal{C}$ is of *polynomial dimension*, i.e., there is a polynomial $d(n)$ such that $|\{c \cap \Sigma^{\leq n} \mid c \in \mathcal{C}\}| \leq 2^{d(n)}$ for all $n \geq 0$.

2. There is a polynomial-time algorithm called a *polynomial-time hypothesis finder* for $\mathcal{C}$ which produces a hypothesis from a sequence of examples such that it is consistent with the given examples.

Moreover, the polynomial-time hypothesis finder for $\mathcal{C}$ is a learning algorithm satisfying (1) and (2) if $\mathcal{C}$ satisfies 1.

The following lemma can be easily shown.

Lemma 1 Let T be a decision tree over regular patterns and T_v be a subtree of T at node v. We denote T_v by $\pi(T_0, T_1)$, where π is the label of node v and T_0, T_1 are the left and right subtrees of T_v, respectively. Let S be a set of strings and let T' be the tree obtained from T by replacing T_v with T_0 at node v. If no string in S matches π, then $L(T) \cap S = L(T') \cap S$.

Proof of Theorem 1.

First we show that the concept class DTRP(d, k) is of polynomial dimension. Let DTRP$(d, k)_n = \{L \cap \Sigma^{\leq n} \mid L \in \text{DTRP}(d, k)\}$ for $n \geq 0$. We evaluate the cardinality of DTRP$(d, k)_n$. Let π be a regular pattern with $|\pi| > n + k$, then no string of length at most n matches π. By Lemma 1, we need to consider only regular patterns of length at most $n + k$. The number of such patterns is roughly bounded by $(|\Sigma| + 1)^{n+k}$. Since a tree of depth bounded by d has at most $2^d - 1$ internal nodes and at most 2^d leaves, $|\text{DTRP}(d, k)_n| \leq ((|\Sigma| + 1)^{n+k})^{2^d-1} \cdot 2^{2^d}$ This shows that the dimension of DTRP$(d, k)_n$ is $O(n)$.

Next we show that there is a polynomial-time hypothesis finder for DTRP(d, k). Let P and N be the sets of strings which appear in positive and negative examples, respectively. Let $\Pi(k, P, N)$ be the set of regular patterns π up to renaming of variables such that it contains at most k variable occurrences and $\pi\theta$ is a substring of some s in $P \cup N$. By Lemma 1, we need to consider only patterns in $\Pi(k, P, N)$ in order to find a decision tree over regular patterns which is consistent with P and N. Then $|\Pi(k, P, N)| \leq \sum_{s \in P \cup N}((|s|^2)^{k+1})$. Therefore the number of possible trees is bounded by $(|\Pi(k, P, N)|)^{2^d-1} \cdot 2^{2^d}$, which is bounded by a polynomial with respect to the input length $\sum_{s \in P \cup N} |s|$.

It is known that, given a regular pattern π and string w, we can decide in polynomial time whether w matches π or not. Therefore, given a string w and a decision tree T over k-variable regular patterns whose depth is at most d, we can decide whether $w \in L(T)$ or not in polynomial-time.

The required polynomial-time algorithm enumerates decision trees T over regular patterns in $\Pi(k, P, N)$ with depth at most d. Then it checks whether $s \in L(T)$ for each $s \in P$ and $t \notin N$ for each $t \in N$. If such a tree T is found, the algorithm outputs T as a hypothesis. □

We should say that the polynomial-time learning algorithm in the proof of Theorem 1 exhausts an enormous amount of time and is not suited for practical use.

We may understand the relationship of Algorithms 1 and 2 in Section 3 to Theorem 1 in the following way: When we set $\Pi(P, N)$ to be the family of k-variable regular patterns made from P and N, Algorithms 1 and 2 run sufficiently fast in practical use (of course, in polynomial-time) and produce a decision tree over k-variable regular patterns which classifies given positive and negative examples. But the produced decision tree is not guaranteed to be of depth at most d. Hence, these algorithms are not any learning algorithm in the exact sense of (2).

However, experiences tell that these algorithms usually find small enough decision trees over regular patterns in our experiments on transmembrane domains. For the class DTRP(d, k), Theorem 2 asserts that if a polynomial-time algorithm $\mathcal{A}$ produces a decision tree over k-variable regular patterns with depth at most d which classifies given positive and negative examples then it is a polynomial-time learning algorithm. In this sense, we may say that Algorithms 1 and 2 are polynomial-time algorithms for DTRP(d, k) which *often* produce reasonable hypotheses although there is no mathematical proof showing how often such small hypotheses are obtained. This aspect is very important and useful when we are concerned with machine discovery.

Ehrenfeucht and Haussler [1989] have considered learning of decision trees of a fixed rank. For learning decision trees over regular patterns, the restriction by rank can be shown to have no sense. Instead, we consider the depth of a decision tree. It is also reasonable to put a restriction on the number of variables in a regular pattern. It has been shown that the class of regular pattern languages is not polynomial-time learnable unless **NP** $\neq$ **RP** [Miyano *et al.* 1991]. Therefore, unless restrictions such as bound on the number of variables in a regular pattern are given, we may not expect any positive results for polynomial-time learning.

6 Conclusion

We have shown that the idea of combining regular patterns and decision trees works quite well for transmembrane domain identification. The experiments also have shown the importance of negative motifs.

A union of regular patterns is regarded as a special form of a decision tree called a decision list. We have reported in [Arikawa *et al.* 1992] that the union of small number of regular patterns can also recognize transmembrane domains with high accuracy. However, the time exhausted in finding hypotheses in [Arikawa *et al.* 1992] is much larger than that reported in this paper.

Our system constructs a decision tree over regular patterns just from strings called positive and negative examples. We need not take care of which attributes to specify as in ID3. Therefore it can be applied to another classification problems for proteins and DNA sequences. We believe that our approach provides a new application

of algorithmic learning to Molecular Biology.

We are now in the process of examining our method for some other related problems such as predicting the secondary structure of proteins.

References

[Arikawa *et al.* 1992] S. Arikawa, S. Kuhara, S. Miyano, A. Shinohara and T. Shinohara. A Learning Algorithm for Elementary Formal Systems and its Experiments on Identification of Transmembrane Domains. In *Proc. 25th Hawaii Int. Conf. on Sys. Sci*, IEEE, Hawaii, 1992. pp. 675–684.

[Bairoch 1991] A. Bairoch, PROSITE: A Dictionary of Sites and Patterns in Proteins, sl Nucleic Acids Res., Vol. 19 (1991), pp. 2241–2245.

[Blumer *et al.* 1989] A. Blumer, A. Ehrenfeucht, D. Haussler and M.K. Warmuth. Learnability and the Vapnik-Chervonenkis Dimension. *JACM*, Vol. 36 (1989), pp. 929–965.

[Ehrenfeucht and Haussler 1989] A. Ehrenfeucht and D. Haussler. Learning Decision Trees from Random Examples. *Inform. Comput.*, Vol. 82 (1989), pp. 231–246.

[Engelman *et al.* 1986] D.M. Engelman, T.A. Steiz and A. Goldman. Identifying Nonpolar Transbilayer Helices in Amino Acid Sequences of Membrane Proteins. *Ann. Rev. Biophys. Biophys. Chem.*, Vol. 15 (1986), pp. 321–353.

[Gusev and Chuzhanova 1990] V. Gusev, N. Chuzhanova. The Algorithms for Recognition of the Functional Sites in Genetic Texts. In *Proc. 1st Workshop on Algorithmic Learning Theory*, Tokyo, 1990. pp. 109–119.

[Hartmann *et al.* 1989] E. Hartmann, T.A. Rapoport and H.F. Lodish. Predicting the Orientation of Eukaryotic Membrane-Spanning Proteins. In *Proc. Natl. Acad. Sci. U.S.A.*, Vol. 86 (1989), pp. 5786–5790.

[Holly and Karplus 1989] L.H. Holly and M. Karplus. Protein Secondary Structure Prediction with a Neural Network, In *Proc. Natl. Acad. Sci. USA*, Vol. 86 (1989), pp. 152–156.

[Kyte and Doolittle 1982] J. Kyte and R.F. Doolittle. A Simple Method for Displaying the Hydropathic Character of Protein. *J. Mol. Biol.*, Vol. 157 (1982), pp. 105–132.

[Lipp *et al.* 1989] J. Lipp, N. Flint, M.T. Haeuptle and B. Dobberstein. Structural Requirements for Membrane Assembly of Proteins Spanning the Membrane Several Times. *J. Cell Biol.*, Vol. 109 (1989), pp. 2013–2022.

[Miyano *et al.* 1991] S. Miyano, A. Shinohara and T. Shinohara. Which Classes of Elementary Formal Systems are Polynomial-Time Learnable? In *Proc. 2nd Algorithmic Learning Theory*, Tokyo, 1991. pp. 139–150.

[Natarajan 1989] B.K. Natarajan, On Learning Sets and Functions. *Machine Learning*, Vol. 4 (1989), pp. 67–97.

[PIR] Protein Identification Resource, National Biomedical Research Foundation.

[Quinlan 1986] J.R. Quinlan, Induction of Decision Trees. *Machine Learning*, Vol. 1 (1986), pp. 81–106.

[Quinlan and Rivest 1989] J.R. Quinlan and R. L. Rivest. Inferring Decision Trees using the Minimum Description Length Principle. *Inform. Comput.*, Vol. 80 (1989), pp. 227–248.

[Rao and Argos 1986] J.K.M. Rao and P. Argos. A Confirmational Preference Parameter to Predict Helices in Integral Membrane Proteins. *Biochim. Biophys. Acta*, Vol. 869 (1986), pp. 197–214.

[Shinohara 1982] T. Shinohara. Polynomial Time Inference of Pattern Languages and its Applications. In *Proc. 7th IBM Symp. Mathematical Foundations of Computer Science*, 1982. pp. 191–209.

[Shinohara 1983] T. Shinohara. Polynomial Time Inference of Regular Pattern Languages. In *Proc. RIMS Symp. Software Science and Engineering* (Lecture Notes in Computer Science, Vol. 147), 1983. pp. 115–127.

[Utgoff 1989] P.E. Utgoff. Incremental Induction of Decision Tree. *Machine Learning*, Vol. 4 (1989), pp. 161–186.

[Valiant 1984] L. Valiant. A Theory of the Learnable. *Commun. ACM*, Vol. 27 (1984), pp. 1134–1142.

[Von Heijine 1988] G. von Heijine. Transcending the Impenetrable: How Proteins Come to Terms with Membranes. *Biochim. Biophys. Acta*, Vol. 947 (1988), pp. 307–333.

[Wu *et al.*] C.H. Wu, G.M. Whiston and G.J. Montllor. PROCANS: A Protein Classification System Using a Neural Network, *IJCNN Int. Joint Conf. Neural Networks*, Vol. 2 (1990), pp. 91-96.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Efficient Induction of Version Spaces through Constrained Language Shift

Claudio Carpineto

Fondazione Ugo Bordoni
Via Baldassarre Castiglione 59, 00142 - Rome, ITALY
fubdpt5@itcaspur.bitnet

Abstract

A large hypothesis space makes the version space approach, like any other concept induction algorithm based on hypothesis ordering, computationally inefficient. Working with smaller composable concept languages rather than one large concept language is one way to attack the problem, in that it allows us to do part of the induction job within the more convenient languages and move to the less convenient languages when necessary. In this paper we investigate the use of multiple concept languages in a version space approach. We define a graph of languages ordered by the standard set inclusion relation, and provide a procedure for efficiently inducing version spaces while shifting from small to larger concept languages. We apply this method to the attribute languages of a typical conjunctive concept language (i.e., a conjunctive concept language defined on a tree-structured attribute-based instance space) and compare its complexity to that of a standard version space algorithm applied to the full concept language. Finally we contrast our approach with other work on language shift, outlining an alternative highly-constrained strategy for searching the space of new concepts which is *not* based on constructive operators.

1 Introduction

Of all the algorithms for incremental concept induction that are based on the partial order defined by generality over the concept space, the candidate elimination (CE) algorithm [Mitchell 1982] is the best known exemplar. The CE algorithm represents and updates the set of all concepts that are consistent with data (i.e. the version space) by maintaining two sets, the set S containing the maximally specific concepts and the set G containing the maximally general concepts. The procedure to update the version space is as follows. A positive example prunes concepts in G which do not cover it and causes all concepts in S which do not cover the example to be generalized just enough to cover it. A negative example prunes concepts in S that cover it and causes all concepts in G that cover the example to be specialized just enough to exclude it. As more examples are seen, the version space shrinks; it may eventually reduce to the target concept provided that the concept description language is consistent with the data.

This framework has been later improved along several directions. The first is that of incorporating the domain knowledge available to the system in the algorithm; this has resulted in feeding the CE algorithm with analytically-generalized positive examples (e.g., [Hirsh 1989], [Carpineto 1990]), and analytically-generalized negative examples (e.g., [Carpineto 1991]). Another research direction is to relax the assumption about the consistency of the concept space with data. In fact, like many other learning algorithms, the CE algorithm uses a restricted concept language to incorporate bias and focus the search on a smaller number of hypotheses. The drawback is that the target concept may be contained in the set of concepts that are inexpressible in the given language, thus being unlearnable. In this case the sets S and G become empty: to restore consistency the bias must be weakened adding new concepts to the concept language [Utgoff 1986]. Thirdly, the CE algorithm suffers from lack of computational efficiency, in that the size of S and G can be exponential in the number of examples and the number of parameters describing the examples [Haussler 1988]. Changes to the basic algorithm have been proposed that improve efficiency for some concept language [Smith and Rosenbloom 1990].

In this paper we investigate the use of *multiple* concept languages in a version space approach. By organizing the concept languages into a graph corresponding to the relation *larger-than* implicitly defined over the sets of concepts covered by the languages, we have a framework that allows us to shift from small to larger concept languages in a controlled manner. This provides a powerful basis to apply a general divide-and-conquer strategy to improve the efficiency of a standard version space approach in which the concept description language is factorizable. The idea is to start out with the smallest concept languages (i.e., the factor languages) and, once the version spaces induced over them have become inconsistent with the data, to move along the graph of product languages to the maximally small concept languages that restore consistency. Working with smaller concept languages may greatly reduce the size of S and G, thus resulting in a neat improvement in efficiency. On the other hand, use of several languages in parallel and language shifts negatively affect complexity. Therefore the two main objectives of the paper are : (1) define a set of languages and a procedure for inducing version spaces after any language shift efficiently, (2) show that in some cases this method may be applied to reduce the complexity of the standard CE algorithm. Since this framework supports version-space induction over a set of concept languages, it can also be suitable to handle inconsistency when the original concept language is too small. More generally, it suggest an alternative approach to inductive language shift in which the search for useful concepts to be added to the concept language is not based on constructive operators. This aspect is also discussed in the paper.

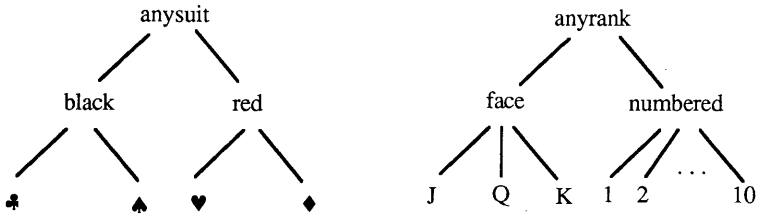


Fig. 1. Two concept languages in the playing cards domain.

The rest of the paper is organized as follows. In the next section we define a graph of conjunctive concept languages and describe the learning problem with respect to it. Then we present the learning method. Next, we apply the method to the factor languages of a conjunctive concept language defined on a tree-structured attribute-based instance space, and evaluate its utility. Finally we compare this work to other approaches to factorization in concept induction and to inductive language shift.

2 The learning problem

We first introduce the notions that characterize our learning problem. In the following concepts are viewed as sets of instances and languages as sets of concepts.

A concept c_1 is *more general than* a concept c_2 if the set of instances covered by c_1 is a proper superset of the set of instances covered by c_2.

A language $\mathbf{L}_1$ is *larger than* a language $\mathbf{L}_2$ if the set of concepts expressible in $\mathbf{L}_1$ is a proper superset of the set of concepts expressible in $\mathbf{L}_2$.

In the playing cards domain, which we shall use as an illustration, two possible concept languages are: $\mathbf{L}_1$ = {anysuit, black, red, ♣, ♠, ♥, ♦ } and $\mathbf{L}_2$ = {anyrank, face, numbered, J, Q, K, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. The relation more-general-than over the concepts present in each language is shown in fig1.

The *product* $\mathbf{L}_{1,2}$ of two factor languages $\mathbf{L}_1$ and $\mathbf{L}_2$ is the set of concepts formed from the conjunctions of concepts from $\mathbf{L}_1$ and $\mathbf{L}_2$ (examples of product concepts are 'anyrank-anysuit', 'anyrank-black', etc). The number of concepts in the product language is therefore the product of the number of concepts in its factors. Also, a concept $c_{c1',c2'}$ in the language $\mathbf{L}_{1,2}$ is more general than (>) another concept $c_{c1'',c2''}$ if and only if $c_1' > c_1''$ and $c_2' > c_2''$.

With n initial languages it is possible to generate $\sum_{k=1,n} n! / (n - k)! k! = 2^n - 1$ product languages (see fig. 2). Moreover, given that the superconcept 'any' can always be added to each factor language, the relation larger than over this set of languages can be immediately established, for each product language is larger than any of its factor languages.

The learning problem can be stated as follows.

Given
A set of factor concept languages
A set of positive instances.
A set of negative instances.

Incrementally Find
The version spaces in the set of product concept languages that are consistent with data and that contain the smallest number of factors.

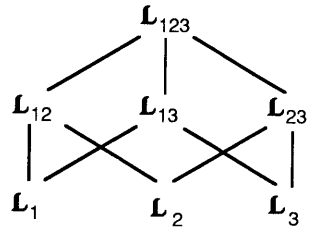


Fig. 2. The graph of product languages with three factor languages.

3. The learning method

In this approach concept learning and language shift are interleaved. We process one instance at a time, using a standard version space approach to induce consistent concepts over *each* language of the current set (initially, the n factor languages). During this inductive phase some concept languages may become inconsistent with the data. When every member of the current set of languages has become inconsistent with data, the language shifting algorithm is invoked. It iteratively selects the set of maximally small concept languages that are larger than the current ones (i.e. the two-factored languages, the three-factored languages, etc.) and computes the new version spaces in these languages. It halts when it finds a consistent set of concept languages (i.e. a set in which there is at least one consistent concept language); then it returns control to the inductive algorithm to process additional examples.The whole process is iterated as long as the set of current languages can be further specialised (i.e. until the n-factored language has been generated). We call this algorithm Factored Candidate Elimination (FCE) algorithm. The top-level FCE algorithm is presented in table 1.

The core of the algorithm is the procedure to find the new consistent version spaces in the product languages (in italics in table 1). The difficulty is that the algorithm for inducing concepts over a language (the inductive algorithm) is usually distinct from the algorithm for adding new terms to the language itself (the language-shifting algorithm). In

Table 1: The top-level FCE algorithm

Input:	An instance set {I}. A set of partially ordered concept languages {L} formed by n given one-factored languages and their products.
Output:	The version spaces in the set of languages {L} that are consistent with {I} and that contain the smallest number of factors.
Variables:	$\{L_S\}_k$ is the subset of (unordered) languages in {L} which have k factors. $\{VS\}_k$ is a set of version spaces, with $\|VS_k\| = \|L_{Sk}\|$. $\{L_S,VS\}_k$ is the set of pairs obtained pairing the corresponding elements in $\{L_S\}_k$ and $\{VS\}_k$.
Function:	CE(i,l,vs) takes an instance, a concept language and a version space and returns the updated version space.

```
FCE([I],{L})
    K=1.
    {VS}1 = {LS}1.
    For each instance i in {I},
        For each (ls,vs) in {LS,VS}k
            vs = CE(i,ls,vs).
        If all the version spaces in {VS}k are empty
            Then Repeat
                If K=n
                    Then Return failure
                K=K+1.
                For each ls in {LS}k,
                    find the new version space vs associated with it.
            Until at least one vs is not empty.
```

general, the inductive algorithm has to be run again over the instance set after any change made by the language-shifting algorithm ([Utgoff 1986], [Matheus and Rendell 1989], [Pagallo 1989], [Wogulis and Langley 1989]). In this case, however, in defining the procedure to induce the new consistent concepts after any language shift, we take advantage of the features of the particular inductive learning algorithm considered (i.e. the CE algorithm) and of the properties of language "multiplication". The two key facts are that the CE algorithm makes an explicit use of concept ordering and that concepts in any product language preserve the order of concepts in its factors. This makes it possible to modify the basic CE algorithm with the aim of computing the set of consistent concepts in a product language as a function of some appropriate concept sets induced in its factors.

The concept sets computed in each factor language which will be utilized during language shift are the following. First, for each language we compute the set S*. S* contains the most specific concepts in the language that cover *all* positive examples, regardless of whether or not they include any negative examples. Second, for each language *and each negative example*, we compute the set G*. G* contains the most general concepts in the language that do not cover the negative example, regardless of whether or not they include all positive examples.

These operations can be better illustrated with an example. Let us consider again the playing cards domain and suppose that we begin with the two concept languages introduced above - rank ($\mathbf{L}_1$) and suit ($\mathbf{L}_2$). Let us suppose the system is given one positive example - the Jack of spades - and two negative examples - the Jack of hearts and the Two of spades. We compute the two corresponding version spaces (one for each language), the sets S* (one for each language), and the sets G* (one for each language and for each negative example) in parallel. In particular, the sets S* and G* can be immediately determined, given the ordering over each language's members. The inductive phase is pictured in fig.3 (f stands for face, b for black,etc).

The three instances cause both of the version spaces to reduce to the empty set. The next step is therefore to shift to the set of maximally small concept languages that are larger than $\mathbf{L}_1$ and $\mathbf{L}_2$ (in this case the product $\mathbf{L}_{12}$) and check to see if it contains any concepts consistent with data. The problem of finding the version space in the language $\mathbf{L}_{12}$ can be subdivided into the two tasks of finding the lower boundary set S_{12} (i.e. the set of the most specific concepts in $\mathbf{L}_{12}$ that are consistent with data) and the upper boundary set G_{12} (i.e. the set of the most general concepts in $\mathbf{L}_{12}$ that are consistent with data).

Computation of S_{12}

Because a product concept contains an instance if and only if all of its factor concepts contain the instance, the product of S_1* and S_2* returns the most specific factor concepts that include all positive instances. By discarding those that also cover negative examples, we get just the set S_{12}. If the set becomes empty, then the product language is also inconsistent with the data. More specific concepts, in fact, cannot be consistent because they would rule out some positive example. More general concepts cannot be consistent either, for they would cover some negative examples. In our example, as there is only one positive example, the result is trivial : $S_{12} = \{J\spadesuit\}$.

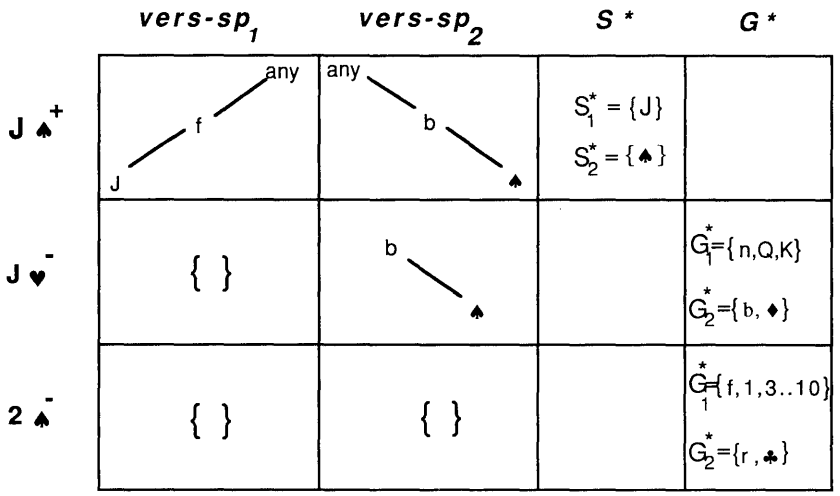


Fig. 3. Concept sets computed during the inductive phase.

Computation of G_{12}

Rather than generating and testing for consistency all the product concepts more general than the members of S_{12}, the set G_{12} is computed using the sets G* As for each negative example there must be *at least one* factor concept in each consistent product concept which does not cover the negative example, and because we seek the maximally general consistent product concepts, the idea is to use the members of the sets G* as upper bounds to find the factor concepts present in such maximally general product concepts.

The algorithm is as follows. It begins by dropping from the sets G* the elements that cannot generate factor concepts that are more general than those contained in S_{12}. Then, it (a) finds all the conjunctions of concepts in the reduced sets G* such that each negative instance is ruled out by at least one concept, and (b) checks if there are more general consistent conjunctions. Step (a) requires conjoining each factor concept in each G* (it will rule out at least one negative example) with all the combinations of factor concepts in the other G*'s which rule out the remaining negative examples. Step (b) requires generalising (with the value 'any') the factor concepts in the conjunctions found at the end of step (a) which do not contribute to rule out any negative example.The resulting set of conjunctions, if any is found, coincides with the set G_{12}, in that there cannot be more general product concepts consistent with data. However, it may not be possible to find a consistent concept conjoining the members of the G*'s. In this case we are forced to specialise the members of the G*'s to the extent required so that they rule out more negative instances, and to iterate the procedure (in the limit, we will get the set S_{12}).

In our example there are just two factors and only two negative instances. The initial sets G* are:

J ♥ -	{n, Q, K}	{b, ♦}
2 ♠ -	{f, 1, 3,.., 10}	{r, ♣}

The simplification with the set S_{12} returns:

J ♥ -	{ }	{b}
2 ♠ -	{f}	{ }

Step (a) in this case reduces to the *union of* the conjunction of G_1* relative to instance 1 and G_2* relative to instance 2 *and* the conjunction of G_1* relative to instance 2 and G_2* relative to instance 1. The result ({fb}) does not need be generalized (step (b)) for both 'f' and 'b' contribute to rule out (at least) one negative example. Also, in this case, the specialisation procedure is not needed because we have been able to find a consistent conjunction: G_{12}={fb}. The overall version space in the language $\mathbf{L}_{12}$ is shown in fig.4.

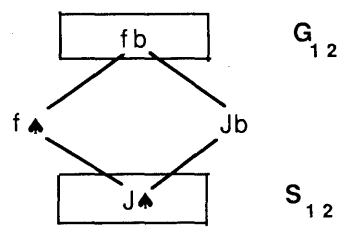


Fig.4. The version space in the product language after the constructive phase.

4 Evaluation

There are two ways in which the factored CE algorithm (FCE) can be used to reduce the complexity of the standard CE algorithm. Either we use a graph-factoring algorithm [Subramanian and Feigenbaum, 1986] to find the factors of a given concept space (provided that it is factorable), or we choose a concept language that can be naturally decomposed into factor languages. Here we evaluate the utility of the FCE algorithm with respect to a simple but widely used concept language that has this property. We consider a conjunctive concept language defined on a tree-structured attribute-based instance space. We assume the number of attributes be *n*, each attribute with *l* levels and branching factor *b* (the case can be easily extended to nominal and linear attributes, considering that a nominal attribute can be converted in a tree-structured attribute using a dummy root 'any-value', and that a linear attribute can be considered as a tree-structured attribute with branching factor = 1). Each term of the concept space is a conjunction of n values, one for each attribute; the total number of terms in the concept space is $[(b^l - 1) / (b - 1)]^n$. It is worth noting that with such a concept language the set S of the version space will never contain more than one element [Bundy *et al.* 1985]. Even in this case, however, Haussler [1988] has shown that the size of the set G can still be exponential, due to its fragmentation.

In the following we compare the CE algorithm applied to this full conjunctive concept language to the FCE algorithm applied to its attribute languages. While their relative performances are equivalent , in that in order to find all the concepts consistent with data in the full concept language it suffices to eventually compute the boundaries of the n-factored version space, their time complexity may strongly vary. The gain/loss in efficiency ultimately depends on the number of instances that each intermediate language is able to account for before it becomes inconsistent. In the best case all the induction is done within the smallest languages,and language shift to larger languages is not necessary. In the worst case no consistent concepts are induced in the smaller languages, so that all the induction is eventually done within the full concept language.

To make a quantitative assessment we have to make assumptions about a number of factors in addition to the structure of factor and product languages, including target concept location, training instance distribution, cost of matching concepts to training instances. We consider the *worst case* convergency to the target concept in the full concept language. This amounts to say that after the first positive instance (the first instance must be positive in the CE algorithm) there are only negative instances, and that each of them causes only one concept to be removed from the version space until it shrinks to the target concept (i.e., the first positive instance). In terms of the full concept language ordering this means that general concepts are removed earlier than any of their more specific concepts. Furthermore, we assume that the generality of the attribute values in the concepts dropped from the version space decreases uniformly. More precisely, we assume that if an attribute value in a dropped concept is placed at level k in the corresponding attribute tree, then the values of that attribute in the remaining consistent concepts are placed at most at level k+1. This presentation of training instances has the effect of maximizing the amount of instances that each intermediate language can take in before it becomes inconsistent.

As for the cost of matching concepts to instances and other concepts we assume that it is the same in all languages.

We can now analyse the complexity in the two approaches. As done in [Mitchell 1982], the time complexity bounds indicate bounds on the number of comparisons between concepts and instances, and comparisons between concepts.

CE algorithm with full conjunctive concept language. Let q be the number of negative instances, g the largest size of G. Following [Mitchell 1982], in our case the key term is $O(g^2q)$. The maximum size of G is given by the largest number of unordered concepts that can be found in the version space after the first positive instance. This number turns out to be $O(n^2l)$. To illustrate, first we must note that the version space after the first positive instance will contain the concepts more general than the instance, therefore the admissible values for each attribute will be the l values in the attribute tree that are placed in the chain linking the attribute value in the instance to the root of the attribute tree. When n = 2 there are at most l ways to choose a pair of values from two ordered sets of size l in such a way that the pairs are unordered. When n increases, this number comes to be multiplied by n / (n -2)! 2! . In fact, considering that two n-factored concepts are unordered if they contain at least two factor concepts with different orderings, all the possible unordered n-factored concepts can be obtained considering the same combinations as in the l original unordered concepts for each possible way of choosing a pair of attributes from among the n attributes. The maximum size of G is therefore $O(n^2l)$. The complexity of the CE algorithm is $O(n^4l^2q)$.

FCE algorithm with attribute languages. In this case several concept languages are active at once. For each negative instance we have to update in parallel at most $\max_k [n! / (n-k)!k!]$, that is $O(n^2)$, version spaces. Given our hypothesis on instance distribution, the g value of the intermediate version spaces will be 1 for the one-factored languages, 2 for the two-factored languages, .., n for the n-factored languages. The largest value of g is n, and the relative complexity factor for each version space is therefore $O(n^2)$. Thus the time taken to induce version spaces within the set of active languages is at most $O(n^2n^2q) = O(n^4q)$.

The total time complexity can be calculated adding the time taken by language shift to the time taken by concept induction alone. The cost of shifting the concept languages is given by the number of language shifts (2^n) multiplied by the cost of any single language shift. The time taken by any single language shift becomes constant if we modify the FCE algorithm's inductive phase by labelling each member of each G* and any of its more specific concepts with all the negative instances it does not cover. In this way, in fact, the operations described in the procedure to compute the G set in any product language will no longer involve any matching between concepts and instances. On the other hand, the cost of labelling must now be added to the cost of language shift. The labelling we introduced requires matching each negative instance against the members of n G*'s (we keep only the G*'s relative to the initial factor languages), where each G* contains only one member (in our case, in fact, as there is only one positive instance, we can immediately remove the concepts that are not more general than the positive instance from the G*'s,

at an additional cost of O(qnbl)), and repeat for all the l more specific concepts of each member of G* (i.e., the concepts contained in the chain of admissible values relative to that G*'s factor language). Therefore labelling takes in all O(qnl) + O(qnbl) = O(qnbl). The time complexity of language shift is $O(2^n)$ + O(qnbl). The overall time complexity is therefore $O(n^4q)$ + $O(2^n)$ + O(qnbl), which, for practical values of n, b, and l, approximates to $O(n^4q)$.

In sum, we have $O(n^4l^2q)$ in the CE algorithm versus $O(n^4q)$ in the FCE algorithm. The effect of using the FCE algorithm with the chosen instance distribution appears to be that of blocking the fragmentation of G due to l. It is also worth noting that the factor $O(n^2)$ in the FCE algorithm due to the presence of multiple languages can be reduced by reducing the number of intermediate product languages employed. This would, on the other hand, be counteracted by an increase of the factor $O(n^2)$ due to the g of the intermediate languages. Here is a trade-off between using few concept languages and using many concept languages in a given range. The fewer the concept languages, the less the amount of computation devoted to parallel induction and language shift. The more the concept languages, the more likely it is that a smaller amount of induction will be done within the largest concept languages, which are the least convenient. Experimentation might help investigate this kind of trade-off .

5 Relation to factorization in concept induction

Factorization with smaller concept languages in the CE algorithm has been first explored in [Subramanian and Feigenbaum 1986] and [Genesereth and Nilsson 1987]. Although we were inspired by their work, our goals, methods and assumptions are different. First, in [Subramanian and Feigenbaum 1986] and [Genesereth and Nilsson 1987].language factorization has been used with the aim of improving efficiency during the phase of experiment generation, whereas we have investigated its utility during the earlier and more important stage of version-space induction from given examples and counter-examples. Second, while they have primarily addressed the problem of factoring a version space and assessing credit over its factors, we have focussed on language shift during version-space induction over a set of available factor and product languages. Third, their approach relies on the assumption that the given concept langage is factorable into *independent* concept languages[1]. By contrast, when applying the FCE algorithm directly to the attribute languages of a conjunctive concept language it is not necessary the attribute languages be independent. For example, the two factor languages we have used as an illustration throughout the paper (L1 and L2) happen to be two independent languages[2]; however, we could well apply the FCE algorithm to the concept language $\mathbf{L}_B$ we introduced earlier along with the concept language $\mathbf{L}_C$ = {anyrank, odd, even, 1, 3, 5, 7, 9, J, K, 2, 4, 6, 8, 10, Q}, these two languages being not independent (the intersection of the concept "2" in $\mathbf{L}_B$ and the concept "odd" in $\mathbf{L}_C$ is empty, for instance). Using non-independent factor languages, as their product may contain a large number of empty or redundant concepts, may badly affect the performance when the FCE algorithm is applied to recover from inconsistency due to use of small concept languages. But it does not seem to affect the result when the FCE algorithm is used to improve efficiency with respect to the full conjunctive concept language.

[1]Two concept languages L_A and L_B are *independent* if membership in any of the concepts from L_A does not imply or deny membership in any of the concepts in L_B. This definition implies that for every concept a in L_A and every concept b in L_B the intersection of a and b is neither empty nor equal to either concept.Two independent concept languages are unordered with respect to the larger-than relation.

6 Relation to inductive language shift

As mentioned earlier, the FCE algorithm can also be seen as a method for introducing new concepts to overcome the limitations of a set of restricted concept languages (i.e., the factor languages). It does so by creating another set of larger concept languages (i.e., the product languages) to constrain the search for new useful concepts. This is a significant departure from the search strategy usually employed in most approaches to inductive language shift. Regardless of the specific goal pursued - many systems deal with improvement of some quality measures of the learned descriptions rather than with their correctness - "the problem of new terms" [Dietterich *et al.* 1982] or "constructive induction" [Michalski 1983] is in general tackled by defining a set of appropriate *constructive operators* and carrying out a depth-first search through the space of the remaining concepts to find useful (e.g., consistent, more concise, more accurate) extensions to be added to the given language. Furthermore, since the number of admissible extensions is generally intractably large, most of the approaches to constructive induction rely on various heuristics to reduce the number of candidate additional concepts and/or to cut down the search (e.g, [Matheus and Rendell 1989], [Pagallo 1989], [Wogulis and langley 1989]).

By contrast, we compute and keep all the admissible language extensions (in a given set of extensions) that restore consistency with data, rather than considering one or few plausible language extensions at a time. Just as the relation *more general than* that is implicitly defined over the terms of a concept language may allow efficient representation and updating of all consistent concepts [Mitchell 1982], so too the relation *larger than* that is implicitly defined over a set of languages may provide the framework to efficiently organize the small-to-large breadth-first search of useful languages. These considerations suggest that an alternative abstract model for language shift can be formulated, in which the search for new concepts, rather than being based on the use of constructive operators, is driven by the ordering of a set of candidate concept languages (work in preparation).

[2] It is often the case that attribute choice reflects independencies in the world, thus giving rise to actual independent factor languages.

7. Conclusion

We have presented the FCE algorithm for efficiently inducing version spaces over a set of partially-ordered concept languages. The utility of this algorithm is twofold: improving the efficiency of version-space induction if the initial concept language is decomposable into a set of factor languages, and inducing consistent version spaces if a set of concept languages inconsistent with data is initially available. In this paper we have focussed on the former. We have applied the FCE algorithm to the task of inducing version spaces over a conjunctive concept language defined on a tree-structured attribute-based instance space, and we have evaluated when it leads to a reduction in complexity.

Acknowledgements

Part of this work was done while at the Computing Science Department of the University of Aberdeen, partially supported by CEC SS project SC1.0048.C(H). I would like to thank Derek Sleeman and Pete Edwards for their support and for useful discussions on this topic. The work was carried out within the framework of the agreement between the Italian PT Administration and the Fondazione Ugo Bordoni

References

[Bundy *et al.* 1985] A. Bundy, B. Silver, D. Plummer. An analytical comparison of some rule-learning problems. *Artificial Intelligence,* Vol. 27, No. 2 (1985), pp. 137-181.

[Carpineto 1990] C. Carpineto. Combining EBL from success and EBL from failure with parameter version spaces. In *Proc. 9th ECAI,* Pitman, London, 1990, pp. 138-140.

[Carpineto 1991] C. Carpineto. Analytical negative generalization and empirical negative generalization are not cumulative: a case study. In *Proc. EWSL-1991, Lecture Notes on Artificial Intelligence*, Springer-Verlag, Berlin, 1991, pp. 81-88.

[Dietterich *et al* 1982] T. Dietterich, B. London, K. Clarkson, R. Dromey. Learning and inductive inference. In Cohen & Feigenbaum (Eds.) *The Handbook of Artificial Intelligence,* Morgan Kaufmann, Los Altos, 1982.

[Genesereth and Nilsson 1987] M. Genesereth, N. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, Los Altos, 1987.

[Haussler 1988] D. Haussler. Quantifying inductive bias: Artificial Intelligence learning algorithms and Valiant's learning framework. *Artificial Intelligence*, Vol. 36, No.2 (1988), pp. 177-221.

[Hirsh 1989] H. Hirsh. Combining Empirical and Analytical Learning with Version Spaces. In *Proc. 6th Int.Workshop on Machine Learning.* Morgan Kaufmann, Los Altos, 1989, pp. 29-33.

[Matheus and Rendell 1989] C. Matheus, L. Rendell. Constructive induction on decision trees. In *Proc. 11th IJCAI*, Detroit, Morgan Kaufmann, Los Altos, 1985, pp. 645-650 .

[Michalski 1983] R. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, Vol. 20, 1983, pp. 111-161.

[Mitchell 1982] T. Mitchell. Generalization as Search. *Artificial Intelligence*, Vol. 18, 1982, pp. 203-226.

[Pagallo 1989] G. Pagallo. Learning DNF by Decision Trees. In *Proc.11th IJCAI*, Morgan Kaufmann, Los Altos, pp. 639-644.

[Smith and Rosenbloom 1990] B. Smith, P. Rosenbloom. Incremental Non-Backtracking Focusing: A Poliniomally Bounded Generalization Algorithm for Version Spaces. In *Proc. 8th AAAI*, Morgan Kaufmann, Los Altos, pp. 848-853.

[Subramanian and Feigenbaum 1986] D. Subramanian, J. Feigenbaum. Factorization in Experiment Generation. In *Proc. 5th AAAI*, Morgan Kaufmann, Los Altos, 1986, pp. 518-522.

[Utgoff 1986] P. Utgoff. Shift of bias for inductive concept learning. In Michalski *et al.* (Eds), *Machine Learning II.* Morgan Kaufmann, Los Altos, 1986, pp. 107-148.

[Wogulis and Langley 1989] J. Wogulis, P. Langley. Improving efficiency by learning intermediate concepts. In *Proc. 11th IJCAI*, Morgan Kaufmann, Los Altos, pp. 657-662.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Theorem Proving Engine and Strategy Description Language

Massimo Bruschi

State University of Milan - Computer Science Department
Via Comelico 39, 20135 Milan, Italy
e-mail: mbruschi@imiucca.csi.unimi.it

Abstract

The concepts of strategy description language (*SDL*) and theorem proving engine (*TPE*) are introduced as architectural and applicative tools in the design and use of an automated theorem proving system. Particular emphasis is given to the use of an SDL as a research tool as well as a way to use a prover both as a batch or as an interactive program. In fact, the availability of an interpreter for such a language offers the possibility of having a system able to cover both of these usages, giving to the user some way of choosing the granularity of the steps the prover must take. Three examples are given to show possible applications. Their purpose is to show its usefulness for expressing and testing new ideas. Some interesting capabilities of an SDL are applied to highlight how it allows the treatment of self-analysis on the state of the search space. Examples of these are the definition of a self-adaptive search and a tree pruning strategy. All the definitions we give reflect a running Prolog prototype and inherit much from the Prolog style and structure.

1 Introduction

The uses of and the interest in automated theorem proving have grown markedly in the preceding decade. The cause rests in part with faster computers, easy access to workstations, portable and powerful automated theorem-proving programs, and successes with answering open questions. Various researchers in the field conjecture that far more power is needed to attack the deep problems of mathematics and logic that are currently out of reach.

Although some of the needed increase in effectiveness will result from even faster computers, many state that the real advances will result from the formulation of new and diverse strategies. Because we feel that the ease of comparing, analyzing, and formulating such strategies would be enhanced if an appropriate abstract language and theory were available, we undertake here the development of such a language. Perhaps the abstraction and language will lead to needed insights into the nature of strategy of diverse types. In addition, because of its relation to this language, here we also provide an abstract treatment of theorem-proving programs as engines. This abstraction may enable researchers to analyze the differences, similarities, and sources of power among the radically diverse program designs.

The idea for developing a strategy description language (SDL) usable to define search strategies for a theorem prover was born when we began to study the application of parallelism to ATP. One proposal was to run many theorem provers on the same problem but with different search strategies. Having different strategies expressed as programs would mean having, as input of each prover process, the couple $< theorem, search - algorithm >$.

The development of a language requires the definition of an abstract machine to execute its programs, requiring an interpreter for the language. Our experiences and previous work with Prolog has suggested its use for the realization of a prototype.

One simple way to build an interpreter is to define a kernel module offering the basic services. This led us to the definition of a theorem-proving engine (TPE). Next, we developed a theorem prover having an SDL interpreter and a TPE as basic modules. *zSDL* is the name of our SDL.

Generally, we conjecture that an SDL might benefit by having one (or more) of the basic attitudes and of being procedural, functional, and logical. It should also be able to focus on the operations with different granularity as well as directing the prover process, controlling details of different level of complexity. As a sample model we can think at production systems in AI, and say that an SDL could be used to describe the control side of such a system. There can be as many SDL languages as different production systems.

The language we defined did not result from a deep analysis of the cited aspects; instead, it has been driven by the underlying structure of the TPE we developed, by the fact that is realized in Prolog, and by the wish to define the language *on the field* so that it could be run. One of the nice things about Prolog is that you can develop executable meta-languages.

2 A theorem proving engine

A TPE is a program module devoted to maintain and operate a knowledge base *(KB)* of logical formulas and a set of indexes on them. We think of these indexes as sets of references (or ids) to the formulas. The sets are distinguished by name. Each formula is retained together with various information about it.

A TPE can perform two basic activities: *inference* and *reduction*. The object of the first activity is to deduce new knowledge, gathering it by considering various subsets of the formulas in the KB. The object of the second activity is to keep the size (or the *weight*) of the KB as small as possible, by discarding redundant information. We require that every successful call to the inference process *(IP)* also calls the reduction process *(RP)*.

To better define the activities of a TPE, we focus on a possible minimal interface to such a module. We assume that the TPE finds the KB initialized with a given input set of formulas and that each operation maintains appropriately the indexes. We shall extend this interface gradually in the paper.

The kernel functions of a TPE can be:

- *(TPE.1)* - **enable**$(+Rule)$
- *(TPE.2)* - **disable**$(+Rule)$:

  A TPE is thought to offer a set of inference and reduction rules, each referred with a name. An IP will then apply the set of all the active inference rules, and the RP will only use the active reduction rules. These two functions are used to control the *activity sets*. For simplicity we assume the calls can also accept a list of rule names.

- *(TPE.3)* - **superpose**$(+Id_1, +Id_2)$:

  Its purpose is to activate the IP. It will superpose the formula referred to by Id_1 on the one referred to by Id_2 using all the active inference rules. We use the concept of superposition because it implies the ordering of the arguments, which is sometimes required. In this respect the general form of a single inference (as well as reduction) rule is thought of as

$$\frac{Id_1, Id_2}{\{N_1, N_2, \ldots, N_m\}}$$

  meaning that this rule takes as premises two formula references and produces a set of new references associated to the formulas resulting from the actual application. Consider for example the binary resolution inference rule. It takes two clauses and generates a set of resolvents. So, if we consider the clausal formulas referred to by Id_1 and by Id_2, the reference set $\{N_1, N_2, \ldots, N_m\}$ will refer to their resolvents (if any). Rules with single premises are called with **superpose**(Id, Id).

- *(TPE.4)* - **delete**$(+Id)$:

  It is used to delete the formula referred to by Id from the KB and from the indexes. This operation, combined with a superposition call, can be used to realize transformation processes on the KB. Consider for example the standard CNF transformation. It replaces a formula with a (satisfiability) equivalent set of clauses. We can model this by calling an inference rule with only one premise to generate the set of clauses and then delete the premise. As a matter of fact we think of this operation as reversible. See the next operation.

- *(TPE.5)* - **undelete**$(+Id)$:

  It is called to recover an earlier deletion of a formula. We can think of it as a special inference rule that uncovers a formula. It can be useful in adaptive searches. Suppose for example we are using a weighting strategy to discard newly generated formulas if they exceed a fixed weight. Using the `delete/1` call we can simply hide the formula from the KB and the indexes and later recover it if, for example, the search ends with a consistency status.

As a matter of fact the indexes on the formula KB have a dominant role for understanding the entire idea. In the next section we will make clearer this role.

3 zSDL: a strategy description language

Indexes, as sets of references to KB's formulas, are the basic objects of the language zSDL, which uses id-sets as the basic elements to refer to nodes and to describe the visit of the search tree.

The underlying idea is that an SDL requires some mechanism to represent a proof tree, for the ideal search strategy for proving a given theorem is the description of the precise steps the reasoning module must follow to reach the proof nodes in the search tree. With an SDL we must be able to speak about the nodes of the tree (the formulas) and the relations between them (how to reach the parents of each node, following the ancestor relation, as well as how to reach the children of a node, following the descendants relation). Another useful property might be the ability to know the level of a tree node, in order to define a (partial) ordering between the steps made to reach the proof (a sequence of parallelizable steps).

From these observations we chose to use sets of nodes as the basic description objects. And zSDL turned out to be, in some sense, a sets-operations oriented language. We will refer to a generic zSDL set of references to formulas to mean either an id-set or an index. A set is referred by a (unique) name. It is something like a variable of type id-set.

In zSDL we can apply to the id-sets all of the common operations and relations on sets, plus some special (procedural) ones like assignments, evaluation, etc. The following is a list of these functions, giving in addition some of the syntax of zSDL (recall that it is a Prolog by-product). In zSDL an id-set is represented as a Prolog list.

The Prolog variable names implicitly define the types of the operators in the following way:

SetName: the name of the variable that refers to the set.

SetExpr: an expression on sets, which can be an explicit set (list), a *SetName* or an expression built up using the defined operations.

Var: a Prolog non-instantiated variable.

ElemOrVar: a Prolog variable *(Var)* eventually instantiated *(Elem).*

Notice that the *SetExpr* are evaluated.

◇ *(zSDL.1) - set operations* :

$+SetExpr_A$.# $+SetExpr_B$ % *union*
$+SetExpr_A$.+ $+SetExpr_B$ % *weak union*
$+SetExpr_A$.* $+SetExpr_B$ % *intersection*
$+SetExpr_A$.- $+SetExpr_B$ % *difference*

The weak union makes no checks on repetitions.

◇ *(zSDL.2) - set relations* :

$?ElemOrVar$.? $+SetExpr$ % *membership*
$+SetExpr_A$.=< $+SetExpr_B$ % *containment*
$+SetExpr_A$.< $+SetExpr_B$ % *strict containment*
$+SetExpr_A$.= $+SetExpr_B$ % *equality*

Notice that, using the Prolog negation, we also have the negations of these relations

◇ *(zSDL.3) - set procedures* :

$+SetName$:= $+SetExpr$ % *assignment*
$-Var$.$\mp SetName$ % *extract 1st element*
$-Var$ .. $+SetExpr$ % *evaluate*
:. $+SetName$ % *destroy the set*

The *pop* operation treats the set as a stack.

As an example, in a zSDL-Prolog session we could have:

```
| ?- a := [1,2,3],
     b := a .- [3,4,5].

yes
| ?- A .. a,
     B .. b,
     X .. b .# [6].

A = [1,2,3],
B = [1,2],
X = [1,2,6]
```

in which you see how zSDL sets are permanent objects, contrary to the classical Prolog variables.

This level of basic operations on (id-)sets must be enriched by statements to permit interaction with the TPE. We will show the basic calls zSDL defines to run an IP by developing the Prolog code that can realize it.

We are looking for a statement responsible for executing the actual inference steps applicable on some given id-sets. Consider the zSDL syntax

◇ *(zSDL.4) - directed superposition* :

$+SetExpr_A$ ++> $+SetExpr_B$

After the evaluation of the id-set expressions the general form of a call can be thought of as

$$[A_1, A_2, \ldots, A_n] \text{ ++> } [B_1, B_2, ..., B_m].$$

Obviously we expect this search to consider all the pairs, i.e. the TPE must be directed to try all the following superpositions:

$$\begin{array}{cccc} < A_1, B_1 >, & < A_1, B_2 >, & \ldots, & < A_1, B_m > \\ < A_2, B_1 >, & < A_2, B_2 >, & \ldots, & < A_2, B_m > \\ \vdots & \vdots & \ddots & \vdots \\ < A_n, B_1 >, & < A_n, B_2 >, & \ldots, & < A_n, B_m > \end{array}$$

This can be realized by the following straightforward Prolog code:

```
SetExprA ++> SetExprB :-
   Ai .? SetExprA,
   Bj .? SetExprB,
   superpose(Ai,Bj),
   stop_search.
SetExprA ++> SetExprB.
```

The only new predicate we used is `stop_search/0`. In fact, one omitted item in the TPE interface we have observed is a test to control the status of the KB. Therefore, we extend the TPE interface with

• *(TPE.6)* - **proof_found**(−*Int*) :

Used to ask the status of the KB. The number of found proof(s) is given.

You can think of `stop_search/0` as built from a `proof_found/1` call followed by an appropriate comparison and by any other (eventually) necessary operations.

In addition to the `++>/2` operator, zSDL also defines the syntax

◇ *(zSDL.5)* - *superposition* :

+$SetExpr_A$ <+> +$SetExpr_B$

With the `<+>/2` operator each couple is also reversed (except for the $<X,X>$ ones).

As we commented, the general form of an inference rule in zSDL is thought to be

$$\frac{Id_1, Id_2}{[N_1, N_2, \ldots, N_m]}.$$

The actual application of such a rule is called by

$[Id_1]$ ++> $[Id_2]$.

The first missing item is a way to get, in a zSDL program, the id-set of the generated formulas. With a typical Prolog attitude, we can generalize this problem.

A superposition goal on id-sets is like evaluating a high-level function on a set. The relation that links the input and the output sets is different from the classical ones, for it is related to some properties of the objects in the sets and not to the sets themselves. This simply implies that the actual module responsible of the evaluation of these relations is not the classical one. And we know that that module must be the TPE. So we are looking for a syntax like

◇ *(zSDL.6)* :

?*Index* ::= +*TPE_Goal*,

where a *TPE_Goal* can be, as an example, a superposition call. Notice that we defined the new operator `::=/2` in order to switch the evaluation to the right module. The call also suggests a possible model for the computation of the goal. In fact a goal of the TPE is generally requested to produce a new index (say a *dynamic* index) that is updated during the actual evaluation of the goal. Consider the following code.

```
Given ::= TPE_Goal :-
   new_dynamic_index(NewSet),
   call(TPE_Goal),
   ( var(Given),
     Given .. NewSet
   ; Given := NewSet  ),
   del_dynamic_index(NewSet).
```

It asks the TPE to release a new dynamic index that will be updated during the execution of the given TPE_Goal to hold the result of the evaluation. This result is then properly assigned to the input Given argument and finally the dynamic index is cleared. This asks for the extension of the TPE interface with the two following calls

• *(TPE.7)* - **new_dynamic_index**(−*SetName*) :

Ask the TPE to extend the sets of active indexes. *SetName* will be used to refer to this new dynamic id-set. The complementary call is

• *(TPE.8)* - **del_dynamic_index**(+*SetName*) :

It is used to remove the index referred by *SetName* from the set of the dynamic indexes known by the TPE.

With the new zSDL operator we can now use the following statement to sketch the application of an inference rule:

NewIds ::= $[Id_1]$ ++> $[Id_2]$.

where *NewIds* will be instantiated to the right instance of $[N_1, N_2, \ldots, N_m]$, even possibly the empty id-set. Notice that the `::=/2` operator works for each TPE goal.

The last extension we will give before going through some examples of an application of zSDL focus on a way to have a local specification of the inference rules we wish to apply in a TPE goal. The zSDL syntax is:

◇ *(zSDL.7)* :

+*TPE_Goal* ./ +*Inferences*,

defines a TPE evaluation *modulo* a given set of inference rules.

Suppose for example we wish to superpose clauses 3 and 15 only by binary resolution (`binary_res`). Consider the following code

```
TPE_Goal ./ Inferences :-
   Active .. enabled_inferences,
   disable(Active),
   enable(Inferences),
   call(TPE_Goal),
   disable(Inferences),
   enable(Active).
```

With this new operator we can express the preceding problem as

```
Resolvents ::= [3] ++> [15] ./ binary_res.
```

The code we have given assumes that the `enable/1` and `disable/1` calls in the TPE interface maintain one set, called `enabled_inferences`, collecting the names of the active inference rules.

So in zSDL the more general IP activation call to the TPE is

$$NewIds \ ::= \ Expr_A \ \texttt{<+>} \ Expr_A \ ./ \ Infs.$$

which will give in *NewIds* the id-set of all the formulas derivable by applying the chosen inferences to all the pairs of formulas implicitly referred to by the id-set expressions.

4 A simple zSDL program: the breadth-first strategy

Time has come to give the first example of the use of zSDL to describe a classical strategy: the breadth-first search. We suppose that the TPE is already active and some input formulas are present in the KB. An index called `input` collects the references to those statements.

In the breadth-first search the next level of the tree is filled with all the conclusions given by superposing the last level with all the existing levels. The search stops with complete search or, for example, with a proof. The zSDL program is

```
breadth_first :-
  levels := input,
  last   := input,
  while( ( \+ stop_search,
           \+ last .= [] ),
       ( Next ::= last <+> levels,
         last := Next,
         levels #= last ) ).
```

The two indexes, `levels` and `last`, refer to the entire tree and to its last level, respectively. The `while/2` is the classical cyclic structure you found in each procedural language. Its syntax is

◇ *(zSDL.8)* - **while**(+*Condition*, +*Goal*) .

After the initialization of the values to the input references, the program repeatedly fills the `Next` level of the search tree, superposing the `last` level with all the nodes. Then the `Next` level becomes the `last` and is also added to the references of the entire tree. The `#=` notation resemble the C language style assignments. Similarly zSDL accepts the operators `+=`, `-=` and `*=`. Notice also that the instances of the Prolog variable(s) in the `while/2` statement are released between the cycles.

The preceding algorithm can be improved by thinking of the cases it generates. When we superpose the last level with the entire tree, we must note that all of the nodes in `last` are already in `levels`. Furthermore, if we apply the `<+>` operator to superpose an id-set on itself, we try all of the pairs twice. So, a better program is

```
breadth_first :-
  last   := input,
  others := [],
  while( ( \+ stop_search,
           \+ last .= [] ),
       ( LL ::= last ++> last,
         L0 ::= last <+> others,
         others := last .+ others,
         last := LL .+ L0 ) ).
```

In this definition the `last` index refers again to the last level of the tree while `others` refers to the rest of the tree. At each step `last` is superposed on itself (with the oriented operation `++>`) and then with the upper levels of the tree. You might also note that in this way we can substitute the use of the standard union with the weak one (append) as no repetitions are possible in the references in the indexes.

In addition to the while statement, zSDL defines some other basic control structure:

◇ *(zSDL.9)* - **foreach**(+*Generator*, +*Goal*) :

Goal is executed for all the solutions of the given *Generator*.

◇ *(zSDL.10)* - **repeat**(+*Goal*, +*Condition*) :

Goal is executed at least once and re-executed while *Condition* fails.

◇ *(zSDL.11)* - **iF**(+*Condition*, +*Goal*) :

Goal is executed only if the *Condition* holds. It always succeeds.

This list is given only for completeness: the reader might note that zSDL programs are basically extended Prolog programs and that all the structures definable on the underlying Prolog machine can be used by zSDL programs.

However, we think that one real important aspect of the $< TPE, zSDL >$ Prolog-based architecture comes from its direct executabilty on a Prolog machine. The global proving system loses the property to be batch or interactive: a proof search is directed by the execution of goals, and the granularity of these steps can vary from the single superposition to the entire search.

5 More complex applications

The availability of a language like zSDL adds to the ease of implementing and experimenting with new ideas, for example, non-standard search strategies. To illustrate the value of using of zSDL, and to introduce some additional features of this language, we now focus on three somewhat complex programs. The first defines an *adaptive*, weighting-based, search strategy. The second introduces some atypical *deletion* strategy into the search. The last one shows how to define a strategy (oriented) *tailored* to a given inference rule.

5.1 A weight-based adaptive strategy

By weighting (w) strategies we refer to those algorithms structured to consider the length, or weight, of the formulas. Examples of w-functions are: the number of symbols in a formula, the number of (positive, negative, total) literals in a clause, as well as linear functions built on these or other values. The general behavior of a w-strategy is to filter the retention in the KB of a newly generated formula, according to the given w-function. Formulas that are too heavy are discarded. The underlying intuitive idea is that if a proof can be obtained without the use of heavy formulas, then such formulas can be discarded.

We shall not consider the well-known subproblems that the subsumption operation can lead to, which vary with the w-function adopted. Instead, we consider one of the practical difficulties in the application of these strategies, namely, choosing the appropriate threshold (upper bound on weights) to use for deciding which formulas to discard. The solution we propose follows this simple idea: the threshold can be increased, when the search stops generating formulas, and set to the lightest weight template in the set of the w-deleted formulas. In this sense the search is adaptive: it adapts to the performance of the program.

Let us first show the mechanisms provided by the TPE to support w-strategies. Each formula is stored with a `weight template`. An internal function, namely, `weight(+`*Formula*`,-`*W_Template*`)`, is used by the TPE to calculate it. Such a template consists of a 4-integers tuple ($N-P-T-S$) that counts Negative_Literals, Positive_Literals, Total_Literals and Symbols, where the first three values are "0" if the formula is not a clause. The TPE offers some calls in order to define weighting-based strategies:

- *(TPE.9)* - **max_weights**(*?W_Template*) :

  The call can be used both to access the current reference w-template (if *W_Template* is an uninstantiated variable at the call) or to set a new value for it. The new given *W_Template* will be used by the w-filter operation to decide which new formulas to accept or discard. All the values for the new formulas must be less or equal to the threshold ones fixed by the given *W_Template*. The value of a variable will be considered greater than each integer.

- *(TPE.10)* - **formula_weight**(*+Id, −W_Template*) :

  Accesses the given formula(s) to get their weight template(s).

The basic behavior of the strategy we are going to write is straightforward. At each time we choose the lightest not yet used formula in the KB to be superposed with all the already used ones. Than we move the given formula to the set of the used ones (say "done") while the new generated formulas are added to the first set (say "to_do"). We can express this with the following zSDL program:

```
to_do := [],
done  := [],
Input .. input,
add_ordered(Input,to_do),
while( ( \+ stop_search,
         \+ to_do .= [] ),
     ( Lightest .~ to_do,
       add_ordered([Lightest],done),
       New ::= [Lightest] <+> done,
       add_ordered(New,to_do) ) ).
```

As one sees, we solved the problem of getting the lightest formula in a set by extracting the first element from an ordered set. The expected side effect of an `add_ordered(`*Set,SetName*`)` call is to build an ordered union of *Set* and *SetName* (into *SetName*) according to the weight of the corresponding formulas. We can obtain this with:

```
add_ordered([],_SetName).
add_ordered(Set,SetName) :-
   Xet .. SetName,
         %% gets a list of Count-Id pairs
   get_counts(Set,SetCs),
   get_counts(Xet,XetCs),
   append(SetCs,XetCs,YetCs),
         %% sorts by counts
   keysort(YetCs,ZetCs),
         %% removes the counts
   pop_counts(ZetCs,Zet),
   SetName := Zet.
```

where the `get_counts/2` call accesses the weights-template of the formulas to get the symbol counts (obviously, one can choose different approaches).

To extend our strategy to be self-adaptive we have to solve certain problems:

- ⋆ how to get information on the deleted formulas;
- ⋆ how to choose some initial value for the reference w-template.

The first problem rests entirely on the TPE behavior, as the "over-weight" deletions are embedded into its operations. Our system maintains a set of structures, indexed by weights-template, to have the references to the deleted formulas. The call

- *(TPE.11)* - `queue(wdel(?W_Template),?Queue)` :

  Queue holds the ids of all the deleted formulas sharing the same it W_Template.

We first give the extended program that realizes the self-adaptive search, and then we discuss its main steps.

```
self_adaptive :-
   input_weighting,
   to_do :≈ [],
   done  :≈ [],
   Input .. input,
   add_ordered(Input,to_do),
   while( ( \+ stop_search,
            (  \+ to_do .≈ []
            ;  q_exists(wdel(_))  ) ),
     once (  to_do .≈ [],
             lightest_deleted(Count),
             closest_wtemplate(Count,NewWT),
             max_weights(NewWT),
             deleted :≈ [],
             add_deleted(Count,deleted),
             Unhide .. deleted,
             Restored ::≈ undelete(Unhide),
             add_ordered(Restored,to_do)
             Lightest .˜ to_do,
             add_ordered([Lightest],done),
             New ::≈ [Lightest] <+> done,
             add_ordered(New,to_do) ) ).
```

The first difference concerns the while condition: it now considers the possible presence of formulas deleted by weight, so the search is complete only if no deleted formulas remain. The `lightest_deleted/1` call accesses the deletion queue, searching for the lightest-weight formula. Its definition can be:

```
lightest_deleted(Count) :-
   setof( SymCount,
          queue(wdel(N-P-L-SymCount),Q),
          Deleted ),
   sort(Deleted,[Count|_Others]).
```

The `closest_wtemplate/2` call is responsible for deciding the value for the new reference weights-template, or, in other words, for the "size of the adaptation-step". The following definition builds the new template in order to accept all the formulas with the given deleted smallest *symbol* count.

```
closest_wtemplate(Count,Template) :-
   setof( N-P-L-Count,
          queue(wdel(N-P-L-Count),Q),
          Deleted ),
   max_weights(CurrentWT),
   max4([CurrentWT|Deleted],Template).
```

where the call to `max4/2` builds the `Template` given by the maximal values for each count.

The `add_deleted/2` call is conceptually similar to the `add_ordered/2` call, but works with the deletion queue. Its definition can be:

```
add_deleted(Count,SetName) :-
   (  queue(wdel(N-P-L-Count),Queue),
      SetName += Queue,
      q_del(wdel(N-P-L-Count)),
      fail
   ;  true  ).
```

It collects into `SetName` all the references to the deleted formulas with the given symbol `Count` and deletes the corresponding queue (`q_del/1`).

So, in the while loop of our program, the `to_do` id-set is extended either by newly inferred formulas or by reactivating the lightest deleted ones (if any).

A last point addresses the choice of the initial values for the reference w-template. A strategy that has given us interesting results fixes the values by looking at the counts of the input formulas and choosing the lowest values among them. Its definition is:

```
input_weighting :-
   Input .. input,
   formula_weight(Input,WTs),
   max4(WTs,Template),
   max_weights(Template).
```

5.2 A pruning strategy

This second example of the applications of the zSDL language is given to show how it can be used to define some self-analytical activity for the proving process. In other words we can use it to reason about the current state of the search during the execution.

A well-known problem each ATP program must face is the possible explosion of the search space, which can occur for various reasons. Here we do not study this topic, nor do we suggest that our program has a deep impact on the solution of the general problem. Our goal is only to show how an SDL language can be useful in different research areas of ATP.

We observe that our pruning strategy is based on the addresses of the *undeterminism in the order of application of the inference steps.* On the other hand, the use of reduction rules comes from the wish to have a KB capture the same logical consequences with a smaller possible representation "size". Consider now a generic search process and suppose a reduction step occurs. With "reduction" we will refer to the results of an operation able to change the structure of a formula, maintaining its logical value. Generally speaking a reduction step will reformulate a formula by "reducing" its complexity and/or size. This transformation will in general involve other formulas used as a base for the logical reformulation. As an example, consider the following steps on two generic clauses

$$\dfrac{\dfrac{[1]\,\neg A \mid B, [2]\,\neg A \mid \neg B \mid C}{[3]\,\neg A \mid C}\ \textit{binary resolution}}{\textit{delete}\ 2}\ \textit{subsumption}$$

We can view this step as the application of a reduction rule that uses [1] to transform [2] into [3]. We note that the satisfiability of the overall KB is preserved, i.e. the operation maintains the logical truth of the set of formulas.

Suppose next that such a reduction has occurred during a search, say a formula $\mathcal{F}$ has been reduced to $\mathcal{F}'$. There now exists a potential set of formulas whose generation depends on the order in which the search process has been executed: this set consists of all of the descendants of $\mathcal{F}$ that have not contributed to the generation of $\mathcal{F}'$, or, more precisely, the set

$$\textit{by\_inference}(\textit{descendants}(\mathcal{F})) - \textit{ancestors}(\mathcal{F}').$$

(We note that we must leave all the descendants of $\mathcal{F}$ given by reduction as those are formulas originally not in the set generated by $\mathcal{F}$).

Pruning this set (if not empty) could perhaps make the proof longer, as the proof could be reachable rapidly by using one of the formulas we deleted, but it will not preclude the possibility of finding the proof if there is one.

The effectiveness of this pruning strategy depends mainly on the effectiveness and the applicability of reduction steps in a proof, and so it relies directly on the structure of the search space (given by the formulas asserting the theorem).

Let us now see how we can implement this operation by using zSDL and the mechanisms of the TPE. First of all we formalize the calls the TPE defines (and zSDL inherits) to access various relations on the content of the KB. We already announced some of them in section 2.

- *(TPE.12)* - **parents**$(+Id, -Parents)$
- *(TPE.13)* - **ancestors**$(+Id, -Ancestors)$
- *(TPE.14)* - **children**$(+Id, -Children)$
- *(TPE.15)* - **descendants**$(+Id, -Descendants)$:

  Being *Id* the reference to a formula, these calls will respectively return the id-set of its parents, ancestors, children, and descendants, with respect to the current KB. We note that the given id-set may contain references to currently inactive formulas (deleted for some reason). All these relations will consider both inference as well as reduction steps.

- *(TPE.16)* - **by_reduction**$(+IdSet, -ByRed)$:

  Given an *IdSet* this call selects which referred formulas have been produced by application of a reduction rule, building the id-set *ByRed* with their ids.

- *(TPE.17)* - **replace**$(?NewId, ?Id)$:

  The call succeeds if *NewId* refers to a formula that replaces an old one (referred by *Id*) following a reduction step. Otherwise the call fails.

The proposed pruning strategy acts like a filter on the result of a superposition call: at each step it checks if the new formulas are given by reduction, in which case it tries to apply the deletion. So, we are going to extend the superposition control level of zSDL with a meta-call realizing the pruning.

```
pruning_derive(SetA,Mode,SetB) :-
   XA .? SetA,
   XB .? SetB,
   once (
      Given ::= derive([XA],Mode,[XB]),
      by_reduction(Given,ByRed),
      foreach( NId .? ByRed, (
         replace(NId,Id),
         ancestors(NId,NIdAnc),
         descendants(Id,IdDes),
         by_reduction(IdDes,IdDesByRed),
         IdDesByInf .. IdDes .- IdDesByRed,
         DelSet .. IdDesByInf .- NIdAnc,
         delete(DelSet) ) ) ),
   stop_search.
pruning_derive(SetA,Mode,SetB).

derive(SetA,(<+>),SetB) :- SetA <+> SetB.
derive(SetA,(++>),SetB) :- SetA ++> SetB.
```

The schema is quite simple. Each by-reduction child (`NId`) of a superposition call is related to the formula it replaces (`Id`). Then the set of the by-inference descendant of `Id` is reduced by the set of the `NId` ancestors. Notice how the $\textit{by\_inference}(\textit{descendant}(\mathcal{F}))$ set is evaluable as $\textit{descendants}(\mathcal{F}) - \textit{by\_reductions}(\textit{descendants}(\mathcal{F}))$.

5.3 A hyperresolution-oriented search strategy

Our last example uses zSDL to define a strategy specifically oriented to work with a given inference rule, namely, the inference rule hyperresolution.

The efficiency of an ATP system comes from the efficiency of all of the different components of the program, from the basic unification and match algorithms to the KB management, and so on. With some "tough" inference rule, it also heavily relies on the ability of the search strategy to control its application ensuring a complete search without repeating steps. Hyperresolution is one such inference rule.

Hyperresolution considers a basic clause (called nucleus) that has one or more negative literals. An inference step occurs when a set of positive unit clauses (called satellites) is found that simultaneously unify with all of the negative literals of the nucleus. It is simple to see how hyperresolution will not generate new nuclei (for the rule cannot produce a clause containing negative literals) while it can generate new satellites. So, the set of potential satellites change dynamically during the search, and a good strategy must ensure a complete covering partition (with multiple occurrences) of this set without repeating trials.

We first explain how we implemented the hyperresolution inference rule in our system (we call it hy_p). As usual the rule has two arguments: the first must be a satellite and the second a nucleus. If a unification is found between the given satellite and one of the negative literals in the nucleus, then the set of the current active satellites is partitioned and superposed on the remaining negative literals.

This behavior suggests the development of a search strategy driven by the generation of new satellites. In fact, we can visit the search space by levels, generate all the possible hyperresolvents, choose from them the new satellites, and use those to drive the search in the next level. As those satellites are new, the partitions we will try are new too, and no repetition in the trials occur. The basic shape of the strategy can be:

```
hyper_strategy :-
   Input .. input,
   get_satellites(Input,Sats),
   get_nuclei(Input,Nucs),
   last_sats := Sats,
   nucs := Nucs,
   while( ( \+ stop_search,
            \+ last_sats .= [] ), (
      New ::= last_sats ++> nucs ./ hy_p,
      get_satellites(New,NewSats),
      last_sats := NewSats ) ).
```

The `get_satellites/2` and `get_nuclei/2` calls are used to choose from an id-set the subset of formula-references corresponding, respectively, to valid satellites and nuclei. Notice how these calls can be defined by using the `formula_weight/2` call and testing the negative and positive literal counts accordingly.

As a matter of fact, the algorithm we have given follows closely the general schema of a breadth-first search. So, it can be simply extended to consider the application of more inference rules, intermixing the searches with the control, the `enable/1`, and the `disable/1` operations permitted.

6 Conclusions

This work introduces the concepts of Theorem Proving Engine and Strategy Description Language as architectural and applicative tools in the design and use of an automated theorem-proving program.

The definitions we give reflect a running Prolog system, named zEN2, and,, because of this fact, they inherit a Prolog style structure.

Particular emphasis is given to the use of an SDL as a research tool as well as a way to reinterpret the use of a theorem prover as a batch or as an interactive program. In fact, the availability of an interpreter for such a language offers the possibility of having a system able to cover both of these usages, giving to the user some way of choosing the granularity of the steps the prover must take.

Three examples are given to show the possible application of an SDL. Their purpose is to show its usefulness for expressing and testing new ideas. Some interesting capabilities of zSDL are applied to highlight how it allows the treatment of self-analysis on the state of the search space. Examples of these are the definition of the self-adaptive search and the pruning strategy.

Acknowledgments

The author is very grateful to Larry Wos, Bill McCune and Gianni Degli Antoni for their comments. This work was partially supported by the CEE ESPRIT2 KWICK Project and partially by a grant of the Italian Research Council. Most part of the work was done while the author was visiting the Mathematics and Computer Science Division of the Argonne National Laboratory.

References

[**Henschen** *et al.* **1974**] L.Henschen, R.Overbeek and L.Wos. A Theorem Proving Language for Experimentation. *Communications of the ACM*, Vol. 17 No. 6 (1974)

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A New Algorithm for Subsumption Test

Byeong Man Kim*, Sang Ho Lee**, Seung Ryoul Maeng*, and Jung Wan Cho*

* Department of Computer Science & Center for Artificial Intelligence Research
Korea Advanced Institute of Science and Technology, Dae-Jeon, Korea

** Database Section
Electronics and Telecommunications Research Institute, Dae-Jeon, Korea

Abstract

To reduce the number of generated clauses in resolution-based deduction systems, subsumption has been around quite for a long time in the automated reasoning community. It is well-known that use of the subsumption sharply improves the effectiveness of theorem proving. However, subsumption tests can be very expensive because they should be applied repeatedly and are relatively slow. There have been several researches to overcome the expensiveness of subsumption. One of them is the s-link test based on the connection graph procedure. In the s-link test, it is essential to find a set of pairwise strongly compatible matching substitutions between literals in two clauses. This paper presents an improved algorithm of the s-link test with a new object, called *strongly compatible list*. By use of the strongly compatible lists and appropriate bit operations on them, the proposed algorithm reduces the possible combinations of matching substitutions between literals as well as improves the pairwise strongly compatible test itself. Two other subsumption algorithms and our algorithm are analyzed in terms of the estimated maximal number of string comparisons. Our analysis shows that the worst-case time complexity of our algorithm is much lower than the other algorithms.

1 Introduction

Logical Reasoning (or theorem proving) is the key to solving many puzzles, to solving problems in mathematics, to designing electronic circuits, to verifying programs, and to answering queries in deduction systems. Logical reasoning is a process of drawing conclusions that follows logically from the supplied facts. Since the first-order predicate logic is generally sufficient for logical reasoning and offers the advantage of being partially decidable, it is widely used in automated reasoning.

There have been a number of approaches to show that a formula is a logical consequence of a set of formulas. Notable among them is Robinson's resolution principle [Robinson 1965] which is very powerful and uses only one inference rule. Many refinements of the resolution principle based on graph have been proposed to increase the efficiency [Kowalski 1975, Sickel 1976, Andrew 1981, Bibel 1981, Kowalski 1979]. One of them is Kowalski's connection graph proof procedure [Kowalski 1975, Kowalski 1979] which has some distinct advantages over previous approaches based upon resolution.

1. Once an initial connection graph is constructed all information is present as to which literals are potentially resolvable so that no further search for unifiable complementary literals is needed.

2. Application of a deletion operation can result in further deletion operations, thus potentially leading to a snowball effect which reduces the graph rapidly. The probability of this effect rises with the number of deletion rules available.

3. The presence of the complete search space during connection graph proof procedure suggests the opportunity to use parallel evaluation strategies [Loganantharaj 1986,Loganantharaj 1987,Juang 1988] to improve the efficiency.

Various deletion strategies [Munch 1988,Gottlob and Leitsch 1985,Chang and Lee 1973] are suggested to reduce the number of clauses generated in theorem proving (automated reasoning). A very powerful deletion rule in resolution-based deduction systems is the subsumption [Eisinger 1981, Wos 1986]. The subsumption is used not only to discard a newly deduced clause when a copy already has been retained, but also to discard other types of unneeded information. The use of subsumption sharply improves the effectiveness of theorem proving, as illustrated by the benchmark problem, Sam's Lemma [Wos 1986].

However, the use of subsumption can be quite expensive because it must be repeated very often and is relatively slow [Wos 1986]. There have been two approaches for overcoming the expensiveness of subsumption. One is to reduce the number of necessary subsumption tests [Eisinger 1981], and the other is to improve the subsumption test itself [Gottlob and Leitsch 1985, Stillman 1973]. Eisinger [Eisinger 1981] proposes the s-link test which is based on the principal ideas of the connection graph proof procedure. His method provides an efficient preselection which singles out clauses D that do not possess the appropriate links to the clause C. Having preselected the candidates, we need to compose matching substitutions from literals in clause C to literals in clause D to find a matcher θ from C to D. In some cases many compositions are possible and hence the search for θ becomes quite expensive. Socher [Socher 1988] improves the search procedure by imposing restrictions on the possible matching substitutions.

In this paper we propose an improved s-link test with a new object, called *strongly compatible list*. By use of the strongly compatible lists and appropriate bit operations on them, the proposed algorithm reduces the

possible combinations of matching substitutions between literals as well as improves the pairwise strongly compatible test itself. Two subsumption algorithms (Eisinger, Socher) and our algorithm are analyzed in terms of the estimated maximal number of string comparisons. Our analysis shows that the worst-case time complexity of our algorithm is much lower than the other algorithms.

In the next chapter, preliminary definitions and the s-link test are presented. A new subsumption algorithm based on strongly compatible lists and its related works and analysis are given in Chapter 3 and Chapter 4, respectively. In Chapter 5, our works are summarized.

2 Preliminaries

We assume that the readers are familiar with materials in [Chang and Lee 1973]. A variable starts with an upper case letter and a constant starts with a lower case letter.

Definition 2.1 A *substitution* σ is a mapping from variables to terms.

We represent a substitution σ with $s_i\sigma = t_i$ for each i $(1 \leq i \leq n)$ by the set of pairs $\{t_1/s_1, \cdots, t_n/s_n\}$, and represent the composition of substitution of σ and τ by $\sigma \bullet \tau$. For convenience, we denote $\sigma_1 \bullet \cdots \bullet \sigma_n$ by $\bullet_{i=1}^{n}\sigma_i$.

Definition 2.2 Two substitutions σ and τ are *strongly compatible*, if $\sigma \bullet \tau = \tau \bullet \sigma$.

Definition 2.3 Substitutions $\sigma_1, \cdots, \sigma_n$ are *pairwises-trongly compatible*, if any two substitutions $\sigma_i, \sigma_j \in \{\sigma_1, \cdots, \sigma_n\}$ are strongly compatible.

Definition 2.4 A *matching substitution* from a term (or a literal) s to a term (or a literal, respectively) t is a substitution μ such that $s\mu = t$.

Definition 2.5 $uni(C, l_i, D)$ is a set of all matching substitutions mapping a literal l_i in clause C onto some literal in clause D.

For example, given $C = \{p(X,Y), q(Y,c)\}$ and $D = \{p(a,b), p(b,a), q(a,c)\}$, we have $uni(C, p(X,Y), D) = \{\{a/X, b/Y\}, \{b/X, a/Y\}\}$ and $uni(C,\ q(Y,c),\ D) = \{\{a/Y\}\}$.

Definition 2.6 If there is a τ with $\theta = \sigma \bullet \tau$ for any other unifier θ for s and t, σ is a *most general unifier* (mgu) for s and t.

To reduce the search space in theorem proving, redundant clauses must be removed. The *redundant* clause means a clause whose removal does not affect the unsatisfiability. The redundant clause includes a tautology or a subsumed clause. The subsumption can be defined in two ways.

Definition 2.7 A clause C_1 *subsumes* another clause C_2 if C_1 logically implies C_2.

Definition 2.8 A clause C_1 *θ-subsumes* another clause C_2 if $|C_1| \leq |C_2|$ and there is a substitution θ such that $C_1\theta \subseteq C_2$.

It has been shown [Gottlob and Leitsch 1985,Loveland 1978] that these two definitions are not equivalent. If we use the first definition, then most of the resolution-based proof procedures are not complete because a clause always subsume its factors. In this paper we are concerned only with the θ-subsumption.

In order to perform a subsumption test on given two clauses, we must find a matcher θ such that $C\theta \subseteq D$. It is well known that finding such θ is NP-complete [Gottlob and Leitsch 1985] and the search for θ may become expensive. There have been some efforts to reduce the cost of finding a matcher θ [Gottlob and Leitsch 1985,Socher 1988,Chang and Lee 1973,Eisinger 1981,Stillman 1973]. One of them is the s-link test based on the connection graph procedure. The subsumption test based on the s-link is provided by the following theorem [Eisinger 1981]:

Theorem 2.1 Let $C = \{l_1, \ldots, l_n\}$ and D be clauses. Then C θ-subsumes D if and only if $|C| \leq |D|$ and there is an n-tuple $(\sigma_1, \ldots, \sigma_n) \in \times_{i=1}^{n} uni(C, l_i, D)$ such that all σ_i $(1 \leq i \leq n)$ are pairwise strongly compatible.

Example 2.1 *(of Theorem 2.1 [Socher 1988])* Given a set $\{C,\ D_1,\ D_2,\ D_3\}$ of clauses with $C = \{p(X,Y), q(Y,c)\}$, $D_1 = \{p(a,c), r(b,c)\}$, $D_2 = \{p(U,V), q(V,W)\}$ and $D_3 = \{p(a,b), p(b,a), q(a,c)\}$ one want to find out, which clauses are subsumed by C. D_1 can be excluded because the literal $q(Y,c)$ in C is not unifiable with any literal in D_1, that is, there is no s-link from $q(Y,c)$ to a literal in D_1. D_2 cannot be a candidate because $uni(C, q(Y,c), D_2) = \{\}$. For D_3 we obtain the two pairs (σ_1, τ) and (σ_2, τ), where $\sigma_1 = \{a/X,\ b/Y\}$, $\sigma_2 = \{b/X,\ a/Y\}$ and $\tau = \{a/Y\}$. From these two pairs only (σ_2, τ) is strongly compatible and thus C subsumes D_3. □

As shown in Example 2.1, in order to find clauses that are subsumed by a clause $C = \{l_1, \ldots, l_m\}$, first we have to preselect clauses that are connected to every literals in C by s-links of a connection graph. If D is such clause then each literal in C is unifiable with some literals in D. For such candidate D, we need to perform a pairwise strongly compatible test on all elements of $\times_{i=1}^{m} uni(C, l_i, D)$.

3 A New Subsumption Algorithm Based on Strongly Compatible Lists

The s-link test [Eisinger 1981] for long clauses with more than one matching substitution for each literal may require an expensive search of all elements of the Cartesian product.

We define *the strongly compatible list* of matching substitutions in order to improve the s-link test. With the strongly compatible lists, we can single out useless matching substitutions and improve the pairwise strongly compatible test itself.

The following three bit operations are used in this paper.

$w_1 + w_2$	:	bitwise disjunction of w_1 and w_2
$w_1 * w_2$	:	bitwise conjunction of w_1 and w_2
$\overline{w}$	:	bitwise complementation

where w_i is a bit sequence. For convenience, we denote $w_1 + \cdots + w_n$ by $+_{i=1}^{n} w_i$. Similarly, we denote $w_1 * \cdots * w_n$ by $*_{i=1}^{n} w_i$.

To test whether the given two matching substitutions are strongly compatible, we need the following definition.

Definition 3.1 Let $\{v_1, \cdots, v_n\}$ be an ordered set of variables in clause C, and a matching substitution σ between literals in clauses C and D be $\{t_1/s_1, \cdots, t_m/s_m\}$. $\delta(\sigma)$ is an n-length list such that the ith element is t_j if $v_i = s_j$, ϕ otherwise. $\delta(\sigma) = (t_1, \cdots, t_n)$ indicates that substitution σ does not substitute for variable v_i if t_i is ϕ, otherwise it substitutes t_i for v_i.

Example 3.1 Let $C = \{l_1, l_2\}$ and $D = \{k_1, k_2, k_3\}$ with $l_1 = p(X, Y)$, $l_2 = p(Y, Z)$, $k_1 = p(a, b)$, $k_2 = p(a, c)$, $k_3 = p(d, b)$ and $\{X, Y, Z\}$ be an ordered set of variables in C. We want to find all the matching substitutions between literals in C and literals in D. Then, we can obtain that $\sigma_1 = \{a/X, b/Y\}$, $\sigma_2 = \{a/X, c/Y\}$, $\sigma_3 = \{d/X, b/Y\}$, $\sigma_4 = \{a/Y, b/Z\}$, $\sigma_5 = \{a/Y, c/Z\}$, and $\sigma_6 = \{d/Y, b/Z\}$ for each (l_i, k_j) where $1 \leq i \leq 2$ and $1 \leq j \leq 3$. By Definition 3.1 we obtain that $\delta(\sigma_1) = (a, b, \phi)$, $\delta(\sigma_2) = (a, c, \phi)$, $\delta(\sigma_3) = (d, b, \phi)$, $\delta(\sigma_4) = (\phi, a, b)$, $\delta(\sigma_5) = (\phi, a, c)$, and $\delta(\sigma_6) = (\phi, d, b)$. □

If two matching substitution σ_1 and σ_2 are strongly compatible, they should not substitute for same variables differently. That is, if σ_1 substitutes a term t for a variable v then σ_2 has to substitute the term t for the variable v or does not have to substitute for the variable v. This can be formally described in Lemma 3.1.

Lemma 3.1 Let $\{v_1, \cdots, v_n\}$ be an ordered set of variables in clause C, and σ_1 and σ_2 be matching substitutions from literals in clause C to literals in clause D. σ_1 and σ_2 are strongly compatible if and only if $\pi_i(\delta(\sigma_1)) = \phi \vee \pi_i(\delta(\sigma_2)) = \phi \vee \pi_i(\delta(\sigma_1)) = \pi_i(\delta(\sigma_2))$ for each i $(1 \leq i \leq n)$, where $\pi_i(X)$ is a selection function which returns the ith element of list X.

(Proof) ($\leftarrow$) Each case is considered separately.

(i) in the case $\pi_i(\delta(\sigma_1)) = \phi$
Since $v_i\sigma_1 = v_i$ and clauses C and D are variable-disjoint, $v_i(\sigma_1 \bullet \sigma_2) = (v_i\sigma_1)\sigma_2 = v_i\sigma_2 = (v_i\sigma_2)\sigma_1 = v_i(\sigma_2 \bullet \sigma_1)$.

(ii) in the case $\pi_i(\delta(\sigma_2)) = \phi$
Since $v_i\sigma_2 = v_i$ and clauses C and D variable-disjoint, $v_i(\sigma_2 \bullet \sigma_1) = (v_i\sigma_2)\sigma_1 = v_i\sigma_1 = (v_i\sigma_1)\sigma_2 = v_i(\sigma_1 \bullet \sigma_2)$.

(iii) in the case $\pi_i(\delta(\sigma_1)) = \pi_i(\delta(\sigma_2))$
Since $v_i\sigma_1 = v_i\sigma_2$ and clauses C and D are variable-disjoint, $v_i(\sigma_1 \bullet \sigma_2) = (v_i\sigma_1)\sigma_2 = (v_i\sigma_2)\sigma_2 = v_i\sigma_2 = v_i\sigma_1 = (v_i\sigma_1)\sigma_1 = (v_i\sigma_2)\sigma_1 = v_i(\sigma_2 \bullet \sigma_1)$.

From (i), (ii), and (iii), if $\pi_i(\delta(\sigma_1)) = \phi$ or $\pi_i(\delta(\sigma_2)) = \phi$ or $\pi_i(\delta(\sigma_1)) = \pi_i(\delta(\sigma_2))$ for each i $(1 \leq i \leq n)$, then σ_1 and σ_2 are strongly compatible.

($\rightarrow$) Assume that $\pi_i(\delta(\sigma_1)) \neq \phi$, $\pi_i(\delta(\sigma_2)) \neq \phi$, and $\pi_i(\delta(\sigma_1)) \neq \pi_i(\delta(\sigma_2))$. By Definition 3.1, σ_1 and σ_2 contain s_1/v_i and s_2/v_i, respectively, where $s_1 \neq s_2$. Hence, $v_i(\sigma_1 \bullet \sigma_2) \neq v_i(\sigma_2 \bullet \sigma_1)$ (i.e. $\sigma_1 \bullet \sigma_2 \neq \sigma_2 \bullet \sigma_1$). This is contradictory to that σ_1 and σ_2 are strongly compatible. The proof is completed. □

Lemma 3.1 suggests a new method for testing whether the given two matching substitutions σ_1 and σ_2 are strongly compatible. That is, without calculating $\sigma_1 \bullet \sigma_2$ and $\sigma_2 \bullet \sigma_1$, we can determine whether σ_1 and σ_2 are strongly compatible by only comparing $\delta(\sigma_1)$ with $\delta(\sigma_2)$. For example, we can know that σ_1 and σ_4 in Example 3.1 are not strongly compatible because $\pi_2(\delta(\sigma_1)) \neq \phi \wedge \pi_2(\delta(\sigma_4)) \neq \phi \wedge \pi_2(\delta(\sigma_1)) \neq \pi_2(\delta(\sigma_4))$.

Definition 3.2 Let $\{v_1, \cdots, v_m\}$ be an ordered set of variables in clause C, and let $\{\sigma_1, \cdots, \sigma_n\}$ be an ordered set of matching substitutions from literals in clause C to literals in clause D. $P_i(X)$, $1 \leq i \leq m$, is an n-bit sequence such that its j'th bit is 1 if the i'th element of $\delta(\sigma_j)$ is X or ϕ, otherwise 0 for each j $(1 \leq j \leq n)$. Especially, when X is ϕ all bits of $P_i(X)$ are 1.

Example 3.2 From Example 3.1, we have $P_1(a) = 110111$, $P_1(d) = 001111$, $P_1(\phi) = 111111$, $P_2(a) = 000110$, $P_2(b) = 101000$, $P_2(c) = 010000$, $P_2(d) = 000001$, $P_3(\phi) = 111111$, $P_3(b) = 111101$, and $P_3(c) = 111010$. In this case, $P_1(a) = 110111$ indicates that variable v_1 having value a is compatible with substitutions 1,2,4,5,6 but not with the substitution 3. □

Let $\{\sigma_1, \cdots, \sigma_n\}$ be an ordered set of matching substitutions from literals in clause C to literals in clause D and m be the number of variables in C. Matching substitutions which are strongly compatible with σ_i, $1 \leq i \leq n$, can be represented by an n-bit sequence which is calculated by the following function $\beta(\sigma_i)$:

$$\beta(\sigma_i) = *_{j=1}^{m} P_j(\pi_j(\delta(\sigma_i))).$$

We call $\beta(\sigma_i)$ the *strongly compatible list* for σ_i.

Lemma 3.2 Let $\{v_1, \cdots, v_m\}$ be an ordered set of variables in clause C, and $\{\sigma_1, \cdots, \sigma_n\}$ be an ordered set of matching substitutions between literals in clauses C and literals in clause D. $\sigma \in \{\sigma_1, \cdots, \sigma_n\}$ and σ_k, $1 \leq k \leq n$, are strongly compatible if and only if the k'th bit of $\beta(\sigma)$ is 1.

(Proof) We must show that if the k'th bit of $*_{i=1}^{m} P_i(\pi_i(\delta(\sigma)))$ is 1 then σ and σ_k are strongly compatible, and also show that if σ is strongly compatible with σ_k then the k'th bit of $*_{i=1}^{m} P_i(\pi_i(\delta(\sigma)))$ is 1.

($\leftarrow$) From the fact that the k'th bit of $*_{i=1}^{m} P_i(\pi_i(\delta(\sigma)))$ is 1, the k'th bit of $P_i(\pi_i(\delta(\sigma)))$ is 1 for each i $(1 \leq i \leq m)$. By Definition 3.2, $\pi_i(\delta(\sigma)) = \phi$ or $\pi_i(\delta(\sigma)) = \pi_i(\delta(\sigma_k))$ or $\pi_i(\delta(\sigma_k)) = \phi$ for each i $(1 \leq i \leq m)$. Therefore, by Lemma 3.1, σ and σ_k are strongly compatible.

($\rightarrow$) From Lemma 3.1, $\pi_i(\delta(\sigma)) = \phi$ or $\pi_i(\delta(\sigma)) = \pi_i(\delta(\sigma_k))$ or $\pi_i(\delta(\sigma_k)) = \phi$ for each i $(1 \leq i \leq m)$. By Definition 3.2, the k'th element of $P_i(\pi_i(\delta(\sigma)))$ is 1 for each i $(1 \leq i \leq m)$. Therefore, the k'th bit of $*_{i=1}^{m} P_i(\pi_i(\delta(\sigma)))$ is 1. □

Example 3.3 From Example 3.2, we can obtain $\beta(\sigma_i)$ for each i $(1 \leq i \leq 6)$ as follows:

$\beta(\sigma_1) = P_1(a) * P_2(b) * P_3(\phi) = 110111 * 101000 * 111111 = 100000$

$\beta(\sigma_2) = P_1(a) * P_2(c) * P_3(\phi) = 110111 * 010000 * 111111 = 010000$

$\beta(\sigma_3) = P_1(d) * P_2(b) * P_3(\phi) = 001111 * 101000 * 111111 = 001000$

$\beta(\sigma_4) = P_1(\phi) * P_2(a) * P_3(b) = 111111 * 000110 * 111101 = 000100$

$\beta(\sigma_5) = P_1(\phi) * P_2(a) * P_3(c) = 111111 * 000110$

$* 111010 = 000010$
$\beta(\sigma_6) = P_1(\phi) * P_2(d) * P_3(b) = 111111 * 000001 * 111101 = 000001.$
From this, we know that each $\sigma_i (1 \leq i \leq 6)$ is strongly compatible with only itself. □

Some matching substitutions do not contribute to construct a matcher θ such that $C\theta \subseteq D$. If such matching substitutions can be identified and removed before the actual pairwise strongly compatible tests, we can reduce the effort to find a matcher θ. One class of such matching substitutions can be defined as follows:

Definition 3.3 Let $C = \{l_1, \cdots, l_m\}$ and D be clauses and σ a matching substitution mapping a literal in C onto a literal in D. If there is an $l_k \in \{l_1, \cdots, l_m\}$ such that any matching substitution in $uni(C, l_k, D)$ is not strongly compatible with σ then σ is *useless*.

Intuitively we know that a matching substitution σ is useless if the number of 1s in $\beta(\sigma)$ is less than m, the number of literals of C. But the number of 1s in $\beta(\sigma)$ is not always less than m though σ is useless. Let $C = \{l_1, \cdots, l_m\}$ and D be clauses and $\{\sigma_1, \cdots, \sigma_n\}$ be a set of matching substitutions from literals in C to literals in D. We can represent $uni(C, l_k, D)$ by an n-bit sequence M_{l_k} such that its i'th bit is 1 if $\sigma_i \in uni(C, l_k, D)$, otherwise 0 for each k and i $(1 \leq k \leq m, 1 \leq i \leq n)$. Given these n-bit sequences, we can easily test whether a matching substitution σ is useless, that is, if there is an l_k such that $M_{l_k} * \beta(\sigma) = 0$ then σ is useless.

Example 3.4 From Example 3.1, we have $uni(C, l_1, D) = \{\sigma_1, \sigma_2, \sigma_3\}$ and $uni(C, l_2, D) = \{\sigma_4, \sigma_5, \sigma_6\}$ and thus $M_{l_1} = 111000$ and $M_{l_2} = 000111$. We have $\beta(\sigma_k)$ for each k $(1 \leq k \leq 6)$ as shown in Example 3.3. Since $\beta(\sigma_i) * M_{l_2} = 0$ for each i $(1 \leq i \leq 3)$ and $\beta(\sigma_j) * M_{l_1} = 0$ for each j $(4 \leq j \leq 6)$, all σ_k $(1 \leq k \leq 6)$ are useless. □

Theorem 3.1 Let $C = \{l_1, \cdots, l_m\}$ and D be clauses. If $\sigma \in uni(C, l_k, D)$ $(1 \leq k \leq m)$ is a useless matching substitution then there is no θ such that $C\theta \subseteq D$ and $\theta = \sigma_1 \bullet \cdots \sigma_{k-1} \bullet \sigma \bullet \sigma_{k+1} \cdots \bullet \sigma_m$ where $\sigma_i \in uni(C, l_i, D)$ for each i $(1 \leq i \leq m, i \neq k)$.
(Proof) Let σ be a member of $uni(C, l_k, D)$ and a useless matching substitution. Suppose that there is a θ such that $C\theta \subseteq D$ and $\theta = \sigma_1 \cdots \bullet \sigma_{k-1} \bullet \sigma \bullet \sigma_{k+1} \cdots \bullet \sigma_m$ where $\sigma_i \in uni(C, l_i, D)$. By Theorem 2.1, $\sigma_1, \cdots, \sigma_{k-1}, \sigma, \sigma_{k+1}, \cdots, \sigma_m$ must be pairwise strongly compatible and thus σ is strongly compatible with $\sigma_i \in uni(C, l_i, D)$ for each i $(1 \leq i \leq m, i \neq k)$. $\sigma \in uni(C, l_k, D)$ is strongly compatible with itself. Therefore each $uni(C, l_i, D)$ for each i $(1 \leq i \leq m)$ has at least one matching substitution which is strongly compatible with σ. This is contradictory to that σ is a useless matching substitution. Hence, there is no such θ. □

By Theorem 3.1, it is not necessary to perform pairwise strongly compatible test on useless matching substitutions. If $\sigma \in uni(C, l_k, D)$ is a useless matching substitution then we can remove σ from $uni(C, l_k, D)$ without changing the result of subsumption tests. In Example 3.4, we know that clause C does not subsume clause D without pairwise strongly compatible tests, since all σ_i are useless.

Given two clauses C and D, there may be more than one matcher θ such that $C\theta \subseteq D$. To test that C subsumes D, we only find a matcher, that is, we have no need to find all matchers. By this property we can remove more matching substitutions.

Definition 3.4 Let $C = \{l_1, \cdots, l_m\}$ and D be clauses and $\{\sigma_1, \cdots, \sigma_n\}$ be an ordered set of matching substitutions from literals in C to literals in D. If $\sigma_i, \sigma_j \in uni(C, l_r, D)$ for some r $(1 \leq r \leq m)$ and σ_j is strongly compatible with each σ_k which is strongly compatible with σ_i then σ_j *includes* σ_i, denoted by $\sigma_i \leq_\sigma \sigma_j$, where $k \neq i$ and $k \neq j$.

Let $\{\sigma_1, \cdots, \sigma_n\}$ be an ordered set of matching substitutions and $\gamma(\sigma_i)$ be $\beta(\sigma_i) * \overline{\mu_i^n}$, where $\mu_i{}^n$ is the n-bit sequence such that the value of its i'th bit is 1 and all remaining bits are 0. Then, we can easily test the $\leq_\sigma$-relation by bit operations, i.e. if $\gamma(\sigma_i) * \overline{\gamma(\sigma_j)} = 0$ then $\sigma_i \leq_\sigma \sigma_j$.

Example 3.5 Let $C = \{p(X), q(Y)\}$, $D = \{p(a), p(b), q(a), q(b)\}$ and $\{X, Y\}$ be an ordered set of variables in C. Then we have $\sigma_1 = \{a/X\}$, $\sigma_2 = \{b/X\}$, $\sigma_3 = \{a/Y\}$, and $\sigma_4 = \{b/Y\}$. By Definition 3.1, we have $\delta(\sigma_1) = (a, \phi)$, $\delta(\sigma_2) = (b, \phi)$, $\delta(\sigma_3) = (\phi, a)$, $\delta(\sigma_4) = (\phi, b)$. We can calculate the following strongly compatible lists of matching substitutions:
$\beta(\sigma_1) = P_1(a) * P_2(\phi) = 1011 * 1111 = 1011$
$\beta(\sigma_2) = P_1(b) * P_2(\phi) = 0111 * 1111 = 0111$
$\beta(\sigma_3) = P_1(\phi) * P_2(a) = 1111 * 1110 = 1110$
$\beta(\sigma_4) = P_1(\phi) * P_2(b) = 1111 * 1101 = 1101.$
From this strongly compatible lists, we can obtain $\gamma(\sigma_1)$ and $\gamma(\sigma_2)$ as follows:
$\gamma(\sigma_1) = \beta(\sigma_1) * \overline{\mu_1^4} = 1011 * 0111 = 0011$
$\gamma(\sigma_2) = \beta(\sigma_2) * \overline{\mu_2^4} = 0111 * 1011 = 0011.$
Since $\gamma(\sigma_1) * \overline{\gamma(\sigma_2)} = 0$, we obtain the relation $\sigma_1 \leq_\sigma \sigma_2$. Similarly, we obtain the relation $\sigma_3 \leq_\sigma \sigma_4$. □

Theorem 3.2 Let $C = \{l_1, \cdots, l_m\}$ and D be clauses, $\sigma \in uni(C, l_k, D)$ and $\sigma' \in uni(C, l_k, D)$ for some k $(1 \leq k \leq m)$, $\sigma \leq_\sigma \sigma'$ and $\sigma_i \in uni(C, l_i, D)$ for each i $(1 \leq i \leq m, i \neq k)$. If there is a θ such that $C\theta \subseteq D$ and $\theta = \sigma_1 \bullet \cdots \bullet \sigma_{k-1} \bullet \sigma \bullet \sigma_{k+1} \cdots \bullet \sigma_m$, then there is a θ' such that $C\theta' \subseteq D$ and $\theta' = \sigma_1 \bullet \cdots \bullet \sigma_{k-1} \bullet \sigma' \bullet \sigma_{k+1} \cdots \bullet \sigma_m$.
(Proof) Let us suppose that there is a θ such that $C\theta \subseteq D$ and $\theta = \sigma_1 \bullet \cdots \bullet \sigma_{k-1} \bullet \sigma \bullet \sigma_{k+1} \cdots \bullet \sigma_m$. Then, $\sigma_1, \cdots \sigma_{k-1}, \sigma, \sigma_{k+1} \cdots \sigma_m$ are pairwise strongly compatible by Theorem 2.1. Since σ' includes σ, σ' is strongly compatible with $\sigma_1, \cdots, \sigma_{k-1}, \sigma_{k+1}, \cdots, \sigma_m$. Thus $\sigma_1, \cdots, \sigma_{k-1}, \sigma', \sigma_{k+1}, \cdots, \sigma_m$ are pairwise strongly compatible. Therefore, there is a θ' such that $C\theta' \subseteq D$ and $\theta' = \sigma_1 \bullet \cdots \bullet \sigma_{k-1} \bullet \sigma' \bullet \sigma_{k+1} \cdots \bullet \sigma_m$ by Theorem 2.1. □

By Theorem 3.2, we do not need perform a strongly compatible test on the combinations of matching substitutions which contain a matching substitution σ_1 such that $\sigma_1 \in uni(C, l_i, D)$, $\sigma_2 \in uni(C, l_i, D)$, and $\sigma_1 \leq_\sigma \sigma_2$. In Example 3.5, we can remove σ_1 and σ_3 because σ_2 and σ_4 include σ_1 and σ_3, respectively.

As Theorem 3.1 (useless theorem) and Theorem 3.2 (included theorem) suggest, we can remove the useless or included matching substitution before we take a pairwise strongly compatible test. We call a matching substitution which is either useless or included *unnecessary*.

One phenomenon we want to point out is that a matching substitution becomes unnecessary due to the propagation of deletion, so needs to be deleted. Therefore we should keep deleting unnecessary matching substitutions until there is no more such matching substitution. For examples, let σ_1, σ_2 and σ_3 be matching substitutions from literals in C to literals in D, and let the number of literals in C be 3. Suppose that σ_1 is strongly compatible with σ_2 and σ_3, and σ_2 is not strongly compatible with σ_3. Then σ_1 is not a useless matching substitution. However, the removal of useless matching substitutions, σ_2 and σ_3, causes σ_1 to be a useless matching substitution and thus it can be removed.

Let $C = \{l_1, \cdots, l_n\}$, and D clause. Then, in the worst case $O(n^2)$ strongly compatible tests will be needed for each combination $(\sigma_1, \cdots, \sigma_n) \in \times_{i=1}^{n} uni(C, l_i, D)$ in order to check C subsumes D. However, given $\beta(\sigma_i)$ we can enhance the performance of a subsumption test by the following theorem.

Theorem 3.3 Let $C = \{l_1, \cdots, l_m\}$ and D be clauses, $\{\sigma_1, \cdots, \sigma_n\}$ be a set of matching substitutions from literals in C to literals in D, and $\{\sigma_{x_1}, \cdots, \sigma_{x_m}\}$ be a subset of $\{\sigma_1, \cdots, \sigma_n\}$. There is a $\theta = \sigma_{x_1} \bullet \cdots \bullet \sigma_{x_m}$ such that $C\theta \subseteq D$ and $\sigma_{x_k} \in uni(C, l_k, D)$ for each k $(1 \leq k \leq m)$ if only if $*_{k=1}^{m} \beta(\sigma_{x_k}) * +_{k=1}^{m} {\mu_{x_k}}^n = +_{k=1}^{m} {\mu_{x_k}}^n$.

(Proof) ($\leftarrow$) Since $*_{k=1}^{m} \beta(\sigma_{x_k}) * +_{k=1}^{m} {\mu_{x_k}}^n = +_{k=1}^{m} {\mu_{x_k}}^n$, by Lemma 3.2, $\sigma_{x_1}, \cdots, \sigma_{x_m}$ are strongly compatible with each of $\{\sigma_{x_1}, \cdots, \sigma_{x_m}\}$. Therefore $\sigma_{x_1}, \cdots, \sigma_{x_m}$ are pairwise strongly compatible. Thus, by Theorem 2.1, there is a $\theta = \sigma_{x_1} \bullet \cdots \bullet \sigma_{x_m}$ such that $C\theta \subseteq D$ and $\sigma_{x_k} \in uni(C, l_k, D)$ for each k $(1 \leq k \leq m)$
($\rightarrow$) By Theorem 2.1, $\sigma_{x_1}, \cdots, \sigma_{x_m}$ are pairwise strongly compatible. Therefore, by Lemma 3.2, the x_i bit of $\beta(\sigma_{x_k})$ for each i, k $(1 \leq i, k \leq m)$ is 1. Thus $*_{k=1}^{m} \beta(\sigma_{x_k}) * +_{k=1}^{m} {\mu_{x_k}}^n = +_{k=1}^{m} {\mu_{x_k}}^n$. □

Now we can formulate a new algorithm that returns a pairwise strongly compatible set $\{\sigma_1, \cdots, \sigma_m\}$ such that $(\sigma_1, \cdots, \sigma_m) \in \times_{i=1}^{m} uni(C, l_i, D)$ if exists, otherwise return $\{\}$. The detail algorithm, *Pairwise Strongly Compatible Test* (PSCT), is described in Figure 1 and it can be summarized as follows:

1. Calculate the strongly compatible list for each matching substitution.
2. Remove unnecessary matching substitutions until there is no such matching substitution.
3. Find out an m-tuple $(\sigma_1, \cdots, \sigma_m)$ such that $*_{k=1}^{m} \beta(\sigma_k) * +_{k=1}^{m} {\mu_k}^n = +_{k=1}^{m} {\mu_k}^n$.

Example 3.6 Given $C = \{p(X,Y), r(Y,Z), s(X,Z)\}$ and $D = \{p(b,a), p(a,b), r(a,d), r(b,c), s(a,d), s(a,c)\}$ we want to find out a substitution θ such that $C\theta \subseteq D$. Let $\{X, Y, Z\}$ be an ordered set of variables in C. Then, we can obtain that

$M_{p(X,Y)} = 110000$, $M_{r(Y,Z)} = 001100$, $M_{s(X,Z)} = 000011$,
$\beta(\sigma_1) = 101000$, $\beta(\sigma_2) = 010111$, $\beta(\sigma_3) = 101010$,
$\beta(\sigma_4) = 010101$, $\beta(\sigma_5) = 011010$, $\beta(\sigma_6) = 010101$.

Since $\beta(\sigma_1) * M_{s(X,Z)} = 0$, σ_1 is removed and thus the strongly compatible lists are adjusted as follows:

$\beta(\sigma_2) = 010111$, $\beta(\sigma_3) = 001010$, $\beta(\sigma_4) = 010101$,
$\beta(\sigma_5) = 011010$, $\beta(\sigma_6) = 010101$.

Since $\beta(\sigma_3) * M_{p(X,Y)} = 0$, σ_3 is useless. By further removing the useless matching substitution σ_3, we can obtain that

$\beta(\sigma_2) = 010111$, $\beta(\sigma_4) = 010101$, $\beta(\sigma_5) = 010010$,
$\beta(\sigma_6) = 010101$.

Since $\beta(\sigma_5) * M_{r(Y,Z)} = 0$, σ_5 is useless and thus removed. Consequently we can obtain following strongly compatible lists:

$\beta(\sigma_2) = 010101$, $\beta(\sigma_4) = 010101$, $\beta(\sigma_6) = 010101$.

Since $\beta(\sigma_2) * \beta(\sigma_4) * \beta(\sigma_6) * 010101 = 010101$, σ_2, σ_4 and σ_6 are pairwise strongly compatible. Thus, there is a substitution $\theta = \sigma_2 \bullet \sigma_4 \bullet \sigma_6 = \{a/X, b/Y, c/Z\}$. □

Our Algorithm PSCT
Input: clauses $C = \{l_1, \cdots, l_m\}$ and D
Output: a pairwise strongly compatible set $\{\sigma_1, \cdots, \sigma_m\}$ such that $(\sigma_1, \cdots, \sigma_m) \in \times_{i=1}^{m} uni(C, l_i, D)$

1. Calculate $\beta(\sigma)$ for all $\sigma \in \bigcup_{i=1}^{m} uni(C, l_i, D)$.
2. Let I be an n-bit sequence such that all its bits are 0.
   (a) for each $\sigma_k \in \bigcup_{i=1}^{m} uni(C, l_i, D)$, if σ_k is useless then
      i. remove σ_k.
      ii. $\mathrm{I} = \mathrm{I} + {\mu_k}^n$.
   (b) for each $\sigma_k \in \bigcup_{i=1}^{m} uni(C, l_i, D)$, if there is a σ_l such that $\sigma_k \leq_\sigma \sigma_l$ then
      i. remove σ_k.
      ii. $\mathrm{I} = \mathrm{I} + {\mu_k}^n$.
   (c) for each $\sigma_k \in \bigcup_{i=1}^{m} uni(C, l_j, D)$, $\beta(\sigma_k) = \beta(\sigma_k) * \overline{I}$.
3. If $uni(C, l_i, D) = \{\}$ for some i, then return $\{\}$.
4. Repeat step 2$\sim$3 until there is no unnecessary matching substitution.
5. For each m-tuple $(\sigma_{i_1}, \cdots, \sigma_{i_m})$ where $\sigma_{i_k} \in uni(C, l_k, D)$,
   if $*_{k=1}^{m} \beta(\sigma_{i_k}) * +_{k=1}^{m} {\mu_{i_k}}^n = +_{k=1}^{m} {\mu_{i_k}}^n$, then return $\{\sigma_{i_1}, \cdots, \sigma_{i_m}\}$.
6. return $\{\}$.

Figure 1: Algorithm PSCT

4 Related Works and Analysis

This section compares our algorithm with the two existing s-link tests, namely Eisinger's algorithm and Socher's algorithm. The analysis is based on the number of string comparisons to determine whether a clause $C = \{l_1, \cdots, l_m\}$ subsumes a clause D. To measure the complexity of three algorithms, we use the following symbols:

r: the maximal arity of predicate symbols occurring in literals in clauses C and D.

N_C: the number of distinct variables in a literal in clauses C and D.

N_D: the number of distinct terms which are substituted for a variable in clause C.

N_S: the number of strongly compatible tests needed to see whether m matching substitutions between literals are pairwise strongly compatible.

N_P: the number of pairwise strongly compatible tests needed to find a matcher θ such that $C\theta \subseteq D$.

To simplify the analysis, we assume that the number of matching substitutions in each $uni(C, l_i, D)$ $(1 \leq i \leq m)$ is equal and let it be k.

In Eisinger's algorithm, subsumption tests for long clauses with more than one matching substitution for each literal may require an expensive search of all elements of the Cartesian product. Since compositions of substitutions are needed to see whether two given substitutions are strongly compatible and $O(N_C^2)$ string comparisons are needed for each strongly compatible test, $O(N_S N_C^2)$ string comparisons are needed for each pairwise strongly compatible test. Thus $O(N_P N_S N_C^2)$ string comparisons are needed for the subsumption test. Since $k^2 \leq N_P \leq k^m$, $1 \leq N_S \leq \frac{m(m-1)}{2}$, and $1 \leq N_C \leq r$, in the worst case $N_C = r$, $N_S = m(m-1)/2$ and $N_P = k^m$, so the worst-case time complexity of Eisinger's is $O(k^m m^2 r^2)$.

Socher proposes an improvement of the s-link test for subsumption of two clauses [Socher 1988]. He improves the search for θ such that $C\theta \subseteq D$ by imposing a restriction on the possible matching substitutions. It is based on the idea of giving the variables and literals of a clause a characteristic property, which in fact denotes information about the occurrences of variables in various argument positions of a literal. An order for these characteristic is defined and it is shown that the order is compatible with the matching substitution σ from C to D. Thus all matching substitutions that do not respect the order can be singled out. However, he does not improve the pairwise strongly compatible test itself and thus does not reduce the worst-case time complexity of the s-link test.

In some cases Socher's algorithm can not single out a matching substitution which is either useless or included matching substitution. For example, let $C = \{p(X,Y), q(Y,X)\}$ and $D = \{p(a,c), p(b,d), q(c,b), p(d,a)\}$ be given. No matching substitution is singled out because the characteristic matrices of literals p and q in C are equal to those ones in D. However, all matching substitutions are useless in our approach, so no pairwise strongly compatible test is performed.

By using strongly compatible lists and bit operations, we improve the pairwise strongly compatible test and thus reduce the worst-case time complexity of the s-link test. $O(km^2 N_C N_D)$ string comparisons are needed to calculate all strongly compatible lists. $O(m)$ bit-conjunctions are needed for a pairwise strongly compatible test when m strongly compatible lists are given. Thus, $O(km^2 N_C N_D)$ string comparisons and $O(mN_P)$ bit-conjunctions are needed for a subsumption test. Since $k^2 \leq N_P \leq k^m$, $1 \leq N_C \leq r$, and $1 \leq N_D \leq km$, in the worst case $N_P = k^m$, $N_C = r$ and $N_D = km$, so the worst-case time complexity is $O(k^2 r m^3$ string comparisons $+ mk^m$ bit-conjunctions).

Let n be the ratio of the time complexity of a string comparison to the time complexity of a bit-conjunction. Then, in the case that $k^2 r m^3$ is greater than $\frac{mk^m}{n}$, the worst-case time complexity of our algorithm is $O(k^2 r m^3)$ and we can reduce the worst-case time complexity of Eisinger's algorithm by $O(\frac{k^{m-2}r}{m})$. In the other case, the worst-time complexity of our algorithm is $O(\frac{mk^m}{n})$ and we can reduce the worst-case time complexity of Eisinger's algorithm by $O(mr^2 n)$.

5 Conclusions

Subsumption tests for long clauses with more than one matching substitution for each literal may require an excessive search for all elements in the Cartesian product. We have presented a new subsumption algorithm, called PSCT algorithm, which has a lower worst-case time complexity than the existing methods. The efficiency of our algorithm is based on the following facts.

1. Construction of strongly compatible lists allows us to identify unnecessary matching substitutions at the early stage of the subsumption test. Such matching substitutions are removed and are not involved at the actual pairwise strongly compatible test to come. This filtering process reduces the number of possible combinations of matching substitutions clearly.

2. As for the pairwise strongly compatible test itself, the test is carried out efficiently due to the appropriate bit operations on the strongly compatible lists which are already constructed.

The approaches [Socher 1988, Eisinger 1981] that actually compose the matching substitutions to check pairwise compatibility are considered to be slow and expensive. In most cases our approach outperforms others [Socher 1988, Eisinger 1981] even though it may involve the cost overhead for computation of the strongly compatible lists of matching substitutions. Furthermore, it should be noted that our subsumption algorithm can be used in general theorem proving approach even though it is described in the context of the connection graph proof procedure in this paper.

References

[Andrew 1981] P. B. Andrews, Theorem Proving via General Matings, *Journal of ACM* **28** (2) (1981) 193-214.

[Bibel 1981] W. Bibel, On Matrices with Connections, *Journal of ACM* **28** (4) (1981) 633-645.

[Chang and Lee 1973] C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving* (Academic Press, New York, 1973).

[Eisinger 1981] N. Eisinger, Subsumption and Connection Graph, in: *Proceedings of the IJCAI-81* (1981) 480-486.

[Gottlob and Leitsch 1985] G. Gottlob and A. Leitsch, On the Efficiency of Subsumption Algorithms, *Journal of ACM* **32** (2) (1985) 280-295.

[Juang et al. 1988] J. Y. Juang, T. L. Huang, and E. Freeman, Parallelism in Connection Graph-based Logic Inference, in: *Proceedings of the 1988 International Conference on Parallel Processing* **2** (1988) 1-8.

[Kowalski 1975] R. Kowalski, A Proof Procedure using Connection Graphs, *Journal of ACM* **22** (4) (1975) 572-595.

[Kowalski 1979] R. Kowalski, *Logic for Problem Solving* (North Holland, Oxford, 1979).

[Loganantharaj 1986] R. Loganantharaj, Parallel Theorem Proving with Connection Graphs, in: *Proceedings of 8th International Conference on Automated Deduction* (1986) 337-352.

[Loganantharaj 1987] R. Loganantharaj, Parallel Link Resolution of Connection Graph Refutation and its Implementation, in: *Proceedings of International Conference on Parallel Processing* (1987) 154-157.

[Loveland 1978] D. Loveland, *Automated Theorem Proving: A Logical Basis* (North-Holland, Amsterdam, 1978).

[Munch 1988] K. H. Munch, A New Reduction Rule for the Connection Graph Proof Procedure, *Journal of Automated Reasoning* **4** (1988) 425-444.

[Robinson 1965] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *Journal of ACM* **12** (1) (1965) 23-41.

[Sickel 1976] S. Sickel, A Search Technique for Clause Interconnectivity Graphs, *IEEE Transaction on Computers* **25** (8) (1976) 823-835.

[Socher 1988] R. Socher, A Subsumption Algorithm Based on Characteristic Matrices, in: *Proceedings of 9th International Conference on Automated Deduction* (1988) 573-581.

[Stillman 1973] R. B. Stillman, The Concept of Weak Substitution in Theorem-Proving, *Journal of ACM* **20** (4) (1973) 648-667.

[Wos 1986] L. Wos, Automated Reasoning: Basic Research Problems, Argonne National Laboratory, Technical Memorandum No.67, March 1986.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

On the Duality of Abduction and Model Generation

Marc Denecker * Danny De Schreye †

Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium.
e-mail : {marcd, dannyd}@cs.kuleuven.ac.be

Abstract

We present a duality relationship between abduction for definite abductive programs and model generation on the only-if part of these programs. As was pointed out by Console et all, abductive solutions for an abductive program correspond to models of the only-if part. We extend this observation by showing that the procedural semantics of abduction itself can be interpreted dually as a form of model generation on the only-if part. This model generation extends Satchmo with an efficient treatment of equality. It is illustrated how this duality allows to improve current procedures for both abduction and model generation by transferring technical results known for one of these computational paradigms to the other.

1 Introduction

The work we report on this paper was motivated by some recent progress made in the field of Logic Programming to formalize abductive reasoning as logic deduction (see [Console *et al.*, 1991] and [Bry, 1990]). In [Kowalski, 1991], R. Kowalski presents the intuition behind this approach. He considers the following simple definite abductive logic program:

P = { wobbly-wheel ← flat-tyre.
wobbly-wheel ← broken-spokes.
flat-tyre ← punctured-tube.
flat-tyre ← leaky-valve. }

where the predicates broken-spokes, punctured-tube and leaky-valve are the abducibles. Given a query Q = ← *wobbly-wheel*, abductive reasoning allows to infer the assumptions:

S_1 = { punctured-tube },
S_2 = { leaky-valve }, and
S_3 = { broken-spokes } .

These sets of assumptions are abductive solutions to the given query $\leftarrow Q$ in the sense that for each S_i, we have that $P \cup S_i \models Q$.

Kowalski points out that we can equally well obtain these solutions by deduction, if we first transform the abductive program $P \cup \{Q\}$ into a new logic theory T. The transformation consists of taking the only-if part of every definition of a non-abducible predicate in the Clark-completion of P and by adding the negation of Q. In the example, we obtain the (non-Horn) theory T:

T = { wobbly-wheel → flat-tyre, broken-spokes.
flat-tyre → punctured-tube, leaky-valve.
wobbly-wheel ← }

Minimal models for this new theory T are:

M_1 = { wobbly-wheel, flat-tyre, punctured-tube },
M_2 = { wobbly-wheel, flat-tyre, leaky-valve },
and
M_3 = { wobbly-wheel, broken-spokes }.

Restricting these models to the atoms of the abducible predicates only, we precisely obtain the three abductive solutions S_1, S_2 and S_3 of the original problem.

The above observation points to an interesting issue; namely the possibility of linking these dual declarative semantics by completely equivalent dual procedures. Figure 1 shows this duality between an SLD+ Abduction tree (see [Cox and Pietrzykowski, 1986]) and the exectution tree of Satchmo, a theorem prover based on model generation ([Manthey and Bry, 1987]).

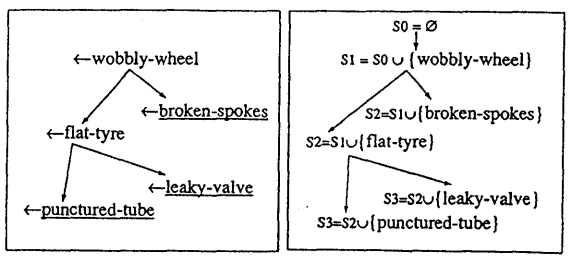


Figure 1: Procedural Duality of Abduction and Satchmo

*supported by the Belgian "Diensten voor Programmatie van Wetenschapsbeleid", under the contract RFO-AI-03
†supported by the Belgian National Fund for Scientific Research

Although this example illustrates the potential of using deduction or more precisely, model generation, as a formalisation of abductive reasoning, an obvious restriction of the example is that it is only propositional. Would this approach also hold for the general case of definite abductive programs? An example of a non-propositional program and its only-if part is given in figure 2.

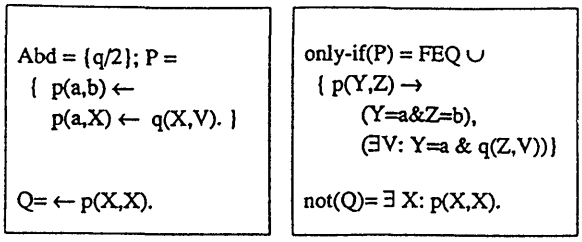


Figure 2: A predicate example

The theory only-if(P) consists not only of the only-if part of the definitions of the predicates but comprises also the axioms of Free Equality (FEQ), also known as Clark Equality ([Clark, 1978]). The abductive solutions and models of only-if(P) are displayed in figure 3.

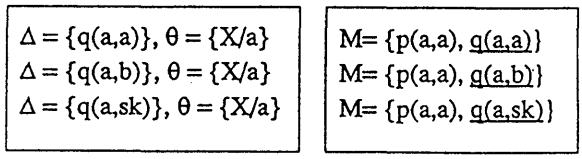


Figure 3: Abductive solutions and models

The duals of the abductive solutions are again identical to models of only-if(P). This example suggests that at least the duality on the level of declarative semantics is maintained. However, on the level of procedural semantics, some difficulties arise. The SLD+Abduction derivation tree is given in figure 4.

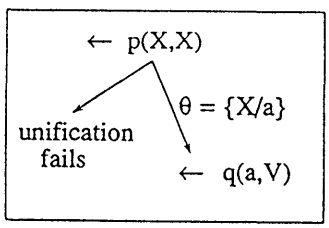


Figure 4: Abductive derivation tree

After skolemisation of the residue $\leftarrow q(a, V)$, we obtain the third abductive solution. With respect to the model generation, the theory only-if(P) is not clausal, however the extension of Satchmo, Satchmo_1 ([Bry, 1990]), can deal with such formulas directly (without normalisation to clausal form). Without dealing with the technical details of the computation, figure 5 presents the computation tree.

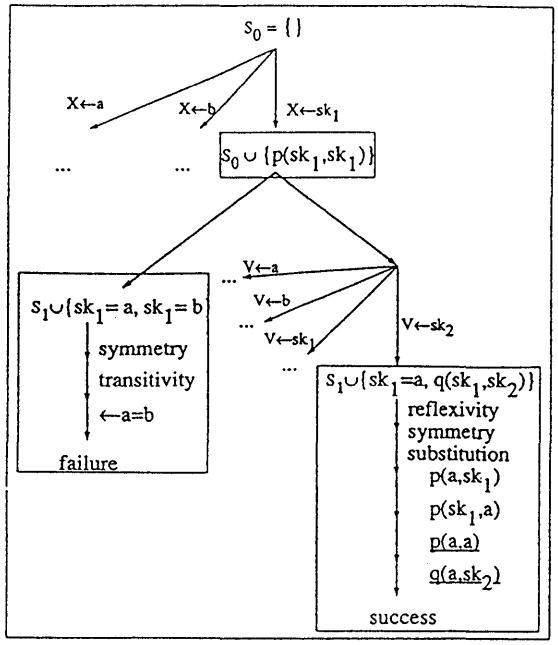


Figure 5: Execution tree of Satchmo_1

Globally, the structure of the SLD+abduction tree of figure 4 can still be seen in the Satchmo_1-tree. Striking is the *duality* of variables in the abductive derivation and skolem constants in the model generation. However, one difference is that the Satchmo_1 tree comprises many additional inference steps due to the application of the axioms of FEQ[1]. In the abductive derivation these additional steps correspond to the unification operation (e.g. on both left-most branches, the failure of the unification of {X=a, X=b} corresponds to the derivation of the inconsistency of the facts $\{sk_1 = a, sk_1 = b\}$).

Another difference is that the generated model

$$\{p(a,a), q(a,sk_2),\ p(sk_1,a), p(a,sk_1), p(sk_1,sk_1),\ q(sk_1,sk_2), sk_1 = a, a = sk_1, a = a, sk_1 = sk_1, sk_2 = sk_2\}$$

is much larger than the model which is dual to the abuctive solution. Satchmo_1 generates besides the atoms of this model also all logical implications of FEQ, comprising all substitutions of a by sk_1. It is clear that in general this will lead to an exponential explosion.

However, observe that we obtain the desired model by *contracting* sk_1 and a in the generated model. Therefore, extending Satchmo_1 with methods for dynamic contraction of equal elements would solve the efficiency problem and would restore the duality on the level of declarative semantics.

Contraction of a model is done by taking one unique witness out of every equivalence class of equal terms and

[1]Improper use of Satchmo_1: Equality in head of rule.

replacing all terms in the facts of the model by their witnesses. Techniques from Term Rewriting can be used to implement this. The procedural solution is to consider the set of inferred equality facts as a Term Rewriting System (TRS), to transform the set to an equivalent *complete* TRS, and to *normalise* all facts in the model using this complete TRS, and this after each forward derivation step in Satchmo_1.

This procedure may seem alien to Logic programming, but the contrary is true. As a mather of fact, the proposed procedure appears to be exactly the dual of techniques used in SLD+Abduction:

- the completion procedure corresponds dually to unification.
  The dual of the mgu (by replacing variables by skolem constants) is the completion of the set of equality atoms.
- the normalisation corresponds dually to applying the mgu.

Therefore, incorporating these techniques in Satchmo_1 would also restore the duality on the level of procedural semantics.

The research reported in this paper started as a mathematical exercise in duality. However, there are clearly spinoffs. One application is the extension of Satchmo_1 with efficient treatment of equality. We propose a framework for model generation under an arbitrary equality theory and we formally proof the duality of SLD+abduction in the instance of the framework, obtained by taking FEQ as the equality theory. Also for abduction there are spinoffs. An illustration of this is found in the context of planning as abduction in the event calculus. The event calculus contains a clause, saying that a property holds at a certain moment if there is an earlier event which initiates this property, and the property is not terminated (clipped) in between:

$$holds\_at(P,T) \leftarrow happens(E), initiates(E,P), \\ E < T, \neg clipped(E,P,T).$$

A planner uses this clause to introduce new events which initialise some desired property. Technically this is done by first skolemising and then abducing the *happens* goal. However, skolemisation requires explicit treatment of the equality predicate as an abducible satisfying FEQ ([Eshghi, 1988]). The techniques proposed in this paper allow efficient treatment of the abduced equality atoms, and provide a declarative semantics for it.

The paper is structured as follows. In section 2, we present the class of theories for which the model generation is designed. Section 3 recalls basic concepts of Term Rewriting. In section 4, the framework for model generation is presented and important semantic results are formulated. In section 5, the duality with abductive reasoning is formalised. Section 6 discusses future and related work. Due to space restrictions, all proofs are omitted. We refer to [Denecker and De Schreye, 1991] for the explicit proofs.

2 Extended programs.

In this section we introduce the formalism for which the model generation will be designed. This formalism should at least contain any theory that can be obtained as the only-if part of the definition in the Clark-completion of definite logic programs. The extended clause formalism introduced below, generalises both this kind of formulas and the clausal form.

Definition 2.1 *Let L be a first order language.*
An extended clause *or* rule *is a closed formula of the type:* $\forall(G_1,\ldots,G_k \rightarrow E_1,\ldots,E_l)$
where E_i has the general form:

$$\exists Y_1,..,Y_m : s_1 = t_1 \& \ldots \& s_g = t_g \& F_1 \& \ldots \& F_h$$

such that all G_i are atoms based on L, all F_i are non-equality atoms based on L

Definition 2.2 *An extended program is a set of extended clauses.*

Interestingly, the extended clause formalism can be proven to provide the full expressivity of first order logic. Any first order logic theory can be translated to a logically equivalent extended program, in the sense that they share exactly the same models. (Recall that the equivalence between a theory and its clausal form is much weaker: the theory is consistent iff its clausal form is consistent.)

In the sequel, the theory of general equality (resp. the theory of Free Equality), for a first order language L will be denoted EQ(L) (resp. FEQ(L)). A theory T, based on L, is called a *theory with equality* if it comprises EQ(L). A theory T, based on L is called an *equality theory* if it is a theory with equality in which "=" is the only predicate symbol in all formulas except for the substitution axioms of EQ(L).

3 Concepts of Term Rewriting.

The techniques we intend to develop for dealing with equality, are inspired by Term Rewriting. However, work in this area is too restricted for our purposes, because the concepts and techniques assume the general equality theory EQ underlying the term rewriting. To be able to deal with FEQ, we extend the basic concepts for the case of an arbitrary underlying equality theory E. In the sequel, equality and identity will be denoted distinctly when ambiguity may occur, resp. by "=" and "≡". We assume the reader to be familiar with basic notions of TRS's (see

e.g. [Dershowitz and Jouannaud, 1989]). We just recall some general ideas. A TRS γ associates to each term s a reduction tree in which each branch consists of successive applications of rewrite rules of γ. If γ is *noetherian*, these trees are all finite. If moreover γ is *Church-Rosser* or *confluent*, all leaves of the reduction tree of any term t contain the same term, called the normalisation of t and denoted $t.\gamma$. In Term Rewriting, such a TRS is called *complete*. Below we extend this concept.

Definition 3.1 *Let E be an equality theory based on a language L, γ a Term Rewriting System based on L.*

γ is complete wrt to <L,E> iff γ is noetherian and Church-Rosser and, moreover, <L,E+γ> has a Least Herbrand Model, which consists of all ground atoms $s = t$ constructed from terms in HU(L) such that $s.\gamma \equiv t.\gamma$.

This definition extends the normal definition in Term Rewriting by the third condition. However, for E = EQ, it has been proved that this property is implied by the noetherian and Church-Rosser properties (for a proof see [Huet, 1980]). Of course this is not the case for an arbitrary equality theory (as FEQ).

Definition 3.2 *A completion of a TRS γ wrt <L,E> is:*

- $\{\Box\}$ *if <L,E+γ> is inconsistent*
- *a complete TRS γ_c, such that <L,E> $\models \gamma \leftrightarrow \gamma_c$*

Our framework for model generation is developed for logical theories consisting of two components, an extended program P and an *underlying* equality theory E. This distinction reflects the fact that the model generation mechanism applies only to the extended clauses of P, while E is dealt with in a procedural way, using completion and normalisation. However, in order to make this possible, E should satisfy severe conditions, which are formulated in the following definition.

Definition 3.3 *An* equality theory with completion, *E, based on a language L, is a clausal equality theory equipped with a* language independent *completion procedure.*

The latter condition means that if γ is a ground Term Rewriting System based on an extension L' of L by skolem constants, and γ_c is the completion of γ wrt to <L',E>, then for any further extension L" of L' by skolem constants, γ_c is still the completion of γ wrt <L",E>.

We denote γ_c as TRS-comp(γ).

4 A framework for Model Generation

Informally a model generator constructs a sequence[1] $(Cl_d, j_d)_1^n$, where Cl_d is the ground instance of a rule applied after d steps, and j_d the index indicating the conclusion of Cl_d that was selected, an increasing sequence of sets of *asserted* ground facts $(M_d)_0^n$ of non-equality predicates, a sequence of complete Term Rewriting Systems $(\gamma_d)_0^n$, each of which is equivalent with the set of *asserted* equality facts, and an increasing sequence of sets of skolem constants $(Sk_d)_0^n$, obtained by skolemizing the existentially quantified variables. Formally:

Definition 4.1 *Let L be a language, L_{sk} an infinite countable alphabet of skolem constants, T an extended program based on L consisting of an equality theory with completion E with completion function TRS-comp and P an extended program.*

An Nondeterministic Model Generator with Equality *(NMGE) K is a tuple of four sequences $(Sk_d)_0^n$, $(M_d)_0^n$, $(\gamma_d)_0^n$ and $(Cl_d, j_d)_1^n$ where $n \in \mathbb{N} \cup \{\infty\}$. The sequences satisfy the following conditions:*

1. $M_0 = Sk_0 = \{\}; \gamma_0 = TRS\text{-}comp(\{\})$
2. *for each d such that $0 < d \leq n$, Cl_d, j_d, Sk_d, M_d and γ_d are obtained from Sk_{d-1}, M_{d-1} and γ_{d-1} by applying the following steps:*

   (a) Selection of rule and conclusion

   Define LHM_{d-1} as:
   $LHM(<L+Sk_{d-1}, EQ(L)+ M_{d-1}+ \gamma_{d-1}>)$
   Select nondeterministically a ground normal *instance of a rule of P*

   $$Cl_d = "G_1, \ldots, G_k \rightarrow E_1, \ldots, E_l"$$

   such that $G_1, \ldots, G_k$ hold in LHM_{d-1}. If $l = 0$, define $Sk_d = \{\}$, $M_d = \gamma_d = \{\Box\}$ and $n = d$.

   Otherwise, select nondeterministically a conclusion E_j from the head $E_1, \ldots, E_l$. Define $j_d = j$. We say that the rule Cl_d applies [with its j_d'th conclusion].

   (b) Skolemisation

   Let E_{j_d} be of the form: $\exists Y_1, \ldots, Y_m$:
   $s_1 = t_1 \& \ldots \& s_g = t_g \& F_1 \& \ldots \& F_h$
   Replace $Y_1, \ldots, Y_m$ by unique skolem constants $sk_1, \ldots, sk_m$ from $L_{sk} \backslash Sk_{d-1}$. Define $Sk_d = Sk_{d-1} \cup \{sk_1, \ldots, sk_m\}$

   (c) Completion

   Define $\gamma_d = TRS\text{-}comp(\gamma_{d-1} + \{s_1 = t_1, \ldots, s_g = t_g\})$. If γ_d is $\{\Box\}$ then define $M_d = \{\Box\}$ and $n = d$.

[1] $(A_d)_i^n$ denotes a sequence $(A_i, \ldots, A_n)$

(d) Normalisation+Assertion
Define $M_d = M_{d-1}.\gamma_d + \{F_1, \ldots, F_h\}.\gamma_d$, obtained by computing the normal form of all facts in these sets.

K is failed if n is finite and $\gamma_n = M_n = \{\Box\}$. This situation occurs when Cl_n is a negative clause, or when $E+\gamma_{n-1}+\{s_1 = t_1, \ldots, s_g = t_g\}$ is inconsistent.
If K is not failed then K is called successful.

Not all NMGE's are interesting. For example, the empty NMGE (({}), ({}), (TRS-comp({})), ()) trivially satisfies the definition of an NMGE, but will not generate a model if P contains one positive extended clause, i.e. an extended clause with empty body. In that case the empty NMGE is an example of an unfair NMGE: there exists a rule with a true body, but which is never applied.

Definition 4.2 *A NMGE K is fair iff K is failed or if the following conditions are satisfied:*

- *K is successful.*
- *If $Cl = G_1, \ldots, G_k \rightarrow E_1, \ldots, E_l$ is a ground instance of a rule of P based on $L+L_{sk}$, and there exists a d such that Cl is based on $L + Sk_d$ and the body of Cl holds in LHM_d then there exists a d' such that $E_1 \vee \ldots \vee E_l$ holds in $LHM_{d'}$.*

Property 4.1 *$(Sk_d)_0^n$ is a monotonically increasing sequence. $(LHM_d)_0^n$ is a monotonically increasing sequence.*

An NMGE performs a fixpoint computation, the result of which can be seen as an interpretation of the language L and, as we later show, a model of <L,P+E>.

Definition 4.3 *The fixpoint of an NMGE K is $\cup_0^n LHM_d$ and is denoted by $K\uparrow$. The skolem set used by K is $\cup_0^n Sk_d$ and is denoted by Sk(K). $K\uparrow$ defines an interpretation of L in the following way:*

- *domain: HU(L+Sk(K))*
- *for each constant c of L: $K\uparrow(c) \equiv c$*
- *for each functor f/n of L: $K\uparrow(f/n)$ is the function which maps terms $t_1, \ldots, t_n$ of $HU(L + Sk(K))$ to $f(t_1, \ldots, t_n)$.*
- *for each predicate of L: $K\uparrow(p/n)$ is the set of $p(t_1, \ldots, t_n)$ facts in $K\uparrow$.*

Corollary 4.1 *If K is a finite successful NMGE K of length n, then $K\uparrow = LHM_n$*

Theorem 4.1 (Soundness) *If K is a fair NMGE, then $K\uparrow$ is a model for <L,P+E> and P+E is consistent (a fortiori).*

We say that $K\uparrow$ is the model generated by K.

To state the completeness result, we require an additional concept: the NMGE-Tree. Analogously with the concept of SLD-Tree, an NMGE-Tree is a tree of NMGE's obtained by applying all different conclusions of one rule in the descendents of a node.

Definition 4.4 *Let L be a language, E an equality theory with completion, P an extended program based on L, and L_{sk} an alphabet of skolem constants.*
An NMGE-Tree (NMGET) T for <L,P+E> is a tree such that:

- *Each node is labeled with a tuple (Sk,M,γ) where Sk is a skolem set, M a set of non-equality facts based on L+Sk, and γ is a ground TRS based on L+Sk.*
- *To each non-leaf N, a ground instance Cl of a rule of P is associated. For each conclusion with index j in the head of Cl, there is an arc leaving from N which is labeled by (Cl,j).*
- *The sequence of labels on the nodes and arcs on each branch of T constitute an NMGE.*

Definition 4.5 *An NMGET is fair if each branch is fair.*

Definition 4.6 *An NMGET is failed if each branch is failed.*

Observe that a failed NMGET contains only a finite number of nodes. Also if T is inconsistent then because of the soundness Theorem 4.1, each fair NMGET is failed.

As a completeness result, we want to state that for any model of P+E, the NMGE contains a branch generating a *smaller* model. In a context of Herbrand models, the smaller-than relation can be expressed by set inclusion. However, because of the existential quantifers and the resulting skolem constants, we cannot restrict to Herbrand models only. In order to define a smaller-than relation for general models, we must have a mechanism to compare models with a different domain. A solution to this problem is provided by the concept of *homomorphism.*

Definition 4.7 *Let I_1, I_2 be interpretations of a language L with domains D_1, D_2.*
A homomorphism from I_1 to I_2 is a mapping h: $D_1 \rightarrow D_2$ which satisfies the following conditions:

- *For each functor f/n $(n \geq 0)$ of L and $x, x_1, \ldots, x_n \in D_1$: $x \equiv I_1(f/n)(x_1, \ldots, x_n) \Rightarrow h(x) \equiv I_2(f/n)(h(x_1), \ldots, h(x_n))$*
- *For each predicate symbol p/n $(n \geq 0)$ of L and $x_1, \ldots, x_n \in D_1$: $I_1(p/n)(x_1, \ldots, x_n) \Rightarrow I_2(p/n)(h(x_1), \ldots, h(x_n))$*

Intuitively a homomorphism is a mapping from one domain to another, such that all positive information in the first model is maintained under the mapping. Therefore the homomorphsims in the class of models of a theory can be used to represent a "...*contains less positive information than...*" relation. We denote the fact that there exists a homomorphism from interpretation I_1 to I_2 by $I_1 \preceq I_2$. This notation captures the intuition that I_1 contains less positive information than I_2.

For NMGET's we can proof the following powerful completeness result.

Theorem 4.2 (Completeness) *Let E be an equality theory with completion, P an extended program, both based on L. Let L_{sk} be an alphabet of skolem constants.*

1. *There exists a fair NMGET for <L,P+E>.*
2. *For each model M of <L,P+E> and each fair NMGET T, there exists a succesful branch K of T such that $K\uparrow \preceq M$.*

We refer to [Denecker and De Schreye, 1991] for a constructive proof of this strong result. As a corollary we obtain the following reformulation of a traditional completeness result.

Corollary 4.2 *If <L,P+E> is consistent then in each fair NMGET there exists a succesful branch.*

If there exists a failed NMGET for <L,P+E>, then <L,P+E> is inconsistent, and all fair NMGET's are failed.

The completeness result does not imply that all models are generated. For example for P = {p←q}, the model {p,q} is not generated by an NMGE. The following example shows that different NMGET's for the same theory might generate different models.

Example $P = \{\ p, q\leftarrow \quad p\leftarrow\ \}$

Depending on which of these clauses is applied first, we get two different nonredundant NMGET's. If $p\leftarrow$ is applied first, then $p, q\leftarrow$ holds already and is not applied anymore. So we get an NMGET with one branch of length 1. On the other hand if $p, q\leftarrow$ was selected first, then two branches exist and we get the solutions {p} and {p,q}.

Therefore it would be interesting if we could characterize a class of models which are generated by each NMGET. The second item of the completeness Theorem 4.2 gives some indication: for any given model M, some succesful branch of the NMGET generates a model with less positive information than M. For the clausal case, models with no redundant positive information are minimal Herbrand Models. From this observation one would expect that for a clausal program, each fair NMGET generates all minimal models. Indeed, the following completeness theorem holds:

Theorem 4.3 (Minimal Herbrand models) *If P is clausal, then for each fair NMGET T, each minimal Herbrand model is generated by a branch in T .*

We have extended the concept of minimal model for general logic theories and proved the completeness of NMGE in the sense that each fair NMGET T generates all minimal models. We refer to [Denecker and De Schreye, 1991].

5 Duality of SLD+Abduction and Model Generation.

The NMGE framework allows to formalise the observations that were made in the introduction. We first introduce the notion of a dualisation more formally.

Definition 5.1 *Let L be a first order language, L_{sk} an alphabet of skolem constants, V_{sk} a dual alphabet of variables such that a bijection $D : L_{sk} \rightarrow V_{sk}$ exists.*

The dualisation mapping D can be extended to a mapping from $HU(L+L_{sk}) \cup HB(L+L_{sk})$ to the set of terms based on $L+V_{sk}$ by induction on the depth of terms:

- *for each constant c of L : $D(c) \equiv c$*
- *for each term $t = f(t_1, \ldots, t_n)$: $D(f(t_1, \ldots, t_n)) \equiv f(D(t_1), \ldots, D(t_n))$*

D can be further extended to any formula or set of formulas. Under dualisation, a ground TRS γ based on $L+L_{sk}$ corresponds to an equation set $D(\gamma)$ with terms based on $L+V_{sk}$. γ is said to be in solved form iff $D(\gamma)$ is an equation set in solved form.

An equation set is in solved form iff it consists of equations $x_i = t_i$, such that the x_i's are distinct variables and do not occur in the right side of any equation. So a TRS is in solved form if the left terms are distinct skolem constants of L_{sk} which do not occur at the right. A TRS in solved form can also be seen as the dual of a variable substitution.

Property 5.1 *Let γ be a TRS in solved form. Then γ is complete wrt to <L,FEQ>.*

Theorem 5.1 (Duality completion - unification) *FEQ(L) is an equality theory with completion. The completion procedure is dual to unification. The dual of the completion of a ground TRS γ based on L+Sk, is the mgu of $D(\gamma)$. Or D(TRS-comp(γ)) = mgu($D(\gamma)$).*

As was observed in the introduction, this duality can be extended further to the complete process of SLD+abduction. On a procedural level, each resolution step corresponds dually to a model generation step. The selection of a goal for resolution corresponds dually to

the selection of the extended rule with its condition instantiated with the dual of the goal. The selection of the clause in the resolution corresponds dually to the selection of the corresponding conclusion in the extended rule. The unification of goal with the head of the clause and the subsequent application of the mgu, corresponds to the completion of the dual equations in the conclusion and the subsequent normalisation.

Now we can formulate the duality theorem for SLD+Abduction ([Cox and Pietrzykowski, 1986]) and Model Generation.

Theorem 5.2 *Let L be a first order language, with an alphabet of variables L_v, L_{sk} an alphabet of skolem constants, and D: $L_{sk} \rightarrow L_v$ a duality bijection between skolem constants and variables. Let P be a definite abductive program based on L.*

For any definite query $\leftarrow Q$, an abductive derivation for $\leftarrow Q$ and P can be dually interpreted as a fair NMGE for only-if(P)+$\exists(Q)$. The set of atoms of the generated model, restricted to the abducible predicates is the dual of the abductive solution. The dual of the answer substitution is the restriction of γ_n to the skolem constants dual to the variables in the query.

The following corollary was proved first by Clark ([Clark, 1978]) for normal programs. For the definite case it follows immediately from the theorem above.

Corollary 5.1 *An SLD-refutation for a query $\leftarrow Q$, and a definite program P without abducibles is a consistency proof of $\exists(Q)$+only-if(P). A failed SLD-tree for a ground query $\leftarrow Q$ and P is an inconsistency proof of $\exists(Q)$+only-if(P), and therefore of $\exists(Q)$+comp(P).*

6 Discussion

A current limitation of the duality framework is its restriction to definite abductive programs. In the future we will extend it to the case of normal abductive procedures. The extended framework will then describe a duality between an SLDNF+Abduction procedure and a form of model generation.

The SLDNF+Abduction procedure can be found by proceeding as for the definite case. There we started from pure SLD and definite programs without abduction, we dualised it and obtained the NMGE method, which under dualisation yields an SLD+Abduction procedure. At present we have performed (on an informal basis) the dualisation of SLDNF for normal programs without abduction. Under dualisation, the resulting model generation procedure gives a natural extension of SLDNF for abductive programs. The abductive procedure incorporates skolemisation for non-ground abducibles goals and efficient treatment of abduced equality atoms by the methods presented earlier. Integrity constraints can be represented by adding for any integrity constraint IC, the rule: "$false \leftarrow not(IC)$.",transforming these rules to a normal program using the transformation of Lloyd-Topor ([Lloyd and Topor, 1984]), and adding the literal *not false* to the query.

A prototype of this method has been implemented. An interesting experiment was its extension to an abductive planner based on the event calculus. Our prototype planner was able to solve some hard problems with context dependent events, problems that are not properly solved by existing systems ([Shanahan, 1989], [Missiaen, 1991]).

In [Denecker and De Schreye, 1992], we proved the soundness of the procedure with respect to Completion semantics, in the sense that for any query $\leftarrow Q$ and generated solution Δ:

$$P + \Delta \models Q$$

This implies the soundness of the procedure with respect to the Generalised Stable Model semantics of [Kakas and Mancarella, 1990b]: a generated solution can be extended in a natural way to a generalised stable model of the abductive program. As a completeness result we proved that the procedure generates all *minimal* solutions when the computation tree is finite.

Related to our work, [Bry, 1990] also indicates a relationship between abduction and model generation. However, while we propose a relationship on the object level, there it is argued that abductive solutions can be generated by model generation on the abductive program augmented with a fixed metatheory.

In [Console *et al.*, 1991], another approach is taken for abduction through deduction. An abductive procedure is presented which for a given normal abductive program P and query $\leftarrow Q$, derives an *explanation formula* E equivalent with Q under the completion of P:

$$comp(P) \models (Q \Leftrightarrow E)$$

The explanation formula is built of abducibles predicates and equality only. It characterises all abductive solutions in the sense that for any set Δ of abducible atoms, Δ is an abductive solution iff it satisfies E.

Although this approach departs also from the concept of completion, it is of a totally different nature. In the first place, our approach aims at contributing to the procedural semantics of abduction. This is not the case with the work in [Console *et al.*, 1991]. Another difference is that this approach is restricted to queries with a finite computation tree. If the computation tree contains an infinite branch, then the explanation formula cannot be computed.

In [Kakas and Mancarella, 1990a], an abductive procedure for normal abductive programs has been defined. A restriction of this method is that abducible goals can only be selected when they are ground. As argued in section 1, this poses a serious problem for applications such

as planning. The methods presented here allow to overcome the problem by skolemisation of nonground goals and efficient treatment of abduced equality facts.

Recently, an planning system based on abduction in the event calculus has been proposed in [Missiaen, 1991]. The underlying abductive system incorporates negation as failure, skolemisation for non-ground abducible goals and efficient treatment of abduced equality facts. However, the system shows some problems with respect to soundness and completeness. Experiments indicated that these problems are solved by our prototype planner.

Finally, we want to draw attention to an unexpected application of the duality framework. In current work on abduction, the theory of Free Equality is implicitly or explicitly present. What happens if FEQ is replaced by general equality EQ and the equality predicate is abducible? The result is an uncommon form of abduction illustrated below. Take the program P = $\{r(a)\leftarrow\}$. For this program, the query $\leftarrow r(b)$ has a successful abductive derivation.

$\leftarrow \underline{r(b)}$ $\quad \Delta = \{\}$

$\square$ $\quad \Delta = \{b = a\}$

$\leftarrow r(b)$ succeeds under the abductive hypothesis {b= a}. The duality framework provides the technical support for efficiently implementing this form of abduction. The only difference with normal abduction is that the completion procedure for FEQ -the dual of unification- must be replaced by a completion procedure for EQ, for example Knuth-Bendix completion.

To conclude, we have presented a duality between two computation paradigms. This duality allows to transfer technical results from one paradigm to the other and vice versa. One application that was obtained was an efficient extension of model generation with equality. Transferring these methods back to abduction, we obtained techniques for dealing with non-ground abducible goals and efficient treatment of abduced equality atoms. We discussed experiments indicating that the extension of the duality framework for the case of normal programs is extremely useful for obtaining an abductive procedure for normal abductive programs.

7 Acknowledgements

We thank Krzysztof Apt, Eddy Bevers, Maurice Bruynooghe and Francois Bry for helpful suggestions.

References

[Bry, 1990] F. Bry. Intensional updates: Abduction via deduction. In *proc. of the intern. conf. on Logic Programming 90*, pages 561–575, 1990.

[Clark, 1978] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and databases*, pages 293–322. Plenum Press, 1978.

[Console *et al.*, 1991] L. Console, D. Theseider Dupre, and P. Torasso. On the relationship between abduction and deduction. *journal of Logic and Computation*, 1(5):661–690, 1991.

[Cox and Pietrzykowski, 1986] P.T. Cox and T. Pietrzykowski. Causes for events: their computation and application. In *proc. of the 8th intern. conf. on Automated Deduction*, 1986.

[Denecker and De Schreye, 1991] Marc Denecker and Danny De Schreye. A framework for indeterministic model generation with equality. Technical Report 124, Department of Computer Science, K.U.Leuven, March 1991.

[Denecker and De Schreye, 1992] Marc Denecker and Danny De Schreye. A family of abductive procedures for normal abductive programs, their soundness and completeness. Technical Report 136, Department of Computer Science, K.U.Leuven, 1992.

[Dershowitz and Jouannaud, 1989] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, vol.B*, chapter 15. North-Holland, 1989.

[Eshghi, 1988] K. Eshghi. Abductive planning with event calculus. In R.A. Kowalski and K.A. Bowen, editors, *proc.of the 5th ICLP*, 1988.

[Huet, 1980] G. Huet. confluent reductions: Abstract properties and applications to term rewriting systems. *journal of the Association for Computing Machinery*, 27(4):797–821, 1980.

[Kakas and Mancarella, 1990a] A.C. Kakas and P. Mancarella. Database updates through abduction. In *proc. of the 16th Very large Database Conference*, pages 650–661, 1990.

[Kakas and Mancarella, 1990b] A.C. Kakas and P. Mancarella. Generalised stable models: a semantics for abduction. In *proc. of ECAI-90*, 1990.

[Kowalski, 1991] R.A. Kowalski. Logic programming in artificial intelligence. In *proceedings of the IJCAI*, 1991.

[Lloyd and Topor, 1984] J.W. Lloyd and R.W. Topor. Making prolog more expressive. *Journal of logic programming*, 1(3):225–240, 1984.

[Manthey and Bry, 1987] R. Manthey and F. Bry. A hyperresolution-based proof procedure and its implementation in prolog. In *proc. of the 11th German workshop on Artificial Intelligence*, pages 221–230. Geseke, 1987.

[Missiaen, 1991] L. Missiaen. *Localized abductive planning with the event calculus*. PhD thesis, Department of Computer Science, K.U.Leuven, 1991.

[Shanahan, 1989] M. Shanahan. Prediction is deduction but explanation is abduction. In *IJCAI89*, page 1055, 1989.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Defining Concurrent Processes Constructively *

Yukihide Takayama
Kansai Laboratory, OKI Electric Industry Co., Ltd.
Crystal Tower, 1-2-27 Shiromi, Chuo-ku, Osaka 540, Japan
takayama@kansai.oki.co.jp, takayama@icot.or.jp

Abstract

This paper proposes a constructive logic in which a concurrent system can be defined as a proof of a specification. The logic is defined by adding stream types and several rules for them to an ordinary constructive logic. The unique feature of the obtained system is in the $(MPST)$ rule which is a kind of structural induction on streams. The $(MPST)$ rule is based on the idea of largest fixed point inductions, but the formulation of the rule is quite different and it allows to define a concurrent process as a Burge's mapstream function with a good intuition on computation. This formulation is possible when streams are viewed as sequences not infinite lists. Also, our logic has explicit nondeterminacy but we do not introduce any extralogical device. Our nondeterminacy rule, $(NonDet)$, is actually a defined rule which uses inherent nondeterminacy in the traditional intuitionistic logic. Several techniques of defining stream based concurrent programs are also presented through various examples.

1 Introduction

Constructive logics give a method for formal development of programs, e.g., [C+86, HN89]. Suppose, for example, the following formula: $\forall x : D_1.\exists y : D_2.\ A(x,y)$. This is regarded as a specification of a function, f, whose domain is D_1 and the codomain is D_2 satisfying the input-output relation, $A(x,y)$, that is, $\forall x : D_1.\ A(x, f(x))$ holds. This functional interpretation of formulas is realized mechanically. Namely, if a constructive proof of the formula is given, the function, f, is extracted from the proof with q-realizability interpretation [TvD88] or with Curry-Howard correspondence of types and formulas [How80]. This programming methodology will be referred to as *constructive programming* [SK90] in the following.

Although constructive programming has been studied by many researchers, the constructive systems which can handle concurrency are rather few. This is mainly because most of the constructive logics have been formalized as intuitionistic logics, and the intuitionism itself does not have explicit concurrency besides proof normalization corresponding to the execution of programs [Got85]. For example, QJ [Sat87] is an intuitionistic programming logic for a concurrent language, Quty. However, when we view QJ as a constructive programming system, concurrency only appears in the operational semantics of Quty.

Linear Logic [Gir87] gives a new formulation of constructive logic which is not based on intuitionism. This is the first constructive logic which can handle concurrency at the level of logic. The logic was obtained by refining logical connectives of traditional intuitionistic or classical logic to introduce drastically new connectives with the meaning of parallel execution. In Linear Logic, formulas are regarded as processes or resources and every rule of inference defines the behavior of a concurrent operation. Linear Logic resembles Milner's SCCS [Mil89] in this respect.

We take intermediate approach between QJ and Linear Logic in the sense of not throwing away but extending intuitionistic logic. The advantage of this approach is that the functional interpretation of logical connectives in the traditional constructive programming based on intuitionism is preserved, and that both the sequential and concurrent parts of programs are naturally described as constructive proofs. To this end, we take the stream based concurrent programming model [KM74]. We introduce stream types and quantification over stream types. A formula is regarded as a specification of a process when it is a universally or an existentially quantified over stream types, and otherwise it represents a specification of a sequential function, properties of processes or linkage relation between processes. A typical process, $\forall X.\exists Y.A(X,Y)$ where X and Y are stream variables, is regarded as a stream transformer. Most of the rules of inference are those of ordinary constructive programming systems, but rules for nondeterminacy and for stream types are also introduced. Among them, a kind of structural induction on stream types called $(MPST)$ is the heart of our extended system: With $(MPST)$, stream transformers can be defined as Burge's mapstream functions [Bur75].

*This work was supported by ICOT as a joint research project on theorem proving and its application.

T. Hagino [Hag87] gave a clear categorical formalization of stream types (infinite list types or lazy types) whose canonical elements are given by a schema of map-stream functions, but relation between his formulation and logic is not investigated. N. Mendler and others [PLS86] introduced lazy types and the type checking rules for them into an intuitionistic type theory preserving the propositions-as-types principle in the sense that an empty type can exist even in the extended type theory. However, they do not give sufficient rules of inference for proving specification of stream handling programs. Reasoning about stream transformer can be handled with a largest fixed point induction as was demonstrated by P. Dybjer and H. P. Sander [DSS89]. However, their system is designed as a program verification system not as a constructive programming system. Although q-realizability interpretation for program extraction can be defined for the coinduction rule [KT91], the rule seems rather difficult to use for proving specifications. The reason is that the coinduction rule deeply depends on the notion of bisimulation, so that in the proof procedure one must find a stronger logical relation included in the more general logical relation and that is not always an easy task.

The $(MPST)$ rule is based on a similar idea to the coinduction rule: one must find a new logical relation and a new function to prove the conclusion. However, what one must find has a clear intuitive meaning as the components of a concurrent process. Therefore, the $(MPST)$ rule shows an intuitive guideline on how to construct a concurrent process.

Section 2 explains how a concurrent system is specified in logic. A process is specified by the $\forall X.\exists Y.A(X,Y)$ type formula as in the traditional constructive programming. The rest of the sections focus on the problem of defining processes which meet the specifications. Section 3 formulates streams and stream types. Streams are viewed as infinite lists or programs which generate infinite lists at the level of underlying programming language. At the logical reasoning level, streams are sequences, namely, total functions on natural numbers. This two level formulation of streams enables to introduce $(MPST)$ which will be given in section 4. Section 5 presents the rest of the formalism of the whole system. The realizability interpretation which gives the program extraction algorithm from proofs will be defined. Several examples will be given in section 6 to demonstrate how stream based concurrent programming is performed in our system.

Notational preliminary: We assume first order intuitionistic natural deduction. Equalities of terms, typing relations $(M : \sigma)$, and $\top$ (true) are atomic formulas. The domain of the quantification is often omitted when it is clear from the context. Sequences of variables are denoted as $\overline{x}$ or $\overline{X}$. $M_x[N]$ denotes substitution of N to the variable, x, occurring freely in M. $M_{\overline{x}}[\overline{N}]$ denotes simultaneous substitution. $FV(M)$ is the set of free variables in M. $(::)$ denotes the (infinite) list constructor. Function application is denoted $ap(M,N)$ or $M(N)$. $M^n(N)$ denotes $\underbrace{M(\cdots M(}_{n} N)\cdots)$.

2 Specifying Concurrent Systems in Logic

The model of concurrent computation in this paper is as follows: A concurrent system consists of processes linked with streams. A process interacts with other processes only through input and output streams. The configuration of processes in a concurrent system is basically static and finite, but in some cases, which will be explained later, infinitely many new processes may be created by already existing processes. A process is regarded as a transformer (stream transformer) of input streams to an output stream, and it is specified by the $\forall\overline{X} : I_{\sigma_1,\cdots,\sigma_n}.\ \exists Y : I_\tau.\ A(\overline{X},Y)$ type of formula where I_σ denotes the type of streams over the type σ, but its definition will be given later. $I_{\sigma_1,\cdots,\sigma_n}$ is an abbreviation of $I_{\sigma_1}\times\cdots\times I_{\sigma_n}$, $\overline{X}$ and Y are input and output streams, and $A(\overline{X},Y)$ is the relation definition of input and output streams.

The combination of two processes, $\forall X.\exists Y.\ A(X,Y)$ and $\forall P.\exists Q.\ B(P,Q)$, by linking the stream Y and P is described by the following proof procedure:

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\Sigma_1\\ \forall X.\exists Y.\ A(X,Y)\end{matrix}}{\exists Y.\ A(X,Y)}(\forall E) \quad \Pi_0}{\exists Y.\exists\alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists E)^{(1)}}{\forall X.\exists Y.\exists\alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\forall I)$$

where $\Pi_0 \stackrel{\text{def}}{=}$

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\Sigma_2\\ \forall P.\exists Q.\ B(P,Q)\end{matrix}}{\exists Q.\ B(Y',Q)}(\forall E) \quad \Pi_1}{\exists Y.\exists\alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists E)^{(2)}$$

and $\Pi_1 \stackrel{\text{def}}{=}$

$$\dfrac{\dfrac{\dfrac{[A(X,Y')]^{(1)}\ [B(y,Q')]^{(2)}}{A(X,Y')\ \&\ B(Y',Q')}(\&\ I)}{\exists\alpha.\ A(X,\alpha)\ \&\ B(\alpha,Q')}(\exists I)}{\exists Y.\exists\alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists I)$$

and Σ_1 and Σ_2 are the definition of process $\forall X.\exists Y.A(X,Y)$ and $\forall P.\exists Q.B(P,Q)$.

This is a typical proof style to define a composition of two functions. Thus, a concurrent system is also specified by $\forall X.\exists Y.\ A(X,Y)$ type formula. X and Y are input and output streams of the whole concurrent system, and α is an internal stream.

All these things just realize the idea that functions can be viewed as a special case of processes. In the following, we focus on the problem of how to define a process (stream transformer) as a constructive proof.

3 Formulation of Streams

We give in this section the definition of the stream types C_σ and I_σ, and consider the semantics of quantification over I_σ.

3.1 Two Level Stream Types

A stream can be viewed at least in three ways: an infinite list, an infinite process, and an output sequence of an infinite process, namely, a total function on natural numbers. The formal theories of lazy functional programming such as [PL86] and [Hag87] can be regarded as the theories of concurrent functional programming based on the first two points of view on streams. Our system uses a lazy typed lambda calculus as the underlying programming language and has lazy types as *computational stream types*. Computational stream types are only used as the type system for the underlying language. In proving specifications of stream transformers, we use *logical stream types* which are based on the third point of view on streams. In other words, we have two kinds of streams: computational streams at the programming language level, and logical streams at the logical reasoning level. We denote a computational stream type C_σ and a logical stream type I_σ. The following is the basic rules for computational stream types. The idea behind them is similar to that behind the lazy type rules in [PL86]. We confuse the meaning of the infinite list constructor, (::), and will use this also as an infinite cartesian product constructor. We abbreviate $M \stackrel{\sigma}{=} N$ for $M = N$ *in* σ in the following.

$$C_\sigma = \sigma \times C_\sigma \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash S : C_\sigma}{\Gamma \vdash (M :: S) : C_\sigma}$$

$$\frac{\Gamma \vdash M \stackrel{\sigma}{=} N \quad \Gamma \vdash S \stackrel{C_\sigma}{=} T}{\Gamma \vdash (M :: S) \stackrel{C_\sigma}{=} (N :: T)} \qquad \frac{\Gamma \vdash (M :: S) \stackrel{C_\sigma}{=} (N :: T)}{\Gamma \vdash M \stackrel{\sigma}{=} N}$$

$$\frac{\Gamma \vdash (M :: S) \stackrel{C_\sigma}{=} (N :: T)}{\Gamma \vdash S \stackrel{C_\sigma}{=} T} \qquad \frac{\Gamma, z : T \vdash M : T}{\Gamma \vdash \nu z.\ M : T}$$

where T is C_σ or $\tau \to C_\sigma$.

ν is the fixed point operator only used for describing a stream as an infinite process (infinite loop program). The reduction rule for ν-terms is defined as expected. hd and tl are the primitive destructor functions on streams.

$$\frac{\Gamma \vdash M : C_\sigma}{\Gamma \vdash hd(M) : \sigma} \qquad \frac{\Gamma \vdash M : C_\sigma \qquad \Gamma \vdash n : nat}{\Gamma \vdash tl^n(M) : C_\sigma}$$

$$\frac{\Gamma \vdash X : C_\sigma}{\Gamma \vdash X \stackrel{C_\sigma}{=} (hd(X) :: tl(X))}$$

$$\frac{\Gamma \vdash (M :: S) : C_\sigma}{\Gamma \vdash hd((M :: S)) \stackrel{\sigma}{=} M} \qquad \frac{\Gamma \vdash (M :: S) : C_\sigma}{\Gamma \vdash tl((M :: S)) \stackrel{C_\sigma}{=} S}$$

$$\frac{\Gamma, n : nat, tl^n(S) \stackrel{C_\sigma}{=} tl^n(T) \vdash S \stackrel{\sigma^n \times C_\sigma}{=} T}{\Gamma \vdash S \stackrel{C_\sigma}{=} T}$$

$$\frac{\Gamma, n : nat \vdash hd(tl^n(S)) \stackrel{\sigma}{=} hd(tl^n(T))}{\Gamma \vdash S \stackrel{C_\sigma}{=} T}$$

$$\frac{\Gamma \vdash M_z \stackrel{C_\sigma}{=} N_w[z]}{\Gamma \vdash \nu z.\ M_z \stackrel{C_\sigma}{=} \nu w.\ N_w}$$

Before giving the definition of logical stream types, note that the type, $nat \to \sigma$, is isomorphic to C_σ, namely,

Proposition 1: *Let σ be any type, then Let $\varphi : (nat \to \sigma) \to C_\sigma$ be $\varphi(M) \equiv ap(\nu z.\lambda n.\ (M(n) :: z(n+1)), 0)$ for arbitrary $M : nat \to \sigma$, and let $\psi(N) \equiv \lambda n.\ hd(tl^n(N))$ for arbitrary $N : C_\sigma$. Then,*
(1) For arbitrary $M : nat \to \sigma$, $\varphi(M) : C_\sigma$ and $\psi(\varphi(M)) = M$ in $nat \to \sigma$;
(2) For arbitrary $N : C_\sigma$, $\psi(N) : nat \to \sigma$ and $\varphi(\psi(N)) = N$ in C_σ.

A logical stream type, I_σ, has the same elements as $nat \to \sigma$, but the elements are viewed differently, namely, viewed as streams:

$$\frac{\Gamma \vdash M : nat \to \sigma}{\Gamma \vdash M : I_\sigma} \qquad \frac{\Gamma \vdash M : I_\sigma}{\Gamma \vdash M : nat \to \sigma}$$

This means that any (total) function on the natural number type nat definable in the underlying programming language is regarded as a stream. A similar idea is formulated with regard to formulas:

$$\frac{\Gamma \vdash \forall n : nat. \exists x : \sigma.\ A(n, x)}{\Gamma \vdash \exists Y : I_\sigma . \forall n : nat.\ A(n, Y(n))}(ST)$$

The equality between streams is extensional. That is

$$\frac{\Gamma \vdash X : I_\sigma \quad Y : I_\sigma \quad \forall n : nat.\ X(n) = Y(n)}{\Gamma \vdash X \stackrel{I_\sigma}{=} Y}$$

The following rule, (CON), characterizes a kind of continuity of stream transformers and is used for justifying $(MPST)$ rule given later.

$$\frac{(a)\ \Gamma \vdash F : I_{\sigma_1,\cdots,\sigma_k} \to I_{\sigma_1,\cdots,\sigma_k} \quad (b)\ \Gamma \vdash \forall \overline{X} : I_{\sigma_1,\cdots,\sigma_k} . \forall n : nat.\ A(n, F(\overline{X})) \Rightarrow A(n+1, \overline{X})}{\Gamma \vdash \forall \overline{X} : I_{\sigma_1,\cdots,\sigma_k} . \forall n : nat.\ A(0, F^n(\overline{X})) \Rightarrow A(n, \overline{X})}$$

A logical stream also has, hd, tl and (::), which simulate those accompanied with C_σ:

$$hd(X) \stackrel{\text{def}}{=} X(0) \text{ for } X : I_\sigma$$

$$tl^n(X) \stackrel{\text{def}}{=} \lambda m.X(m+n) \text{ for } X : I_\sigma$$

$$(M :: S)(0) \stackrel{\text{def}}{=} M$$

$$(M :: S)(n) \stackrel{\text{def}}{=} S(n-1) \text{ for } n > 0$$

Note that $X(n) = hd(tl^n(X))$ for arbitrary $X : I_\sigma$ and $n : nat$. All the rules for hd, tl and (::) in computational streams also hold for these defined functions and the constructor for logical streams.

3.2 Quantification over Logical Stream Types

There is a difficulty in defining the meaning of quantification over (logical) stream types. The standard intuition-

istic interpretation of, say, existential quantification over a type, σ, $\exists x : \sigma.A(x)$ is that "we can explicitly give the object, a, of type σ such that $A(a)$ holds". However, as a stream is a partial object we can only give an approximation of the complete object at any moment. Therefore we need to extend the familiar interpretation of quantification over types. In fact, Brouwer's theory of choice sequences [TvD88] in intuitionism provides us with the meaning of quantification over infinite sequences.

There are two principles in Brouwer's theory, the *principle of open data* and the *principle of function continuity*. The principle of open data, which informally states that for independent sequences any property which can be asserted must depend on initial segments of those sequences only, gives the meaning of the quantification of type $\forall X.\exists y.A(X,y)$. That is, for an arbitrary sequence, X, there is a suitable initial finite segment, X_0, of X such that $\exists y.\ A(X_0,y)$ holds. The principle of function continuity gives the meaning of the quantification of type $\forall X.\exists Y.A(X,Y)$. Assume the case of natural number streams (total functions between natural number types). The function continuity is stated as follows:

$$\forall X.\exists Y.\ A(X,Y) \Rightarrow \exists f : K.\ \forall X.\ A(X, f|X)$$

where $f|X = Y$ is an abbreviation of $\forall x : nat.\ f(x :: X) = Y(x)$ and K is the class of functions that take initial finite segment of the input sequences and return the values. This means that every element of Y is determined with a suitable initial finite segment of X.

These principles meet our intuition of functions on streams and stream transformers very well. $\forall X : I_\sigma.\exists y : \tau.A(X,y)$ represents a function on streams over σ, but we would hardly ever try to define a function which returns a value after taking *all* the elements of an input stream. Also, we would expect a stream transformer, $\forall X : I_\sigma.\exists Y : I_\tau.A(X,Y)$, calculate the elements of the output stream, Y, gradually by taking finitely many elements of the input stream, X, at any step of the calculation.

Note that this semantics also meets the proof method used in [KM74]: To prove a property $P(X)$ on a stream X, we first prove P for an initial finite subsequence, X_0, of X ($\vdash P(X_0)$) and define $\vdash P(X)$ to be $\lim_{X_0 \to X} P(X_0)$.

4 Structural Induction on Logical Streams

As streams can be regarded as infinite lists, we would expect to extend the familiar structural induction on finite lists to streams. However, a naive extension of the structural induction on finite lists does not work well. If we allow the rule below,

$$\frac{\Gamma, A(tl(X)) \vdash] A(X)}{\Gamma \vdash \forall X : I_\sigma.\ A(X)}(SI)$$

the following wrong theorem can be proved:

WrongTheorem: $\forall X : I_{nat}.\ B(X)$
where $B(X) \stackrel{\text{def}}{=} \exists n : nat.\ X(n) = 100$.
Proof: By (SI) *on* $X : I_{nat}$. *Assume* $B(tl(X))$. *Then, there is a natural number* k *such that* $tl(X)(k) = X(k+1) = 100$. *Then* $B(X)$.∎

This proof would correspond to the following uninteresting program: $foo = \lambda X.\ foo(tl(X))$. This is because the naive extension of the structural rule on finite lists does not maintain the continuity of the function on streams. Therefore, we need a drastically different idea in the case of infinite lists. One candidate is the coinduction rule (a largest fixed point induction) as in [DS89]: $(B \Rightarrow \Phi_P[B]) \Rightarrow (B \Rightarrow \nu P.\Phi)$ where $\nu P.\Phi$ denotes the largest fixed point of $P = \Phi$. $\forall X : I_\sigma.A(X)$ part will be described with $\nu P.\Phi$ type formulas, and one must find a suitable logical relation B to prove the conclusion. But searching B will not always be an easy task: we wish the searching task decomposed into more than one smaller tasks each of which has clear and intuitive meaning of computation. Therefore, we take another approach: the $(MPST)$ rule.

4.1 Mapstream Functions as Stream Transformers

Recall that the motivation of pursuing a kind of structural induction on streams is to define stream transformers as proofs, and stream transformers can be realized as Burge's mapstream functions. A schema of mapstream functions is described in typed lambda calculus as follows:

$$P \equiv \lambda M^{\tau\to\sigma}.\lambda N^{\tau\to\tau}.\lambda x^{\tau}.\ ((M\ x) :: (((P\ M)\ N)\ (N\ x)))$$

If we give the procedures M and N, we obtain a mapstream function. Note that, from the viewpoint of continuity, these procedures should be as follows:

$M \equiv$ "Fetch initial segment, X_0, of the input stream, X, to generate the first element of the output stream. "

$N \equiv$ "Prepare for fetching the next finite segment input stream interleaving, if necessary, other stream transformer between the original input stream and the input port.

This suggests that if a way to define M, N, and P as proof procedures is given, one can define stream transformers as constructive proofs.

4.2 A Problem of Empty Stream

Before giving the rule of inference for defining stream transformers, a little more observation of stream based programming is needed. Assume a filter program on natural number streams realized as a mapstream function:

$$flt_a \equiv \lambda X.\ if\ (a|hd(X))\ then\ flt_a(tl(X))\ else\ (hd(X) :: flt_a(tl(X)))$$
$$\equiv \lambda X.((M\ X) :: (((P\ M)\ N)(NX)))$$

where $(a|hd(X))$ is true when $hd(X)$ can be divided by a (a natural number) and

$$M \equiv \lambda X.\, if\ (a|hd(X))\ then\ M(tl(X))\ else\ hd(X)$$
$$N \equiv \lambda X.\, if\ (a|hd(X))\ then\ N(tl(X))\ else\ tl(X)$$

For example, $flt_5((5 :: 5 :: 5 :: 5 :: \ldots))$ is an empty sequence because the evaluation of $M(5 :: 5 :: 5 :: 5 :: \ldots)$ does not terminate. This contradicts the principle of open data explained in 3.2. To handle such a case, we introduce the notion of complete stream. The idea is to regard flt_5, for example, always generating some elements even if the input stream is $(5 :: 5 :: \ldots)$.

Def. 1: Complete types
Let σ be any type other than a stream type, then $\sigma_\perp$ denotes a type σ together with the bottom element $\perp_\sigma$ (often denoted just $\perp$) and it is called a *complete type*.

Def. 2: Complete stream types
A stream type, I_σ or C_σ, is called complete when σ is a complete type.

flt_5 is easily modified to a function from C_{nat} to $C_{nat_\perp}$, and then $flt_5((5 :: 5 :: \ldots))$ will be $(\perp :: \perp :: \ldots)$ which is practically an empty stream.

4.3 The $(MPST)$ rule

Based on the observations in the previous sections, we introduce a rule $(MPST)$ for defining stream transformers. The rule is formulated in natural deduction style, but the formula, A, in the specification of a stream transformer, $\forall X.\exists Y.\ A(X,Y)$, is restricted. In spite of the restriction, the rule can handle a fairly large class of specifications of stream transformers as will be demonstrated later. The rule is as follows:

$$\frac{\begin{array}{l}(a)\ \forall X : I_\sigma.\exists a : \tau.\ M(X,a) \\ (b)\ \forall X : I_\sigma.\forall a : \tau.\ \forall S : I_\tau.\ (M(X,a) \Rightarrow A(0,X,(a :: S))) \\ (c)\ \exists f : I_\sigma \to I_\sigma.\ \forall X : I_\sigma.\forall Y : I_\tau.\forall n : nat. \\ \qquad\qquad (A(n,f(X),tl(Y)) \Rightarrow A(n+1,X,Y))\end{array}}{\forall X : I_\sigma.\exists Y : I_\tau.\forall n : nat.\ A(n,X,Y)}$$

where M is a suitable predicate and $A(n,X,Y)$ must be a rank 0 formula [HN89]. We can easily extend the rule to the multiple input stream version. We do not give the precise definition of rank 0 formulas here, but the intention is that we should not expect to extract any computational meaning from $A(n,X,Y)$ part. This restriction comes from purely technical reason, but does not degenerate the expressive power of the rule from the practical point of view because we usually need only to define a stream transformer program but not the verification code corresponding to $A(n,X,Y)$ part. The technical reason for the side condition of $(MPST)$ is as follows: $(MPST)$ is in fact a derived rule with (ST) and (CON), so that q-realizability interpretation defined in the next section is carried out using the interpretation of those rules. The difficulty resides in the interpretation of the (CON) rule, but if we restrict the formula $A(n,\overline{X})$ in (CON) to be rank 0, the interpretation is trivial. This condition corresponds to to side condition of $(MPST)$.

The intuitive meaning of $(MPST)$ is as follows. As explained in 4.1, a mapstream function is defined when M and N procedure are given. (a) is the specification of the M procedure, f_M, and (b) means that f_M certainly generates the right elements of the output stream. The N procedure, f_N, is defined as the value of existentially quantified variable, f, in (c). (c) together with (b) intuitively means the following: for $X : I_\sigma$ (input stream) and $Y : I_\tau$ (output stream), let us call a pair, $(f_N^n(X), tl^n(Y))$, the *nth f_N-descendant of* (X,Y). Then, for arbitrary $n : nat$, $A(n,X,Y)$ speaks about nth f_N-descendant of (X,Y), and $A(n,f_N(X),tl(Y))$ actually speaks about $n+1$th f_N-descendant of (X,Y).
If f_N is a stream transformer, this means that the process (stream transformer) defined by $(MPST)$ generates another processes dynamically.

Note that, as we must give a suitable formula, M, to prove the conclusion, $(MPST)$ is essentially a second order rule.

5 The Formal System

This section presents the rest of the formalization of our system briefly.

5.1 Non-deterministic λ-calculus

The non-deterministic λ-calculus is a typed concurrent calculus based on parallel reduction and this is used as the underlying programming language. The core part is almost the same as that given in [Tak91]. It has natural numbers, booleans (T and F), L and R as constants. Individual variables, lambda-abstractions, application, sequences of terms ($(M_1,...,M_n)$ where M_i are terms), *if-then-else*, and a fixed point operator (μ) are used as terms and program constructs. The reduction rules for terms are defined as expected, and if a term, M, is reducible to a term, N, then M and N are regarded as equal. Also, several primitive functions are provided for arithmetic operations and for the handling of sequences of terms such as projection of elements or subsequences from a sequence of terms. The type structure of the calculus is almost that of simply typed λ-calculi. *nat* (natural number type), *bool* (boolean types), and 2 (type of L and R) are primitive types and $\times$ (cartesian product) and $\to$ (arrow) are used as type constructors. The type inference rules for this fragment of the calculus are defined as expected. In addition to them, computational streams, computational stream types and a special term called *coin flipper* is introduced to describe concurrent computation of streams. For the reduction strategy, ν-

terms in section 3.1 are lazily evaluated.

The coin flipper is a device for simulating nondeterminacy. It is a term, $\bullet$, whose computational meaning is given by the following reduction rule:

$$\bullet \triangleright L \text{ or } R$$

That is, $\bullet$ reduces to L or R in a nondeterministic way. This is like flipping a coin, or can be regarded as hiding some particular decision procedure whose execution may not always be explained by the reduction mechanism.

$\bullet$ is regarded as an element of 2^+, a super type of 2. The elements of 2 have been used to describe the decision procedure of *if-then-else* programs in the program extraction from constructive proofs in [Tak91] as *if* $T = L$ *then* M *else* N. Nondeterminacy arises when T is replaced by $\bullet$. The intentional semantics of $\bullet$ is *undefined*. 2^+ enjoys the following typing rules:

$$L : 2^+ \qquad R : 2^+ \qquad \bullet : 2^+$$

5.2 Rules of Inference

(1) Logical Rules

The rules for logical connectives and quantifiers are those of first order intuitionistic natural deduction with mathematical induction.

(2) Rules for Nondeterminacy

$$\bullet = L \vee \bullet = R \qquad \frac{A \quad A}{A}(NonDet)$$

$(NonDet)$ is actually a derived rule: This is obtained by proving A by divide and conquer on $\top \vee \top$. $(NonDet)$ means that if two distinct proof of A are given, one of them will be chosen in a nondeterministic way. This is the well-known nondeterminacy both in classical and intuitionistic natural deduction.

(3) Auxiliary Rules

$$\frac{M : \sigma \to \sigma \quad a : \sigma \quad n : nat}{ap(M^n, a) : \sigma} \qquad \frac{f : \sigma_1 \to \tau_1 \quad g : \sigma_2 \to \tau_2}{f \times g : \sigma_1 \times \sigma_2 \to \tau_1 \times \tau_2}$$

5.3 Realizability Interpretation

The realizability defined in this section is a variant of q-realizability [TvD88].

A new class of formulas called realizability relations is introduced to define q-realizability.

Def. 3: Realizability relation

A *realizability relation* is an expression in the form of $\overline{a}$ q A, where A is a formula and $\overline{a}$ is a finite sequence of variables which does not occur in A. $\overline{a}$ is called a *realizing variables* of A. For a term M, M q A, which reads "a term M realizes a formula A", denotes $(\overline{a}$ q $A)_{\overline{a}}[M]$, and M is called a *realizer* of A.

A type is assigned for each formula, which is actually the type of the realizer of the formula.

Def. 4: $type(A)$

Let A be a formula. Then, a type of A, $type(A)$, is defined as follows:

1. $type(A)$ is empty, if A is rank 0;
2. $type(A \,\&\, B) \stackrel{\text{def}}{=} type(A) \times type(B)$;
3. $type(A \vee B) \stackrel{\text{def}}{=} 2^+ \times type(A) \times type(B)$;
4. $type(A \Rightarrow B) \stackrel{\text{def}}{=} type(A) \to type(B)$;
5. $type(\forall x : \sigma.\ A) \stackrel{\text{def}}{=} \sigma \to type(A)$;
6. $type(\exists x : \sigma.\ A) \stackrel{\text{def}}{=} \sigma \times type(A)$;

Proposition 2: *Let A be a formula with a free variable x. Then, $type(A) = type(A_x[M])$ for any term M of the same type as x.*

Def. 5: q-realizability

1. If A is a rank 0 formula, then $()$ q $A \stackrel{\text{def}}{=} A$;
2. $\overline{a}$ q $A \Rightarrow B \stackrel{\text{def}}{=} \forall b : type(A).(A \,\&\, b$ q $A \Rightarrow \overline{a}(b)$ q $B)$;
3. $(a, \overline{b})$ q $\exists x : \sigma.\ A \stackrel{\text{def}}{=} a : \sigma \,\&\, A_x[a] \,\&\, \overline{b}$ q $A_x[a]$;
4. $\overline{a}$ q $\forall x : \sigma.\ A \stackrel{\text{def}}{=} \forall x : \sigma.\ (\overline{a}(x)$ q $A)$;
5. $(z, \overline{a}, \overline{b})$ q $A \vee B$
   $\stackrel{\text{def}}{=} (z = L \,\&\, A \,\&\, \overline{a}$ q $A \,\&\, \overline{b} : type(B))$
   $\vee (z = R \,\&\, B \,\&\, \overline{b}$ q $B \,\&\, \overline{a} : type(A))$ provided that A and B are distinct or $A = B$ with A and B not rank 0;
6. $\bullet$ q $A \vee A \stackrel{\text{def}}{=} A$ if A is rank 0;
7. $(\overline{a}, \overline{b})$ q $A \,\&\, B \stackrel{\text{def}}{=} \overline{a}$ q $A \,\&\, \overline{b}$ q B.

Proposition 3: *Let A be any formula. If $\overline{a}$ q A, then $\overline{a} : type(A)$.*

Theorem: *Soundness of realizability:*
Assume that A is a formula. If A is proved, then there is a term, T, such that T q A can be proved in a trivially extended logic in which realizability relations are regarded as formulas, and $FV(T) \subset FV(A)$.

The proof of the theorem gives the algorithm of program extraction from constructive proofs. The program extracted from $(NonDet)$ is *if* $\bullet = L$ *then* M *else* N where M and N are the program extracted from the subproofs of two premises. From a proof by $(MPST)$, the program $\lambda X.\lambda m.ap(f_M, f_N^m(X))$ is extracted where f_M and f_N are as explained in section 4.3. Other part of the extraction algorithm can be seen in [Tak91].

6 Examples

The basic programming technique with $(MPST)$ is demonstrated in this section. In the following, we write X_n for $X(n)$ when X is a stream.

6.1 Simple Examples

A process which doubles each element of the input natural number stream is defined as follows:

SPEC 1: $\forall X : I_{nat}.\exists Y : I_{nat}.\forall n : nat.\ Y_n = 2 \cdot X_n$
Proof: The proof is continued by $(MPST)$. Let $M(X,a) \stackrel{\text{def}}{=} a = 2 \cdot hd(X)$, and (a) and (b) are easily proved. (c) is proved by letting $f = \lambda X.\ tl(X)$. ∎

The program extracted from the proof is $\lambda X.\lambda m.\ 2 \cdot hd(tl^m(X))$ which is, by the isomorphism φ, extensionally equal to $\nu z.\lambda X.\ (2 \cdot hd(X) :: z(tl(X)))$.

A process which takes the successive two elements at once from the input stream and outputs the sum of them is defined as follows:

SPEC 2: $\forall X : I_\sigma.\exists Y : I_\sigma.\forall n : nat.\ Y_n = X_{2\cdot n} + X_{2\cdot n+1}$
Proof: By $(MPST)$. Let $M(X,a) \stackrel{\text{def}}{=} a = hd(X) + hd(tl(X))$ and (a) and (b) are easily proved. (c) is proved by letting $f \stackrel{\text{def}}{=} \lambda X.\ tl^2(X)$. ∎

The program extracted from the proof is
$\lambda X.\lambda m.\ hd(tl^{2\cdot m}(X)) + hd(tl^{2\cdot m+1}(X))$
which is extensionally equal to $\nu z.\lambda X.\ (hd(X) + hd(tl(X)) :: z(tl^2(X)))$.

6.2 Parameterized Processes and Complete Stream Types

A filter process defined below removes all the elements of the input stream, X, which can be divided by a fixed natural number p. This process is an example of parameterized processes. The definition uses the complete stream type and the rank 0 formula technique.

SPEC 3: $\forall p : nat.\forall X : I_{nat}.\exists Y : I_{nat_\perp}.$
$\forall n : nat.\ \Diamond A(p,n,X,Y)$

where $A(p,n,X,Y) \stackrel{\text{def}}{=} ((p|X_n)\ \&\ Y_n = \perp) \vee (\neg(p|X_n)\ \&\ Y_n = X_n)$ and $\Diamond$ is the rank 0 operator.
Proof: Let $p : nat$ be arbitrary, and
$\forall X.\exists Y.\forall n.\ \Diamond A(p,n,X,Y)$ will be proved by $(MPST)$. Let $M(X,a) \stackrel{\text{def}}{=} ((p|hd(X))\ \&\ a = \perp) \vee (\neg(p|hd(X))\ \&\ a = hd(X))$. (a) is proved by divide and conquer with regard to $(p|hd(X)) \vee \neg(p|hd(X))$. (b) is proved easily, and (c) is proved by letting $f = \lambda X.\ tl(X)$. ∎

The program extracted from the proof is
$\lambda p.\lambda X.\lambda m.\ ap(f_M, f_N^m(X))$
where $f_M \stackrel{\text{def}}{=} \lambda X.\ if\ (p|hd(X))\ then\ \perp\ else\ hd(X)$ and $f_N \stackrel{\text{def}}{=} \lambda X.\ tl(X)$. Precisely, $(p|hd(X))$ should be a decision procedure for $(p|hd(X))$.

6.3 Dynamic Invocation of Processes

The following example, a program which extracts only prime numbers in the input stream, is one of the typical examples of dynamic creation of new processes.

SPEC 4: $\forall X : I_{nat}.\exists Y : I_{nat_\perp}.\forall n : nat.\ \Diamond A(n,X,Y)$
where
$$A(n,X,Y) \stackrel{\text{def}}{=} (PR(X_n)\ \&\ Y_n = X_n) \vee (\neg PR(X_n)\ \&\ Y_n = \perp)$$
and
$$PR(m) \stackrel{\text{def}}{=} \forall n : nat.\ (2 \le n < m \Rightarrow \neg(\exists d : nat.\ m = d \cdot n)).$$

Proof: By $(MPST)$. Let $M(X,a) \stackrel{\text{def}}{=} (PR(hd(X))\ \&\ a = hd(X)) \vee (\neg PR(hd(X))\ \&\ a = \perp)$. (a) is proved by divide and conquer with regard to $PR(hd(X)) \vee \neg PR(hd(X))$. (b) is proved easily. The proof of (c) is a little complex. Let $f \equiv \lambda X.\ if\ PR(hd(X))\ then\ flt(hd(X),tl(X))\ else\ tl(X)$ where $flt(p,X)$ is the filter process developed in the previous subsection. Then, for arbitrary $X : I_{nat_\perp}$ and $n : nat$ the following hold: 1. $PR(f(X)_n) \Rightarrow PR(tl(X)_n)$; 2. $\neg PR(f(X)_n) \Rightarrow \neg PR(tl(X)_n)$; 3. $PR(f(X)_n) \Rightarrow f(X)_n = tl(X)_n$. These can be proved by divide and conquer on $PR(hd(X)) \vee \neg PR(hd(X))$. Then, from $A(n, f(X), tl(Y)) \Leftrightarrow (PR(f(X)_n)\ \&\ Y_{n+1} = f(X)_n) \vee (\neg PR(f(X)_n)\ \&\ Y_{n+1} = \perp)$, $A(n+1,X,Y)$ can be proved. Then, (c) is proved. ∎

The program extracted from this proof is
$\lambda X.\lambda m.\ ap(f_M, f_N^m(X))$
where $f_M \stackrel{\text{def}}{=} \lambda X.\ if\ PR(hd(X))\ then\ hd(X)\ else\ \perp$ and $f_N \stackrel{\text{def}}{=}$
$\lambda X.\ if\ PR(hd(X))\ then\ flt(hd(X),tl(X))\ else\ tl(X)$.
This program performs load distribution in the following way. When a prime number, p, is found in the input stream, X, this program invokes a filter process, flt_p making X as the input stream of flt_p and take the output stream of flt_p as the new input stream.

6.4 Nondeterminacy

The stream merge operation is a typical example of nondeterminacy which can also be defined by $(MPST)$. However, because of the condition (c) on $A(n,(X,Y),Z)$, our specification is weaker than that of the merge operation. It only specifies that all the elements of the output stream come from the input streams. The rest of the criteria for a merge operation, namely, all the elements of the input streams occur in the output stream preserving the order of the input elements without repetition and loss, depends on how the formula M is defined in (a) and how f is defined for (c) in the premises of $(MPST)$.
SPEC 5: $\forall(X,Y)$: $I_{\sigma,\sigma}.\exists Z : I_\sigma$.

$$\forall n : nat.\ \Diamond A(n,(X,Y),Z)$$

where $A(n,(X,Y),Z) \stackrel{\text{def}}{=} (\exists m : nat.\ Z_n = X_m) \vee (\exists l : nat.\ Z_n = Y_l)$

Proof: By $(MPST)$. Let $M((X,Y),a) \stackrel{\text{def}}{=} a = hd(X)$, then the proofs of (a) and (b) are straightforward. (c) is proved as follows: Let $(X,Y) : I_{\sigma,\sigma}$, $Z : I_\sigma$ and $n : nat$ be arbitrary. Then, $A(n,(tl(X),Y),tl(Z)) \equiv (\exists m.\ tl(Z)_n = tl(X)_m) \vee (\exists l.\ tl(Z)_n = Y_l) \Leftrightarrow (\exists m.\ Z_{n+1} = X_{m+1}) \vee (\exists l.\ Z_{n+1} = Y_l)$. This implies $(\exists m'.\ Z_{n+1} = X_{m'}) \vee (\exists l.\ Z_{n+1} = Y_l) \equiv A(n+1,(X,Y),Z)$. Similarly, $A(n,(Y,tl(X)),tl(Z)) \Rightarrow A(n+1,(X,Y),Z)$ is proved. Then, two distinct proofs of (c) are given. Then, by $(NonDet)$, (c) is proved. ∎

The program extracted from this proof is
$\lambda(X,Y).\lambda m.\ ap(f_M, f_N^m(X,Y))$
where $f_M \stackrel{\text{def}}{=} \lambda X.hd(X)$ and $f_N \stackrel{\text{def}}{=} \lambda(X,Y).\ if\ \bullet = L\ then\ (tl(X),Y)\ else\ (Y,tl(X))$.

7 Conclusion and Future Works

An extension of constructive programming to stream based concurrent programming was proposed in this paper. The system has lazy types at the level of programming language and logical stream types, which are types of sequences viewed as streams, at the level of logic. This two level formulation of streams enables to formulate a purely natural deduction style of structural induction on streams $(MPST)$ in which concurrent processes (stream transformers) are defined as proofs. The $(MPST)$ rule allows to develop the proof of a specification with a good intuition on the concurrent process to be defined, and the rule seems to be easier to handle than the largest fixed point induction. Also, nondeterminacy was introduced at the level of logic using the inherent nondeterminacy of proof normalization in intuitionistic logic.
For the future work, as seen in the example of a merger process, the side condition for $(MPST)$ should be relaxed to handle larger varieties of concurrent processes.

References

[Bur75] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice–Hall, 1986.

[DS89] P. Dybjer and H. P. Sander. A Functional Programming Approach to the Specification and Verification of Concurrent Systems. *Formal Aspects of Computing*, 1:303 – 319, 1989.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1987. North–Holland.

[Got85] S. Goto. Concurrency in proof normalization and logic programming. In *Internatioinal Joint Conference on Artificial Intelligence '85*, 1985.

[Hag87] T. Hagino. A Typed Lambda Calculus with Categorical Type Constructors. In *Category Theory and Computer Science, LNCS 283*, 1987.

[HN89] S. Hayashi and H. Nakano. *PX : A Computational Logic.* The MIT Press, 1989.

[How80] W. A. Howard. The formulas-as-types notion of construction. In *Essays on Combinatory Logic, Lambda Calculus and Formalism, eds. J. P. Seldin and J. R. Hindley.* Academic Press, 1980.

[KM74] G. Kahn and D. B. MacQueen. The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress 74.* North–Holland, 1974.

[KT91] S. Kobayashi and M. Tatsuta. private communication. 1991.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[PL86] N. Mendler P. Panangaden and R. L.Constable. Infinite Objects in Type Theory. In *Symposium on Logic in Computer Science'86*, 1986.

[Sat87] M. Sato. Quty: A Concurrent Language Based on Logic and Function. In *Fourth International Conference on Logic Programming*, pages 1034–1056. The MIT Press, 1987.

[SK90] M. Sato and Y. Kameyama. Constructive Programming in SST. In *Proceedings of the Japanese-Czechoslovak Seminar on Theoretical Foundations of Knowledge Information Processing*, pages 23–30, INORGA, 1990.

[Tak91] Y. Takayama. Extraction of Redundancy-free Programs from Constructive Natural Deduction Proofs. *Journal of Symbolic Computation*, 12(1):29–69, 1991.

[TvD88] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction.* Studies in Logic and the Foundation of Mathematics 121 and 123. North–Holland, 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Realizability Interpretation of Coinductive Definitions and Program Synthesis with Streams

Makoto Tatsuta

Research Institute of Electrical Communication,
Tohoku University,
2-1-1 Katahira, Sendai 980, JAPAN
e-mail: tatsuta@riec.tohoku.ac.jp

Abstract

The main aim of this paper is to construct a logic by which properties of programs can be formalized for verification, synthesis and transformation of programs.

This paper has 2 main points. One point is realizability interpretation of coinductive definitions of predicates. The other point is an extraction of programs which treat streams.

An untyped predicative theory $\mathbf{TID}_\nu$ is presented, which has the facility of coinductive definitions of predicates and is based on a constructive logic. Properties defined by the greatest fixed point, such as streams and the extensional equality of streams, can be formalized by the facility of coinductive definitions of predicates in $\mathbf{TID}_\nu$.

q-realizability interpretation for $\mathbf{TID}_\nu$ is defined and the realizability interpretation is proved to be sound.

By the realizability interpretation, a program which treats streams can be extracted from a proof of its specification in $\mathbf{TID}_\nu$. General program extraction theorem and stream program extraction theorem are presented.

1 Introduction

Our main aim is to construct a logic by which we can formalize properties of programs for verification, synthesis and transformation of programs. In this paper, we concentrate on formalization of programs with streams and present a theory $\mathbf{TID}_\nu$.

Coinductive definitions are very important for this purpose. Properties of streams are represented semantically by the greatest fixed point. The predicate representing what a stream is and the extensional equality of streams are defined semantically by the greatest fixed point. These properties defined by the greatest fixed point can be formalized by coinductively defined predicates and coinduction.

μ-calculus has been studied to formalize programs with streams for verification [3]. μ-calculus has the facility of coinductive definitions of predicates and coinduction and is based on classical logic.

In this paper, we present a theory $\mathbf{TID}_\nu$, which has the facility of coinductive definitions of predicates and coinduction and is based on a constructive logic. By these facilities we can formalize properties of programs with streams in $\mathbf{TID}_\nu$.

Our theory $\mathbf{TID}_\nu$ is based on a constructive logic because we want to use the facility of program extraction by realizability for $\mathbf{TID}_\nu$. Program extraction is one of the benefits we get when we use a constructive formal theory to formalize properties of programs. Program extraction is to get a program from a constructive proof of its specification formula. One method of program extraction is to use realizability interpretation. In **PX**[4], for example, a LISP program is extracted from a proof of its specification formula by realizability interpretation.

By the facility of coinductive definitions of predicates and realizability interpretation, we can synthesize programs with streams naturally in $\mathbf{TID}_\nu$ using theorem proving techniques.

This paper has 2 main points. One point is realizability interpretation of coinductive definitions. The other point is an extraction of programs with streams.

We present an untyped predicative theory $\mathbf{TID}_\nu$, which has coinductive definitions of predicates and is based on a constructive logic. We define q-realizability interpretation of $\mathbf{TID}_\nu$. We show that the realizability interpretation is sound. We present general program extraction theorem and stream program extraction theorem.

The soundness proof is based on the early version of this paper [8]. The soundness theorem was proved also in [5]. Both works are independent.

In Section 2, we define a theory $\mathbf{TID}_\nu$. In Section 3, we briefly explain how useful the facility of coinductive definitions of predicates is to formalize streams. In Section 4, we discuss a model of $\mathbf{TID}_\nu$ and prove its consistency. In Section 5, we present q-realizability interpretation of $\mathbf{TID}_\nu$ and prove the soundness theorem. In Section 6, we give general program extraction theorem, stream

program extraction theorem for $\mathbf{TID}_\nu$ and an example of program synthesis.

2 Theory $\mathbf{TID}_\nu$

We present a theory $\mathbf{TID}_\nu$ in this section. It is the same as Beeson's **EON** [1] except for the axioms of coinductive definitions of predicates.

In this paper, we choose combinators as the target programming language for simplicity since we want to concentrate on the topic of coinductive definitions of predicates. We suppose that the evaluation strategy of combinators is lazy or call by name because we represent a stream by an infinite list, which is a non-terminating term. We omit also the formalization of the lazy or call by name evaluation strategy in $\mathbf{TID}_\nu$ for simplicity.

Definition 2.1. (Language of $\mathbf{TID}_\nu$)
The language of $\mathbf{TID}_\nu$ is based on a first order language but extended for coinductive definitions of predicates.

The constants are:

$\mathbf{K}$, $\mathbf{S}$, p, p_0, p_1, 0, s_N, p_N, d.

We choose combinators as a target programming language for simplicity. $\mathbf{K}$ and $\mathbf{S}$ mean the usual basic combinators. We have natural numbers as primitives, which are given by 0, a successor function s_N and a predecessor function p_N. We also have paring functions p, p_0 and p_1 as built-in, which correspond to cons, car and cdr in LISP respectively. d is a combinator judging equality of natural numbers and corresponds to an if-then-else statement in a usual programming language.

We have only one function symbol:

$\mathbf{App}$

whose arity is 2. It means a functional application of combinators.

Terms are defined in the same way as for a usual first order logic. For terms s, t, we abbreviate $\mathbf{App}(s,t)$ as st. For terms s, t, we also use an abbreviation $\langle s,t\rangle \equiv \mathsf{p}st$, $t_0 \equiv \mathsf{p}_0 t$ and $t_1 \equiv \mathsf{p}_1 t$.

The predicate symbols are:

$\bot$, N, $=$.

We have predicate variables, which a first order language does not have. The predicate variables are:

$X, Y, Z, \ldots, X^*, Y^*, Z^*, \ldots$.

Each predicate variable has a fixed arity.

We use an abbreviation $\lambda x.t$ which is constructed by combinators in the usual way. We also abbreviate $\mathrm{Y}(\lambda x.t)$ as $\mu x.t$ where $\mathrm{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

Definition 2.2. (Formula)
We define a formula A, a set $S_+(A)$ of predicate variables which occur positively in A and a set $S_-(A)$ of predicate variables which occur negatively in A.

1. If a, b are terms,
   $\bot$, $\mathrm{N}(a)$, $a = b$
   are formulas. Then
   $S_+(\bot) = S_-(\bot) = \phi$,
   $S_+(\mathrm{N}(a)) = S_-(\mathrm{N}(a)) = \phi$,
   $S_+(a = b) = S_-(a = b) = \phi$.

2. If X is a predicate variable whose arity is n, $X(x_1, \ldots, x_n)$ is a formula and
   $S_+(X(x_1, \ldots, x_n)) = \{X\}$,
   $S_-(X(x_1, \ldots, x_n)) = \phi$.

3. $A \,\&\, B$, $A \vee B$, $A \to B$, $\forall x A$, $\exists x A$ are formulas if A and B are formulas in the same way as a first order language. Then
   $S_+(A \,\&\, B) = S_+(A \vee B) = S_+(A) \cup S_+(B)$,
   $S_-(A \,\&\, B) = S_-(A \vee B) = S_-(A) \cup S_-(B)$,
   $S_+(A \to B) = S_-(A) \cup S_+(B)$,
   $S_-(A \to B) = S_+(A) \cup S_-(B)$,
   $S_+(\forall x A) = S_+(\exists x A) = S_+(A)$,
   $S_-(\forall x A) = S_-(\exists x A) = S_-(A)$.

4. $(\nu X.\lambda x_1 \ldots x_n.A)(t_1, \ldots, t_n)$ is a formula where X is a predicate variable whose arity is n, A is a formula, $t_1, \ldots, t_n$ are terms and X is not in $S_-(A)$. Then
   $S_+((\nu X.\lambda x_1 \ldots x_n.A)(t_1, \ldots, t_n)) =$
   $S_+(A) - \{X\}$,
   $S_-((\nu X.\lambda x_1 \ldots x_n.A)(t_1, \ldots, t_n)) = S_-(A)$.

$\bot$ means contradiction. $\mathrm{N}(a)$ means that a is a natural number. $a = b$ means that a equals to b.

The last case corresponds to coinductively defined predicates. Remark that X and $x_1, \ldots, x_n$ may occur freely in A. The intuitive meaning of a formula

$(\nu X.\lambda x_1 \ldots x_n.A(X, x_1, \ldots, x_n))(t_1, \ldots, t_n)$

is as follows: Let P be a predicate of arity n such that P is the greatest solution of an equation

$P(x_1, \ldots, x_n) \leftrightarrow A(P, x_1, \ldots, x_n)$.

Then $(\nu X.\lambda x_1 \ldots x_n.A(X, x_1, \ldots, x_n))(t_1, \ldots, t_n)$ means $P(t_1, \ldots, t_n)$ intuitively.

We abbreviate a sequence as a bold type symbol, for example, $x_1, \ldots, x_n$ as $\mathbf{x}$.

Example 2.3.
We give an example of a formula. We assume the arity of a predicate variable P is 1. Then

$(\nu P.\lambda x.x = \langle x_0, x_1\rangle \,\&\, x_0 = 0 \,\&\, P(x_1))(x)$

is a formula.

Among many axioms and inference rules of $\mathbf{TID}_\nu$, we discuss only inference rules of coinductive definitions of predicates here. The rest of axioms and inference rules are almost the same as **EON** [1] and we only list them in Appendix A.

Definition 2.4. (Coinductive Definitions)
Let $\nu \equiv \nu P.\lambda \mathrm{x}.A(P)$ where x is a sequence of variables whose length is the same as the arity of a predicate variable P and $A(P)$ is a formula displaying all the occurrences of P in a formula A. Suppose that $C(\mathrm{x})$ is a formula displaying all the occurrences of variables x in the formula.

We have the following axioms:

$$\forall \mathrm{x}(\nu(\mathrm{x}) \to A(\nu)), \qquad (\nu 1)$$

$$\forall \mathrm{x}(C(\mathrm{x}) \to A(C)) \to \forall \mathrm{x}(C(\mathrm{x}) \to \nu(\mathrm{x})). \qquad (\nu 2)$$

$\nu P.\lambda x.A(P)$ means the greatest fixed point of the function from a predicate P to a predicate $\lambda x.A(P)$.

We define a theory TID^- as a theory TID_ν except for the 2 axioms of coinductive definitions of predicates.

3 Coinductive Definitions of Predicates

We explain coinductive definitions of TID_ν and show some examples of formalization of streams by coinductive definitions.

Proposition 3.1.
Let ν be $\nu X.\lambda \mathrm{x}.A(X)$. Then

$$\forall \mathrm{x}(\nu(\mathrm{x}) \leftrightarrow A(\nu)) \qquad (\nu 1')$$

holds.

Proof 3.2.
By $(\nu 1)$, we get $\nu(\mathrm{x}) \to A(\nu)$. By letting C be $\lambda \mathrm{x}.A(\nu)$ in $(\nu 2)$, $A(\nu) \to \nu(\mathrm{x})$ holds. □

This proposition shows that $\nu P.\lambda \mathrm{x}.A(P)$ is the solution of the following recursive equation of a predicate P:

$$P(\mathrm{x}) \leftrightarrow A(P).$$

$(\nu 2)$ says that $\nu P.\lambda \mathrm{x}.A(P)$ is the greatest solution of this equation or the greatest fixed point of the function $\lambda P.\lambda x.A(P)$.

Streams can be formalized by coinductive definitions [3]. Therefore we can formalize streams in TID_ν.

We represent a stream by an infinite list $\langle a, s\rangle$ constructed by pairing where a is the first element of the stream, s is the rest of the stream. In this representation, if s is a stream, we can get the first element of s by s_0 and the rest by s_1.

We present an example of bit streams. A bit stream is a stream whose elements are 0 or 1. We will define a predicate $BS(x)$ which means that x is a bit stream. When we write down a formula $BS(x)$ in a naive way, BS itself occurs in the body of the definition as follows:

$$BS(x) \leftrightarrow x = \langle x_0, x_1\rangle \ \&\ (x_0 = 0 \vee x_0 = 1) \ \&\ BS(x_1).$$

BS is a solution P of the following equation for a predicate P

$$P(x) \leftrightarrow$$

$$x = \langle x_0, x_1\rangle \ \&\ (x_0 = 0 \vee x_0 = 1) \ \&\ P(x_1) \qquad (1)$$

or the fixed point of the function

$$\lambda P.\lambda x.x = \langle x_0, x_1\rangle \ \&\ (x_0 = 0 \vee x_0 = 1) \ \&\ P(x_1). \qquad (2)$$

There may be many solutions P for (1). For example, $\lambda x.\bot$ is one solution of (1), though it is not our intended solution. $\lambda x.\bot$ is the least solution. Our intended solution is the greatest solution of (1) or the greatest fixed point of (2). Hence we have the solution in TID_ν and it is represented as follows:

$$BS \equiv \nu P.\lambda x.x = \langle x_0, x_1\rangle \ \&\ (x_0 = 0 \vee x_0 = 1) \ \&\ P(x_1).$$

Let $\overline{0}$ be $\mu s.\langle 0, s\rangle$. $\overline{0}$ represents the zero stream whose elements are all 0. We can show $BS(\overline{0})$ by coinduction $(\nu 2)$. Let C be $\lambda x.(x = \overline{0})$ in $(\nu 2)$, then we have

$$\forall x(x = \overline{0} \to x = \langle x_0, x_1\rangle \ \&\ (x_0 = 0 \vee x_0 = 1) \ \&\ x_1 = \overline{0}) \to \forall x(x = \overline{0} \to BS(x)).$$

By definition of $\overline{0}$,

$$\forall x(x = \overline{0} \to x = \langle x_0, x_1\rangle \ \&\ (x_0 = 0 \vee x_0 = 1) \ \&\ x_1 = \overline{0})$$

holds and we have

$$\forall x(x = \overline{0} \to BS(x)).$$

Let $x = \overline{0}$ then we get $BS(\overline{0})$.

The coinductive definitions of predicates play an important role also to represent predicates of properties of streams [3, 6]. We will define the extensional equality $s \approx t$ for streams s and t. This equality can be represented by the coinductive definitions of predicates. $\approx$ is the greatest solution of the following equation for a predicate P:

$$P(x, y) \leftrightarrow x_0 = y_0 \ \&\ P(x_1, y_1).$$

Therefore $\approx$ can be formalized in TID_ν as follows:

$$\approx \equiv \nu P.\lambda xy.x_0 = y_0 \ \&\ P(x_1, y_1).$$

4 Model of TID_ν

We will briefly explain semantics of TID_ν by giving its intended model.

We will use classical set theory and the well-known greatest fixed point theorem for model construction in this section.

Theorem 4.1. (Greatest Fixed Point)
Suppose S be a set, $p(S)$ be a power set of S. If $f : p(S) \to p(S)$ is a monotone function, there exists a such that $a \in p(S)$ and

1. $f(a) = a$,
2. For any $b \in p(S)$, if $b \subset f(b)$, then $b \subset a$.

a is abbreviated as $\mathrm{gfp}(f)$.

We will construct a model M' of $\mathbf{TID}_\nu$ extending an arbitrary model M of $\mathbf{TID}^-$. Our intended model of $\mathbf{TID}^-$ is the closed total term model whose universe is the set of closed terms [1]. We denote the universe by U.

We will define $\rho \models A$ in almost the same way as for a first order logic where A is a formula and ρ is an environment which assigns a first order variable to an element of U and a predicate variable of arity n to a subset of U^n and which covers all the free first order variables and all the free predicate variables of A. We present only the definition for the case $(\nu P.\lambda \mathbf{x}.A(P))(\mathbf{t})$.

Define F as follows:

$|\mathbf{x}| = n$,

$F : p(U^n) \to p(U^n)$,

$F(X) = \{\mathbf{x} \in U^n \mid \rho[P := X] \models A(P)\}$,

where $\rho[P := X]$ is defined as follows:

$\rho[P := X](P) = X$,

$\rho[P := X](x) = \rho(x)$ if x is not P.

Then $\rho \models (\nu P.\lambda \mathbf{x}.A(P))(\mathbf{t})$ is defined as $\mathbf{t} \in \mathrm{gfp}(F)$. Note that F is monotone since a predicate variable P occurs only positively in $A(P)$.

Theorem 4.2.

If $\mathbf{TID}_\nu \vdash A$, then $\rho \models A$ for any environment ρ which covers all the free variables of A.

Theorem 4.3.

$\mathbf{TID}_\nu$ is consistent.

5 q-Realizability Interpretation of $\mathbf{TID}_\nu$

We will explain motivation of our realizability. We start with a usual $\mathbf{q}$-realizability and try to interpret $(\nu P.\lambda x.A(P))(x)$. Let ν be $\nu P.\lambda x.A(P)$ and then $\nu(x) \leftrightarrow A(\nu, x)$ holds. We want to treat $\nu(x)$ and $A(\nu, x)$ in the same manner. So we require $(e \ \mathbf{q} \ \nu(x)) \leftrightarrow (e \ \mathbf{q} \ A(\nu, x))$. Therefore it is very natural to define $(e \ \mathbf{q} \ \nu(x))$ as $\nu^*(e, x)$ where $\nu^*(e, x)$ is the greatest solution of a recursive equation for a predicate variable X^*:

$X^*(e, x) \leftrightarrow (e \ \mathbf{q} \ A(\nu, x))[(r \ \mathbf{q} \ \nu(y)) := X^*(r, y)]$.

where $[(r \ \mathbf{q} \ \nu(y)) := X^*(r, y)]$ of the right hand side means replacing each subformula $(r \ \mathbf{q} \ \nu(y))$ by a subformula $X^*(r, y)$ in a formula $(e \ \mathbf{q} \ A(\nu, x))$. We get the following definition of our realizability by describing syntactically this idea.

Our realizability in this paper is an extension of Grayson's realizability. We can also define usual $\mathbf{q}$-realizability of coinductively defined predicates in the same way as in this paper.

Definition 5.1. (Harrop formula)

1. Atomic formulas $\bot$, $\mathbf{N}(a)$ and $a = b$ are Harrop.
2. If A and B are Harrop, then $A \,\&\, B$, $C \to B$, $\forall x A$ and $(\nu P.\lambda \mathbf{x}.A)(\mathbf{t})$ are also Harrop.

Since a Harrop formula does not have computational meanings, we can simplify the $\mathbf{q}$-realizability interpretation of them.

Definition 5.2. (Abstract)

1. A predicate constant of arity n is an abstract of arity n.
2. A predicate variable of arity n is an abstract of arity n.
3. If A is a formula, $\lambda x_1 \ldots x_n.A$ is an abstract of arity n.

We identify $(\lambda x_1 \ldots x_n.A)(t_1, \ldots, t_n)$ with $A[x_1 := t_1, \ldots, x_n := t_n]$ where $[x_1 := t_1, \ldots, x_n := t_n]$ denotes a substitution.

Definition 5.3. (q-realizability Interpretation)

Suppose A is a formula, $P_1, \ldots, P_n$ is a sequence of predicate variables whose arities are $m_1, \ldots, m_n$ respectively and $F_1, G_1, \ldots, F_n, G_n$ is a sequence of abstracts whose arities are $m_1, m_1+1, \ldots, m_n, m_n+1$ respectively.

$(e \ \mathbf{q}_{P_1,\ldots,P_n}[F_1, G_1, \ldots, F_n, G_n] \ A)$

is defined by induction on the construction of A as follows.

We abbreviate $\mathbf{q}_{P_1,\ldots,P_n}[F_1, G_1, \ldots, F_n, G_n]$ as $\mathbf{q}'$, $\mathbf{q}_{P_1,\ldots,P_n,P}[F_1, G_1, \ldots, F_n, G_n, F, G]$ as $\mathbf{q}'_P[F, G]$, $F_1, \ldots, F_n$ as $\mathbf{F}$ and $P_1, \ldots, P_n$ as $\mathbf{P}$.

1. $(e \ \mathbf{q}' \ A) \equiv e = 0 \,\&\, A_{\mathbf{P}}[\mathbf{F}]$ where A is Harrop.
2. $(e \ \mathbf{q}' \ P_i(\mathbf{t})) \equiv F_i(\mathbf{t}) \,\&\, G_i(e, \mathbf{t})$.
3. $(e \ \mathbf{q}' \ Q(\mathbf{t})) \equiv Q(\mathbf{t}) \,\&\, Q^*(e, \mathbf{t})$ where $Q \not\equiv P_i \ (1 \le i \le n)$.
4. $(e \ \mathbf{q}' \ A \,\&\, B) \equiv (e_0 \ \mathbf{q}' \ A) \,\&\, (e_1 \ \mathbf{q}' \ B)$.
5. $(e \ \mathbf{q}' \ A \vee B) \equiv \mathbf{N}(e_0) \,\&\, (e_0 = 0 \to (e_1 \ \mathbf{q}' \ A)) \,\&\, (e_0 \neq 0 \to (e_1 \ \mathbf{q}' \ B))$.
6. $(e \ \mathbf{q}' \ A \to B) \equiv (A \to B)_{\mathbf{P}}[\mathbf{F}] \,\&\, \forall q((q \ \mathbf{q}' \ A) \to (eq \ \mathbf{q}' \ B))$.
7. $(e \ \mathbf{q}' \ \forall x A(x)) \equiv \forall x(ex \ \mathbf{q}' \ A(x))$.
8. $(e \ \mathbf{q}' \ \exists x A(x)) \equiv (e_1 \ \mathbf{q}' \ A(e_0))$.
9. $(e \ \mathbf{q}' \ (\nu X.\lambda \mathbf{x}.A(X))(\mathbf{t})) \equiv (\nu X^*.\lambda e\mathbf{x}.(e \ \mathbf{q}'_X[\nu_{\mathbf{P}}[\mathbf{F}], X^*] \ A(X)))(e, \mathbf{t})$ where $\nu \equiv \nu X.\lambda \mathbf{x}.A(X)$.

In the above definition, ${}_{P_1,\ldots,P_n}[F_1, G_1, \ldots, F_n, G_n]$ means a substitution. Our realizability interpretation is something like a realizability interpretation with a substitution.

Proposition 5.4.
Let $\nu \equiv \nu P.\lambda \mathbf{x}.A(P)$.

1. $\forall \mathbf{x} r((r \ \ \mathbf{q} \ \ \nu(\mathbf{x})) \leftrightarrow (r \ \ \mathbf{q} \ \ A(\nu)))$.

2. $\lambda \mathbf{x} r.r \ \ \mathbf{q} \ \ \forall \mathbf{x}(\nu(\mathbf{x}) \rightarrow A(\nu))$.

Proof 5.5.
By the definition of $\mathbf{q}$-realizability and $(\nu 1')$. □

Definition 5.6.
For a formula A, a predicate variable P and a term f, we define a term $\sigma_A^{P,f}$ by induction on the construction of A as follows:

1. A is a Harrop formula, then $\sigma_A^{P,f} \equiv \lambda r.0$.

2. $A \equiv P(\mathbf{t})$, then $\sigma_A^{P,f} \equiv \lambda r.f\mathbf{t}r$.

3. $A \equiv Q(\mathbf{t})$, then $\sigma_A^{P,f} \equiv \lambda r.r$ if $Q \not\equiv P$.

4. $A \equiv A_1 \ \& \ A_2$, then $\sigma_A^{P,f} \equiv \lambda r.\langle \sigma_{A_1}^{P,f} r_0, \sigma_{A_2}^{P,f} r_1 \rangle$.

5. $A \equiv A_1 \vee A_2$, then $\sigma_A^{P,f} \equiv \lambda r.\langle r_0, \mathbf{d}r_0 0 \sigma_{A_1}^{P,f} \sigma_{A_2}^{P,f} r_1 \rangle$.

6. $A \equiv A_1 \rightarrow A_2$, then $\sigma_A^{P,f} \equiv \lambda r q.\sigma_{A_2}^{P,f}(r(\sigma_{A_1}^{P,f} q))$.

7. $A \equiv \forall x A_1(x)$, then $\sigma_A^{P,f} \equiv \lambda r x.\sigma_{A_1(x)}^{P,f}(rx)$.

8. $A \equiv \exists x A_1(x)$, then $\sigma_A^{P,f} \equiv \lambda r.\langle r_0, \sigma_{A_1(r_0)}^{P,f} r_1 \rangle$.

9. $A \equiv (\nu Q.\lambda \mathbf{x}.A_1)(\mathbf{t})$, then $\sigma_A^{P,f} \equiv (\mu g.\lambda \mathbf{x} r.\sigma_{A_1}^{Q,g}(\sigma_{A_1}^{P,f} r))\mathbf{t}$ where $Q \not\equiv P$.

Proposition 5.7.
Let $\nu \equiv \nu P.\lambda \mathbf{x}.A(P)$. Then
$\lambda q.\mu f.\lambda \mathbf{x} r.\sigma_{A(P)}^{P,f}(q\mathbf{x}r) \ \ \mathbf{q}$
$\forall \mathbf{x}(C(\mathbf{x}) \rightarrow A(C)) \rightarrow \forall \mathbf{x}(C(\mathbf{x}) \rightarrow \nu(\mathbf{x}))$
holds.

We prove it in Appendix B.

Theorem 5.8. (Soundness Theorem)
If $\mathbf{TID}_\nu \vdash A$, we can get a term e from the proof of $\vdash A$ and $\mathbf{TID}_\nu \vdash (e \ \ \mathbf{q} \ \ A)$ holds where all the free variables of e are included in all the free variables of A.

Proof 5.9.
By induction on the proof of $\vdash A$. The case of the axiom $(\nu 1)$ is proved by Proposition 5.4. The case of the axiom $(\nu 2)$ is proved by Proposition 5.7. □

6 Program Synthesis with Streams

In this section, we give general program extraction theorem, stream program extraction theorem for $\mathbf{TID}_\nu$ and an example of program synthesis.

Program synthesis by theorem proving techniques has been studied both in typed theories [2] and untyped theories [4]. For untyped theories, realizability interpretation is used as the foundation of program synthesis by theorem proving techniques. In Section 3, we showed that streams and programs which treat streams can be formalized in $\mathbf{TID}_\nu$ by the facility of coinductively definitions of predicates. In Section 5, we showed that realizability interpretation can be defined for $\mathbf{TID}_\nu$ and the interpretation is sound. Hence we can synthesize programs which treat streams by theorem proving techniques in $\mathbf{TID}_\nu$ using realizability interpretation.

We represent streams by infinite lists constructed by pairing. We represent a specification of a program by a formula:
$\forall x(A(x) \rightarrow \exists y B(x,y))$
where x is an input, y is an output, $A(x)$ is an input condition and $B(x,y)$ is an input output relation.

Theorem 6.1. (Program Extraction)
Suppose that we prove a specification formula $\forall x(A(x) \rightarrow \exists y B(x,y))$ of a program in $\mathbf{TID}_\nu$ and we have a realizer j such that
$\forall x(A(x) \rightarrow (jx \ \ \mathbf{q} \ \ A(x)))$.
Then we can get a program f and a proof of
$\forall x(A(x) \rightarrow B(x, fx))$
effectively from the proof of the specification formula.

Proof 6.2.
Since the specification formula is proved in $\mathbf{TID}_\nu$, by soundness theorem of $\mathbf{q}$-realizability interpretation we have a realizer e such that
$e \ \ \mathbf{q} \ \ \forall x(A(x) \rightarrow \exists y B(x,y))$
holds. Let f be $\lambda x.(ex(jx))_0$. Then the claim holds. □

We can synthesize programs in the following steps:

1. We write down a specification formula.

2. We prove the specification formula in $\mathbf{TID}_\nu$.

3. We extract a program from the proof.

The program extraction theorem says that the third step can be automated completely.

Example 6.3.
We show an example of the program which gets a stream of natural numbers and returns a stream whose each element is the element of the input stream plus one.

The predicate $NS(x)$ which says that x is a stream of natural numbers can be represented in $\mathbf{TID}_\nu$ by the facility of coinductive definitions of predicates as follows:

$NS \equiv \nu X.\lambda x.x = \langle x_0, x_1\rangle \,\&\, \mathbf{N}(x_0) \,\&\, X(x_1)$.

The input condition of the specification is a formula $NS(x)$.

The input output relation of the specification is a formula $ADD1(x,y)$ which is defined as follows:

$ADD1 \equiv \nu X.\lambda xy.y_0 = x_0 + 1 \,\&\, X(x_1, y_1)$.

The specification formula is:

$\forall x(NS(x) \to \exists y ADD1(x,y))$.

We have one problem for this program synthesis method. The coinduction cannot be applied to the part $\forall x(NS(x) \to \ldots)$ in the above example. We cannot prove $\exists y ADD1(x,y)$ by the coinduction in general. Therefore the realizer of the coinduction cannot give a loop structure for the program. On the other hand, a realizer of the induction principle plays an important role for this approach of program synthesis since the realizer corresponds to a loop structure of a program [4, 7]. Therefore we need the new method by which a realizer of the coinduction also corresponds to a loop structure and is useful.

Then we need more specialized program extraction method for programs with streams in which the coinduction is useful. We give one solution for this problem by the next theorem.

We put 2 restrictions on the theorem: One is that the input condition $A(x)$ must be the form $(\nu X.\lambda x.x = \langle x_0, x_1\rangle \,\&\, \tilde{A}(x_0) \,\&\, X(x_1))(x)$ for some $\tilde{A}$. The other is that the input output relation $B(x,y)$ must be the form $(\nu X.\lambda xy.\tilde{B}(x,y_0) \,\&\, X(x_1,y_1))(x,y)$ for some $\tilde{B}$. These restrictions require an input condition and an input output relation are uniform over data and they are natural when we suppose that an input x and an output y are both streams.

Theorem 6.4. (Stream Program Extraction)

Suppose that the specification formula is $\forall x(A(x) \to \exists y B(x,y))$,

$A \equiv \nu X.\lambda x.x = \langle x_0, x_1\rangle \,\&\, \tilde{A}(x_0) \,\&\, X(x_1)$,

$B \equiv \nu X.\lambda xy.\tilde{B}(x,y_0) \,\&\, X(x_1,y_1)$

and we have a term j such that $\forall x(A(x) \to (jx \;\mathbf{q}\; A(x)))$. Then we define

$B^\circ \equiv \nu X.\lambda x.\exists z\tilde{B}(x,z) \,\&\, X(x_1)$.

If we have e such that

$e \;\mathbf{q}\; \forall x(A(x) \to B^\circ(x))$,

we can get a term F such that

$\forall x(A(x) \to B(x, Fx))$

where

$filter \equiv \mu f.\lambda x.\langle x_{00}, fx_1\rangle$,

$F \equiv \lambda x.filter(ex(jx))$.

We prove it in Appendix C.

By this theorem, we can synthesize programs in the following steps:

1. We write down a specification formula $\forall x(A(x) \to \exists y B(x,y))$.
2. We prove the corresponding formula $\forall x(A(x) \to B^\circ(x))$ in $\mathbf{TID}_\nu$.
3. We extract a program $\lambda x.filter(ex(jx))$ from the proof where e is a realizer of the corresponding formula $\forall x(A(x) \to B^\circ(x))$.

In the second step, we can apply the coinduction to prove the part $B^\circ(x)$ since $B^\circ(x)$ is defined by coinductive definitions. Therefore a realizer of the coinduction can correspond to a loop structure of the program.

Example 6.5.

We treat the same example as above again. The specification formula is a formula $\forall x(NS(x) \to \exists y ADD1(x,y))$. Hence the formula $ADD1^\circ(x)$ is:

$$ADD1^\circ \equiv \nu X.\lambda x.\exists z(z = x_0 + 1) \,\&\, X(x_1). \tag{3}$$

Therefore the corresponding formula we must prove is:

$$\forall x(NS(x) \to ADD1^\circ(x)). \tag{4}$$

If we prove this formula in $\mathbf{TID}_\nu$, we can get the program which satisfies the specification by stream program extraction theorem.

The conditions of the theorem hold for this case. We can put $j \equiv \lambda x.\mu s.\langle 0, s\rangle$ since

$\forall x(NS(x) \to (\mu s.\langle 0, s\rangle \;\mathbf{q}\; NS(x)))$.

We prove (4) in the following way here: Firstly, we prove

$$\forall x(NS(x) \to \exists z(z = x_0 + 1) \,\&\, NS(x_1)). \tag{5}$$

This is proved by letting z be $x_0 + 1$. Secondly, by letting C be NS in $(\nu 2)$ for $ADD1^\circ$, we have

$$\forall x(NS(x) \to \exists z(z = x_0 + 1) \,\&\, NS(x_1)) \to \forall x(NS(x) \to ADD1^\circ(x)). \tag{6}$$

Finally, by (5) and (6), we get (4).

We calculate realizers corresponding to the above proofs as follows: The realizer corresponding to the proof of (5) is:

$e_1 \equiv \lambda xr.\langle\langle x_0 + 1, 0\rangle, r_{11}\rangle$,

$e_1 \;\mathbf{q}\; \forall x(NS(x) \to \exists z(z = x_0 + 1) \,\&\, NS(x_1))$.

The realizer corresponding to the proof of (6) is:

$e_2 \equiv \lambda q.\mu f.\lambda xr.\sigma(qxr)$,

$e_2 \;\mathbf{q}\; \forall x(NS(x) \to \exists z(z = x_0 + 1) \,\&\, NS(x_1)) \to \forall x(NS(x) \to ADD1^\circ(x))$

where

$\sigma \equiv \lambda r.\langle\langle r_{00}, r_{01}\rangle, fx_1r_1\rangle$.

The realizer corresponding to the proof of (4) is:

$e \equiv e_2e_1$,

$e \;\mathbf{q}\; \forall x(NS(x) \to ADD1^\circ(x))$.

We get

$$e = \mu f.\lambda x r.\langle\langle x_0 + 1, 0\rangle, f x_1 r_{11}\rangle.$$

The extracted program is:

$$\begin{aligned} Fx &= \mathit{filter}(ex(jx)) \\ &= \mathit{filter}(fx(\mu s.\langle 0, s\rangle)) \\ &= (\mu g.\lambda x.\langle x_0 + 1, g x_1\rangle)x \end{aligned}$$

where $f \equiv \mu f.\lambda x r.\langle\langle x_0 + 1, 0\rangle, f x_1 r_{11}\rangle$. This is the program we expect.

Remark that the realizer e_2 of the coinduction (6) gives a loop structure of the program F.

Acknowledgments

I would like to thank Mr. Satoshi Kobayashi and Mr. Yukiyoshi Kameyama for careful comments. I'm deeply grateful to Prof. Masahiko Sato for invaluable discussions and comments.

References

[1] M. Beeson, *Foundations of Constructive Mathematics* (Springer, 1985).

[2] R.L. Constable et al., *Implementing Mathematics with the Nuprl Proof Development System* (Prentice-Hall, 1986).

[3] P. Dybjer and H.P. Sander, A Functional Programming Approach to the Specification and Verification of Concurrent Systems, *Formal Aspects of Computing* **1** (1989) 303–319.

[4] S. Hayashi and H. Nakano, **PX**: *A Computational Logic* (MIT Press, Cambridge, 1988).

[5] S. Kobayashi, Inductive/Coinductive Definitions and Their Realizability Interpretation, Manuscript (1991).

[6] R. Milner, *Communication and Concurrency* (Prentice Hall, 1989).

[7] M. Tatsuta, Program Synthesis Using Realizability, *Theoretical Computer Science* **90** (1991) 309–353.

[8] M. Tatsuta, Realizability Interpretation of Greatest Fixed Points, Manuscript (1991).

[9] M. Tatsuta, Monotone Recursive Definition of Predicates and Its Realizability Interpretation, *Proceedings of Theoretical Aspects of Computer Software*, LNCS **526** (1991) 38–52.

Appendix

A Axioms and Inference Rules of TID_ν

The logical axioms and inference rules are the same as the ones of a usual intuitionistic logic.

Axioms for Equality:

$$\forall x(x = x) \tag{E1}$$

$$\forall x, y(x = y \,\&\, A(x) \to A(y)) \tag{E2}$$

Axioms for Combinators:

$$\forall x, y(\mathbf{K}xy = x) \tag{C1}$$

$$\forall x, y, z(\mathbf{S}xyz = xz(yz)) \tag{C2}$$

Axioms for Pairing:

$$\forall x, y(\mathbf{p}_0(\mathbf{p}xy) = x) \tag{P1}$$

$$\forall x, y(\mathbf{p}_1(\mathbf{p}xy) = y) \tag{P2}$$

Axioms for Natural Numbers:

$$\mathbf{N}(0) \tag{N1}$$

$$\forall x(\mathbf{N}(x) \to \mathbf{N}(\mathbf{s}_\mathbf{N}x)) \tag{N2}$$

$$\forall x(\mathbf{N}(x) \to \mathbf{p}_\mathbf{N}(\mathbf{s}_\mathbf{N}x) = x) \tag{N3}$$

$$\forall x(\mathbf{N}(x) \to \mathbf{s}_\mathbf{N}x \neq 0) \tag{N4}$$

$$A(0) \,\&\, \forall x(\mathbf{N}(x) \,\&\, A(x) \to A(\mathbf{s}_\mathbf{N}x)) \to \forall x(\mathbf{N}(x) \to A(x)) \tag{N5}$$

Axioms for $\mathbf{d}$:

$$\forall x, y, a, b(\mathbf{N}(x) \,\&\, \mathbf{N}(y) \,\&\, x = y \to \mathbf{d}xyab = a) \tag{D1}$$

$$\forall x, y, a, b(\mathbf{N}(x) \,\&\, \mathbf{N}(y) \,\&\, x \neq y \to \mathbf{d}xyab = b) \tag{D2}$$

B Proof of Soundness Theorem

Lemma B.1.

(1) If a predicate variable P occurs only positively in a formula A,

$$(r \;\; \mathbf{q}_P[F, \lambda y\mathbf{x}.(y \;\; \mathbf{q} \;\; C(\mathbf{x}))] \;\; A) \to (\sigma_A^{P,f} r \;\; \mathbf{q}_P[F, \lambda y\mathbf{x}.\exists r((r \;\; \mathbf{q} \;\; C(\mathbf{x})) \,\&\, y = f\mathbf{x}r)] \;\; A).$$

(2) If a predicate variable P occurs only negatively in a formula A,

$$(r \;\; \mathbf{q}_P[F, \lambda y\mathbf{x}.\exists r((r \;\; \mathbf{q} \;\; C(\mathbf{x})) \,\&\, y = f\mathbf{x}r)] \;\; A) \to (\sigma_A^{P,f} r \;\; \mathbf{q}_P[F, \lambda y\mathbf{x}.(y \;\; \mathbf{q} \;\; C(\mathbf{x}))] \;\; A).$$

Proof B.2.

We prove (1) and (2) simultaneously by induction on the construction of A. □

Proof B.3. (of 5.7)

Let $\nu \equiv \nu P.\lambda\mathbf{x}.A(P)$.

Suppose

$$\forall\mathbf{x}(C(\mathbf{x}) \to A(C)),$$

$$q \;\; \mathbf{q} \;\; \forall\mathbf{x}(C(\mathbf{x}) \to A(C))$$

and let

$$f \equiv \mu f.\lambda\mathbf{x}r.\sigma_{A(P)}^{P,f}(q\mathbf{x}r).$$

We show

$$f \;\; \mathbf{q} \;\; \forall\mathbf{x}(C(\mathbf{x}) \to \nu(\mathbf{x})).$$

Let $\nu^*(r, \mathbf{x}) \equiv (r \quad \mathbf{q} \quad \nu(\mathbf{x}))$. It is sufficient to show
$\forall \mathbf{x} r((r \quad \mathbf{q} \quad C(\mathbf{x})) \to \nu^*(f\mathbf{x}r, \mathbf{x}))$.
This is equivalent to
$\forall \mathbf{x} y(\exists r((r \quad \mathbf{q} \quad C(\mathbf{x})) \,\&\, y = f\mathbf{x}r) \to \nu^*(y, \mathbf{x}))$.
By $(\nu 2)$, it is sufficient to show
$\forall \mathbf{x} y(\exists r((r \quad \mathbf{q} \quad C(\mathbf{x})) \,\&\, y = f\mathbf{x}r) \to$
$(y \quad \mathbf{q}_P[\nu, \lambda y\mathbf{x}.\exists r((r \quad \mathbf{q} \quad C(\mathbf{x})) \,\&\, y = f\mathbf{x}r)] \quad A(P)))$.
This is equivalent to
$\forall \mathbf{x} r((r \quad \mathbf{q} \quad C(\mathbf{x})) \to$
$(f\mathbf{x}r \quad \mathbf{q}_P[\nu, \lambda y\mathbf{x}.\exists r((r \quad \mathbf{q} \quad C(\mathbf{x})) \,\&\, y = f\mathbf{x}r)]$
$A(P)))$.
Fix $\mathbf{x}$ and r and assume
$r \quad \mathbf{q} \quad C(\mathbf{x})$.
We show
$f\mathbf{x}r \quad \mathbf{q}_P[\nu, \lambda y\mathbf{x}.\exists r((r \quad \mathbf{q} \quad C(\mathbf{x})) \,\&\, y = f\mathbf{x}r)] \quad A(P)$.
By the assumption about q,
$q\mathbf{x}r \quad \mathbf{q} \quad A(C)$.
Hence
$q\mathbf{x}r \quad \mathbf{q}_P[C, \lambda y\mathbf{x}.(y \quad \mathbf{q} \quad C(\mathbf{x}))] \quad A(P)$.
By positivity and $\forall \mathbf{x}(C(\mathbf{x}) \to \nu(\mathbf{x}))$,
$q\mathbf{x}r \quad \mathbf{q}_P[\nu, \lambda y\mathbf{x}.(y \quad \mathbf{q} \quad C(\mathbf{x}))] \quad A(P)$.
By Lemma B.1,
$\sigma^{P,f}_{A(P)}(q\mathbf{x}r) \quad \mathbf{q}_P[\nu, \lambda y\mathbf{x}.\exists r((r \quad \mathbf{q} \quad C(\mathbf{x})) \,\&\, y = f\mathbf{x}r)]$
$A(P)$.
By $f\mathbf{x}r = \sigma^{P,f}_{A(P)}(q\mathbf{x}r)$, we have
$f\mathbf{x}r \quad \mathbf{q}_P[\nu, \lambda y\mathbf{x}.\exists r((r \quad \mathbf{q} \quad C(\mathbf{x})) \,\&\, y = f\mathbf{x}r)] \quad A(P)$.
□

C Proof of Stream Extraction Theorem

Lemma C.1.
Suppose that
$A \equiv \nu X.\lambda x.x = \langle x_0, x_1\rangle \,\&\, \tilde{A}(x_0) \,\&\, X(x_1)$,
$B \equiv \nu X.\lambda xy.\tilde{B}(x, y_0) \,\&\, X(x_1, y_1)$,
$B^\circ \equiv \nu X.\lambda x.\exists z\tilde{B}(x, z) \,\&\, X(x_1)$.
Then
$\forall f(\forall x(A(x) \to (fx \quad \mathbf{q} \quad B^\circ(x))) \to$
$\forall x(A(x) \to B(x, \mathit{filter}(fx))))$
holds.

Proof C.2.
By only rules of **NJ**, the above goal is equivalent to
$\forall xy(\exists f(\forall x(A(x) \to (fx \quad \mathbf{q} \quad B^\circ(x))) \,\&$
$A(x) \,\&\, y = \mathit{filter}(fx)) \to B(x, y))$.
By $(\nu 2)$, it is sufficient to show
$\forall xy(\exists f(\forall x(A(x) \to (fx \quad \mathbf{q} \quad B^\circ(x))) \,\&$
$A(x) \,\&\, y = \mathit{filter}(fx)) \to$
$\tilde{B}(x, y_0) \,\&\, \exists g(\forall x(A(x) \to (gx \quad \mathbf{q} \quad B^\circ(x))) \,\&$
$A(x_1) \,\&\, y_1 = \mathit{filter}(gx_1)))$.
By only rules of **NJ**, it is equivalent to
$\forall xf(\forall x(A(x) \to (fx \quad \mathbf{q} \quad B^\circ(x))) \,\&\, A(x) \to$
$\tilde{B}(x, (\mathit{filter}(fx))_0) \,\&$
$\exists g(\forall x(A(x) \to (gx \quad \mathbf{q} \quad B^\circ(x))) \,\&$
$A(x_1) \,\&\, (\mathit{filter}(fx))_1 = \mathit{filter}(gx_1)))$. (7)
We will prove it.
Fix x and f and assume that
$\forall x(A(x) \to (fx \quad \mathbf{q} \quad B^\circ(x)))$, (8)
$A(x)$. (9)
By (8) and (9), $(fx \quad \mathbf{q} \quad B^\circ(x))$ holds. Hence
$((fx)_{01} \quad \mathbf{q} \quad \tilde{B}(x, (fx)_{00})) \,\&\, ((fx)_1 \quad \mathbf{q} \quad B^\circ(x_1))$ (10)
holds. Therefore $\tilde{B}(x, (\mathit{filter}(fx))_0)$ holds since $(\mathit{filter}(fx))_0 = (fx)_{00}$.
Put g be $\lambda y.(f\langle x_0, y\rangle)_1$. We will show $\forall y(A(y) \to (gy \quad \mathbf{q} \quad B^\circ(y)))$. Fix y and assume that $A(y)$. By the definition of A,
$A(x) \leftrightarrow x = \langle x_0, x_1\rangle \,\&\, \tilde{A}(x_0) \,\&\, A(x_1)$
and
$A(\langle x_0, y\rangle) \leftrightarrow \tilde{A}(x_0) \,\&\, A(y)$
hold. By this and (9), $\tilde{A}(x_0)$ holds. Hence $A(\langle x_0, y\rangle)$ holds. Combined it with (8), we get $(f\langle x_0, y\rangle \quad \mathbf{q} \quad B^\circ(\langle x_0, y\rangle))$. Hence $((f\langle x_0, y\rangle)_1 \quad \mathbf{q} \quad B^\circ(y))$ and $(gy \quad \mathbf{q} \quad B^\circ(y))$ hold. Therefore we get $\forall y(A(x) \to (gy \quad \mathbf{q} \quad B^\circ(y)))$.
By (9), $A(x_1)$ holds. Since, in general, $(\mathit{filter}(s))_1 = \mathit{filter}(s_1)$ holds, we get $(\mathit{filter}(fx))_1 = \mathit{filter}((fx)_1) = \mathit{filter}(gx_1)$. Therefore (7) holds. □

Proof C.3. (of Theorem 6.4)
By the assumptions and the definition of $\mathbf{q}$-realizability, $\forall x(A(x) \to (ex(jx) \quad \mathbf{q} \quad B^\circ(x)))$ holds. Letting f be $\lambda x.ex(jx)$ in Lemma C.1, we get $\forall x(A(x) \to B(x, Fx))$. □

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

MLOG: A STRONGLY TYPED CONFLUENT FUNCTIONAL LANGUAGE WITH LOGICAL VARIABLES

Vincent Poirriez*

Université de Reims
INRIA-Ecole Normale Supérieure PARIS, FRANCE

Abstract

A new programming language called MLOG is introduced. MLOG is a conservative extension of ML with logical variables. To validate our concepts, a compiler named CAML Light FLUO was implemented. Numerous examples are presented to illustrate the possibilities of MLOG. The pattern-matching of ML is kept for λ-calculus bindings and an unification primitive is introduced for the logical variables bindings. A suspension mechanism allows cohabitation of pattern-matching and logical variables. Though the evaluation strategy for the application is fixed, the order for evaluation of the parts of pairs and application remains free. MLOG programs can be evaluated in parallel with the same result obtained irrespective of the particular order of evaluation. This is guaranteed by the Church Rosser property observed by the evaluation rules. As a corollary, a strict λ-calculus with explicit substitutions on named variables is shown to be confluent. A completely formal operational semantics of MLOG is given in this paper.

1 Introduction

Many attempts have been made at integrating functional and logical tools in the same language. It actually seems worthwile to combine the strengths of the two paradigms, allowing the programmer to choose the most appropriate tool to resolve his problem. The approach we have followed is to add "logical" tools to a well-known strongly typed functional language: ML. To validate our ideas and to demonstrate that MLOG is a realistic proposal, we have implemented a compiler for MLOG named "CAML Light FLUO". It is an extension of the CAML Light system of X.Leroy[Leroy 90]. Logical variables and unification serve two goals in logical languages: to handle partially defined values, and to provide a resolution mechanism. The implementation of logical variables and unification is a required step to implement a resolution mechanism, so we bypass that second goal and focus on the first one. MLOG is an extension of ML with built-in logical variables instantiable once, and unification. We allow a fruitful cohabitation of logical variables and ML pattern matching by introducing a suspension mechanism, thus when an application cannot be evaluated due to a lack of information, the application is suspended. In the designing of MLOG, we strive to obtain a *conservative extension of ML.* Pure ML programs are not penalized by the extension. This result is obtained by limiting the domain of logical variables and suspensions to specified *logical types.* Moreover, MLOG inherits from ML a strong system of types and a safety property for the execution of well-typed programs. Thus the programmer does not waste energy in checking types. In this article, we trace the execution of programs that illustrate that synchronisation algorithms, demand driven computation, algorithms using potentially infinite data structures or partially instantiated values are easily written in MLOG. Then we focus on the confluence property. In MLOG, the strategy for the evaluation of an application is strict evaluation: i.e. we impose the evaluation of the argument before reducing the application. Nevertheless, some freedom remains in the order of evaluation of a term: both parts of an application or of a pair for example. Then MLOG is independent of the implementation choices and it can be implemented on a parallel machine. As we fix the strategy for the evaluation of the applications, we can name variables without risking clashes. A complete operational semantics is given in appendix. A subset limited to the functional part of these rules is a strict λ-calculus with explicit substitutions and named variables that verify the Church Rosser property. That calculus is a very simple formalism and as it is confluent, it is a good candidate to describe any implementation of strict λ-calculus, even a parallel one.

2 MLOG syntax and examples

We describe here the added syntax to ML. As MLOG is an extension of ML, all programs of ML are programs of

*Projet Formel BP 105 Domaine de Voluceau
78153 Rocquencourt Cedex , FRANCE.
poirriez@margaux.inria.fr

MLOG. For clearness, we limit ourselves to a mini-ML. All examples are produced by a session of our system CAML Light FLUO. Note that # is the prompt and ;; the terminator of our system.

2.1 Syntax

The language we consider is λ-calculus with pattern-matching, concrete types (either built-in, as *int* or *string*, or declared by the user), constructors, the `let` construct and the conditional. We first define the set P of *programs* of MLOG. We assume the existence of a countable set Var of term variables, with typical elements x, y, and a disjoint countable set C of constructors, with typical elements c. Some constructors are predefined: integers, strings, booleans (true, false) and `()`, the element of type unit. In the following, i ranges over integers and s over strings. The syntax of patterns, with typical element p, is:
$p ::= x \mid c \mid (p_1, \ldots, p_n) \mid c\ p$
As in ML, we limit ourselves to linear patterns. The syntax of programs, with typical elements a, b, is:

$$a ::= x \mid c \mid a\ b \mid (a_1, \ldots, a_n) \mid let\ x = a\ in\ b \mid a; b$$
$$(function\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n) \mid undef \mid unif$$

$a; b$ is the ML notation for a sequence, it means evaluate a then evaluate b and return the value of b. The last two constructs are specific to MLOG: *undef* is a generator of fresh logical variables; *unif* is the unification primitive. `let_var u in ...` is syntactic sugar for `let u = undef in ...`.

2.2 Types

In MLOG, the programmer has to declare specially the types that may contain undefined objects (that is, logical variables and suspensions). The notion of *logical type*, is introduced. We assume given a countable set of type variables $TVar$, with typical elements *'a*, *'b*, a disjoint countable set of variables over logical types $LTVar$ with typical elements *'a?*, *'b?* and two countable sets of type constructors with typical elements ident and lident. The sets of logical types $\mathcal{L}$, with typical element τ_i, and types $\mathcal{T}$ (typical element t_i) are recursively defined by:

$\tau_i ::= {'a?} \mid [t_i]$ lident
and
$t_i ::= \tau_i \mid bool \mid int \mid string \mid unit \mid t_i \rightarrow t_j \mid t_i * t_j \mid$
$[t_i]$ ident

Note that $\mathcal{L}$ is a strict subset of $\mathcal{T}$. Expressions to declare new type are :

type ['a,...,'k]ident = c [of t_i][| ... |c' [of t_j]] |
type logic ['a,...,'k] lident = c [of t_i][| ... |c' [of t_j]]

where [ ] surround optional expressions. A logical type is declared by the new key-word: `type logic`. The type `void` below has a unique value `void` and logical variables of type `void` may be declared. The type `void` is isomorphic to the type `unit` except that no logical variable can be declared in `unit`. A value of the type `Bool` below is `True`, `False`, or a free logical variable that will possibly be instantiated later to either `True` or `False`.

```
#type logic void = void;;
 Type void defined.
#type logic Bool = True | False;;
 Type Bool defined.
```

The following rules govern type variable instantiations: (1) `'a` may be instantiated by any type (including `'b?`); (2) `'a?` may be instantiated by any logical type; (3) `'a?` may not be instantiated by a non logical type.

We write "a : t_i" the program a of type t_i. Thus, the set of MLOG programs is in fact the subset of the well-typed programs $P_{\mathcal{T}}$ of P defined by the familiar ML type system. We just have to specify that : (1) undef : *'a?*; (2) unif : $'a \rightarrow' a \rightarrow void$. Fortunately, as far as types are concerned, logical variables and assignable constructs are quite close, we have adapted to logical variables previous work done for typing assignable objects in ML. We have directly applied the idea of Pierre Weis and Xavier Leroy [LeroyWeis 91], and, using their notion of cautious generalization, we get an extension of the ML type system to logical variables that is *sound*:

Theorem 1 *No evaluation of a well-typed program can leads to a run-time type error.*

Thus CAML Light Fluo has a type-checker that infers and checks the types of programs.

2.3 Examples

We give below very simple examples to illustrate the semantics of unification and logical variables in MLOG. First logical variables are instantiable once, when the unification fails, the exception `Unify` is raised:

```
#let (u:Bool) = undef;;
Value u : Bool u = ?
#unif u True; unif u False;;
- : void  Uncaught exception: Unify
#u;;
- : Bool - = True
```

CAML Light FLUO prints "?" for a free logical variable. Rational trees are allowed; `unif` does not perform any occur-check. Moreover, `unif` does not loop when unifying rational trees. The type `'a stream` below implements the potentially infinite lists.

```
#type logic 'a stream = Nil |St of 'a * 'a stream;;
Type stream defined.
#let (u:int stream) = undef;;
Value u : int stream u = ?
#unif u (St(1,u));u;;
- : int stream
- = St (1, St (1, St (1, St (1,Interrupted.
```

The printing of u was interrupted by a system break. At that point we can use classical technics used in the logical languages, see for example in the appendix the classical functional quicksort program, except that difference lists are used instead of lists to improve the concatenation of sorted sublists.

2.4 Suspensions: an intuitive semantics

Consider first the example below:

```
#let neg = function True -> False |False -> True;;
Value neg : Bool -> Bool
#let b,exp = let_var u in (u, neg u);;
Value b : Bool Value exp : Bool  b = ? exp = ...
```

b is a new free logical variable of type Bool. The application cannot match u with True or False: u is free. So what is the meaning of exp? The answer is: the application neg u is suspended. Thus, exp is a suspension of type Bool[1]. A suspension is a first class citizen in MLOG. It may be handled in data structures, and used in other expressions.

```
#let exp' = unif exp False;;
Value exp' : void  exp' = ...
```

Since exp is a suspension, MLOG cannot perform the unification of exp with False. Therefore this unification is also suspended[2]. Let us now instantiate b with True, and look at exp and exp'.

```
#unif b True; exp,exp';;
Value - : Bool * void  -: (False,void)
```

We have to clarify when a suspension is awakened. Awakening a suspension could be delayed until it is actually needed. We must define when such an evaluation is needed :

```
#let (a,b,e) = let_var a,b in
 (a,b,(function True ->(unif a True))b);;
Value a : Bool  Value b : Bool Value e: void
a = ?  b = ?  e = ...
```

e is suspended waiting for the instantiation of b.

```
#unif b True;;
Value - : void - = void
```

As b is instantiated, e can be awakened. If we choose to wake up a suspension only if its value is needed, e remains suspended and then a remains free. If the value of a is needed, nothing indicates that the evaluation of e will instantiate a. This motivates our choice to *wake up all suspended evaluations that can be awakened.* Another motivation is that, if an expression is suspended, it is because its evaluation was needed and unfortunately was stopped by lack of information. So if we look at a:

```
#a;; Value - : Bool - = True
```

The example above illustrates the fine control on evaluation allowed by the suspension mechanism. The application is performed and then a is instantiated only when b is instantiated.

3 A confluence result

To give an operational semantics for MLOG we have to deal with bindings of λ-calculus variables, bindings of logical variables and suspensions. We give here a simple formalism that allows us to keep named parameters and we show that this calculus is strongly confluent[3]. In this section we neglect types.

3.1 A strict calculus with environment

We store bindings of parameters in environments. We call $E\Lambda$ the set of terms with environments. As our calculus is strict, we specialize a subset Val of $E\Lambda$ which is the set of the values handled by the language. Typical elements of Val and $E\Lambda$ are respectively noted v and t.

e ::= [] | (x,v)::e
v ::= c | c(v) | (v,v') | (function ...).e
t ::= c | c(t) | (t,t') | t(t') |a.e

3.2 Logical variables, substitutions and suspensions

Now we have to extend the set Val with logical variables. We assume the existence of a countable set U disjoint with V and C with typical element $u(i)$, distinct logical variables have distinct indexes. We call $LVal$ and $EL\Lambda$ the obtained sets of values and terms with environments. To manage the bindings of logical variables we define substitutions as functions from U to $EL\Lambda$. We will use greek letters to note substitutions. We call the domain of σ and note $dom(\sigma)$ the set $\{u(i)\ s.t.\ \sigma(u(i)) \neq u(i)\}$. We will note $\sigma \circ \alpha$ the composition of substitutions. The MLOG pattern matching algorithm has to deal with logical variables. It has to

[1]Note that CAML Light FLUO prints suspensions as "...".

[2]That is why the type of the result of unif has to be a logical type. We do not want to have suspension in a non logical type.

[3]Recall that if no strategy for application is imposed, name clash may occurs. To avoid that problem, the names of variables can be replaced by numbers "à la De Bruijn"[AbadiCaCuLe 90, HardinLevy 90]

access to the pointed value when it checks a bound variable, it fails with *Unknown* when it tries to match a free logical variable with a construct pattern. We define the match of a term t with a pattern *pat* in the substitution σ and note $\Phi_\sigma(pat, t)$ as the list of appropriate bindings of parameters of pat. Recall that patterns are linear. We define now a sequential pattern matching without entering into the optimization of the algorithm[4].

if $\Phi_\sigma(p_0, t) = e$ then $\Phi s_\sigma(i, p_0 :: pl, t) = i, e$
if $\Phi_\sigma(p_0, t) = Unknown$ then $\Phi s_\sigma(i, p_0 :: \_, t) = i, Unknown$
if $\Phi_\sigma(p_0, t) = fail$ then $\Phi s_\sigma(i, p_0 :: [], t) = i, fail$
if $\Phi_\sigma(p_0, t) = fail$ and $pl \neq []$then
$\Phi s_\sigma(i, p_0 :: patl, t) = \Phi s_\sigma(i + 1, patl, t)$

When the pattern matching fails with *Unknown*, we suspend the application. We do not want to have to go throughout the term to wake up suspensions or to duplicate suspensions when reducing application. On other hand, we note that both free logical variables and suspensions are holes in the term that will be plugged in when more information is broadcast. So we replace the new suspension by a logical variable $u(j)$ (with $j < 0$ to recall that it is created for a suspension) and we bind $u(j)$ with the suspension in a dedicated substitution α(See rules **Susp** and **ASusp** in figure 2). As explained above, unification may build rational trees, thus a naive recursive application of a substitution to a term may loop. We define $\sigma^*(t)$ as the recursive application of σ to t that does not substitute a logical variable if it has already been substituted in a prenex occurrence of t. More precisely, we call M the set of the logical variables of $dom(\sigma)$ already met, σ^* is defined by:

$\sigma^* = \emptyset \vdash \sigma^*$ and
$M \vdash \sigma^*(u(i)) = u(i)$ *if* $u(i) \in M$ or $u(i) \in dom(\sigma)$
$M \vdash \sigma^*(u(i)) = \{u(i)\} \cup M \vdash \sigma^*(\sigma(u(i)))$ *if* $u(i) \notin M$
$M \vdash \sigma^*(c) = c$
$M \vdash \sigma^*(t(t')) = (M \vdash \sigma^*(t))(M \vdash \sigma^*(t'))$
$M \vdash \sigma^*(t, t') = (M \vdash \sigma^*(t), M \vdash \sigma^*(t'))$
$M \vdash \sigma^*(p.e) = (M \vdash \sigma^*(p).M \vdash \sigma^*(e))$

3.3 Unification

The used unification procedure is adapted from [Huet 76]. We do not discuss here the whole algorithm but the three following points deserve mention: (1) We do not want to open the Pandora's Box of higher order unification, so when we compare closures we limit ourselves to physical identity (we assume an appropriate primitive *eq*).(2) When the procedure has to unify a suspension with any other term, it stops and returns *susp*[5]. (3) When the procedure has to unify a free logical variable with a construct term, the unification is performed even if a suspension occurs in the term. We define $unif_{\sigma_0}(t, t')$ by:
(a) $unif_{\sigma_0}(t, t') = \sigma$ iff the unification procedure applied to$\{(t, t')\}$ with the initial substitution σ_0 succeeds and builds the substitution σ.
(b) $unif_{\sigma_0}(t, t') = fail$ iff the unification procedure applied to $\{(t, t')\}$ with the initial substitution σ_0 stops with $fail$.
(c) $unif_{\sigma_0}(t, t') = susp(u(i))$ iff the unification procedure applied to $\{(t, t')\}$ with the initial substitution σ_0 stops with $susp(u(i))$.
The following result holds:

Theorem 2 *For all terms t, t' $unif_{\sigma_0}(t, t')$ terminates and:(a) if t and t' are not unifiable in the initial substitution σ_0, then $unif_{\sigma_0}(t, t') = fail$ or $susp(\_)$; (b) otherwise if there is at least one pair of the form $(u(j), t")$ with $j < 0$ built then $unif_{\sigma_0}(t, t') = susp(\_)$ (c) else $unif_{\sigma_0}(t, t') = \sigma$ which is the most general unifier of (t, t'), moreover there is no cycle in σ of the form $\sigma^*(u(i)) = u(i)$.*

3.4 Confluence of the reduction over $EL\Lambda$

The reduction has to account for the bindings of logical variables and those of logical variables created for the suspensions. Moreover, it has to deal with waking up the suspensions. Thus we define $\rightarrow$ as the smallest relation over $EL\Lambda \times substitutions \times substitutions \times substitutions$ that verifies the rules given in figures 1 and 2 in appendix . A 4-tuple is note by $< t, \sigma, \alpha, \Gamma >$ where t is the term to reduce. The substitution σ stores the bindings of unified logical variables and updated suspensions. The valuation α stores the suspensions (recall they are bound to $u(j)$ with $j < 0$). The substitution Γ stores the suspensions of which evaluations are running. We use the classical notation $\stackrel{*}{\rightarrow}$ and $\stackrel{n}{\rightarrow}$ for reflexive transitive closure of $\rightarrow$ and for derivations of length n. We first have two lemmas that say that no term of the form (a.e).e' is produced and that the term component of a normal form is a value.

Lemma 1 *Let a be a program and $< a.[], \emptyset, \emptyset, \emptyset > \stackrel{n}{\rightarrow} < t, \sigma, \alpha, \Gamma >$. For all subterms of t of the form $t'.e$, t' is a program.*

Lemma 2 *Let a be a program and $< a.[], \emptyset, \emptyset, \emptyset > \stackrel{*}{\rightarrow} < t, \sigma, \alpha, \Gamma >$ such that $< t, \sigma, \alpha, \Gamma >$ is a normal form. Then t is a value.*

We can deduce from these lemmas that all bindings in σ bind a variable with a value. Let us remark now that if no suspension rule is applied, as we do not reduce under a λ and we impose a strict calculus we have strong confluence for our reduction rules.

[4]The interested reader is referred to [Laville 88] and [PuelSuarez 90] for presentation of optimized algorithms in the framework of functional lazy evaluation. Such algorithms may be of some interest for our language as they avoid useless tests and then avoid useless suspensions.

[5]*susp* is returned even if the procedure has to unify a free logical variable and a suspension.

Proposition 1 *Let* $< t, \sigma, \alpha, \Gamma > \rightarrow < t_1, \sigma_1, \alpha, \Gamma_1 >$ *and* $< t, \sigma, \alpha, \Gamma > \rightarrow < t_2, \sigma_2, \alpha, \Gamma_2 >$ *two reduction using respectively rules* $r1$ *and* $r2$ *with* ri *not a suspension rule. Then we have by the application of respectively* $r2$ *and* $r1$*:* $< t_1, \sigma_1, \alpha, \Gamma_1 > \rightarrow < t_3, \sigma_3, \alpha, \Gamma_3 >$ *and* $< t_2, \sigma_2, \alpha, \Gamma_2 > \rightarrow < t_3, \sigma_3, \alpha, \Gamma_3 >$

An important corollary of that result is that if we restrict ourselves to the functional subset of MLOG, we have describe a strong confluent calculus with explicit substitutions and named variables. That calculus is rather simple (all that concerns logical variables and suspensions is unnecessary) and describes all implementations of a strict λ-calculus, even a parallel one.

Remark that $\rightarrow$ is not strongly confluent on the whole language. That is illustrated by the example below where the choice is between **UnifT** and **Susp** and the diagram cannot be closed in one step as even if **UnifT** is chosen after **Susp** waking up the suspension remains to be done.

$$< ((fun\ c \rightarrow c').[]\ u(1), unif\ u(1)\ c), \emptyset, \emptyset, \emptyset >$$

We can see the use of a rule **Susp**, **ASusp** or **USusp** as the translation of subterm from the term to Γ. From a reduction point of view we can say that these rules do not work. Thus the idea is to define an equivalence between four_uples $< t, \sigma, \alpha, \Gamma >$ which is stable for these suspension rules and then show the strong confluence of $\rightarrow$ up to that equivalence.

Definition 1 $< t, \sigma, \alpha, \Gamma > \equiv < t', \sigma', \alpha', \Gamma' >$ *iff*

1. *there exists a permutation* $\mathcal{P}$ *over positive variable index such that* $(\sigma \circ \alpha \circ \Gamma)^*(t) = \mathcal{P}(\sigma' \circ \alpha' \circ \Gamma')^*(t')$
2. *and for all* $u(i)$ *in* $dom(\sigma)$ *with* $i > 0$, $(\sigma \circ \alpha \circ \Gamma)^*(u(i)) = \mathcal{P}(\sigma' \circ \alpha' \circ \Gamma')^*(u(\mathcal{P}(i)))$
3. *and for all* $u(i)$ *in* $dom(\alpha) \cup dom(\Gamma)$ *or there exists* $j < 0$ *such that* $u(j)$ *in* $dom(\alpha') \cup dom(\Gamma')$ *and* $(\sigma \circ \alpha \circ \Gamma)^*(u(i)) = \mathcal{P}(\sigma' \circ \alpha' \circ \Gamma')^*(u(j))$*, either there exists a subterm* t'_i *of* t' *such that* $(\sigma \circ \alpha \circ \Gamma)^*(u(i)) = \mathcal{P}(\sigma' \circ \alpha' \circ \Gamma')^*(t'_i)$

   and vice versa for all $u(i)$ *in* $dom(\alpha') \cup dom(\Gamma')$

or $t = t' = failwith(s)$

Thus we have verified the Church Rosser property (the proof is in appendix C):

Theorem 3 *If* $< t, \sigma, \alpha, \Gamma >$ *has a normal form for* $\rightarrow$ *then it is unique up to* $\equiv$

Remark that if we add types as defined in the section above, the rules have not to be modified and the result holds.

4 MLOG: a conservative extension of ML

The fact that the type of `undef` is `'a?` ensures that no logical variable occurs in a non-logical type. That is not enough to ensure that no suspension of a non-logical type is built. Fortunately, we handle type information when we compile the pattern matching. Thus we have the following rules for the application:

Let f be a function of type $t_1 \rightarrow t_2$: (1) if type t_1 is a non-logical type, then do not do any test to check if the argument is a free variable or a suspension. (2) if type t_1 is a logical type, then (21) first, test if the argument is a bound logical variable or an updated suspension, and access the bound value. (22) if type t_2 is a non-logical type, test if the argument is a free variable or a suspension. If so, raise failure *Unknown*. (23) if type t_2 is a logical type, test if the argument is a free variable or a suspension. If so build and return the appropriate suspension.

Example:

```
#type logic 'a partial = P of 'a;;
 Type partial defined.
#(function (P x) ->x) undef;;
uncaught exception Unknown
```

Theorem 4 *Let* a *be a well-typed program. The evaluation of* a *cannot build a logical variable or a suspension of a non-logical type.*

We can now deduce that MLOG is a conservative extension of ML as pure ML programs need not know for the extension. However, it is clear that with that rule of failure, our calculus is no longuer Church Rosser. To keep that property, we must not use functions from a logical type to a non-logical type. Let call $MLOG^*$ the subset of MLOG that does not contain such functions. Thus, we have the following result.

Proposition 2 *The relation* $\rightarrow$ *is confluent on* $MLOG^*$.

Remark: The counterpart of the conservative property of MLOG is the need to be cautious with logical variables and "functional types". First, for any instances of $'a$ and $'b$ the type $'a \rightarrow' b$ cannot include a logical variable as it is a "pure ML" type. Anyway, it is correct to have logical variables of type $(int \rightarrow int)partial$ as illustrated below.

```
#let app (P h) (P x) = P (h x);;
Value app:('a->'b)partial->'a partial->'b partial
#let (g: (int -> int)partial)=undef;;
Value g : (int -> int) partial  g = ?
#let e2 = app g (P 2);;
Value e2 : int partial  e2 = ...
```

```
#unif g (P (fun x -> x*x));;
- : void  - = void
#e2;;
- : int partial  - = P 4
```

5 Conclusion

We have defined MLOG as an extension of ML. We have shown that it verifies a Church Rosser property and then it may be parallelized or used to simulate parallel processes. Such processes can communicate with each other through shared logical variables and the suspension mechanism allows synchronization. Partial data are handled by MLOG, for example potentially infinite lists can be implemented by the use of free logical variables for the tail of the structure (see example in appendix).

MLOG includes a suspension mechanism, let us now compare it to some other proposals of integration that have made a similar choice. MLOG is close to the language Qute defined by M.Sato and T.Sakurai in [SatoSakurai 86]. However, it differs from it in the following points: (1) its evaluation strategy ensures that the evaluation of a suspended expression will be tried only when needed information is provided; (2) the reduction of an application is allowed even if a subexpression of the argument is suspended, the only condition is that pattern matching succeeds, in that case the binding of the suspension by a logical variable and the storage in α avoid duplication of that suspension.

MLOG is also close to GHC of K.Ueda [Ueda 86], the main difference (except for typing point of view) is that MLOG does not have non-determinism for rule selection and that we have preferred to keep the functional formalism in place of the predicate one as selection of rules is done by pattern matching. However, determinist GHC programs are easily translated in MLOG[6].

The use of a suspension mechanism and the cohabitation of logical variables and functions are common to Le Fun of H.Ait Kaci[Ait Kaci 89] and MLOG. Here the main differences are that Le Fun provides a resolution mechanism based on backtracks and that MLOG is strongly typed.

Perhaps the main difference between MLOG and these related works is that MLOG is a conservative extension of ML. We demonstrate that the type system of ML can be extended to MLOG and we gave a safety property for well typed programs. As a side effect, we have described an operational semantics for strict λ-calculus which uses names for parameters and verifies the Church Rosser property. Therefore it can be used to describe any interpreter of strict λ-calculus, even parallel one. If it seems desirable, further work can be done to provide a resolution mechanism in MLOG. Note that the exhaustive search transformation described by K.Ueda in [Ueda 86] is applicable.

We hope that MLOG is an attractive extension of ML as from a "logical paradigm" point of view it allows handling incomplete data structures and controlled parallel evaluation with the improvement of the ML type system. And from a "functional paradigm" point of view, it respects functional programs with the improvement of partial data and a fair control mechanism.

Acknowledgments: We would like to thanks all members of LIENS-INRIA Formel project for helpful discussions. In particular Therese Hardin for her accurate suggestions to improve our formalism and demonstration.

[6]The author has traduced all programs given by G.Huet in [Huet 88], he found that the use of types and of a functional formalism lead to more clear programs.

A Appendix: MLOG programs

The program below is the classical functional quicksort program, except that difference lists are used instead of lists to improve the concatenation of sorted sublists. This is done by the use of the same variable `r` in both recursive calls of `qsortrec`.

```
#let partition order x =
let rec partrec = function
 Nil -> Nil,Nil
|St(h,t) -> let infl,supl = partrec t in
  if order(h,x) then St(h,infl),supl else infl,St(h,supl)
 in partrec ;;
Value partition :
('a*'b->bool)->'b->'a stream->'a stream*'a stream
#let quicksort order l =
 let rec qsortrec = function
 (Nil,result,sorted) -> (unif result sorted); result
|(St(h,t),presult,sorted) ->
  let infl,supl = partition order h t in
  let_var r in (qsortrec(supl,r,sorted);
                qsortrec(infl,presult,St(h,r)))
 in qsortrec (l,undef,Nil) ;;
Value quicksort:('a*'a->bool)->'a stream->'a stream
```

The following example illustrates the use of potentially infinite lists and demand driven computation. The confluence property allows to parallelize the evaluation of nested applications in the definition of the Hamming sequence of integers of the form $2^i * 3^j * 5^k$[Dijkstra 76].

```
#let mult (P x,P y) = P(x*y);;
Value mult : int partial * int partial -> int partial
#let rec times (u,St(v,r)) = St(mult(u,v),times(u,r));;
Value times:
 int partial*int partial stream->int partial stream
#let rec merge (St(P x,s),St(P y,r)) =
 if x<y then St(P x,merge (s,St(P y,r))) else
 if x>y then St(P y,merge (St(P x,s),r)) else
 St(P x, merge(s,r));;
Value merge: int partial stream*int partial stream ->
               int partial stream
#let rec copy_stream (St(a,b)as s) (St(h,t)) =
  unif a h; copy_stream b t; s;;
Value copy_stream : 'a stream -> 'a stream -> 'a stream
```

```
#let Hamming = let_var r in
 copy_stream
  (St(P 1,merge(merge(times(P 2,r),times(P 3,r)),
                times(P 5,r)))) r;
 r;;
Value Hamming : int partial stream  Hamming = ?
#let rec increase_stream st = function
   0 -> st
 | n -> let_var tail in unif st St(undef,tail);
        increase_stream tail (n-1) ;;
Value increase_stream : 'a? stream -> int -> 'a? stream
#increase_stream Hamming 9; Hamming;;
Value - : int partial stream
- =St(P 1,St(P 2,St(P 3,St(P 4,St(P 5,St(P 6,St(P 8,
      St(P 9,St(P 10,?)))))))))
```

B Reduction rules

Pair1F $$\frac{< t,\sigma,\alpha,\Gamma >\rightarrow< failwith(s),\sigma,\alpha,\Gamma >}{< (t,t'),\sigma,\alpha,\Gamma >\rightarrow< failwith(s),\sigma,\alpha,\Gamma >}$$

Pair2F $$\frac{< t',\sigma,\alpha,\Gamma >\rightarrow< failwith(s),\sigma,\alpha,\Gamma >}{< (t,t'),\sigma,\alpha,\Gamma >\rightarrow< failwith(s),\sigma,\alpha,\Gamma >}$$

Pair1 $$\frac{< t,\sigma,\alpha,\Gamma >\rightarrow< t_1,\sigma_1,\alpha_1,\Gamma_1 >}{< (t,t'),\sigma,\alpha,\Gamma >\rightarrow< (t_1,t'),\sigma_1,\alpha_1,\Gamma_1 >}$$

Pair2 $$\frac{< t',\sigma,\alpha,\Gamma >\rightarrow< t'_1,\sigma_1,\alpha_1,\Gamma_1 >}{< (t,t'),\sigma,\alpha,\Gamma >\rightarrow< (t,t'_1),\sigma_1,\alpha_1,\Gamma_1 >}$$

Figure 1: Structural rules

We assume that we have a function *queue* such that $queue_{\sigma,\alpha}u(i)$ returns all the suspensions in α waiting for instantiation of $u(i)$. The rule **DVar** uses a counter c that is increased each time a new logical variable is created. c is initially at 1. The rules **Susp** and **USusp** use an other counter c_s dedicated to suspensions also initially at 1, they increase α with the new suspension. The rules **UnifT** and **AwUpd** increase σ with the new bindings and increase Γ with the suspensions waiting for these instantiations or update. Note that we remain free to choose the order of evaluation of binary constructs as for $E\Lambda$ (We give in figure 1 the rules for pairs, rules for unification and application are similar.). Moreover, the order of evaluation of terms bound in Γ is also free (see rule **Aw**).

C Demonstration of theoreme 3

Let us give preliminary results.

Lemma 3 *If* $< t,\sigma,\alpha,\Gamma >\rightarrow< t',\sigma',\alpha',\Gamma' >$ *by application of a suspension rule then* $< t,\sigma,\alpha,\Gamma >\equiv< t',\sigma',\alpha',\Gamma' >$

Proposition 3 *If* $< t_1,\sigma_1,\alpha_1,\Gamma_1 >\rightarrow< t'_1,\sigma'_1,\alpha'_1,\Gamma'_1 >$ *by application of a rule distinct of a suspension rule, and if* $< t_1,\sigma_1,\alpha_1,\Gamma_1 >\equiv< t_2,\sigma_2,\alpha_2,\Gamma_2 >$ *then we have* $< t'_2,\sigma'_2,\alpha'_2,\Gamma'_2,>$ *such that* $< t_2,\sigma_2,\alpha_2,\Gamma_2 >\rightarrow< t'_2,\sigma'_2,\alpha'_2,\Gamma'_2 >$ *and* $< t'_1,\sigma'_1,\alpha'_1,\Gamma'_1 >\equiv< t'_2,\sigma'_2,\alpha'_2,\Gamma'_2 >$

Proof: We carefully discuss one case, others are similar:

Env $< x.(x,t) :: \_,\sigma,\alpha,\Gamma >\rightarrow< t,\sigma,\alpha,\Gamma >$

Env0 $< x.(y,t) :: e,\sigma,\alpha,\Gamma >\rightarrow< x.e,\sigma,\alpha,\Gamma >$

Const $< c.e,\sigma,\alpha,\Gamma >\rightarrow< c,\sigma,\alpha,\Gamma >$

AEnv $< (t\ t').e,\sigma,\alpha,\Gamma >\rightarrow< (t.e\ t'.e),\sigma,\alpha,\Gamma >$

UEnv $< (unif\ t\ t').e,\sigma,\alpha,\Gamma >\rightarrow< (unif\ t.e\ t'.e),\sigma,\alpha,\Gamma >$

PEnv $< (t,t').e,\sigma,\alpha,\Gamma >\rightarrow< (t.e,t'.e),\sigma,\alpha,\Gamma >$

DVar $$\frac{< undef.e,\sigma,\alpha,\Gamma >\rightarrow< u(c),\sigma,\alpha,\Gamma > \text{ and } c\leftarrow(c+1)}{}$$

β $$\frac{< t,\sigma,\alpha,\emptyset > \text{ is in } \rightarrow \text{ normal form}\quad \sigma^*(f)=(fun\ p_1\rightarrow a_1\mid\ldots\mid p_n\rightarrow a_n).e,\quad \Phi s_\sigma(1,[p_i],t)=i,e_i}{< f\ t,\sigma,\alpha,\Gamma >\rightarrow< a_i.e_i\ @\ e,\sigma,\alpha,\Gamma >}$$

Susp $$\frac{< t,\sigma,\alpha,\Gamma > \text{ is in } \rightarrow \text{ normal form. } c_s{=}k\quad \sigma^*(f)=(fun\ p_1\rightarrow a_1\mid\ldots\mid p_n\rightarrow a_n).e,\quad \Phi s_\sigma(1,[p_i],t)=Unknown}{< f\ t,\sigma,\alpha,\Gamma >\rightarrow< u(-n),\sigma,(u(-n),\sigma^*(f)\ t)::\alpha,\Gamma > \text{ and } c_s\leftarrow(k+1)}$$

ASusp $$\frac{< t,\sigma,\alpha,\Gamma > \text{ is in } \rightarrow \text{ normal form. } c_s{=}n\quad \sigma^*(f)=u(i)}{< f\ t,\sigma,\alpha,\Gamma >\rightarrow< u(-n),\sigma,(u(-n),u(i)\ t)::\alpha,\Gamma > \text{ and } c_s\leftarrow(n+1)}$$

Fail $$\frac{< t,\sigma,\alpha,\emptyset > \text{ is in } \rightarrow \text{ normal form. } \Phi s_\sigma(1,[p_i],t)=fail\quad \sigma^*(f)=(fun\ p_1\rightarrow a_1\mid\ldots\mid p_n\rightarrow a_n).e,}{< f\ t,\sigma,\alpha,\Gamma >\rightarrow< failwith(Pattern),\sigma,\alpha,\Gamma >}$$

UnifT $$\frac{< t,\sigma,\alpha,\emptyset > \text{ and } < t',\sigma,\alpha,\emptyset > \text{are in } \rightarrow \text{ normal form}\quad unif_\sigma(t,t')=\sigma'\quad \text{Let } L=\emptyset \text{ if } \sigma'=\sigma \text{ or } \sigma'(u(i))=u(j)\quad for\ all\ u(i)\in dom(\sigma')\backslash dom(\sigma)\quad \text{and } L=queue_{\sigma,\alpha}(u(i)) \text{ in other cases}}{< unif\ t\ t',\sigma,\alpha,\Gamma >\rightarrow< void,\sigma',\alpha\backslash L>,L\cup\Gamma >}$$

UnifF $$\frac{< t,\sigma,\alpha,\emptyset > \text{ and } < t',\sigma,\alpha,\emptyset > \text{ are in } \rightarrow \text{ normal form}\quad unif_\sigma(t,t')=fail}{< unif\ t\ t',\sigma,\alpha,\Gamma >\rightarrow< failwith(Unif),\sigma,\alpha,\Gamma >}$$

USusp $$\frac{< t,\sigma,\alpha,\emptyset > \text{ and } < t',\sigma,\alpha,\emptyset > \text{ are in } \rightarrow \text{ normal form}\quad unif_\sigma(t,t')=susp(u(i)),\quad c_s=\text{n}}{< unif\ t\ t',\sigma,\alpha,\Gamma >\rightarrow < u(-n),\sigma,(u(-n),unif\ t\ t')::\alpha,\Gamma > \text{ and } c_s\leftarrow(n+1)}$$

Aw $$\frac{u(i)\in dom(\Gamma) \text{ and } \Gamma(u(i))=t\quad < t,\sigma,\alpha,\emptyset >\rightarrow< t',\sigma',\alpha',\emptyset >\quad \text{and } < t',\sigma',\alpha',\emptyset > \text{ not in normal form}}{< t_0,\sigma,\alpha,\Gamma >\rightarrow< t_0,\sigma',\alpha',\Gamma[u(i)\leftarrow t'] >}$$

AwUpd $$\frac{u(i)\in dom(\Gamma) \text{ and } \Gamma(u(i))=t\quad < t,\sigma,\alpha,\emptyset >\rightarrow< t',\sigma',\alpha',\Gamma'' >\quad \text{and } < t',\sigma',\alpha',\emptyset > \text{ is in normal form}\quad \Gamma'=queue_{\sigma,\alpha}(u(j))}{< t_0,\sigma,\alpha,\Gamma >\rightarrow < t_0,(u(j),t')::\sigma',\alpha'\backslash\Gamma',\Gamma''\cup\Gamma'\cup\Gamma\backslash\{(u(j),t)\} >}$$

AwFail $$\frac{u(i)\in dom(\Gamma) \text{ and } \Gamma(u(i))=t\quad < t,\sigma,\alpha,\emptyset >\rightarrow< failwith(s),\sigma,\alpha,\emptyset >}{< t_0,\sigma,\alpha,\Gamma >\rightarrow< failwith(s),\sigma,\alpha,\Gamma >}$$

Let $< t_1, \sigma_1, \alpha_1, \Gamma_1 >$ be reduced by β applied on a subterm of t_1. Let note that subterm
$(fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e\ v$. By the hypothesis of $\equiv$ we have $(\sigma_2 \circ \alpha_2 \circ \Gamma_2)^*(t_2) = t_1$, thus the corresponding subterm of t_2 is of one of the following forms: $u(i)$; $u(i)\ u(j)$; $(fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e\ w$. We examine the first two forms:
(1): $u(i)$. First as σ_2 binds variable with values, we have $\sigma_2^*(u(i)) = u(j)$ and $u(j) \notin dom(\sigma_2)$. The $\equiv$ hypothesis ensures that $u(j) \notin dom(\alpha_2)$ as in that case the application would be suspended when the rule β applies on t_1. Thus we have: $\sigma_2^*(\Gamma_2(u(j))) = (fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e\ v$. The $\equiv$ hypothesis ensures that the same pattern matchs in both reduction and then application of **Aw** with the rule β on that term clearly leads to an equivalent four_uple.
(2) $u(i)\ u(j)$. The fact that bindings in α_2 and Γ_2 are bindings of logical variable to non value terms ensure that $\sigma_2^*(u(i)) = (fun\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n).e$ and $\sigma_2^*(u(j)) = v$; then β applies on $u(i)\ u(j)$ and leads to an equivalent four_uple.◇ We have now the result of strong confluence of $\rightarrow$ up to $\equiv$,

Theorem 5 *For all* $< t, \sigma, \alpha, \Gamma >$ *such that:*
$< t, \sigma, \alpha, \Gamma > \rightarrow < t_1, \sigma_1, \alpha_1, \Gamma_1 >$
$< t, \sigma, \alpha, \Gamma > \rightarrow < t_2, \sigma_2, \alpha_2, \Gamma_2 >$
There exists $< t'_1, \sigma'_1, \alpha'_1, \Gamma'_1 >$ *and* $< t'_2, \sigma'_2, \alpha'_2, \Gamma'_2 >$ *such that*
$< t_1, \sigma_1, \alpha_1, \Gamma_1 > \overset{0,1}{\rightarrow} < t'_1, \sigma'_1, \alpha'_1, \Gamma'_1 >$
$< t_2, \sigma_2, \alpha_2, \Gamma_2 > \overset{0,1}{\rightarrow} < t'_2, \sigma'_2, \alpha'_2, \Gamma'_2 >$
$< t'_1, \sigma'_1, \alpha'_1, \Gamma'_1 > \equiv < t'_2, \sigma'_2, \alpha'_2, \Gamma'_2 >$

Proof: it is illustrated in figure 3. The cases where at least one reduction use a suspension rule are: if both r_1 and r_2 use suspension rules, then the lemma 3 is enough to conclude. If one r_i use a suspension rule, then we conclude with the proposition 3 and the lemma 3.◇

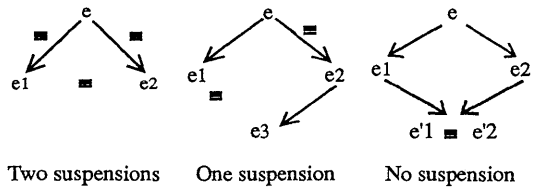


Figure 3: Strong confluence

Proof of the theorem: We show that the diagram of figure 4 holds with the theorem above and by successive inductions on lengths of d_1 and d_2.◇

Remark that the limitation to a strict calculus is necessary. If we permit reducing application without reducing the argument, as some unification may occur in that argument different normal forms are possible. Example:

$< (fun\ (x, y) \rightarrow unif\ x\ True).[](u(1), unif\ u(1)\ False), \emptyset, \emptyset, \emptyset >$

has two normal forms:

$$< void, \{(u(1), True)\}, \emptyset, \emptyset >$$

and $< failwith(Unif), \{(u(1), False)\}, \emptyset, \emptyset >$.

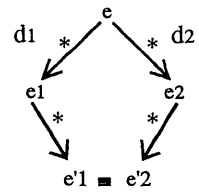


Figure 4: Church Rosser property

References

[AbadiCaCuLe 90] M.Abadi, L Cardelli, P.L. Curien, J.J. Levy, "Explicit substitutions", Proc. Symp. POPL 1990

[Ait Kaci 89] H.Ait-Kaci, R. Nasr, "Integrating Logic and Functional Programming", Lisp and Symbolic Computation,2,51-89 (1989)

[DeGrootLindstrom 86] D DeGroot G Lindstrom (eds),"Logic Programming - Functions, Relations and Equations", Prentice-Hall, New-Jersey, 1986.

[Dijkstra 76] E.W. Dijkstra, " A Discipline of Programming", Prentice Hall, New Jersey, 1976.

[HardinLevy 90] T. Hardin, J.J. Levy, "A Confluent Calculus of Substitutions", third symposium (IZU) on I.A.

[Huet 76] G.Huet, "Résolution d'équations dans les langages d'ordre 1,2,...,ω" Thèse d'état de l'Univ. de Paris 7, 1976

[Huet 88] G. Huet, "Experiments with GHC prototypes", may 1988, unpublished

[Laville 88] Alain Laville, "Implementation of Lazy Pattern Matching Algorithms", ESOP'88,LNCS 300

[Leroy 90] X.Leroy, "The ZINC experiment: an economical implementation of the ML language", INRIA technical report 117, 1990.

[LeroyWeis 91] X.Leroy P.Weis, "Polymorphic type inference and assignment", "Principles of Programming Languages", 1991.

[Poirriez 91] V.Poirriez, "Intégration de fonctionnalités logiques dans un langage fonctionnel fortement typé: MLOG une extension de ML" Thèse, Univ. Paris 7, 1991

[Poirriez 92a] Vincent Poirriez, "FLUO: an implementation of MLOG", Fifth Nordic Workshop on Programming Languages in Tampere, 1992

[SatoSakurai 86] M.Sato et T.Sakurai "QUTE: a functional Language Based on Unification". In [DeGrootLindstrom 86] pp 131-155.

[PuelSuarez 90] A. Suarez and L.Puel, "Compiling pattern matching by term decomposition", LFP'90

[Ueda 86] K. Ueda, " Guarded Horn Clauses", Ph.D.Thesis, Information Engineering Course, Univ. of Tokyo, 1986.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A New Perspective on Integrating Functional and Logic Languages

John Darlington Yi-ke Guo Helen Pull
Department of Computing
Imperial College, University of London
180 Queen's Gate London SW7 2BZ U.K.
E-mail: jd, yg, hmp@doc.ic.ac.uk

February 1992

Abstract

Traditionally the integration of functional and logic languages is performed by attempting to integrate their semantic logics in some way. Many languages have been developed by taking this approach, but none manages to exploit fully the programming features of both functional and logic languages and provide a smooth integration of the two paradigms. We propose that improved integrated systems can be constructed by taking a broader view of the underlying semantics of logic programming. A novel integrated language paradigm, Definitional Constraint Programming (DCP), is proposed. DCP generalises constraint logic programming by admitting user-defined functions via a purely functional subsystem and enhances it with the power to solve constraints over functional programs. This constraint approach to integration results in a homogeneous unified system in which functional and logic programming features are combined naturally.

1 Introduction

During the past ten years the integration of functional and logic programming languages has attracted much research. An extensive survey and classification of their results can be found in [GLDD90]. Traditionally this integration is performed by attempting to integrate the respective semantic logics of functional and logic languages in some way, resulting in a "super logic language". The conventional understanding is that a logic program defines a logical theory and computation is attempting to prove that a query is a logical consequence of this theory. Taking this view, integration is regarded as enhancing the original logic to cope with functional programming features and results in a new logic programming system. In section 2 we survey the main results of this approach. It seems to us that this approach fails to deliver all the features of both functional and logic programming. The main source of inadequacy appears to stem from the respective "intended semantics" assumed for logic and functional languages. It is this intended semantics which we question, motivating our search for a new way of approaching the problem of integrating functional and logic languages.

We show in later sections that if we regard functional programming as defining a higher-order value space, we can extend the conventional constraint logic programming (CLP) framework by using a functional programming language to define the domain over which relations are defined. Thus we combine functional programming with a general CLP framework rather than with the conventional Prolog-like system. We call the resulting language paradigm **Definitional Constraint Programming (DCP)**. We claim that DCP provides a uniform and elegant integration of functional, constraint and logic programming, while preserving faithfully the essence of each of these language paradigms.

In section 3, constraint systems and constraint programming are investigated at a very general level. A constraint logic programming model is then presented in section 4 as a particular constraint programming paradigm. Section 5 presents constraint functional programming (CFP) as a framework which superimposes a solving capability on the functional programming paradigm. The definitional constraint programming paradigm is developed in section 6. We discuss future work in section 7 and make some concluding comments in section 8.

2 Background and Motivation

From the traditional view of logic programming, integrating functional and logic languages is viewed as enhancing the original logic to cope with functional programming features. Most approaches take first-order equational logic as the semantic logic of functional languages and combine it with Horn clause logic. A comprehensive presentation of the theory of Horn clause logic with equality may be found in [GM87] and [Yuk88]. This shows that for every theory in Horn clause logic with equality, its initial model (called the least Herbrand E-model in [Yuk88], and the least Herbrand model in [Sny90]) always exists. Crucially, the initial model is the intended model of a logic programming system, since, according to the Herbrand theorem, the model is complete with respect to solving a query. For a Horn clause with equality program Γ and a query $\exists x_1, \ldots, x_n A_1, \ldots, A_n$ where A_i is an atom or equation, a computational model must verify $\Gamma \models \exists x_1, \ldots, x_n A_1, \ldots, A_n$ by computing an answer substitution θ such that $\Gamma \models \forall(\theta A_1 \wedge \ldots \wedge \theta A_n)$. Such models integrate SLD-resolution with some form of equational deduction such as paramodulation. A complete computational model was proposed recently by Snyder et. al. [Sny90] as a goal directed inference system. Systems which aim to sup-

port the full power of Horn clause logic with equality include Eqlog [GM84], which exploits fully the order-sorted variation of the logic, SLOG [Fri85] in which a completion procedure is used as the computational model, and Yukawa's system [Yuk88] which uses an explicit axiomatization of equality.

The computational difficulties of constructing a practical programming language based on the full Horn clause logic with equality leads us to conclude that this approach is not appropriate. Alternative languages overcome these problems by imposing syntactic and semantic restrictions on the paradigm. They all aim either to restrict the use of, or to weaken, defined equality. An example of the first approach is Jaffer and Lassez's **Logic Programming Scheme** [JL86], in which the equality part of a program is defined separately from predicate definitions. A program uses a first-order equational sublanguage to define abstract data types over which a definite clause subprogram is imposed. Operational models are based on SLD-resolution together with an E-unification procedure which solves equations over the equality defined by the equational subprogram.

Another way to restrict the computational explosiveness of general equational deduction is to use equational clauses as directed rewrite rules. A full discussion may be found in [DO88]. Narrowing [Hul80] (resp. conditional narrowing [DO88]) is employed to solve equations in a rewriting system (resp. conditional rewriting system). Many languages have been developed along this line, e.g. RITE [DP86b], K-Leaf [EGP86]. They represent enhanced Prolog systems in which a "rewrite" relation is defined over the Herband space. Syntactic restrictions guarantee the confluency of this rewrite relation so that equational logic can mimic first order functional programming. In the case of K-Leaf, the Herbrand space is enhanced to include partial terms, thus the lazy evaluation of functional languages may be modeled.

These endeavours have led to the development of several very successful languages and have significantly enriched the state of the art of declarative language design, semantics and implementation. However, we believe that the benefits of this combination are arguable and question how much is gained by enhancing a first-order logic by weakening a higher-order logic. Moreover, even with only first-order equational logic added, the inefficiencies of equational deduction mean that the resulting system is far from practical. This approach to language integration results in a sophisticated theorem prover, which we find unsatisfactory. We suggest, therefore, some fundamental rethinking on the purpose of integrating functional and logic languages.

In fact, the conventional assumption that a logic program defines a logical theory has been criticized in many circumstances because: "there is no reference to the models that the theory is a linguistic device for" [Mes89]. A logical theory may have many models, however when we are programming we always have a particular intended model in mind. This alternative school of thought regards a program as a linguistic description of the intended model; but the model itself is primary. For a Horn clause program, its least Herbrand model is taken as the intended model. Therefore, if a program is regarded as a linguistic description of this model, the canonical denotation of a program is not a first-order theory but a set of relations over the Herbrand space. This view of logic programming has also been taken by researchers wishing to extend Prolog-like systems. Hagiya and Sakurai [MT84] present a formal system for logic programming based on the theory of iterative inductive definitions. A similar approach is taken by Hallnas and Schroeder-Heister to develop the framework of **General Horn Clause Programming** [AEHK89]. Paulson and Smith proposed an integrated system in which a logic subprogram is regarded as an inductive definition of relations [PS89].

This definitional view of logic programming suggests the flexibility to define Horn clauses over arbitrary domains. Relations become constraints over the domain of discourse, which coincides with the general framework of **Constraint Logic Programming** [Smo89]. In this paper, we take this idea one step further by using a functional programming language to define the domain over which relations are defined. A novel definitional constraint programming system is induced in which functions and relations are used together to define constraint systems.

3 Constraint Programming

In this section, we present a framework for constraint programming which has it origins in the seminal work of Steele [Ste80]. From the mathematical point of view, constraints are associated with well-studied domains in which some privileged predicates, such as equality and various forms of inequalities, are available. Relations formed by applying these predicates are regarded as constraints. A constraint may be regarded as a statement of properties of objects; its denotation is the set of objects which satisfy these properties. Therefore, constraints provide a succinct finite representation of possibly infinite sets of objects. We present a simple definition of constraint systems to capture these characteristics.

3.1 Constraint System

Definition 3.1 (Constraint System) *A constraint system is a tuple* $< \mathcal{A}, V, \Phi, I >$ *where*

- $\mathcal{A}$ *is a set of values called the* **domain** *of the system.*
- V *is a set of* **variables**.
- Φ *is a set of* **constraints**.
  We define an $\mathcal{A}$-valuation as a mapping $V \to \mathcal{A}$, and $\mathcal{V}\mathcal{A}$ as the set of all $\mathcal{A}$-valuations. A computable function $\mathcal{V}$ is used to assign to every constraint ϕ a finite set $\mathcal{V}(\phi)$ of variables, which are the variables constrained by ϕ. $Val_{\mathcal{A}}$ denotes the set of all $\mathcal{A}$-valuations.
- *I is an interpretation which consists of a solution mapping $[\![]\!]^I$, mapping every basic constraint ϕ to $[\![\phi]\!]^I$, a set of $\mathcal{A}$-valuations called the* **solutions** *of ϕ and $[\![]\!]^I$ is* **solution closed** *in the sense that*

$$\forall \alpha \in [\![\phi]\!]^I \; \forall x \in \mathcal{V}(\phi) \; \beta(x) = \alpha(x) \Rightarrow \beta \in [\![\phi]\!]^I$$

We now present some examples of constraint systems. The most familiar constraint system in the context of program-

ming languages is perhaps the Herbrand system which is a constraint system over finite labelled trees.

Example 3.1.1 (Herbrand System) *Let Σ be a set of ranked signatures of function symbols and V be a set of constant symbols treated as variables. $T(\Sigma)$ is the ground term algebra consisting of the smallest set of inductively generated Σ-terms. A Herbrand system is a constraint system $< T(\Sigma), V, \Phi, I >$ where Φ consists of all term equations of the form $t_1 = t_2$ for $t_1, t_2 \in T(\Sigma, V)$, where $T(\Sigma, V)$ is the free term algebra, and $[\![t_1 = t_2]\!]^I = \{\alpha \mid \alpha t_1 \equiv \alpha t_2\}$ where $\equiv$ denotes the identity of two terms.*

Example 3.1.2 (Herbrand E-System) *Let Σ, V be as above and E an equational theory over $T(\Sigma, V)$. Then $T(\Sigma)/E$ denotes the quotient term algebra consisting of the finest Σ-congruences over $T(\Sigma)$ generated by E. The constraint system $< T(\Sigma)/E, V, \Phi, I >$ is called the Herbrand E-System where Φ consists of all term equations of the form $t_1 = t_2$ for $t_1, t_2 \in T(\Sigma, V)$ and $[\![t_1 = t_2]\!]^I = \{[\alpha]_E \mid [\alpha t_1]_E = [\alpha t_2]_E\}$, where $[t]_E$ stands for the equivalence class of t in $T(\Sigma)$ and $[\alpha]_E : V \to T(\Sigma)/E$ stands for the corresponding equivalence class of ground term substitutions $\alpha : V \to T(\Sigma)$.*

Constraint systems on various term structures can be regarded as cases of the following general definition of an algebraic constraint system.

Example 3.1.3 (Algebraic Constraint System) *Let A be an algebra equipped with a set of operators Σ and a set of predicates Π. Then the algebra is associated with a constraint system $S_A : <| A |, V, \Phi, I >$ where $| A |$ is the carrier of the algebra and evey constraint in Φ is of the form $p(e_1, \ldots, e_n)$ where every e_i is an A-expression and $p \in \Pi$ is an n-ary predicate in the algebra. $[\![p(e_1, \ldots, e_n)]\!]^I = \{\alpha \mid A, \alpha \models p(e_1, \ldots, e_n)\}$. Examples of algebraic constraints are constraints over term algebras, constraints over arithmetic expressions and constraint systems in boolean algebra.*

Following the idea of associating constraint systems with algebras, predicate logic can be viewed from the constraint system perspective.

Example 3.1.4 (Predicate Logic) *Suppose Σ is the first order signature of symbols, the well-formed Σ-formula are constraints, $\mathcal{V}(\phi) \subseteq V$ are free variables in ϕ with V the set of free (unquantified) variables. $\mathcal{A}$ is given by a Σ- structure (algebra) over which symbols are interpreted. With respect to a particular interpretation, I can be given as $[\![\phi]\!]^I = \{\alpha \mid \mathcal{A}, \alpha \models \phi\}$.*

For any constraint system, the solution of a constraint c can be restricted to a set of variables in c. Given a finite set of variables $W \subseteq_f V$, a valuation with respect to W becomes a partial mapping defined as follows :

$$\alpha_{|W}(x) = \begin{cases} \alpha(x) & x \in W \\ \bot & \text{otherwise} \end{cases}$$

The solutions of a constraint ϕ with respect to W are defined as:

$$[\![\phi]\!]^I_{|W} := \{\alpha_{|W} \mid \alpha \in [\![\phi]\!]^I\}$$

A constraint ϕ is **consistent** in a constraint system iff $[\![\phi]\!]^I \neq \emptyset$. A consistent constraint $\phi \in \Phi$ is **valid** iff $[\![\phi]\!]^I = \mathcal{VA}$. We use the word `true` to denote a valid constraint and `false` to denote an inconsistent constraint. Given a set of constraints, the set inclusion relation of solutions introduces a preorder over constraints that reflects the richness of the information they possess. A constraint ϕ_1 is a W-refinement of a constraint ϕ_2, $\phi_2 \leq_{|W} \phi_1$, iff $[\![\phi_1]\!]^I_{|W} \subseteq [\![\phi_2]\!]^I_{|W}$. ϕ_1 is a refinement of ϕ_2, $\phi_2 \leq \phi_1$, iff $[\![\phi_1]\!]^I \subseteq [\![\phi_2]\!]^I$. The preorder introduces an equivalence relation between constraints. A constraint ϕ is equivalent to a constraint ϕ', denoted $\phi \equiv \phi'$ iff $[\![\phi_1]\!]^I = [\![\phi_2]\!]^I$.

We consider some fundamental operations over constraints.

Definition 3.2 *Let ϕ_1, ϕ_2 be two constraints. Then their conjunction, $\phi_1 \wedge \phi_2$, is a constraint with $\mathcal{V}(\phi_1 \wedge \phi_2) = \mathcal{V}(\phi_1) \cup \mathcal{V}(\phi_2)$ and $[\![\phi_1 \wedge \phi_2]\!]^{\mathcal{I}} = [\![\phi_1]\!]^{\mathcal{I}} \cap [\![\phi_2]\!]^{\mathcal{I}}$; the constraint implication, $\phi_1 \to \phi_2$, is a constraint with $\mathcal{V}(\phi_1 \to \phi_2) = \mathcal{V}(\phi_1) \cup \mathcal{V}(\phi_2)$ and $[\![\phi_1 \to \phi_2]\!]^{\mathcal{I}} = \{\mathcal{VA} - [\![\phi_1]\!]^{\mathcal{I}}\} \cup [\![\phi_2]\!]^{\mathcal{I}}$.*

The definition of binary constraint conjunction can be extended to the conjunction of a set of constraints. A finite set of constraints is called a **goal** when it is interpreted as the conjunction of all its element constraints. A constraint implication $\phi_1 \to \phi_2$ is always valid whenever $[\![\phi_1]\!]^{\mathcal{I}} \subseteq [\![\phi_2]\!]^{\mathcal{I}}$ in which case we say that ϕ_1 entails ϕ_2, denoted $\phi_1 \vdash \phi_2$. It is obvious that $\phi_1 \vdash \phi_2 \iff \phi_2 \leq \phi_1$.

Definition 3.3 *If ϕ is a constraint and $x \in \mathcal{V}(\phi)$, then the existential quantification, $\exists x.\phi$, is a constraint with $\mathcal{V}(\exists x.\phi) = \mathcal{V}(\phi) - \{x\}$ and $[\![\exists x.\phi]\!]^{\mathcal{I}} = \{\alpha \in \mathcal{VA} \mid \exists \beta \in [\![\phi]\!]^{\mathcal{I}}, \alpha_{\mathcal{V}(\phi)-x} = \beta_{\mathcal{V}(\phi)-x}\}$; the negation of ϕ, $\neg\phi$, is a constraint with $\mathcal{V}\neg\phi = \mathcal{V}\phi$ and $[\![\neg\phi]\!]^I = \mathcal{VA} - [\![\phi]\!]^I$.*

Existential quantification provides a means of hiding, by projecting away, information about quantified variables. A constraint system is said to be closed with respect to an operator iff the constraint obtained by applying the operator is always in the system.

3.2 Constraint Solving

The computational task of a constraint system is to solve constraints. This is a constructive procedure which not only verifies that the solution set is non-empty, but transforms it to an equivalent, more informative form, from which solutions are easily derived. Such a form is called a **solved form**. As suggested by Smolka in [Smo91], constraint solving can be modeled by a rewriting system which simplifies a constraint to its equivalent solved form. Since, in the programming context, we are interested in solving goals, rewriting is applied to a set (more precisely, multiset) of constraints. Therefore, to express constraint solving in terms of rewriting we use multiset transformation systems.

Definition 3.4 (Constraint Solver) *A constraint simplification rule is a multiset transformation rule $G \to G'$ where G, G' are multisets of constraints (goals) such that*

$[\![G']\!]^I \subseteq [\![G]\!]^I$. *A constraint solver C is a multiset transformation system containing a set of constraint simplification rules which is* **solution preserving**, *i.e. for n simplification rules of the form* $G \rightarrow G'_i$ *with the same left hand side M (up to renaming), we have :*

$$[\![G]\!]^I = \bigcup_{i=1}^{n} [\![G'_i]\!]^I$$

We call the relation $\xrightarrow{c}$ *a one step simplification and* $\xrightarrow{c}{}^*$ *a simplification derivation. A solved form of a goal G is a goal G' such that G' is a normal form with respect to the constraint solver.*

The set $\mathcal{SF}_G$ of all solved forms of a goal G is complete iff

$$\forall \alpha \in [\![G]\!]^I . \exists G' \in \mathcal{SF}_G \text{ such that } \alpha \in [\![G']\!]^I$$

For a one step simplification $M \rightarrow M'$, it is obvious that simplification is sound.

Lemma 3.4.1 (Soundness of Simplification) *For one step simplification* $G \xrightarrow{c} G', [\![G']\!]^I \subseteq [\![G]\!]^I$.

The following proposition is also straightforward.

Lemma 3.4.2 *For any goal G which is not in normal form and* $\alpha \in [\![G]\!]^I$, *there exists a one step simplification* $G \xrightarrow{c} G'$ *such that* $\alpha \in [\![G']\!]^I$.

To model precisely the idea of simplification and its completeness, the familiar methodology of term rewriting systems is adapted. We require a complexity measure of a goal G with respect to a solution α, $|(G, \alpha)|$. We say that a constraint solver is **well-founded** iff

$$\forall \alpha \in [\![G']\!]^I \; G \xrightarrow{c}{}^* G' \Rightarrow |(G, \alpha)| \geq |(G', \alpha)|$$

For any well-founded solver, a solved form is always reachable for a consistent goal G for a particular solution α. Therefore, it is always possible to enumerate a simplification derivation such that $G \xrightarrow{c}{}^* G'$ with G' in solved form and $\alpha \in [\![G']\!]^I$.

Lemma 3.4.3 (Completeness of Simplification) *If a constraint solver C is well-founded, then for every consistent goal G, there is a set,* $\mathcal{SF}_G$, *which is the complete set of solved forms of G.*

The well-foundness of a constraint solver does not suggest that the complete set of solved forms for a goal is finite. It would be helpful to consider only finite sets of solved forms. For this we need the notion of compactness of constraint systems.

Definition 3.5 (Compactness) *A constraint system is compact iff for every finite set of constraints G :*

$$G \vdash G_1 \iff \exists G_2 \subseteq_f G_1 \text{ such that } G \vdash G_2$$

For any compact constraint system, a stronger completeness condition holds for any well-founded solver.

Lemma 3.5.1 *Let S be a compact constraint system and C be a solver for S. If C is well-founded, for any goal G, G' and* $G \vdash G'$, *there are a finite number of derivations* $G' \xrightarrow{c}{}^* G'_i$ *such that* G'_i *is in solved form and* $G' \vdash \bigvee_{i=1}^{m} G'_i$.

This lemma shows that, in a compact constraint system, any information contained by a goal constraint can always be processed by a finite amount of computation.

A constraint solver is deterministic when the simplification system is confluent. A simplification rule is deterministic in the solver if no other rule in the system has the same left hand side. For a well-founded deterministic solver, a consistent goal has a unique solved form. Constraint solving by a non-deterministic solver can be regarded as a reduction procedure which simplifies a disjunction of goals by rewriting it into an equivalent one. A constraint solver is terminating iff there is no infinite simplification derivation $G \rightarrow G_1 \rightarrow \ldots$. A well-founded constraint solver is decidable iff any unsatisfiable constraint can be simplified to `false`. Thus, a complete constraint solver is a decision procedure for the satisfiability of constraints.

3.3 Constraint Programming

Constraint programming is a declarative programming paradigm in which the task of programming is to define a constraint system and the task of computation is to solve the constraints. Therefore, the declarative semantics of a constraint program is given by determining the domain of discourse and defining the denotation of each constraint as its solution set. Its operational semantics is given by the constraint solver of the system which can be presented as a rewriting system which must be sound with respect to the declarative semantics and preferably complete. A sufficient condition for completeness is well-foundness of the solver. This notion of constraint programming is a generalization of the approach of Steele [Ste80] and Lassez [LM89]. Here the constraint system is assumed to be "built-in" and therefore, "programming" simply means imposing constraints.

When designing a constraint programming language it is essential to develop a systematic way to define constraints and a generic way to construct a solver for each defined constraint system. We give two examples of this generalized definition of constraint programming: Horn Clause Logic Programming and Equational Logic Programming.

Example 3.5.1 (Horn Clause Logic Programming) *A Horn clause program* Γ *defines a constraint system* $< T(\Sigma), V, \Phi, I >$ *where* $T(\Sigma)$ *is the ground term algebra for the signature* Σ *of function symbols (Herbrand space) and* Φ *consists of all positive literals and is closed under renaming, conjunction and existential quantification. I interprets constraints (defined predicates) as relations in the least Herbrand model* M_Γ *of the program :*

$$[\![p(t_1, \ldots, t_n)]\!]^I = \{\alpha : V \rightarrow T(\Sigma) \mid M_\Gamma, \alpha \models p(\alpha t_1, \ldots, \alpha t_n)\}$$

SLD resolution is a well-founded constraint solver which simplifies each consistent goal G into a disjunction of idempotent substitution equations: $\exists X . \bigvee_i S_i$.

This view of Horn clause logic programming is consistent with its traditional presentation. The major divergence is that we take the definitional view of logic programs. In another words, it is a linguistic specification of the intended model of the program, the least Herbrand model. This divergence results in some subtle differences in the properties

of programs. For example, the completeness condition of constraint solving may not hold for all models of a program. Therefore it is not true that $\Gamma \models G \iff \bigvee_i S_i$ in the above example. To get this result a completion procedure must be applied to programs.

Example 3.5.2 (Equational Logic Programming) *An equational program E is a constraint system for solving equations in the quotient term algebra which it defines. A general E-unification procedure is its constraint solver. Following Gallier and Synder's result [Sny90], such a procedure exists and can be represented as multiset transformation system. Moreover, it is also well-founded. Therefore, it is a complete solver which simplifies an equational goal to a (possibly infinite) set of idempotent substitution equations.*

These two logic programming systems show two different ways to construct constraint systems in terms of logical formulas. A constraint system may be defined by a Horn clause logic program using recursive definition rules to define constraints over a fixed underlying domain. By contrast, in the case of equational logic programming, the form of constraints is fixed as equations over terms. An equational logic program forms a Herbrand E-system by defining the domain of discourse along with the interpretation of constraints (i.e. the equality in the domain). The former approach may be seen as a "relational extension" of a basic constraint system comprising the predefined fixed domain of discourse together with some "built in" constraints. In section 4, a systematic framework is constructed for such an extension. On the other hand, an equational logic program can be understood as defining an abstract data type with equations as constraints. This method of defining constraint systems may be called "domain construction" for some fixed constraint relation. In section 5, we propose a way to use functional programs to define the domain of discourse for solving constraints.

These two approaches may be combined to form a powerful constraint programming system in which both the domain of discourse and constraint relations are user-definable. The logic programming scheme [JL87], in which a program is regarded as a relational extension of the Herbrand E-constraint system defined by the equational subprogram, takes this route, although this was not the original semantics of the scheme. We believe that the constraint programming perspective provides a simpler and more intuitive semantic treatment of the scheme. Moreover, if we instantiate the underlying constraint system of the general CLP framework by constraint systems constructed over functional programs we have a powerful, general purpose, definitional constraint programming model which unifies functional and logic programming. This is the main result derived from our generalized view of constraint programming.

4 Constraint Logic Programming

As mentioned in the previous section, from the definitional view of logic programming, a constraint system can easily be integrated into a logic programming system. The resulting **constraint logic programming** system is a definitional logic system which allows a predefined underlying constraint system to be extended by defining relations as new constraints. This formalism, which was first proposed by Höhfeld and Smolka in [Smo89], is more general than the proposal of Jaffer and Lassez [JL87], in which CLP is modelled within the traditional logic programming school. Many restrictions that are imposed on the underlying constraint system in the latter approach, such as the requirement that it is effectively axiomatized in first-order theory, are unnecessary within the definitional framework.

Let $C :< \mathcal{A}, \mathcal{V}, \Phi_c, I_c >$ be a constraint system closed under conjunction, renaming and existential quantification. Given a signature $\mathcal{R}$ as a family of user-defined predicates indexed by their arities, a constraint logic program Γ over C is a set of **constrained defining rules** of the form :

$$\mathcal{P} \leftarrow c_1, \ldots, c_j, B_1, \ldots, B_m$$

$\mathcal{P}$ is an $\mathcal{R}$-atom of form $p(x_1, \ldots, x_n)$ where $p \in \mathcal{R}_n$ is an n-ary user-defined predicate, $c_i \in \Phi_c$ and the B_i are atoms.

An interpretation I of Γ over C is defined by interpreting each predicate symbol $p \in \mathcal{R}$ as a relation p^I over $\mathcal{A}$. An ordering over interpretations is defined by the set inclusion of the relations. That is, for any interpretations I, I', $I \leq I'$ iff $p^I \subseteq p^{I'} \subseteq A^n$ for any n-ary predicate $p \in \mathcal{R}_n$. Thus, the set of all interpretations forms a complete lattice. With respect to a given interpretation, I, any $\mathcal{R}$-atom, $p(x_1, \ldots, x_n)$, can be regarded as a constraint whose solutions are given by :

$$[\![p(x_1, \ldots, x_n)]\!]^I = \{\alpha \mid (\alpha(x_1), \ldots, \alpha(x_n)) \in p^I\}$$

The solution set of a conjunction of atoms, a goal, is the intersection of the solutions of its elements. A defining rule $\mathcal{P} \leftarrow c_1, \ldots, c_j, B_1, \ldots, B_m$ is valid in an interpretation I iff $[\![p(x_1, \ldots, x_n)]\!]^I \supseteq \bigcap_{i=1}^{j} [\![c_i]\!]^{I_c} \cap \bigcap_{i=1}^{m} [\![B_i]\!]^I$

A model of Γ over C is an interpretation in which all rules of the program are valid. The set of all models of a CLP program is closed under intersection and union, therefore, the minimal model exists. We take the minimal model as the intended model of a program. This may be constructed by the standard iteration procedure which computes the inductive closure of relations generated by the clauses in Γ, given as:

$$\begin{aligned} I_0 &= \emptyset \\ I_{n+1} &= \bigcup_{p \in \Pi} p^{I_{n+1}} \end{aligned}$$

where $p^{I_{n+1}}$ defines the denotation of predicate p at the $n+1$th iteration step :

$$p^{I_{n+1}} = \{(\alpha(x_1), \ldots, \alpha(x_n)) \mid \alpha \in \bigcap_{i=1}^{j} [\![c_i]\!]^{I_c} \cap \bigcap_{i=1}^{m} [\![B_i]\!]^{I_n}\}$$

for all $p(x_1, \ldots, x_n) :- c_1, \ldots, c_j, B_1, \ldots, B_m \in \Gamma$. The sequence $I_0, I_1, \ldots, I_k \ldots$ is a chain in the interpretation lattice. The limit of the chain, $I_\Gamma = \cup_{n=0}^{\infty} I_n$, is the minimal model of the program Γ and can be computed as the least fixed point of the iteration procedure. This is presented as the following theorem:

Theorem 4.0.1 *Let $C :< \mathcal{A}, \mathcal{V}, \Phi_C, I_C >$ be a constraint system and Γ a constraint logic program over C. The sequence of interpretations I_n represents a chain in the complete lattice of interpretations of Γ. The limit of the chain is the minimal model of Γ over C.*

In this least model semantics of a CLP program the underlying constraint system is extended to a new constraint system via user-defined constraints. We call this a **relational extension** of a constraint system.

Definition 4.1 (Relational Extension) *Let Γ be a constraint logic program and $\mathcal{R}$ be the signature of user-defined predicates in Γ. Γ constructs the constraint system :*

$$\mathcal{R}(C) :< \mathcal{A}, \mathcal{V}, \Phi_c^{\mathcal{R}}, I_c^{\mathcal{R}} >$$

as a **relational extension** *of the underlying constraint system $C :< \mathcal{A}, \mathcal{V}, \Phi_c, I_c >$ by extending Φ_c to accommodate user-defined relations over $\mathcal{A}$. That is, $\Phi_c^{\mathcal{R}} = \Phi_c \cup \Phi_{\mathcal{R}}$ where $\Phi_{\mathcal{R}}$ contains all $\mathcal{R}$-atoms and*

$$\begin{aligned} [\![\phi]\!]^{I_c^{\mathcal{R}}} &= [\![\phi]\!]^{I_c} \\ [\![p(x_1, \ldots, x_n)]\!]^{I_c^{\mathcal{R}}} &= \{\alpha \mid (\alpha(x_1), \ldots, \alpha(x_n)) \in p^{I_\Gamma}\} \end{aligned}$$

where I_Γ is the minimal model of Γ over C.

A solver for a relationally extended constraint system can be constructed by integrating SLD-resolution with the constraint solver of the underlying constraint system to give **constrained SLD-resolution**. Constrained SLD-resolution rewrites a goal of the form $G = G_c \cup G_{\mathcal{R}}$, where $G_{\mathcal{R}}$ is a finite subset of atoms in $\Phi_{\mathcal{R}}$ and G_c is a finite subset of Φ_c, to its solved forms. This model can be represented by the following multiset transformation rules.

Semantic Resolution:
$\frac{G:\exists X < G_{\mathcal{R}} \cup \{p(s_1,\ldots,s_n)\} \cup G_c >}{G':\exists X \cup Y < G_{\mathcal{R}} \cup \{B_1,\ldots,B_m\} \cup G_c \cup \{c_1,\ldots,c_k\} \cup \{x_1=s_1,\ldots,x_n=s_n\} >}$
where $\forall Y.p(x_1 \ldots x_n) :- c_1, \ldots, c_k, B_1, \ldots, B_m$ is a variant of a clause in a program Γ.

Constraint Simplification: $\frac{G:\exists X < G_{\mathcal{R}} \cup G_c >}{G':\exists X < G_{\mathcal{R}} \cup G_c' >}$
if $\exists X.G_c \longrightarrow_c \exists X.G_c'$ and $\longrightarrow_c$ is the simplification derivation realised by the solver in the underlying system.

Finite Failure: $\frac{G:\exists X G_{\mathcal{R}} \cup G_c}{\texttt{false}}$
if $\exists X.G_c \longrightarrow_c \texttt{false}$.

In this model, semantic resolution generates a new set of constraints whenever a particular program rule is applied. The unification component of SLD-resolution is replaced by solving a set of constraints via the underlying solver. Whenever it can be established that the set of constraints is unsolvable, finite failure results.

For example, the following CLP program [Col87]:

InCap ([], 0)
InCap (i:x, c) :- *InCap* (x, 1.1*c - i)

can be used to compute a series of instalments which will repay capital borrowed at a 10% interest rate. The first rule states that there is no need to pay instalments to repay zero capital. The second rule states that the sequence of *N+1* instalments needed to repay capital c consists of an instalment i followed by the sequence of N instalments which repay the capital increased by 10% interest but reduced by the instalment i. When we use the program to compute the value of m required to repay *$1000* in the sequence *[m, 2m, 3m]*, we compute the solved form of the goal constraint: *InCap ([m, 2m, 3m], 1000)*. One execution sequence is illustrated below, in which $\rightarrow_R$ denotes a semantic resolution rewrite step:

InCap ([m, $2m$, $3m$], 1000)
$\rightarrow_R$ *InCap* (x,1.1c-i), x=[$2m$, $3m$], i=m, c=1000
$\rightarrow_R$ *InCap*(x',1.1c'-i'), x=i':x', c'=1.1c-i,
  x=[$2m$, $3m$], i=m, c=1000
$\rightarrow_c$ *InCap* (x',1.1c'-i'), i'=$2m$,
  x'=[$3m$],i=m, c'=1100-m
$\rightarrow_R$ *InCap*(x'',1.1c''-i''), x'=i'':x'',1.1c'- i'=c''
  , i'=$2m$, x'=[$3m$],i=m,c'=1100-m
$\rightarrow_c$ *InCap* (x'',1.1c''-i''), x''=[], i''=$3m$,i'=$2m$,
  i=m,x'=[$3m$], c'=1100-m, c''=1210-3.1m
$\rightarrow_R$ x''=[],1.1c''-i''=0, i''=$3m$, i'=$2m$, i=m,
  x'=$3m$, c'=1100-m, c''=1210-3.1m
$\rightarrow_c$ 1.1(1210-3.1m)=$3m$
$\rightarrow_c$ m=207+413/641

Constrained SLD-resolution is a sound solver for a relationally extended constraint system and, as proved in [Smo89], it is also well-founded. Therefore, any consistent goal can be simplified to a set of solved forms. Let $\Phi \downarrow\subseteq \Phi$ be the set of all solved forms for Φ. Then it is easy to show that $\Phi_c^{\mathcal{R}} \downarrow\subseteq \Phi_c \downarrow$. Therefore, from the completeness of the model, for any goal G with $\bigvee_{i\geq 1} G_c^i$ as its solved forms, given a goal $G_c' \subseteq_f \Phi_c$ containing only basic constraints, $G_c' \vdash G \Rightarrow G_c' \vdash \bigvee_{i\geq 1} G_c^i$. Moreover, if the underlying constraint system is compact, then $G_c' \vdash \bigvee_{i=1}^n G_c^i$ for some n, i.e. the model has the stronger completeness of section 3.2.

5 Constraint Functional Programming

Constraint functional programming (CFP) is characterized as functional programming, enhanced with the capability to solve constraints over the value space defined by a functional program. An intuitive construction of this language paradigm is presented below.

5.1 Informal CFP

A data type D in a functional program, Γ, can be associated with a constraint system C_D. C_D may contain privileged predicates over D. A CFP system may be formed to extend the constraint solver so that any D-valued expression, which may involve user-defined functions, can be admitted in constraints. A D-valued expression must be evaluated to its normal form with respect to Γ to enable the constraint solver to handle that value.

We give a simple example of this paradigm. We assume a constraint system over lists in which atomic constraints are equations asserting identity over finite lists. A unification algorithm is used as the basic solver for the system. Given a functional program defining the function *++* which concatenates two lists and the function *length* which computes the length of a list:

data $[alpha] = [] \mid alpha : [alpha]$
functions
$++ :: [alpha] \times [alpha] \rightarrow [alpha]$
$length :: [alpha] \rightarrow Num$
$[] ++ z = z$
$(x{:}y) ++ z = x : (y{+}{+}z)$
$length\ [] = 0$
$length\ (x{:}y) = 1 + length\ y$

An extension to the basic solver may be used to solve the constraint:

$$l1 ++ l2 = [a1, a2, \ldots, an],\ length\ l1 = 10$$

to compute the first 10 elements of a the list $[a1, a2, ..., an]$. The solver must apply the function definitions of $++$ and *length* and must guess appropriate instances of the constrained variables. We will show that this procedure itself may be modelled by some new constraints generated during rule application.

Solving constraints over a functional program significantly enhances the expressive power of functional programs to incorporate logic programming features. This idea was central to the **absolute set abstraction** construct which was originally proposed in [DAP86,DG89] as a means to invoke constraint solving and collect solutions. Using the absolute set abstraction notation, the above constraint may be represented as the set-valued expression:

$$\{\ l1 \mid l1 ++ l2 = [a1, a2, \ldots, an],\ length\ l1 = 10\ \}$$

Reddy's proposal of "Functional Logic Programming" languages [Red86] also exploits this solving capability in functional programs. However, his description of functional logic programming as functional syntax with logic operational semantics fails to capture the essential semantic characteristics of the paradigm. The constraint programming approach, as we will show in the following, presents a concise semantical and operational model for the paradigm.

We assume a functional language that is strongly typed, employs a polymorphic type system and algebraic data types, and supports higher-order functions and lazy evaluation. Examples of such languages are Miranda [Tur85] and Haskell [Com90]. To investigate constraint solving we put aside the statical features of a functional language such as its type system, and concentrate on its dynamic semantics. We use a kernel functional language with recursion equation syntax for defining functions. We assume variables ranged over by x and y, a special set of functional variables (identifiers) ranged over by f and g, constructors ranged over by d, constants ranged over by a and b, patterns ranged over by t and s and expressions ranged over by e. A pattern is assumed to be linear, i.e. having no repeated variables. Data terms comprise only constants, constructors and first-order variables. The following syntax defines this tiny functional language:

Program	::=	*Decl* **in** *Exp*
Decl	::=	$ft = e$
		$\mid$ *Decl*; *Decl*
Exp	::=	$x \mid a \mid e_1 e_2 \mid e_1\ op\ e_2$
		$\mid$ **if** e_1 **then** e_2 **else** e_3
Pattern	:=	$x \mid a \mid dt_1, \ldots, t_n$

The language can be regarded as sugared λ-calculus and a program as a λ-expression. The program shown above is an instance of this formalism in which the data statement introduces a list structure with a nullary constructor [] and a binary constructor :, and functions *length* and $++$ which are defined by recursion equations.

The semantics of a functional program is given in the standard way [Sco89]. The semantic domain D of the program is an algebraic CPO which is the minimal solution of the domain equation :

$$D = B_\perp + C(D) + D \rightarrow D$$

D contains the domain $B_\perp$ of basic types (real numbers, boolean values et. al. lifted by $\perp$ which denotes undefinedness), the domain $C(D)$ for constructed data structures which consists of partial terms ordered with respect to the monotonicity of constructors and the domain $D \rightarrow D$ of all continuous functions. A subdomain A of $C(D)$: $A = B_\perp + C(A)$ is distinguished as the domain of data terms in the language (which is defined by the eq-type of ML [Mil84]). We use $\mathcal{T}$ to denote all complete objects of A.

For a functional program, the semantic function $\mathcal{P}[\![]\!]$ computes the value of the program in terms of the function $\mathcal{D}[\![]\!] : Decl \rightarrow (Var \rightarrow D) \rightarrow (Var \rightarrow D)$ which maps function definitions to an environment which associates each function name with its denotation. The function $\mathcal{E}[\![]\!] : Exp \rightarrow (Var \rightarrow D) \rightarrow D$ maps an expression together with an environment $\eta : Var \rightarrow D$ (a D-valuation) to an element of D.

5.3 Evaluating Nonground Expressions

Conventional functional programming involves evaluating a ground expression to its unique normal form by taking a program as a rewriting system. To superimpose a solving capability on the functional programming paradigm, we consider first the extension of functional programming to handle non-ground expressions. The meaning of a non-ground expression is a set of values corresponding to every correctly typed instantiation of its free variables. Narrowing has been proposed as the operational model for computing all possible values of a nonground expressions [Red84]. In the theorem proving context, enumerating narrowing derivations provides a complete E-unification procedure for equational theories defined by convergent rewriting systems. This use of narrowing must be refined for the functional programming context. Due to the lazyness of functional languages, only those narrowing derivations whose corresponding reduction derivations are lazy should be enumerated. This notion of **lazy narrowing** is mentioned by Reddy in [Red84]. A lazy narrowing procedure, **pattern-driven narrowing**, is proposed by Darlington and Guo in [DG90] for evaluating absolute set abstractions. A similar procedure was indepen-

dently developed by You for constructor based equational programming systems [You88]. Here we present a lazy narrowing model following the constraint solving approach. The model is central to the CFP paradigm.

Consider reducing a non-ground expression of form fe by a defining rule $ft = e'$. The environment η should be enhanced to satisfy $\mathcal{E}[\![e]\!]\eta = \mathcal{E}[\![t]\!]\eta$, i.e. η is a solution of the **rewriting constraint** $e = t$. This equality is the so called semantic equality since it is determined by the identity of denotations of components. It is not even semidecidable since it involves verifying the equivalence of partial values. However, since in our problem t is always a linear pattern, a semidecidable solver exists.

Definition 5.1 *The solved form of a rewriting constraint $e = t$ is of the form $\{x_1 = t_1, \ldots, x_n = t_n, y_1 = e_1, \ldots, y_m = e_m\}$ where the $x_i \in \mathcal{V}(e)$ are output variables and the $y_i \in \mathcal{V}(t)$ are input variables. The equation set $\hat{\delta} : \{x_1 = t_1, \ldots, x_n = t_n\}$ is an output substitution equation and $\hat{\theta} : \{y_1 = e_1, \ldots, y_m = e_m\}$ is an input substitution equation. The substitutions δ and θ corresponding to $\hat{\delta}$ and $\hat{\theta}$ are called output substitutions and input substitutions respectively.*

The constraint solver presented below simplifies a rewriting constraint to its solved form. Solving a rewriting constraint realises the bidirectional parameter passing mechanism for narrowing an outmost function application. The algorithm is called **pattern-fitting** [DG89].

Substitution: $\{x = r\} \cup G \Rightarrow \{x = r\} \cup \rho G$
where $\rho = \{x \mapsto r\}$.

Decomposition: $\{de = dt\} \cup G \Rightarrow \{e = t\} \cup G$

Removing: $\{a = a\} \cup G \Rightarrow G$

Failure: $\{d_1 e_1 = d_2 e_2\} \cup G \Rightarrow$ `false`
if $d_1 \neq d_2$.

Constrained Narrowing: $\{fe = ds\} \cup G \Rightarrow \{r = ds, e = t\} \cup G$
where $ft = r \in \Gamma$

Lemma 5.1.1 *The pattern-fitting algorithm is a complete solver for simplifying a rewriting constraint to its solved form.*

For any rewriting constraint $e = t$, a solved form corresponds to a pattern-driven narrowing step $fe \leadsto_\delta \theta e'$ with respect to a defining rule $ft = e'$ where δ is the output substitution and θ is the input substitution associated with the solved form. A pattern-driven narrowing derivation is defined in a standard way by composing the output substitutions of each of its component steps. Note that a one step pattern-driven narrowing derivation contains many narrowing steps due to the need to solve rewriting constraints. Each narrowing step is demand driven and affects an outermost function application. Therefore, we have the following theorem:

Theorem 5.1.1 *For any expression e and term t, if $e \leadsto_\delta t$, then the corresponding reduction derivation $\delta e \to^* t$ is always a lazy derivation. Such a reduction derivation is called a standard reduction in [Hue86]. Enumerating pattern-driven derivations is optimal and complete in the sense that any other derivation is subsumed by a pattern-driven derivation.*

We conclude that pattern-driven narrowing provides a realisation of lazy narrowing. Lazy narrowing extends functional programming with the capability to find for which values of variables in a nonground expression the expression evaluates to a given value. Thus, it introduces the essential solving feature to functional languages. However, on its own it is not enough because "built in" predicates may exist in functional languages, for example equality and various boolean valued primitive functions, for which a dedicated constraint solver is required. If we integrate lazy narrowing with a constraint solver over data terms, the solver is then extended to allow general expressions containing user-defined functions. Therefore, querying a functional program becomes possible. This enhanced functional programming framework may be formalized as the paradigm of constraint functional programming.

5.4 Formalizing CFP

We assume a constraint system $C_\mathcal{T} : (\mathcal{T}, V, \Phi_c, I_c)$ over first-order values, where V is the set of variables over first-order types and Φ_c are constraints consisting of privileged predicates $\mathcal{R}$. Computing the truth value of a ground relation of data terms with respect to $\mathcal{R}$ is decidable. Thus, a predicate ω in $\mathcal{R}$ can always correspond to a boolean valued function f_ω in the language. A functional program may be applied to $C_\mathcal{T}$. This introduces a new syntactic category in the functional program for constraints :

$$\mathit{Constraint} \quad ::= \quad \omega(e_1, \ldots, e_n) \mid \mathit{Constraint}, \mathit{Constraint}$$

where $\omega(x_1, \ldots, x_n) \in \Phi_c$. We use c to range over constraints.

Constraints in $C_\mathcal{T}$ are now enriched to admit general expressions defined by the functional program. A constraint system is **admissible** if it is closed under negation. In the following, we assume the underlying constraint system is admissable. A CFP program is an extension of a functional program with the syntax:

$$\mathit{Program} \quad ::= \quad \mathit{Decl}\ \texttt{in}\ e \mid \mathit{Decl}\ \texttt{in}\ c$$

The semantic function $\mathcal{C}[\![]\!] : \mathit{Constraint} \to \mathcal{P}(\mathit{Env})$ maps constraints to their solution sets:

$$\begin{array}{lcl} \mathcal{C}[\![c_1, c_2]\!] & = & \mathcal{C}[\![c_1]\!] \cap \mathcal{C}[\![c_2]\!] \\ \mathcal{C}[\![\omega(e_1, \ldots, e_n)]\!] & = & \{\eta_{|\cup\mathcal{V}(e_i)} \mid \mathcal{T} \models \omega(\mathcal{E}[\![e_1]\!]\eta, \ldots, \mathcal{E}[\![e_n]\!]\eta)\} \end{array}$$

This semantics reveals constraint solving over a functional language as "computing the environments" in which expressions, when evaluated, satisfy constraints.

The constraint solving mechanism is formed by integrating the solver of $C_\mathcal{T}$ with lazy narrowing, thus enhancing $C_\mathcal{T}$ to handle constraints in the more general universe constructed by a functional program. A scheme for such an integration is presented below. We use the pair (G, C) to represent a goal $G \cup C$ in which C contains rewriting constraints and G contains constraints from the underlying constraint solver G.

Constrained Narrowing: $\frac{(G \cup \{\omega(\ldots, fe, \ldots)\}, C)}{(G \cup \{\omega(\ldots, r, \ldots)\}, C) \cup \{e = t\}}$
where $ft = r \in \Gamma$.

Simplification 1: $\frac{(G, C)}{(G', C)}$
if $G \xrightarrow{c} G'$ where $\xrightarrow{c}$ is a simplification derivation computed by the under lying solver.

Simplification 2: $\frac{(G, C)}{(G, C')}$
if $C \xrightarrow{pt} C'$ where $\xrightarrow{pt}$ stands for a simplification derivation computed by the solver of rewriting constraints.

Failure: $\frac{(G, C)}{\texttt{false}}$
if $G \xrightarrow{c} \texttt{false}$ or $C \xrightarrow{pt} \texttt{false}$

Substitution 1: $\frac{(G, C \cup \{x = e\})}{(\rho G, C \cup \{x = e\})}$
where $\rho = \{x \mapsto e\}$ and $x \in \mathcal{V}(G)$ and $C \cup \{x = e\}$ is in solved form.

Substitution 2: $\frac{(G \cup \{x = t\}, C)}{(G, \rho C \cup \{x = t\})}$
where $\rho = \{x \mapsto t\}$ and $G \cup \{x = t\}$ is in solved form.

Positive Accumulating: $\frac{(G, C \cup \{f_\omega e = \texttt{true}\})}{(G \cup \{\omega(e)\}, C)}$
If $\omega(x) \in \Phi_c$.

Negative Accumulating: $\frac{(G, C \cup \{f_\omega e = \texttt{false}\})}{(G \cup \{\neg\omega(e)\}, C)}$
if $\omega(x) \in \Phi_c$ and the constraint system is admissable.

An initial goal takes the form $(G, \{\})$. Its solved form is of the form (G_n, C_n) where G_n is in solved form with respect to the underlying solver and $\mathcal{V}(G) \notin \mathcal{V}(C)$ and C are solved form rewriting constraints.

The soundness of lazy narrowing guarantees that the enhanced solver is sound. However, it is not in general complete because a functional program may define some boolean-valued functions which have no corresponding constraints in $C_\mathcal{T}$. This problem is similar to that of solving "hard constraints" in general constraint programming. Some ways exist to resolve this problem such as the "waiting-resuming" approach in which the solving of a hard constraint is delayed until its variables are sufficiently instantiated [JL87], or by defining special simplification rules for such constraints. However, for a program in which all boolean-valued functions are consistent with the underlying constraint system, the scheme provides a complete enhanced solver.

The scheme provides a generic model to enhance a constraint system to solve constraints in functional languages. In [Pul90], Pull uses unification on data terms as the underlying solver and combines it with lazy narrowing to solve equational constraints in lazy functional languages. In [JCGMRA91], a more general constraint system over data terms is adopted in which disunification is also exploited to deal with negative equational constraints. This model can be regarded as an instantiation of the scheme by providing unification and disunification as the "built-in" solvers.

CFP represents a constraint programming system of the "domain construction" approach of section 3.3. This means that constraints appear only as computational goals; it is not possible to define new constraints in the system. However, the framework significantly enhances the expressive power of both functional programs and the basic constraint system. Moreover, since a CFP program provides a constraint system in which defined functions behave as operators in some algebra, it is perfectly reasonable to define relations over the system following the philosophy of general constraint logic programming. Therefore, CFP is a "building block" for deriving a fully integrated **Definitional Constraint Programming** system in which both constraints and the domain of discourse are user-definable.

6 Definitional Constraint Programming

We are now in a position to present a unified definitional constraint programming (DCP) framework. A DCP program defines a constraint system by defining its domain of discourse and constraints over this domain. As discussed above, CFP and CLP exhibit, respectively, the power to define domains, and the power to define constraints. Therefore we would expect the unification of these two paradigms to result in a full definitional constraint programming system.

We start by superimposing a functional program onto a privileged constraint system. As shown in the previous section, the functional program defining functions ++ and *length* can be queried to compute the initial segment of a given list. A further abstraction is possible if we take this CFP enriched constraint system as the underlying constraint system for a CLP language. Thus, CFP queries can be used to define relations as new constraints. For example we can define the relation *front*:

$$front\ (n, l, l1)\ \ :-\ \ l1\ ++\ l2 = l, length\ l1 = n$$

to compute the initial segment with length n of an input list l. This systematic integration of CFP and CLP results in a definitional constraint programming system and therefore, can be expressed by the formula $DCP = CLP(CFP)$.

It is straightforward to construct the semantic model of a DCP program. The semantics for its functional component are traditional functional language semantics. The intended model of the relational component is its least model. This may be constructed by computing all ground atoms generated by the program using the "bottom up" iterative procedure presented in theorem 4.0.1 and taking the functionally enhanced constraint system as the underlying constraint system. In terms of the semantic functions defined above the denotation of a defined predicate p in a program Γ can be computed by enumerating the inductive closure of Γ as follows :

$$\begin{aligned} p^0 &= \emptyset \\ p^{I_{n+1}} &= \{\alpha(x_1, \ldots, x_n) \mid \alpha \in \textstyle\bigcap_{i=1}^{n} \mathcal{C}[\![c_i]\!] \cap \bigcap_{j=1}^{m} [\![B_j]\!]^{I_{n+1}}\} \end{aligned}$$

for each $p(x_1, \ldots, x_n) :- c_1, \ldots, c_n, B_1, \ldots B_m \in \Gamma$. $[\![B]\!]^I$ maps B to all solutions of B under the interpretation I for the predicates in B. That is :

$$[\![p(e_1, \ldots, e_n)]\!]^I = \{\eta \mid (\mathcal{E}[\![e_1]\!]\eta, \ldots, \mathcal{E}[\![e_n]\!]\eta) \in p^I$$

Compared with other functional logic systems, this general notion of constraint satisfaction permits us, not only to define equational constraints over finite data terms, but also to introduce more general domain specific constraints. Moreover, partial objects as introduced by lazy functional programming are admissible for constraint solving in the system

as approximations of complete objects. This gives uniform support for laziness in a fully integrated functional logic programming system.

The computational model of the DCP paradigm is simply the instantiation of the underlying constraint solver in constrained SLD-resolution to the CFP solver. Soundness and completeness are a direct result of the properties of these two components.

Clearly then, DCP represents a supersystem of both these paradigms. Both the CLP *InCap* program and the CFP query which computes the initial segment of a list are valid DCP programs and queries. Moreover, the expressive power of each of these individual paradigms is enhanced in the DCP framework. We will demonstrate this with reference to some programming examples.

The "built-in" solver manipulates only first-order objects. In any correctly-typed DCP program, a function-typed variable will never become a constrained variable. Thus, higher-order functional programming features safely inherit their intended use in functional computation without introducing computability problems. The following examples illustrate some of the attractive programming features of this rich language paradigm.

The quicksort algorithm is defined below as a relation which uses difference lists (which appear as pairs of lists *(x, y)*) to perform list concatenation in constant time. The partitioning of the input list is specified naturally as a function, while the ordering function is passed as an argument to the quicksort relation. Within the semantics of DCP, such a functional parameter can be treated as special constant in relation definitions. A primitive function *apply* is assumed which is responsible for the application of such function names to arguments.

functions
partition : $(alpha \rightarrow alpha \rightarrow boolean) \times alpha \times [alpha] \rightarrow ([alpha], [alpha])$
relations
quicksort : $(alpha \rightarrow alpha \rightarrow boolean) \times [alpha] \times ([alpha\ ], [alpha])$

$partition\ (f, n, m : l) =$ **if** $f\ (n, m)$
then $(m : l1, l2)$ **else** $(l1, m : l2)$
where $(l1, l2) = partition\ (f, n, l)$
$partition\ (f, n, []) = ([], [])$

$quicksort\ (f, n : l, (x, y)) :-$
$partition\ (f, n, l) = (l1, l2),$
$quicksort\ (f, l1, (x, n : z)), quicksort\ (f, l2, (z, y))$
$quicksort\ (f, [], (x, x))$

The relation *perms* below shows an interesting and highly declarative way of specifying the permutations problem in terms of constraints over applications of the list concatenation function *++*.

relations
perms : $[alpha] \times [alpha]$
$perms\ (a : l, (l1 ++ (a : l2)) :- (l1 ++ l2) = perms\ l$

The final example shows how the recursive control constructs of higher-order functions may be used to solve problems in the relational component of a DCP language. We use a *reduce* function over lists, together with the "back substitution" technique familiar in logic programming, to find the minimal value in a list and propagate this value to all cells of the list. This is shown via the relation *propagatemin* below, which uses the standard list *reduce* function to find the minimum value, *y*, in the input list and construct a list, *l1*, which is isomorphic to the input list, in which each element is a logical variable *x*.

relations *propagatemin* : $[Int] \times [Int]$
$propogatemin\ l\ l1\ :-$
$reduce\ (f\ x, l, (MaxInt, nil)) = (y, l1), x = y$
where $f\ z\ n\ (m, l2) = (min\ (n, m), z : l2)$

These examples show that as well as being a systematic and uniform integration of constraint, logic and functional programming with a sound semantics, the DCP paradigm displays a significant enhancement of programming expressive power over other integrated language systems. We believe that this pleasing outcome is a direct result of our strenuous effort to identify clearly the essential characteristics of the component language paradigms and to preserve them faithfully in the DCP language construction. We have defined a concrete DCP language, Falcon [GP91]. Many Falcon programming examples appear in [DGP91].

7 Future Work

A very promising area of future research is the use of DCP as the foundation for studying declarative parallel programming. The idea is quite simple. If we keep strictly to the functional computational model for the functional sublanguage of a DCP language, synchronization between functional computation and constraint solving over logic variables becomes possible. Within this concurrent DCP framework, both the logical and the functional sublanguages cooperate to construct objects. The logical component approximates objects by imposing constraints and the functional component constructs objects explicitly. At each step of the construction, the functional part asks for more information and continues the construction if and when that information is available. Otherwise, it suspends and waits until other concurrently executing agents provide the required information.

This behaviour is an important generalization of the traditional **local propagation** model for constraint-based computation [Ste80]. The synchronization mechanism for functional computation obviously follows the data flow school, but the use of constraint computation to enhance incrementally the information of logical variables provides a very attractive general data flow model, i.e. **bi-directional data flow**. This idea originated from the data flow language *Id-Nouveau* [NPA86] in which an array of logical variables is a special structure for synchronising functional computation and constraint solving. This feature is generalised by the concurrent DCP model as the basic principle of programming. Concurrent DCP may be understood as a further development of the concurrent constraint programming framework proposed by Saraswat et. al. [SR90] by exploiting the

elegant concurrent cooperation between functional and logic computation.

Since computation in its functional sublangauge is deterministic, we would expect the efficiency of the system to be much better than a logic programming system. Moreover, since the functional component provides a powerful synchronization mechanism for deduction, with such a "control" mechanism the overall efficiency of the paradigm is promising. This idea of exploiting deterministic computation in a non-deterministic system by constraint propagation is also central to the Andorra model [S.H90] which has been widely accepted recently in the logic programming community. The development of concurrent DCP has led to a very interesting convergence of research on language integration, constraint programming and declarative parallel programming in [GF91].

8 Conclusion

This paper set out to provide an answer to the question of how and why we should integrate functional and logic programming languages. We believe that this should be done not only with the goal of building a more powerful programming system but also aiming at diminishing the drawbacks of the individual language paradigms. An integrated system should not only inherent the features of its components but also, and equally importantly, it should exhibit new distinguishing features as a result of their combination. We have developed a methodology for integration which demonstrates how the essential relational and functional features may be preserved, and have explored the new programming features which arise. The main idea underpinning this work comes from clarification of the intended semantics of logic and functional languages which motivated the insight to use constraints as the glue for their integration. This led us to develop the new language paradigm of definitional constraint programming. We believe that the declarative constraint programming model is a promising language paradigm for the design of future programming languages.

9 Acknowledgements

We are indebted first and foremost to Sophia Drossopoulou and Ross Paterson, our two colleagues on the Phoenix project at Imperial College, for many valuable discussions. We also thank our other colleague on the Phoenix project at Nijmegen University and at GMD Kahlsruhe, particularly Maria Fereira for her cooperation and significant contribution to the recent work on concurrent DCP, and Hendrick Lock for his enlightening discussions on the philosophy of language integration. Many thanks are due to Dr. Hassan Ait-Kaci, Prof. J-L Lassez, Dr. J. Jaffer and Dr. Meseguer for their helpful insights and to all the people in the Advanced Languages and Architectures Section at Imperial College who provide a stimulating working environment. This work was carried out under the European Community ESPRIT funded Basic Research Action 3147 (Phoenix).

References

[AEHK89] M. Aronsson, L-H Eriksson, L. Hallnas, and P. Kreuger. A Survey of GCLA: A Definitional Approach to Logic Programming. In *Proc. of the International Workshop on Extensions of Logic Programming*, volume 475 of *Lecture Notes in Computer Science, Springer Verlag.* Springer, 1989.

[Col87] A. Colmerauer. Opening the Prolog III universe. *Byte, July*, 1987.

[Com90] Haskell Committee. Haskell: A non-strict, purely functional language. Technical report, Dept. of Computer Science, Yale University, April 1990.

[DAP86] J. Darlington, Field. A.J., and H. Pull. The unification of functional and logic languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming*, pages 37–70. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[DG89] J. Darlington and Y.K. Guo. Narrowing and Unification in Functional Programming. In *Proc. of RTA 89*, pages 292–310, 1989.

[DG90] J. Darlington and Y. Guo. Constraint equational deduction. Technical report, Dept. of Computing, Imperial College, March 1990. will be presented in CTRS' 90.

[DGP91] J. Darlington, Y.K. Guo, and H. Pull. A new perspective on integrating functional and logic languages. Technical report, Dept. of Computing, Imperial College, December 1991.

[DO88] N. Dershowitz and M. Okada. Conditional equational programming and the theory of conditional term rewriting. In *Proc. of the FGCS '88, ed. by ICOT*, 1988.

[DP86] N. Dershowitz and D.A. Plaisted. Equational programming. *Machine Intelligence (Mitchie,Hayes and Richards, eds.)*, 1986.

[EGP86] C. Moiso E. Giovannetti, G. Levi and C. Palmidessi. Kernel Leaf: An experimental logic plus functional language - its syntax, semantics and computational model. ESPRIT Project 415, Second Year Report, 1986.

[Fri85] Laurent Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proceeding of the 2nd IEEE Symposium on Logic Programming, Boston*, 1985.

[GF91] Y. K. Guo and M. Fereira. Constraints, Functions and Concurrency. Technical report, Dept of Computing, Imperial College, Sept. 1991. Working Research Notes.

[GLDD90] Y. Guo, H. Lock, J. Darlington, and R. Dietrich. A classification for the integration of functional and logic languages. Technical report, Dept. of Computing, Imperial College and GMD Forchungsstelle an der Universitat Karlsruhe, March 1990. Deliverable for the ESPRIT Basic Research Action No.3147.

[GM84] Joseph A. Goguen and Jose Meseguer. Equality, types, modules, and (why not?) generics for logic programming. *Journal of Logic Programming*, 2:179–210, 1984.

[GM87] Joseph Goguen and Jose Meseguer. Models and equality for logical programming. In *Proc. of TAPSOFT 87*, volume 250 of *Lecture Notes in Computer Science, Springer Verlag*. Springer, 1987.

[GP91] Y.K. Guo and H. Pull. Falcon: Functional And Logic language with CONonstraints–language definition. Technical report, Dept. of Computing, Imperial College, February 1991.

[Hue86] G. Huet. Formal structure for computation and deduction. Technical report, Dept. off Computer Science, Carnegie-Mellon University, May 1986.

[Hul80] Jean-Marie Hullot. Canonical forms and unification. In *5th Conf. on Automated Deduction*. LNCS 87, 1980.

[JCGMRA91] M.T. Hortala-Gonzalez J Carlos Gonzalez-Moreno and Mario Rodriguez-Artalejo. A Functional Logic Language with Higher Order Logic Variiables. Technical Report, Dpto. de Informatica y Automatica UCM, 1991.

[JL86] Joxan Jaffar and Jean-Louis Lassez. Logical programming scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming*, pages 441–467. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Prod. of POPL 87*, pages 111–119, 1987.

[LM89] J-L. Lassez and K. McAloon. A constraint sequent calculus. Technical report, IBM T.J. Watson Research Center, 1989.

[Mes89] Jose Meseguer. General logics. Technical Report SRI-CSL-89-5, SRI International, March 1989.

[Mil84] Robin Milner. A proposal for Standard ML. In *ACM Conference on Lisp and Functional Programming*, 1984.

[MT84] M.Hagiya and T.Sakurai. Foundation of Logic Programming Based on Inductive Definition. *New Generation Computing*, 2(1), 1984.

[NPA86] R. Nikhil, K. Pingali, and Arvind. Id nouveau. Technical report, M.I.T. Laboratory for Computer Science, 1986. CSG Memo 265.

[PS89] L.C. Paulson and A.W. Smith. Logic Programming, Functional Programming and Inductive Definitions. In *Proc. of the International Workshop on Extensions of Logic Programming*, volume 475 of *Lecture Notes in Computer Science, Springer Verlag*. Springer, 1989.

[Pul90] Helen M. Pull. *Equation Solving in Lazy Functional Languages*. PhD thesis, Dept. of Computing, Imperial College, University of London, November 1990.

[Red84] Uday S. Reddy. Narrowing As the Operational Semantics of Functional Languages. In *Proc. of Intern. Symp. Logic Prog. IEEE'*. IEEE, 1984.

[Red86] Uday S. Reddy. Functional Logic Languages, Part 1. In J.H. Fasel and R.M. Keller, editors, *Poceedings of a Workshop on Graph Reduction, Santa Fee*, number 279 in Lecture Notes in Computer Science, Springer Verlag, pages 401–425, 1986.

[Sco89] Dana Scott. Semantic domains and denotational semantics. Lecture Notes of the International Summer School on Logic, Algebra and Computation, Marktoberdorf, 1989. to be published in LNCS series by Springer Verlag.

[S.H90] S.Haridi. A logic programming language based on andorra model. In *New Generation Computing*. 1990.

[Smo89] Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Vom Fachbereich Informatik der Universitat Kaiserlautern, May 1989.

[Smo91] Gert Smolka. Residuation and Guarded Rules for Constraint Logic Programming. Research Report RR-91-13 DFKI, 1991.

[Sny90] W. Snyder. *The Theory of General Unification*. Birkhauser, Boston, 1990.

[SR90] V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proc. 17th Annual ACM Symp. on Principles of Programming Languages. ACM, 1990*, 1990.

[Ste80] G.L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. PhD thesis, M.I.T. AI-TR 595, 1980.

[Tur85] David A. Turner. Miranda: A non-strict language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture*, LNCS 201, pages 1–16, 1985.

[You88] Jia-Huai You. Outer Narrowing for Equational Theories Based on Constructors. In Timo Lepistö and Arto Salomaa, editors, *15th Int. Colloqium on Automata, Languages and Programming*, LNCS 317, pages 727–741, 1988.

[Yuk88] K. Yukawa. Applicative logic programming. Technical Report LP-5, Logic programming Laboratory, June 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Mechanism for Reasoning about Time and Belief

Hideki Isozaki
NTT Basic Research Laboratories
3-9-11, Midoricho, Musashino-shi
Tokyo 180, Japan

Yoav Shoham
Computer Science Department
Stanford University
Stanford, CA 94305, U.S.A.

Abstract

Several computational frameworks have been proposed to maintain information about the evolving world, which embody a *default persistence* mechanism; examples include *time maps* and the *event calculus*. In multi-agent environments, time and belief both play essential roles. Belief interacts with time in two ways: there is the time *at* which something is believed, and the time *about* which it is believed.

We augment the default mechanisms proposed for the purely temporal case so as to maintain information not only about the objective world but also about the evolution of beliefs. In the simplest case, this yields a two-dimensional map of time, with persistence along each dimension.

Since beliefs themselves may refer to other beliefs, we have to think of a statement referring to an agent's temporal belief about another agent's temporal belief (*a nested temporal belief statement*). It poses both semantical and algorithmic problems. In this paper, we concentrate on the algorithmic aspect of the problems. The general case involves multi-dimensional maps of time called *Temporal Belief Maps*.

1 Introduction: *Time Maps* and *Temporal Belief Maps*

In multi-agent environments, time and belief both play essential roles. Belief interacts with time in two ways: there is the time *at* which something is believed, and the time *about* which it is believed. As in the atemporal treatment of belief, beliefs themselves may refer to beliefs (of other agents, or even the same one). For example, in the framework of Agent Oriented Programming [Shoham 1990], at any time the mental state of an agent contains information about the mental states of other agents at various times.

A statement referring to an agent's temporal belief about another agent's temporal belief will be called a *nested temporal belief statement.* An example of it is the sentence "On Wednesday John believed that on the previous Monday Jane believed that on the following Saturday they would clean the house." Nested temporal beliefs pose a number of interesting problems, both semantical and algorithmic. In this paper we concentrate on the latter kind; we propose a computational mechanism called a *Temporal Belief Map*, which functions as a data base of nested temporal beliefs.

Consider a formal language for expressing nested temporal beliefs. A standard construction would extend classical logic with a modal operator $\mathrm{B}_a^t\varphi$ for each agent designator a and time point symbol t, meaning intuitively that at time t the agent a believes φ. To ensure that the modal operator respects the properties of belief (or, more exactly, its crude approximation that has been employed in computer science and AI), various restrictions on this operator have been suggested, and then extensively explored, debated and modified[Hintikka 1962, Griffiths 1967, Konolige 1986]. These include properties such as $\mathrm{B}_a^t(\varphi \supset \psi) \wedge \mathrm{B}_a^t\varphi \supset \mathrm{B}_a^t\psi$ (the 'K' axiom), $\mathrm{B}_a^t\varphi \supset \neg\mathrm{B}_a^t\neg\varphi$ (the 'D' axiom), $\mathrm{B}_a^t\varphi \supset \mathrm{B}_a^t\mathrm{B}_a^t\varphi$ and $\neg\mathrm{B}_a^t\varphi \supset \mathrm{B}_a^t\neg\mathrm{B}_a^t\varphi$ (the '4' and '5' axioms)[Chellas 1980], and others. In addition, although these have been less well studied, further constraint may be imposed on the *change* in belief over time.

We will briefly return to these properties in the next section, but they are not the focus of this paper. Instead, we concentrate on algorithmic issues. Consider first the purely temporal case, without an explicit notion of belief. In principle, capturing the truth of facts over time

should pose no problem; we can use standard data base techniques to capture the fact true at a single point in time, and repeat it for all point. In practice, though, it is impossible, and we will need to use some shortcuts. The representational aspect of the problem appears in the form of the well-known *frame problem* [McCarthy and Hayes 1969]: when you buy a red bicycle, how you conclude that a year later it will still be red, regardless of what happens in the meanwhile — the bike is ridden, the tire is fixed, elections are held — unless it is painted. An axiom stating explicitly that the color does not *change* after each action is called a *frame axiom*; the problem is to capture the persistence of facts without including the numerous possible frame axioms.

The frame problem and related problems have been investigated in detail from the logical point of view (cf. [Shoham 1992]), and most solutions proposed have made use of nonmonotonic logic. Adding belief yields a qualitative increase in difficulty, since beliefs (and lack thereof) tend to persist as well: once you learn something, you will keep it in mind until you forget it or learn incompatible facts. The formal details of the persistence of mental state have not yet been studied as deeply; an initial treatment of it appears in [Lin and Shoham 1992].

As was said, we are interested in the algoirthmic aspects of the problem. Computational complexity of knowledge and belief without time was discussed by [Halpern and Moses 1985]. In the purely temporal case, the question is how to efficiently implement the following persistence principle (throughout this article we will assume discrete time, but the discussion can be adapted to the continuous case as well; we also assume propositional facts, with no variables):

> p^{t+1} holds iff either an event which causes p occurred at time t, or else p^t holds and no event which causes $\neg p$ occurred at time t.

Straightforward embodiment of this rule in backward chaining is too inefficient. In order to determine the truth value of p^t, you do not want to have to check p^{t-1}, p^{t-2}, and so on until you discover that $p^{t-213857}$ is true.

Both *time maps* [McDermott 1982, Dean and McDermott 1987] and *event calculus* [Kowalski and Sergot 1986] provide better alternatives. In particular, time maps rely on keeping track of only the points at which the truth value of the proposition changes, which are sufficient to determine the truth value of all other points. Each event gives rise to a *default persistence*, which ends at the first future point about which a contradictory fact is believed. For example, if an event which causes p occurs at time $t^{[1]}$ (the superscript in $[\ ]$ identifies a given time point), and no other information about p is yet present in the time map, then the two points $t^{[1]}$ and ∞ are associated with p, with a default persistence of p from the first to the second. This may be depicted graphically by Figure 1.

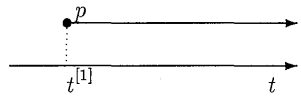


Figure 1: A simple persistence

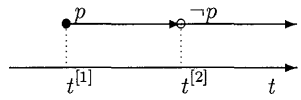


Figure 2: Clipping a persistence

If it is subsequently added that at time $t^{[2]}(> t^{[1]})$ an event happened that causes $\neg p$, $t^{[2]}$ is associated in addition with p; a default persistence of $\neg p$ is assumed between $t^{[2]}$ and ∞, and the persistence of p starting at $t^{[1]}$ is "clipped" at $t^{[2]}$ (Figure 2).

This is a crude description of the operation of time maps, but it suffices to explain the transition to *temporal belief maps* (TBM's), which incorporate an explicit notion of belief.

(Note that we have discussed only persistence into the future. Most of the literature in AI does that, and we too will in this paper. However, persistence into the past can make as much sense, especially when one adds an explicit notion of belief. For example, if you find a book on a desk, you will believe that the book was on the desk a few minutes ago. Most researchers manage to avoid this issue by limiting the form of temporal information. In particular, both time maps and the event calculus embody a certain *causality principle*: the only way new temporal information is added is by a preceding event which *causes* it. Since an explicit cause is known, there is no reason to posit backward persistence, past the cause. For example, we cannot represent the simple fact that the book *was* on the table; we must represent a specific event or action that resulted in that state (such as placing the book

there). The closest one gets to backward persistence is through abductive reasoning, "what would have to be the case previously in order for this fact to hold," positing previous events. In applications such as planning[Allen et al. 1991], this is a reasonable assumption, as in those one is constructing a map of the future based on specific planned events. However, if one is trying to use the mechanism to piece together a map of time on the basis of spotty data, this may prove inappropriate. For example in a framework such as Agent Oriented Programming [Shoham 1990], a major source of new temporal information are INFORM messages from other agents. As a result of these messages, the agent may possess a rich sample of what is true and false over time, but no causal knowledge of the precipitating events. Nevertheless, we will ignore backward persistence in most of the paper. Unless we explicitly state otherwise, we will use the term *persistence* to mean forward persistence.)

Suppose we now wish to represent the evolution of an agent's beliefs. Let us first introduce the notion of *learning*, which will play a role that is analogous to that of an event in time maps. Given this notion, beliefs too will be subject to a persistence rule:

> "The agent believes a fact at time $t+1$ iff he learned it at time t, or else at time t he believed the fact and did not at that time learn that it became false."

(This rule embodies the assumption that agents have perfect memory.) If, in addition, the "fact" itself is temporal, we end up with persistence along two orthogonal dimensions: the time of belief and the time of the property. This is the simple case of a 2-dimensional TBM.

The extension to higher-dimensional TBM's is natural. Such TBM's are obtained by nested belief statements, such as as "John believes today that yesterday he did not believe ..." and "John believes today that tomorrow Mary will believe ..."); both of these example statements induce a 3-dimensional TBM. It turns out that resolving contradictions in a multi-dimensional TBM is somewhat more subtle than in standard time maps, as the following sections will describe.

Here then is the problem we will address. Let us use the notation $\mathrm{L}_a^t\varphi$ to mean that agent a learned φ at t (actually formalizing this notion is tricky, but that is not the concern of this paper; we use the notation merely as shorthand for the English sentence). The input to our problem is assumed to be a collection of data points of the form $\mathrm{L}_{a_1}^{t_1^{[i]}}\mathrm{L}_{a_2}^{t_2^{[i]}}\cdots\mathrm{L}_{a_{n-1}}^{t_{n-1}^{[i]}}p_i^{t_n^{[i]}}$ and $\mathrm{L}_{a_1}^{t_1^{[j]}}\mathrm{L}_{a_2}^{t_2^{[j]}}\cdots\mathrm{L}_{a_{n-1}}^{t_{n-1}^{[j]}}\neg p_j^{t_n^{[j]}}$. In other words, the sequences of agent indices are identical in all the input data, but the time indices are unconstrained (we will see in section 5 why assuming a fixed sequence of agent indices is not limiting). We also assume that the data is *consistent*, that is, it does not contain both $\mathrm{L}_{a_1}^{t_1^{[k]}}\cdots\mathrm{L}_{a_{n-1}}^{t_{n-1}^{[k]}}p_k^{t_n^{[k]}}$ and $\mathrm{L}_{a_1}^{t_1^{[k]}}\cdots\mathrm{L}_{a_{n-1}}^{t_{n-1}^{[k]}}\neg p_k^{t_n^{[k]}}$ for any k. The problem is to define the rules of persistence in this n-dimensional space, that is, to define for any $(t_1, t_2, \ldots, t_n)$ in the space and each fact p, which (if either) of $\mathrm{B}_{a_1}^{t_1}\mathrm{B}_{a_2}^{t_2}\cdots\mathrm{B}_{a_n}^{t_{n-1}}p^{t_n}$ and $\mathrm{B}_{a_1}^{t_1}\mathrm{B}_{a_2}^{t_2}\cdots\mathrm{B}_{a_n}^{t_{n-1}}\neg p^{t_n}$ are supported by the data. (In all of the above, both the agent indices and the time indices may contain repetitions.) Furthermore, we will want our definition to support an efficient mechanism for answering such a query about any point in the space.

Note that both the input form and query form are quite constrained. For example, the input form precludes facts such as "John learned that Mary did not believe φ," ($\mathrm{L}_{\mathrm{John}}\neg\mathrm{B}_{\mathrm{Mary}}\varphi$) without making the stronger statement "John learned that Mary learned $\neg\varphi$." ($\mathrm{L}_{\mathrm{John}}\mathrm{L}_{\mathrm{Mary}}\neg\varphi$)

Similarly, a query "Does John believe that Mary does not believe φ?" ($\mathrm{B}_{\mathrm{John}}\neg\mathrm{B}_{\mathrm{Mary}}\varphi$?) are disallowed, only the stronger query about Mary's believing the negated fact ($\mathrm{B}_{\mathrm{John}}\mathrm{B}_{\mathrm{Mary}}\neg\varphi$?). A positive answer to the second ($\mathrm{B}_{\mathrm{John}}\mathrm{B}_{\mathrm{Mary}}\neg\varphi$) would entail a positive one to the first ($\mathrm{B}_{\mathrm{John}}\neg\mathrm{B}_{\mathrm{Mary}}\varphi$), but a negative answer ($\neg\mathrm{B}_{\mathrm{John}}\mathrm{B}_{\mathrm{Mary}}\neg\varphi$) would shed no light on the first query ($\mathrm{B}_{\mathrm{John}}\neg\mathrm{B}_{\mathrm{Mary}}\varphi$?).

These are extensions we plan to look at in the future.

In the remainder of this paper we will elaborate on this picture. We will explicate the assumptions made about agents, and discuss the multi-dimensional persistence in more detail. The organization is as follows. In section 2 we state the assumptions we make about agents' beliefs, both at single points in time and over periods of time. In section 3 we look closely at persistence in a TBM's with a single datum point. In section 4 we look at TBM's with multiple data points. In section 5 we discuss the extension to data with multiple sequences of agent indices. In section 6 we briefly mention the complexity of the query answering, and in section 7 we briefly mention implementation efforts. We conclude with discussion of related and future work.

2 Assumptions about Belief

We mentioned before that various idealizing assumptions about belief have been made and debated by other researchers, and that the focus of this paper is different from them. Nonetheless a few basic assumptions are essential, and we discuss them here. In the spirit of this paper, we discuss these properties in commonsense terms, rather than in a formal logic.

We have already listed some of the more common restriction on belief: closure of beliefs under tautological implication (as captured by the 'K' axiom), consistency (as captured by the 'D' axiom), and positive and negative introspection (as captured by the '4' and '5' axioms). Since among objective properties (those without a belief operator) we will consider only literals (atomic properties and their negations), the closure property will be irrelevant. Positive and negative introspection will also turn out to impact our results only minimally, as will be discussed in section 5. However, consistency will lie at the heart of the TBM mechanism, and is our first assumption.

Assumption 1 (Consistency) $\mathtt{B}_a^t\varphi$ *and* $\mathtt{B}_a^t\neg\varphi$ *cannot both hold.*

This is the only assumption we will make about a belief at an instance of time.

In addition we have constraints on how beliefs change over time. We first assume that agents do not come to believe facts without explicitly learning them, but that once they learn them, they do not forget them.

Assumption 2 (Causality and Memory) *If at* t *agent* a *does not learn* $\neg\varphi$*, then* $\mathtt{B}_a^{t+1}\varphi$ *holds iff* $\mathtt{B}_a^t\varphi$ *holds.*

Our next assumption is that agents are extremely receptive to new information[Gärdenfors 1988].

Assumption 3 (Gullibility) *If at time* t *agent* a *learns* φ*, then* $\mathtt{B}_a^{t+1}\varphi$ *holds.*

(Of course, in an environement in which agents are supplied with unreliable or dishonest information, this last assumption would be unacceptable, and we would need a more sophistiated criterion to determine which of the two contradictory facts, the previously believed one and the newly learned one, should dominate.)

Our last assumption is that all these properties are 'common knowledge':

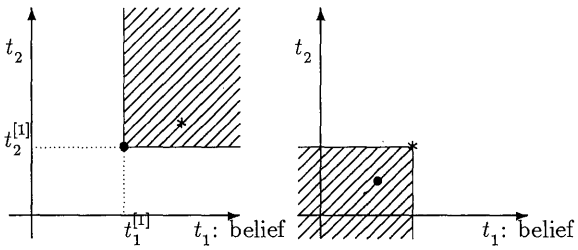


Figure 3: Default region (left) and causal region (right)

Assumption 4 (Common knowledge) *Every agent believes that every agent believes the above properties, that every agent believes that every agent believes them, and so on.*

3 Multi-Dimensional Persistence of a Single Datum

In this section, we consider TBM's induced by a single datum point. We start by considering the non-nested case, in which the datum has the form $\mathtt{L}_a^{t_1^{[1]}} p^{t_2^{[1]}}$ (at time $t_1^{[1]}$ agent a learns that at time $t_2^{[1]}$ property p was (is, will be) true). This induces a 2D TBM, in which the persistences along both axes are uninterrupted and thus do not terminate at all. This situation is represented graphically in Figure 3.

The hatched quarter plane in the left picture, rooted in the point $(t_1^{[1]}, t_2^{[1]})$, is called the *default region* of $(t_1^{[1]}, t_2^{[1]})$. The meaning of this region is that, given only the datum point $\mathtt{L}_a^{t_1^{[1]}} p^{t_2^{[1]}}$, $\mathtt{B}_a^{t_1} p^{t_2}$ holds by default iff (t_1, t_2) lies in that region (i.e., iff $t_1^{[1]} < t_1$ and $t_2^{[1]} < t_2$).

Similarly, if we focus on an affected point (*), all data points affecting it by their forward persistence are distributed in the opposite quarter plane. This is the dual concept of the default region and is called a *causal region* of the affected point. It is depicted graphically in the right picture of the above figure. In this paper we will be concerned mostly default regions.

Finally, although it is only the 2-dimensional case that is so amenable to graphical representation, these concepts extend naturally to the multi-dimensional case. Specifically, given only the datum $\mathtt{L}_{a_1}^{t_1^{[1]}} \cdots \mathtt{L}_{a_{n-1}}^{t_{n-1}^{[1]}} p^{t_n^{[1]}}$, we have that $\mathtt{B}_{a_1}^{t_1} \cdots \mathtt{B}_{a_{n-1}}^{t_{n-1}} p^{t_n}$ holds iff it is the case that $t_1 > t_1^{[1]}, \cdots, t_n > t_n^{[1]}$.

4 Mutiple Data with Incompatible Beliefs

We have so far considered only TBM's induced by a single datum. We now look at the general case in which we have mutiple data. We still assume that all data have the form $\mathrm{L}_{a_1}^{t_1^{[i]}} \cdots \mathrm{L}_{a_{n-1}}^{t_{n-1}^{[i]}} p_i^{t_n^{[i]}}$ or $\mathrm{L}_{a_1}^{t_1^{[j]}} \cdots \mathrm{L}_{a_{n-1}}^{t_{n-1}^{[j]}} \neg p_j^{t_n^{[j]}}$, for some fixed $a_1, \cdots, a_{n-1}$ (again, see section 5 in this connection), but nothing beyond that.

If for any p_k the collection does not contain more than one occurrence of p_k (whether preceded by $\neg$ or not), the situation is simple: the persistence of each fact is independent of the others, and so we construct an independent TBM for each one.

The situation in which multiple occurrence of a p_k exist, but all with the same polarity (that is, either in all data containing p_k the p_k is preceded by $\neg$, or in none), the situation is also simple: the default region is simply the union of the individual regions for each datum containing p_k.

It is the presence of contradictory data that makes the story more interesting. Our assumption of consistency dictates that persistences of contradictory beliefs may not overlap. Without the strong limitations on the form of input data and queries, we would have two problems — to determine which sets of persistences are contradictory, and to resolve the contradiction. For example, we would have to notice that the three sentences $\mathrm{B}_a^t(p \vee q)$, $\mathrm{B}_a^t \neg p$ and $\mathrm{B}_a^t \neg q$ are jointly inconsistent, even though all pairs are consistent. Our restrictions remove this first problem. Since we only consider facts of the form $\mathrm{B}_{a_1}^{t_1} \cdots \mathrm{B}_{a_{n-1}}^{t_{n-1}} p_i^{t_n}$ and $\mathrm{B}_{a_1}^{t_1} \cdots \mathrm{B}_{a_{n-1}}^{t_{n-1}} \neg p_j^{t_n}$, the only fact contradicting $\mathrm{B}_{a_1}^{t_1} \cdots \mathrm{B}_{a_{n-1}}^{t_{n-1}} p_k^{t_n}$ will be $\mathrm{B}_{a_1}^{t_1} \cdots \mathrm{B}_{a_{n-1}}^{t_{n-1}} \neg p_k^{t_n}$, and *vice versa*. When in future work we relax the restrictions on input and queries, we will need a new criterion for determining incompatibility.

Our restrictions do not only render the problem of determining incompatibility trivial, they also simplify the task of resolving it. Since we always have exactly two beliefs contradicting one another, our task reduces to removing one of them; the question is which.[1]

[1] Of course, removing both would also restore consistency, but that would violate our assumption about causality and memory.

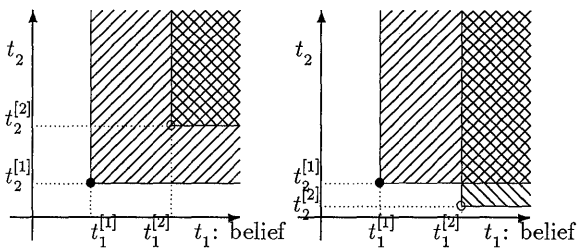


Figure 4: Overlapping default regions ($t_1^{[1]} \neq t_1^{[2]}$, $t_2^{[1]} \neq t_2^{[2]}$)

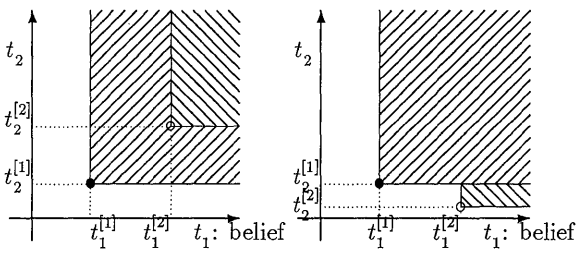


Figure 5: Consistent default regions ($t_1^{[1]} \neq t_1^{[2]}$, $t_2^{[1]} \neq t_2^{[2]}$)

4.1 The 2D Case

The rule for resolving contradictory beliefs in the two dimensional case is derived in a straightforward fashion from the assumptions stated in section 2, and is analogous to the clipping of persistences in simple time maps. We will discuss the case of two data points, but the discussion extends easily to multiple points.

Consider the input data consisting of the two points $\bullet : \mathrm{L}_a^{t_1^{[1]}} p^{t_2^{[1]}}$ and $\circ : \mathrm{L}_a^{t_1^{[2]}} \neg p^{t_2^{[2]}}$. Without loss of generality, assume that $t_1^{[2]} \geq t_1^{[1]}$ holds. We consider the two cases — $t_2^{[2]} \geq t_2^{[1]}$ and $t_2^{[2]} < t_2^{[1]}$ — and assume for now that neither $t_1^{[2]} = t_1^{[1]}$ nor $t_2^{[2]} = t_2^{[1]}$ hold. The default regions of the two points in both cases are shown in the left and right portions of Figure 4, respectively.

In both cases the default regions overlap, which is forbidden, and one of them must be trimmed. In deciding which, we recall the assumption of *gullibility*: right after learning a fact, the agent must believe it. Furthermore, the assumption of memory and causality dictates that the agent must continue to believe it until the next point about which he learns that the fact is false there. This produces the consistent default regions in Figure 5.

> **Example.** If John learns on Monday that on Thursday his house will be painted white ($\bullet$) and on Tuesday he learns that on Friday it will

be painted blue (o), then from Monday until Tuesday John will believe that his house will be white from Thursday until the end of time, and from Tuesday on he will believe that his house will be white from Thursday until Friday (+45° shading), and blue afterwards (−45° shading) (the left picture). (Of course, on Thursday he will learn that the painter had a wedding in Chicago and couldn't come.)

On the other hand (the right picture), if John learns on Monday that on Thursday his house will be painted white (•) and on Tuesday he learns that on *Wednesday* it will be painted blue (o), then from Monday until Tuesday John will still believe that his house will be white from Thursday until the end of time (+45° shading), but from Tuesday he will believe that his house will be blue from Wednesday until Thursday (−45° shading), and leave unaltered his belief that it will be white afterwards (+45° shading). (That will change when the painter, back from Chicago a week later, paints John's house turquoise, since neither white nor blue really go well with olive tree in the yard.)

Note that in either case, the beliefs from Tuesday onwards would not change even if the the two pieces of information were acquired in the opposite order. This is no accident; this Church-Rosser property is true in general of our system.

We now turn to the limiting cases, in which either $t_1^{[2]} = t_1^{[1]}$ holds or $t_2^{[2]} = t_2^{[1]}$ holds. Note that from our assumption about the consistency of the input, at most one of them can hold. Therefore, if $t_1^{[2]} = t_1^{[1]}$ holds, we may assume without loss of generality that $t_2^{[2]} > t_2^{[1]}$. This means that at time $t_1^{[1]} (= t_1^{[2]})$ the agent learned that p first became true (•) and later became false (o). The agent will therefore believe at time $t_1^{[1]} (= t_1^{[2]})$ that p will be true from the first point until the second, and false afterwards. There will be nothing later to change that belief, and thus the default region of p forms an infinite horizontal strip, and the default region of $\neg p$ occupies the quadrant above it (Figure 6).

The case in which $t_2^{[2]} = t_2^{[1]}$ holds is more interesting, since it provides insight into the higher dimensional case.

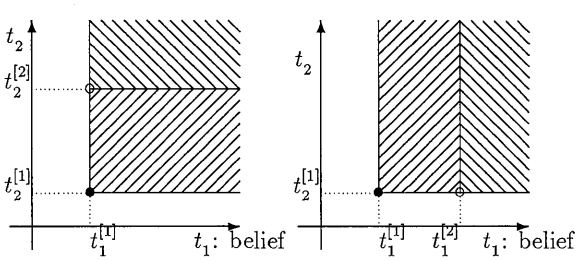


Figure 6: Default regions (left: $t_1^{[1]} = t_1^{[2]}$, right: $t_2^{[1]} = t_2^{[2]}$)

In this case the agent first learned that p became true at some point, and later learned that p became false at that very point. Now in principle we could imagine quite sophisticated criteria to decide which evidence should be given greater credence. However, our assumption of gullibility forces a "recent is better" policy, leading us to accept the later information and abandon the older one. The resulting default regions are shown in the right figure.

4.2 The General Case

We now extend the previous discussion to higher TBM's. We will unfortunately have to do so without the aid of graphics; instead, we will use the following example.

Example. At $t_1^{[1]}$ you learn that at time $t_2^{[1]}$ your son learned that your son's teacher moved to Japan at time $t_3^{[1]}$ ($L_{\text{you}}^{t_1^{[1]}} L_{\text{son}}^{t_2^{[1]}} p^{t_3^{[1]}}$). At time $t_1^{[2]}$ you learn that at time $t_2^{[2]}$ your son learned that his teacher moved to the US at time $t_3^{[2]}$ ($L_{\text{you}}^{t_1^{[2]}} L_{\text{son}}^{t_2^{[2]}} \neg p^{t_3^{[2]}}$) where $t_3^{[2]} > t_3^{[1]}$.

Let $t_1 > \max(t_1^{[1]}, t_1^{[2]})$, $t_2 > \max(t_2^{[1]}, t_2^{[2]})$, and $t_3 > t_3^{[2]} (> t_3^{[1]})$. Then at t_1 you believe that at t_2 your son believes that his teacher is living in the US at t_3. This is true regardless of the relationship between $t_1^{[1]}$ and $t_1^{[2]}$, or the relationship between $t_2^{[1]}$ and $t_2^{[2]}$.

Now consider the same scenario, except that $t_3^{[2]} = t_3^{[1]}$. This means that you believe that your son learned two contradictory facts. However, from the assumption that rules of belief change are common knowledge[2], you know that *your son* will adopt the latest information (as illustrated in the previous figure). Therefore

[2]Note that this is our first use of the common knowledge assumption!

your beliefs about your son's beliefs will depend on the relationship between $t_2^{[1]}$ and $t_2^{[2]}$; if $t_2^{[2]} > t_2^{[1]}$ then you will believe that your son believes that the teacher lives in the US; otherwise you will believe that your son believes that the teacher lives in Japan.

Finally, what will you believe if $t_3^{[2]} = t_3^{[1]}$ and $t_2^{[2]} = t_2^{[1]}$? In this case, *you* will need to break the tie by comparing $t_1^{[1]}$ and $t_1^{[2]}$. Note that they cannot also be equal, as that would violate the assumption that the input data is consistent.

The lesson from this example is clear. To determine whether a point in the hyper-space lies in a particular default region, you should compare the associated time vectors. This ordering is a *reverse lexicographical ordering*, the innermost time being the most significant and the outermost time the least significant.

5 Multiple Sequences of Agent Indices

We have all along assumed one fixed sequence of agent indices in the data: $a_1, \cdots, a_{n-1}$. However, relaxing this limitation is quite simple. Consider data points with multiple sequences of agents indices. Unless we make further assumptions about belief, data with different index sequences will simply not interact. For example, the truth of $\mathrm{B}_a^{t_1^{[1]}} \mathrm{B}_b^{t_2^{[1]}} p^{t_3^{[1]}}$ is completely independent from the truth of any statement that is not of the form $\mathrm{B}_a^{t_1^{[2]}} \mathrm{B}_b^{t_2^{[2]}} x$, where x is an objective sentence (containing no belief operator); in particular, it is consistent with $\mathrm{B}_a^{t_1^{[1]}} \mathrm{B}_c^{t_2^{[1]}} \neg p^{t_3^{[1]}}$. Thus we may simply construct separate TBM's for these different sentences, each obeying our restriction.

However, if we do make further assumptions about belief, we must take greater care. We consider here four possible further assumptions about belief. The first two are the familiar assumptions of introspective capability:

Assumption 5 (Positive introspection) $\mathrm{B}_a^t\varphi$ *holds iff* $\mathrm{B}_a^t\mathrm{B}_a^t\varphi$ *holds.*

Assumption 6 (Negative introspection) $\neg\mathrm{B}_a^t\varphi$ *holds iff* $\mathrm{B}_a^t\neg\mathrm{B}_a^t\varphi$ *holds.*

The other two have to do with beliefs of the agent at different points in time. The first is that not only do agents have memory (which we have already assumed), but they also have perfect memory of past beliefs:

Assumption 7 (Introspection about past beliefs)

$$\mathrm{B}_a^t\mathrm{B}_a^{t-\tau}\varphi \equiv \mathrm{B}_a^{t-\tau}\varphi \qquad \textit{if} \qquad \tau > 0.$$

The last assumption states that agents do not expect their beliefs to change:

Assumption 8 (Belief about stability of beliefs)

$$\mathrm{B}_a^t\mathrm{B}_a^{t+\tau}\varphi \equiv \mathrm{B}_a^t\varphi \qquad \textit{if} \qquad \tau > 0.$$

(Notice that assumptions 5, 7, and 8 can be unified into $\mathrm{B}_a^{t_1}\mathrm{B}_a^{t_2}\varphi \equiv \mathrm{B}_a^{\min(t_1,t_2)}\varphi$.)

We are not arguing on behalf of these assumptions. We list them merely as examples of plausible assumptions one might want to make. The reason we mention them at all is that they violate the property that nested temporal beliefs with different agent indices are independent of one another. For example, under assumption 8, $\mathrm{B}_a^2\mathrm{B}_a^3 p^8$ is contradictory with $\mathrm{B}_a^2\neg p^8$.

Fortunately, these four assumptions allow an easy solution. We simply keep simplifying the sentences by substitution, until no further simplifications are possible. It turns out that no matter what subset of these four we choose, the result of this substitution process is unique (the Church-Rosser property again). More generally, whenever our assumptions allow us to derive a unique *canonical form*, we convert the query and the input data to this canonical form, and then revert to our usual procedure. We have not yet investigated the more complex case in which the canonical form is hard to derive or nonexistent.

6 Complexity

Our definition of default regions was constructive, and allows efficient query answering. We briefly discuss the complexity here. If we assume that comparison of a pair of one-dimensional time points is done in one operation, then comparing two n-dimensional time points requires at most n operations. In ordinary applications, n will be a very small integer. Ordinary people will not think of $n = 5$ cases in their everyday life.

If we have N data points, we can get a sorted list of the data points by the priority based on the *reverse lexicographical ordering*, as explained. This requires only $O(n \cdot N \log_2 N) \simeq O(N \log N)$ operations. Since each agent learns informations gradually, it is useful to use a

heap, a well known balanced tree data structure which can be easily modified to keep ordering.

If we need to identify only the dominant data point in the causal region, even a naive implementation gives it in $O(nN) \simeq O(N)$ operations.

7 Implementation

Our framework can be easily implemented by logic programmming languages such as Prolog as well as ordinary procedural languages such as C. We implemented various versions of this framework in both languages. Backward reasoning mechanism implemented in Prolog employed simplified versions of Kowalski/Sergot's Event Calculus. Forward reasoning mechanism implemented in C employed sorting of an array. As we described before, our algorithm is very fast in simple cases. We intend to implement more complex cases and evaluate their complexity.

As for 2D cases, we have a program which draw a map from a set of data points whose time stamps are given in hour/minutes or minutes/seconds or year/month/day.

Finally, this work has been carried out as part of the research on Agent Oriented Programming. The current simple interpreter, `AGENTO`[Shoham 1990], only has a simple version of standard time maps. We have implemented an experimental agent interpreter which incorporates the ideas of this paper, and hope to report on it in the future.

8 Related Work and Conclusions

The only closely related work of which we are aware, other than the work on time maps and event calculus which we have discussed at length, is Sripada's [Sripada 1991], which was independently developed. Both systems can deal with nested temporal beliefs. Sripada represents a nested temporal belief by a Cartesian product of time intervals, and like us assumes that nested temporal beliefs are consistent. However, he does not consider the notion of default persistence, and therefore not with the resolution of competing default persistences. It would seem that the result of our system could serve as input to his, but we would like to understand his work better before making stronger claims about the relationship to his work.

As should be clear, much more needs to be done. We made it clear that in this work we did not undertake a logical treatment of time, belief and nonmonotonicity. We were also explicit about the limitations of our framework. We hope to do both in the future, as well as demonstrate the pratical utility of this work.

Acknowledgments

We would like to thank AOP group members at Stanford University and the referees of this paper who gave us useful comments. The first author would like to thank Koichi Furukawa at ICOT, Shigeki Goto, Hirofumi Katsuno, and other colleagues at NTT, too.

References

[Halpern and Moses 1985] J. Y. Halpern, Y. Moses. A Guide to the Modal Logics of Knowledge and Belief: Preliminary Draft, pp.480–490, Proc. of IJCAI, 1985.

[Kowalski and Sergot 1986] R. Kowalski, M. Sergot. A Logic-Based Calculus of Events, New Generation Computing, Vol.4, pp.67–95, 1986.

[Lin and Shoham 1992] F. Lin, Y. Shoham. Persistence of Knowledge and Ignorance (in preparation), 1992.

[McCarthy and Hayes 1969] J. McCarthy, P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence, in B. Meltzer and D. Michie (Eds.), Machine Intelligence 4, Edinburgh University Press, pp.463–502, 1969.

[McDermott 1982] D. V. McDermott. A Temporal Logic for Reasoning about Processes and Plans, Cognitive Science, Vol.6, pp.101–155, 1982.

[Dean and McDermott 1987] T. L. Dean, D. V. McDermott. Temporal Data Base Management, Artificial Intelligence, Vol. 32, pp.1–55, 1987.

[Chellas 1980] B. F. Chellas. Modal Logic, An Introduction, Cambridge University Press, 1980.

[Gärdenfors 1988] P. Gärdenfors, D. Makinson. Revisions of Knowledge Systems Using Epistemic Entrenchment, Proc. of the Second Conference on Theoretical Aspects of Reasoning about Knowledge, pp.83–95, 1988.

[Shoham 1990] Y. Shoham. Agent-Oriented Programming, Stanford Technical Report CS-1335-90, 1990.

[Shoham 1992] Y. Shoham. Nonmonotonic Temporal Reasoning, in D. Gabbay (Ed.), The Handbook of Logic in Artificial Intelligence and Logic Programming (to appear), 1992.

[Sripada 1991] S. M. Sripada. Temporal Reasoning in Deductive Database, PhD thesis of Univ. of London, 1991.

[Allen et al. 1991] J. F. Allen, H. A. Kautz, R. N. Pelavin, J. D. Tenenberg. *Reasoning about Plans*, Morgan Kaufmann Publishers, 1991.

[Hintikka 1962] J. Hintikka. *Knowledge and Belief: An Introduction to the Logic of the Two Notions*, Cornell University Press, 1962.

[Griffiths 1967] A. P. Griffiths ed. *Knowledge and Belief*, Oxford University Press, 1967.

[Konolige 1986] K. Konolige. *A Deduction Model of Belief*, Morgan Kaufmann Publishers, 1986.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Dealing with Time Granularity in the Event Calculus

Angelo Montanari (+)(*), Enrico Maim (-), Emanuele Ciapessoni (+), Elena Ratto (+)

(+) CISE, Milano, Italy (-) SYSECA, Paris, France
(*) Current Affiliation: University of Udine, Udine, Italy

Abstract

The paper presents a formalization of the notion of time granularity in a logic-based approach to knowledge representation and reasoning. The work is based on the Event Calculus [Kowalski,86], a formalism for reasoning about time, events and properties using first-order logic augmented with negation as failure. In the paper, it is extended to include the concept of time granularity. With respect to the representation, the paper defines the basic notions of temporal universe, temporal decomposition and coarse grain equivalence. Then, it specifies how to locate events and properties in the temporal universe and how to pair event and temporal decompositions. With respect to the reasoning mechanisms, the paper defines two alternative modalities of performing temporal projection, namely upward and downward projections, that make it possible to switch among coarser and finer granularities.

1 Introduction

The paper presents a formalization of the notion of time granularity in a logic-based approach to knowledge representation and reasoning. The work is based on the Event Calculus, a formalism for reasoning about time, events and properties using first-order logic augmented with negation as failure [Kowalski and Sergot 1986]. In the paper, it is extended to include the concept of time granularity. Informally, granularity can be defined as the resolution power of a representation. In general, each level of abstraction at which knowledge can be represented is characterized by a proper granularity. Providing a formalism with the concept of granularity allows it to embed different levels of knowledge in a representation. In such a way, each reasoning task can refer to the representational level that abstracts from the domain only those aspects relevant to the actual goal. We are interested in time granularity. With respect to the *expressive power*, it allows one to maintain the representations of the dynamics of different processes of the domain that evolve according to different time constants as separate as possible [Corsetti *et al.* 1990]. It also allows one to model the dynamics of a process with respect to different time scales. In such a case time granularity has to be paired with other refinement mechanisms such as process decomposition [Allen 1984], [Kautz and Allen 1986], [Corsetti *et al.* 1991a], [Evans 1990]. Finally, time granularity increases both the temporal distinctions that a language can make and the distinctions that it can leave unspecified. This means that considering two events as simultaneous or temporally distinct, or two time dependent relations as temporally overlapped or disjoint depends on the granularity one refers to. With respect to the *computational power*, it supports different grains of reasoning to deal with incomplete and uncertain knowledge [Allen 1983], [Dean and Boddy 1988]. It also allows one to tailor the visibility of the knowledge base and the reasoning process to the needs of the actual task [Fum *et al.* 1989]. Secondly, it allows one to alternate among different time granularities during the execution of a task in order to solve each incoming problem at a time granularity as coarse as possible [Dean *et al.* 1988]. An example of a limited use of time granularity to expedite the search of large temporal databases is provided by [Dean 1989]. Finally, it allows one to solve a problem at a time granularity coarser than the required one to

Authors' addresses: Angelo Montanari, University of Udine, Mathematics and Computer Science Department, Via Zanon, 6 33100 Udine, ITALY, email: montanari@uduniv.cineca.it; Enrico Maim, SYSECA Temps Reel, Constraint Resolution Research Group, 315 Bureaux de la Colline, 92213 Saint-Cloud Cedex, FRANCE, email: enrico_maim@eurokom.ie; Emanuele Ciapessoni and Elena Ratto, CISE, Artificial Intelligence Section, Division of Systems and Models, Via Reggio Emilia, 39, Segrate (Milano), ITALY, email: kant@sia.cise.it and elena@sia.cise.it. Most work of the first author was done while he was employed in CISE.

cope with the complexity of temporal reasoning. Such a simplification speeds up the reasoning, but implies a relaxation of the precision of the solution. The ratio between the time granularities provides a measurement of the approximation of the achieved result.

In despite of the widespread recognition of its relevance for knowledge representation and reasoning, there is a lack of a systematic framework for temporal granularity. The main references are the paper of Hobbs [1985] on the general concept of granularity and the works of Plaisted [1981], Giunchiglia and Walsh [1989] on abstract theorem proving. Hobbs defines a concept of granularity that supports the construction of simple theories out of more complex ones. He formally introduces the basic notions of relevant predicate set, indistinguishability relations, simplification, idealization and articulation. Such notions are extended and refined by Greer and McCalla [1989], which identify two orthogonal dimensions along which granularity can be interpreted, namely abstraction and aggregation. However, the one and the others reserve little or no attention to time granularity. In particular, Hobbs only sketches out a rather restrictive mapping of continuous time into discrete times using the situation calculus formalism. Conversely, a set-theoretic formalization of time granularity is provided by Clifford and Rao [1988], but they do not attempt to relate the truth value of assertions to time granularity. Finally, Galton [1987] and Shoham [1988] give significant categorizations of assertions based on their temporal properties. These categorizations are strictly related to the concept of time granularity even if it is not explicitly considered.

A first attempt to introduce the notion of time granularity in the Event Calculus is reported in [Evans 1990]. Evans defines a macro-events calculus for dealing with time granularity whose limitations are discussed in section 4.1. Our paper proposes a framework to represent and reason about time granularity in the Event Calculus that generalizes these previous results. It significantly benefits by the work done to formalize the concept of time granularity in TRIO, a logic formalism for specifying realtime systems [Corsetti *et al.* 1991b], [Corsetti *et al.* 1991c], [Montanari *et al.* 1991], and [Ciapessoni *et al.* 1992]. [Maim 1991] and [Maim 1992a] present an alternative approach where the granularity problem is seen as an issue of dealing with ranges and intervals in constraint-based reasoning.

The paper is organized as follows: section 2 presents the original Event Calculus together with its basic extensions, namely types, macro-events and continuous change; section 3 focuses on the representation of time granularity; section 4 details the modalities of reasoning about time granularity.

2 The Event Calculus

The Event Calculus proposes a general approach to represent and reason about events and their effects in a logic framework [Kowalski and Sergot 1986], [EQUATOR 1991]. From a description of events that occur in the real world, it allows one to derive various relationships and the time periods for which they hold. It also embodies a notion of *default persistence*, that is, relationships are assumed to persist until an event occurs which terminates them. As an example, if we know that an aircraft enters a given sector at 10:00hrs and leaves at 10:20hrs, the Event Calculus allows us to infer that it is in that sector at 10:15hrs. More precisely, the Event Calculus takes the notions of event, property, time-point and time-interval as primitives and defines a model of change in which *events* happen at *time-points* and initiate and/or terminate *time-intervals* over which some *property* holds. So, for instance, the events of entering and leaving the sector initiate and terminate the aircraft's property of being in the sector, respectively. Time-points are unique points in time at which events take place instantaneously. In the previous example, the event of entering the sector occurs at 10:00hrs, while the event of leaving the sector occurs at 10:20hrs. They can be specified at different degree of explicitness, e.g. "91/5/24:10:00hrs" to include the full date or just "10:00hrs", but belong to a unique domain. Time-intervals are represented by means of tuples of two time-points. With the same example, we can deduce that the aircraft is in the sector during the time-interval starting at 10hrs and ending at 10:20hrs.

Formally, Event Calculus represents domain knowledge by means of *initiates* and *terminates* predicates that express the effects of events on properties[1]:

initiates(*Event*, *Property*)
terminates(*Event*, *Property*)

In such a way, domain relations are intensionally defined in terms of events and properties *types* [EQUATOR 1991]. Weak forms of the *initiates* and *terminates* predicates, namely *weak-initiates* and *weak-terminates*, have been introduced in [Sergot 1990]. The predicate *weak-terminates* states that a given event terminates a given property unless this property has been already terminated. In a similar way, the predicate *weak-initiates* states that a given event initiates a given property unless this property has been already initiated.

Instances of events and properties are obtained by attaching a time-point *(event, time-point)* and a time-

[1] We adopt the variable convention of the original Event Calculus where constants are distinguished from variables by being denoted by names beginning with upper-case characters.

interval *(property, time-interval)* to event and property types, respectively.

The first Event Calculus axiom we introduce is the *Mholds-for*. It allows us to state that the property p holds maximally (i.e. there is no larger time-interval for which it also holds) over $\langle start, end \rangle$ if an event e occurs at the time *start* which initiates p, and an event e′ occurs at time *end* which terminates p, provided there is no known interruption in between:

Mholds-for(p, ⟨start,end⟩) ←
happens_at(e,start) ∧ initiates(e,p) ∧
happens_at(e′,end) ∧ terminates(e′,p) ∧
end > start ∧ not broken-during(p, ⟨start,end⟩)

In the above axiom, the negation involving the *broken* predicate is interpreted using negation-as-failure. This means that properties are assumed to hold uninterrupted over an interval of time on the basis of failure to determine an interrupting event. Should we later record a terminating event within this interval, we can no longer conclude that the property holds over the interval. This gives us the non-monotonic character of the Event Calculus which deals with default persistence[2].

The predicate *broken-during* is defined as follows:

broken-during(p, ⟨start,end⟩) ←
happens_at(e,t) ∧ start < t ∧
end > t ∧ terminates(e,p)

This states that a given property p ceases to hold at some point during the time-interval ⟨start,end⟩ if there is an event which terminates p at a time t within ⟨start,end⟩.

Event Calculus also defines an *Iholds-for* predicate in terms of *Mholds-for* to state that a property holds over each time-interval included in the maximal one:

Iholds-for(p, ⟨start,end⟩) ←
Mholds-for(p, ⟨a,b⟩) ∧ start ≥ a ∧ end ≤ b

Finally, Event Calculus defines the *holds-at* predicate which is similar to *Iholds-for* except that it relates a property to a time-point rather than a time-interval:

holds-at(p,t) ←
Mholds-for(p, ⟨start,end⟩) ∧
t > start ∧ t < end

In particular, the *holds-at* predicate states that a property is not valid at the time points at which occur the events that initiate and terminate it. This negative conclusion about the validity of properties at the left and right ends of time-intervals properly stands for ignorance. Time granularity will allow us to refine descriptions with respect to finer temporal domains.

[2]To deal with default persistence, [Maim 1992b] presents an approach to constructive negation in constraint-based reasoning.

2.1 Macro-events to Model Discrete Processes

To model discrete processes the basic Event Calculus has been extended with an event decomposition mechanism that allows us to refine event representations [Evans 1990], [EQUATOR 1991]. Evans introduced the notion of *macro-event*, which is a finite event decomposed into a number of sub-events. The connections between a macro-event and its components are formalized in the Event Calculus as follows:

happens_at(e,t) ←
happens_at(e1,t1) ∧ part_of(e1,e) ∧
happens_at(e2,t2) ∧ part_of(e2,e) ∧
happens_at(e3,t3) ∧ part_of(e3,e) ∧
happens_at(e4,t4) ∧ part_of(e4,e)

where the predicate *part_of* is defined by means of appropriate domain axioms.

This axiom allows us to derive the occurrence of a macro-event from the occurrences of its sub-events. It can also be used to abduce the occurrence of sub-events from the occurrence of the macro-event.

2.2 Continuous Change to Model Continuous Processes

The basic Event Calculus is well-equipped to represent discrete processes, but is not so good for representing continuous processes, i.e. processes characterized by a continuous variation in a quantity such as the height of a falling object or the angular position of a crankshaft. Modelling a continuous process in terms of its temporal snapshots, in fact, can be seen a particular case of event decomposition, but cannot be directly done by means of macro-events. To model continuous processes, Event Calculus has been extended with the idea of the *trajectory* of a continuously changing property through a space of values [Shanahan 1990], [Shanahan 1991], [EQUATOR 1991]. Shanahan introduced the notion of 'dynamic' properties, like *moving* of a train. When such a property holds, another property is continuously changing, such as *position* of the train. Continuously changing properties are modelled as *trajectories*. Formally, the *holds-at* axiom which gives value to a continuously changing property is:

holds_at (p,t2) ←
happens_at(e,t1) ∧ initiates(e,q) ∧ t1 < t2 ∧
not broken_during(q, ⟨t1, t2⟩)∧
trajectory(q,t1,p,t2)

In this axiom, the continuously changing property p can be assigned a given value at a time point t2 if an instance of the relevant dynamic property q is initiated at a time point t1 (before t2) and not broken

at some point between t1 and t2. The predicate *trajectory* describes the functional relationship between the continuously changing property and the time that has elapsed since it started to change. It can be seen as a path plotted against time through the corresponding quantity space. The formula $trajectory(q, t1, p, t2)$ represents that property p holds at time $t2$ on the trajectory of the period of continuous change represented by q which start at time $t1$. Such a property p holds only instantaneously and represents that some quantity varying continuously has a particular value. Its definition is *domain specific*. That is, a set of *trajectory* clauses is also part of the description of the domain, along with the domain's *initiates* and *terminates* clauses.

For example, suppose that the angular position of a crankshaft increasing linearly with time whilst the shaft is rotating. If ω is the angular velocity of the crankshaft, we have the following domain axiom:

$$trajectory(rotating, t1, angle(a2), t2) \leftarrow$$
$$holds\_at(angle(a1), t1) \wedge a2 = \omega(t2 - t1) + a1$$

3 Representing Time Granularity

This section first introduces the notion of temporal universe as a set of related, differently grained temporal domains. Such a notion supports the definition of the relations of indistinguishability and distinguishability among the time-points of the domains. Then, it precisely states the linkage between events and properties, and time granularity.

3.1 The Temporal Universe

Providing a representation with time granularity requires introducing a finite set of disjoint *temporal domains* that constitutes the *temporal universe* of the representation:

$$T = \bigcup_{i=1,..,n} T_i$$

The set $\{T_1, T_2, .., T_n\}$ is totally ordered on the basis of the degree of fineness (coarseness) of its elements and, for each i, with $1 \leq i < n$, T_{i+1} is said of a *finer granularity* than T_i. Each domain is *discrete* with the possible exception of the finest domain that may be dense. For the sake of simplicity, we assume that each domain is denumerable. The temporal universe includes at most one dense domain because each dense domain is already at the finest level of granularity, since it allows any degree of precision in measuring time distances. As a consequence, for dense domains we must distinguish granularity from metric, while for discrete domains we can define granularity in terms of set cardinality and assimilate it to a natural notion of metric[3]. For the sake of simplicity, we assume that each domain is denumerable.

For each pair of domains T_i, T_{i+1}, a mapping is defined that maps each time-point of T_i into a time-interval of T_{i+1} (*totality*). mathematical expressions we use the $succ_i(t)$ denotes the It maps contiguous time-points into contiguous, disjoint time-intervals (*contiguity*) preserving the ordering of the domains (*order preserving*) . Moreover, the union set of the time-intervals of T_{i+1} belonging to its range is equal to T_{i+1} (*coverage*). Finally, we assume that the length of time-intervals into which it maps the time-points of T_i is constant (*homogeneity*). This constant, denoted by $\Delta_{i,i+1}$, defines the *conversion factor* between T_i and T_{i+1} which provides a relative measurement of the granularity of T_i and T_{i+1} with respect to each other. A general mapping between T_i and T_j, with T_i coarser than T_j, can be easily obtained by a suitable composition of a number of elementary mappings. It is formally defined in a recursive way in [Corsetti *et al.* 1991a], where it is also shown that the properties of totality, contiguity, order preserving, coverage and homogeneity are preserved.

In general, there are several ways to define these mappings each one satisfying the required properties. According to the intended meaning of the mappings as *decomposition functions*, each time-point of T_i is mapped into the set of time-points of T_{i+1} that compose it. Nevertheless, we are faced by a number of alternative possibilities in settling the reference time-point of each domain. Choosing the one or the other is merely a matter of convention, but it determines the actual form of the mappings. In the following, assume that, for each pair T_i, T_j, the relevant function maps the reference time-point of T_i into a time-interval of T_j whose first element is the reference time-point of T_j (reference time-points alignment assumption).

To include the notion of temporal universe in the Event Calculus, we introduce the predicate *value-metric* which splits each time-point (1st argument) into a metric (2nd argument) and a value (3rd argument) components. Moreover, we express metrics as a subset of integer. Let us consider a temporal universe consisting of hours, minutes and seconds, and assign by convention the metric 1 to the domain of

[3] "Mapping, say, a set of reals into another set of reals would only mean changing the unit of measure with no semantic effect. Just in the same way one could decide to describe geometric facts by using, say, Kmeters and centimetres. However, if Kmeters are measured by real numbers, the same level of precision as with centimetres can be achieved. Instead, the key point in time granularity is that saying that something holds for all days in a given interval does not imply that it holds every second within the 'same' interval" [Corsetti *et al.* 1991c].

seconds (in general, metric 1 is assigned to the finest domain), the metric 60 to the domain of minutes)1 minute corresponds to 60 seconds) and the metric 3600 to the domain of hours (1 hour corresponds to 3600 seconds). As an example, *value-metric(2hrs30m,60,150)* since there are 60 minutes in an hour. Using the predicate *value-metric*, decomposition functions can be defined as follows:

$$
\begin{aligned}
&fine\_grain\_of(<t1,t2>,t)\leftarrow\\
&\quad value\_metric(t,m,v)\wedge\\
&\quad value\_metric(t1,m1,v1)\wedge\\
&\quad value\_metric(t2,m1,v2)\wedge m1\leq m\wedge\\
&\quad v1=v*(m/m1)\wedge v2=(v+1)*(m/m1)-1
\end{aligned}
$$

Given a pair of domains T_i, T_j, with T_i coarser grain of T_j, for each time-point t_j of T_j we also define as its *coarse grain equivalent* on T_i the time-point t_i of T_i such that t_j belongs to the time-interval obtained by applying the corresponding decomposition function to t_i. The uniquity of the coarse grain equivalents can be easily deduced from the definition of the decomposition functions. Coarse grain equivalent functions can also be defined using the predicate *value-metric* as follows:

$$
\begin{aligned}
&coarse\_grain\_of(t2,t1)\leftarrow\\
&\quad value\_metric(t1,m1,v1)\wedge\\
&\quad value\_metric(t2,m2,v2)\wedge\\
&\quad m1\leq m2\wedge v2=(v1*m1)//m2
\end{aligned}
$$

where $(v1*m1)//m2$ denotes the integer division of $(v1*m1)$ by $m2$.

The relationships of temporal ordering can be generalized to make it possible to compare two time-points belonging to different temporal domains as follows:

$$
\begin{aligned}
&is\_after(t2,t1)\leftarrow\\
&\quad value\_metric(t1,m,v1)\wedge\\
&\quad coarse\_grain\_of(t,t2)\wedge\\
&\quad value\_metric(t,m,v)\wedge\\
&\quad v1<v
\end{aligned}
$$

$$
\begin{aligned}
&is\_after(t2,t1)\leftarrow\\
&\quad value\_metric(t2,m,v2)\wedge\\
&\quad coarse\_grain\_of(t,t1)\wedge\\
&\quad value\_metric(t,m,v)\wedge\\
&\quad v<v2
\end{aligned}
$$

The *is_before* predicate can be easily defined in a similar way.

The coarse grain equivalent and the decomposition functions can be viewed as forms of simplification and articulation along the dimension of temporal aggregation, i.e. shifts in focus through part-whole relationships among time-points, respectively. They define distinguishability and undistinguishability relations between any pair of time-points with respect to each domain of the temporal universe.

3.2 Events and Properties in the Temporal Universe

Let us now locate events and properties in the temporal universe. The idea is to directly associate a time granularity with events and to derive the granularity of properties on the basis of the *initiates* and *terminate* relations.

First of all, we give a characterization of events with respect to the temporal universe. With respect to *a given domain*, we distinguish *instantaneous events*, that happen at a time-point, and *events with duration*, that take place over a nonpoint time-interval. Such a distinction among events is a *relative* one, so, for example, passing from a given domain to a finer (coarser) one an instantaneous (with duration) event may become an event with duration (instantaneous).

With respect to *the temporal universe*, we distinguish finite and infinitesimal events. An event is said *finite* if there exists a domain with respect to which it has duration. A finite event thus identifies an implicit level of time granularity: at this level and coarser ones, it is an instantaneous event; at finer levels it is of finite duration. We define such a threshold the *intrinsic time granularity* of the event. An event is said *infinitesimal* if it is instantaneous with respect to every domain[4]. Infinitesimal events are needed for dealing with continuous change [Shanahan 1990]. Let us consider, for instance, a process of continuous change such as sink filling with water. We might associate the occurrence of an event with each new level reached by the filling fluid. If we did this, then there would be no limit to how fine we might choose our temporal grain in order for the events to remain instantaneous. Thus taking this approach we have a need for infinitesimal events. Differently from the previous one, such a distinction among events is an *absolute* one.

To be able to deal with instantaneous events only, we impose that every event is associated with a domain whose granularity is equal to or coarser than the intrinsic one of the event. In such a way, Event Calculus axioms can be still used to reason *within domains*. On the contrary, they are insufficient by themselves to deal with events associated with different domains (*differently grained events*). However, reasoning *across domains* can be brought back to reasoning within domains provided that there exist some rules to relate differently grained events to the same domain. The idea is to integrate macro-events (section 3.3) and continuous change (section 3.4) mechanisms with time granularity, and to define general temporal projection

[4]The absolute instantaneousness of infinitesimal events copes with the same representational problems that suggested to Hayes and Allen the introduction of short time periods (moments) in Allen's Interval Logic [Hayes and Allen 1987].

rules (section 4) that are used by default when neither macro-events nor continuous change decompositions are explicitly given.

3.3 Refining Macro-Events

We define a *unifying framework* for the packaging of events and the granularity of time to describe the temporal relationships between a macro-event and its components. We require that the intrinsic time granularity of a macro-event is coarser than the ones of its sub-events and that its occurrence time is a coarse grain equivalent of the occurrence time of all its sub-events. We also define a number of general operators, called *macro-event constructors*, for specifying temporal relationships among sub-events [EQUATOR 1991](we use the infix notation for macro-event constructors for the sake of simplicity)[5]:

;	sequence
;delay(min,max)	minimum and maximum delay between two events
|	alternative
||	parallelism
$*n$	sequential repetition (n is optional)
$**n$	parallel repetition (n is optional)
[ ]	composition

Let us report here the Event Calculus axiomatization of the basic operators expressing sequence, alternative, and parallelism.

$$happens\_at(e1;e2,t) \leftarrow happens\_at(e1,t1) \wedge happens\_at(e2,t2) \wedge coarse\_grain\_of(t,t1) \wedge coarse\_grain\_of(t,t2) \; is\_after(t2,t1)$$

$$happens\_at(e1|e2,t) \leftarrow happens\_at(e1,t) \wedge not\ happens\_at(e2,t)$$

$$happens\_at(e1|e2,t) \leftarrow not\ happens\_at(e1,t) \wedge happens\_at(e2,t)$$

$$happens\_at(e1||e2,t) \leftarrow happens\_at(e1,t) \wedge happens\_at(e2,t)$$

In general, domain axioms include definition of macro-events in terms of a suitable composition of sub-events. An example of these domain axioms is:

$$happens\_at(e,t) \leftarrow happens\_at([e1;[e2||e3]],t) \wedge part\_of(e1,e) \wedge part\_of(e2,e) \wedge part\_of(e3,e)$$

[5]Dealing with the repetition operators may require the addition of a domain composed of a single time point to the temporal universe (absolutely coarsest domain).

The operator expressing sequence deserves further consideration. It allows us to deduce the occurrence of a macro-event at a time-point of a domain coarser than the domain(s) the occurrence times of its component events (possibly macro-events in their turn) belong to. Such a time-point is a coarse grain equivalent of both the occurrence times of components. Then, the rule for sequential macro-events first executes a comparison of time-points with respect to the finer domain and then it abstracts them into a time-point of the coarser one. The presence of this switching to a coarser domain makes the definition of sequential macro-events *incomplete*. Consider the following example. Given the occurrences of three events $e1$, $e2$ and $e3$ at time-points $2hrs15m$, $2hr42m$ and $2hrs50m$, respectively, we are not able to deduce the occurrence of a sequential event $e1;[e2;e3]$ at time-point $2hrs$ when the temporal universe is {..., hours, minutes, ...}. In fact, there is no way of strictly ordering $e1$ and the macro-event into which $e2$ and $e3$ can be abstracted, because the occurrence time of the macro-event is a coarse grain equivalent of the occurrence time of $e1$. To make it possible to derive the occurrence the macro-event $e1;[e2;e3]$, the temporal universe has to be extended with the domain of 30-minutes (similar considerations hold for the macro-event $[e1;e2];e3$). However, it is easy to find another sequence that cannot be abstracted into a sequential macro-event with respect to the extended temporal universe too. Such an incompleteness is due to the fact that mappings between temporal domains are fixed once and for all and then is inherent to the upward temporal projection involved in macro-event derivation rules (section 4.1).

3.4 Refining Continuous Change

The original approach to continuous change makes the assumptions that the parameters of the trajectory function are set not after $t1$ and are not reset between $t1$ and $t2$. In general, these assumptions are too restrictive. Mechanisms are requested for resetting the parameters of the trajectory function. This allows it to be initiated with parameters values at the start of the property, but also allows the parameters to be changed during the interval of validity of the property. In this way, the trajectory may model 'non-linearities' (e.g. a change in the rate of a linear increase of a temperature) without interrupting the relevant dynamic property (e.g. by splitting a 'temperature rising' property when the rate of rise changes).

To take into account the resetting of parameters the original axiom can be replaced by the following one:

$holds\_at(p,t) \leftarrow$
$value\_metric(t,m,v) \wedge$
$happens\_at(e,t1) \wedge value\_metric(t1,m,v1) \wedge$
$v1 < v \wedge initiates(e,q) \wedge$
$not\ broken\_during(q,\langle t1,t\rangle) \wedge$
$happens\_at(e',t2) \wedge value\_metric(t2,m,v2) \wedge$
$v2 < v \wedge initiates(e',par) \wedge$
$not\ broken\_during(par,\langle t2,t\rangle) \wedge$
$max(t1,t2,ti) \wedge trajectory(q,par,ti,p,t)$

A continuously changing property p can be assigned a given value at a time point t if an instance of the relevant dynamic property q is initiated at a time point t1 (before t) and not broken at some point between t1 and t, the relevant parameter par is (re)set at a time point t2 (before t) and not broken at some point between t2 and t, and the initial value of p is calculated (by the *trajectory* predicate) at the time point ti which is the maximum between t1 and t2 and the 'max' predicate has the obvious definition. The crankshaft example of section 2.2 must be rewritten according to the revisited axiom as:

$trajectory(rotating, velocity(\omega), ti, angle(a), t) \leftarrow$
$value\_metric(ti,m,vi) \wedge value\_metric(t,m,v) \wedge$
$holds - at(angle(ai),ti) \wedge a = \omega(v - vi) + ai$

The indirect recursion on the predicate *trajectory* (or, equivalently, on the predicate *holds_at*) stops when the initial values of the configuration variables, e.g. the angular position, are reached. They can be explicitly asserted or derived from the occurrence of independent events.
The application of the refined axiom for continuous change is not restricted to discrete resetting of parameters; it can be used to deal with continuously changing parameters too. In such a case, the occurrence of the *continuous events* of resetting can be derived from the continuous change of the configuration variables by means of appropriate domain axioms.
Continuous events can be either acquired by the external environment or computed according to explicit laws. In both cases, we generally need to plot them at regular time intervals to make the model computable. Choosing the width of the time interval is equivalent to choosing the time granularity at which describing the process. Then, a change in the frequency of plotting is equivalent to the switching of a continuous process from one time granularity to another.

4 Reasoning with Time Granularity

We distinguished two basic modalities of relating differently grained events, namely upward and downward temporal projections. Upward (downward) projection determines the temporal relations set up by two events e_i and e_j which occur at the time-points $t_i \in T_i$ and $t_j \in T_j$, respectively, with T_i coarser than T_j, by upward (downward) projecting e_j (e_i) on T_i (T_j).

4.1 'Naive' Upward Projection

The 'naive' upward projection is a quite straightforward approach to abstractive temporal reasoning. It states that the upward projection of an event e that occurs at a time-point t of a domain T_j on a domain T_i, coarser than T_j, is accomplished by simply replacing t with its coarse grain equivalent on T_i [Evans 1990]. Then the temporal ordering and distance between two events e_i and e_j which occur at the time-points $t_i \in T_i$ and $t_j \in T_j$, respectively, are determined on the basis of the relation between t_i and the coarse grain equivalent of t_j on T_i. Moreover, if e_i (e_j) precedes e_j (e_i) then the properties initiated by e_i (e_j) and terminated by e_j (e_i) hold over the time-interval of T_i identified by t_i and the coarse grain equivalent of t_j. To formalize upward projection in the Event Calculus, we first extend the definition of the occurrence time of an event as follows:

$happens\_at(e,t1) \leftarrow$
$happens\_at(e,t2) \wedge coarse - grain - of(t1,t2)$

In this way, each event is endowed with several occurrence times belonging to different domains, i.e. the time-point at which it originally occurs and all the coarse grain equivalents of such a point. Combined with the macro-event derivation rules, upward projection allows us to deduce the occurrence of parallel and alternative macro-events at time-points of domains coarser than the domains at which occur their components.
Upward projection can be seen as a *simplification* rule [Hobbs 1985], because it allows us to derive a relation of temporal indistinguishability, i.e. simultaneity, among events from the relation of indistinguishability among time-points defined by coarse grain equivalent functions.

Then, the *Mholds-for* predicate is redefined to constrain the starting and the ending time-points of the time-interval to belong to the same domain:

$Mholds\text{-}for(p, \langle start,end\rangle) \leftarrow$
$happens\_at(e,start) \wedge initiates(e,p) \wedge$
$value\text{-}metric(start,m,vs) \wedge$
$happens\_at(e',end) \wedge terminates(e',p) \wedge$
$value\text{-}metric(end,m,ve) \wedge$
$vs < ve \wedge not\ broken\text{-}during(p, \langle start,end\rangle)$

together with a similar axiom for the predicate *broken-during*.
In such a way, the predicate *Mholds-for* identifies several time-intervals of differents domains over which the

properties initiated and terminated by two differently grained events hold.

In despite of its apparent simplicity upward projection involves a number of semantic assumptions. The most relevant one is related to its application to contradictory events, i.e. events that cannot occur simultaneously. We formally define two events as contradictory if they initiate or terminate incompatible properties. The definition of the relation of incompatibility among properties depends on domain-specific knowledge [Kowalski and Sergot 1986].

Upward projection maintains the weak temporal ordering between events, but it does not always preserve the strict one. Then the logical consistency of the upward projection cannot be guaranteed in the general case, because it may enforce contradictory events to occur at the same time-point in a coarser domain. As a consequence, if two differently grained events are contradictory the coarse grain equivalent of the occurrence time of the fine grained event must be different from the occurrence time of the coarse-grained event. This is guaranteed by the following integrity constraint[6]:

$$\leftarrow happens\_at(e1,t) \wedge happens\_at(e2,t) \wedge contradictory(e1,e2)$$

Moreover, upward projection may change the ratio between the width of time-intervals. That is, given two domains T_i and T_j, with T_i coarser than T_i, the coarse grain equivalents on T_i of two pairs of time-points of T_j which are at the same temporal distance may be at a different one, while the coarse grain equivalents on T_i of two pair of time-points of T_j that are at a different temporal distance may be at the same one.

Such a weakness of the 'naive' upward projection will be overcome refining upward projection according to the downward projection schema we are going to define.

4.2 Downward Projection

The downward projection of an event e that occurs at a time-point t of a domain T_i on a domain T_j finer than T_i is accomplished by applying the following decomposition scheme: for each event e that occurs at a time-point t of T_i there exist two infinitesimal events e_i and e_f that occur at the time-points t_i and t_f of T_j, respectively, and such that (i) $t_i \leq t_f$; (ii) t is the coarse grain equivalent on T_i of both t_i and t_f; (iii) for each property p such that p is terminated by e, there exist an event e_p that occurs at t_p of T_j such that e_p terminates p and $t_i \leq t_p \leq t_f$; (iv) for each property q such that q is initiated by e, there exist an event e_q that occurs at t_q of T_j such that e_q initiates q and $t_i \leq t_q \leq t_f$; (v) the (type of the) event e becomes a dynamic property that is initiated by e_i and terminated by e_f with respect to T_j. Because of an event is defined by the properties that it initiates and terminates, such rules provide the definition of the component events e_i, e_f, e_p and e_q[7].

Downward projection can be seen as an *articulation* rule [Hobbs 1985]. From the relation of distinguishability among time-points of the finer domain introduced by the decomposition function, in fact, it derives a relation of temporal distinguishability among the subevents of a given finite event.

Let us formalize this scheme in the Event Calculus. First of all, we define two functions *begin* and *end* that map a given instance of a macro-event into its initiating and terminating events, respectively. The occurrence of such events can be deduced from the occurrence of the macro-event by means of the following axioms:

$$happens\_at(begin(e,t), time(begin(e,t))) \leftarrow happens\_at(e,t) \wedge coarse-grain-of(t, time(begin(e,t)))$$

$$happens\_at(end(e,t), time(end(e,t))) \leftarrow happens\_at(e,t) \wedge coarse-grain-of(t, time(end(e,t)))$$

together with (condition (ii)):

$$coarse-grain-of(t, time(begin(e,t)))$$
$$coarse-grain-of(t, time(end(e,t)))$$

where $time(begin(e,t))$ and $time(end(e,t))$ denote the occurrence times of $begin(e,t)$ and $end(e,t)$, respectively.

Condition (i) is expressed by the following integrity constraint:

$$\leftarrow is\_after(time(begin(e,t)), time(end(e,t)))$$

Let us now represent e_p and e_q by means of two functions *term* and *in*. For each property p (q), *term* (*in*) maps each instance of a given macro-event into the component event that terminates (initiates) such a property. Using these functions, conditions (iii) and (iv) are codified by the following axioms:

[6] This solution can be generalized by making contradiction dependent on granularities or even on time instants. In such a way, simultaneous occurrence of two events can be classified as contradictory in certain domains, or even in certain time instants of them, only.
The relevant integrity constraint becomes:

$$\leftarrow happens\_at(e1,t) \wedge happens\_at(e2,t) \wedge contradictory(e1,e2,t)$$

[7] When t_i and t_f coincide, the events e_i, e_p, e_q and e_f are merged into the original single event e. This is always the case of the downward projection of infinitesimal events. For instance, the infinitesimal event of switching on the light remains instantaneous with respect to all the domains of the temporal universe composed of {Day, Hour, Minute}.

$terminates(term(e,t,p),p) \leftarrow terminates(e,p)$
$initiates(in(e,t,q),q) \leftarrow initiates(e,q)$

together with:

$\leftarrow is\_after(time(begin(e,t)),time(term(e,t,p))) \vee$
$is\_before(time(end(e,t)),time(term(e,t,p)))$

$\leftarrow is\_after(time(begin(e,t)),time(in(e,t,q))) \vee$
$is\_before(time(end(e,t)),time(in(e,t,q)))$

for each property p and q.
Finally, condition (v) is expressed by the following axioms:

initiates(begin(e,t),e)
terminates(end(e,t),e)

They allow us to state that the property e holds over $\langle time(begin(e,t)), time(end(e,t))\rangle$ by means of the *Mholds-for* axiom. These last axioms provide each temporal object with a twofold event/property characterization. That is, (the type of) an event e, associated with a given domain, may become a dynamic property with respect to a finer domain, and vice versa.
Let us consider, as an example, the event of flying from Milan to Venice. With respect to the domain T_H of *hours* it can be modeled as an instantaneous event that occurs at a time-point t of T_H. Such an event *terminates* the property of *being in Milan* and *initiates* the property of *being in Venice*. With respect to the domain T_M of *minutes*, it can be decomposed into a pair of infinitesimal events $flying_i$ and $flying_f$ that occur at the time-points t_i and t_f of T_M, respectively, with $t_i \leq t_f$, and such that t is the coarse grain equivalent of both. Moreover, $flying_i$ *terminates* the property of *being in Milan* and *initiates* the property of *flying*, while $flying_f$ *terminates* the property of *flying* and *initiates* the property of *being in Venice*.

4.3 'Revised' Upward Projection

The event/property duality introduced by downward projection suggests an extension of the upward projection rules to cope with contradictory events without restrictions. When the coarse grain equivalents of two contradictory events coincide the downward projection schema suggests to merge and replace the events by a *macro-event* corresponding to the conjunction of the properties initiated by the first one and terminated by the second one. Moreover, such a macro-event terminates (initiates) all the properties terminated (initiated) by its first (second) component and every property terminated (initiated) by the second (first) component which is not initiated (terminated) by the first (second) component.
Let us consider, as an example, the events of *leaving station A* and *arriving at station B* of a train. The first one *terminates* the property of the train of *being at station A* and *initiates* the property of *moving*, while the second one *terminates* the property of *moving* and *initiates* the property of *being at station B*. Let be T a domain with respect to which the two events are simultaneous. According to the revised upward projection rules they are merged and replaced by the event of *moving* that *terminates* the property of *being at station A* and *initiates* the property of *being at station B*.

The actual structure of the corresponding macro-event can be given in terms of a *suitable composition* of the component events using macro-event constructors. Consider two contradictory events $e1$ and $e2$. If their temporal ordering is known and meaningful, e.g. $e1$ precedes $e2$, then the corresponding macro-event e is a sequential one, that is, $e1;e2$; if their temporal ordering is meaningless (their global effect does not change even if their ordering changes), and possibly unknown, then the corresponding macro-event is a parallel one, that is $e1||e2$; if their temporal ordering is meaningful and unknown, then the corresponding macro-event is $[e1||e2]|[[e1;e2]|[e2;e1]]$; and so on.
The last one is the case, for instance, of events of rotation around orthogonal axes in the three dimensional space which are not commutative, that is, the final configuration of the rotating system depends on the ordering of their occurrences.

5 Conclusion

The paper made a proposal for embedding the notion of time granularity into a logic-based representation language. Firstly, it enumerated a number of notational and computational reasons that motivate the introduction of time granularity and briefly surveied and discussed the existing relevant literature. Successively, it extended the Event Calculus to deal with time granularity by introducing the concepts of temporal universe, finite and infinitesimal events, macro-event, and continuously changing events and properties. Finally, it provided Event Calculus with the axioms supporting upward and downward temporal projection.

Acknowledgements

We would like to thank Chris Evans of Goldsmiths' College, University of London, and Murray Shanahan of Imperial College for the useful discussions we had with them.
The research for this paper was partially funded by the European Community ESPRIT Program, EQUATOR Project no. 2409 [EQUATOR 1991]. Collaborating organizations are CENA (France), CISE (Italy), EPFL (Switzerland), ERIA (Spain), ETRA (Spain), Ferranti

Computer Systems Ltd. (UK), Imperial College (UK), LABEN (Italy), Politecnico di Milano (Italy), SWIFT (Belgium), SYSECA (France), UCL (UK). The work of CISE was partially funded by the Automatica Research Center (CRA) of the Electricity Board of Italy (ENEL) within the VASTA project too.

References

[Allen 1983] Allen, J., *Maintaining Knowledge about Temporal Intervals*; Communications of the ACM, 26, 11, 1983.

[Allen 1984] Allen, J., *Toward a General Theory of Action and Time*, Artificial Intelligence, Vol. 23, No. 2, July 1984.

[Clifford and Rao 1988] Clifford,J., Rao,A., *A Simple, General Structure for Temporal Domains* in Temporal Aspects in Information Systems, Rolland, C., Bodart, F., Leonard, M. (Eds.), IFIP 1988.

[Ciapessoni *et al.* 1992] Ciapessoni,E., Corsetti, E., Montanari,A., San Pietro,P., *Embedding Time Granularity in a Logical Specification Language for Synchronous Real-Time Systems*; submitted to Science of Computer Programming, North-Holland, January 1992.

[Corsetti *et al.* 1990] Corsetti, E., Montanari, A., Ratto, E., *A Methodology for Real-Time System Specifications based on Knowledge Representation*; in Computational Intelligence, III, N. Cercone, F. Gardin, G. Valle (Eds.), North-Holland, Proc. of the International Symposium, Milan, Italy, 24-28 September, 1990.

[Corsetti *et al.* 1991a] Corsetti, E., Montanari, A., Ratto, E., *Dealing with Different Time Granularities in Formal Specification of Real-Time Systems*; The Journal of Real-Time Systems, Vol. III, Issue 2, June 1991.

[Corsetti *et al.* 1991b] Corsetti, E., Montanari, A., Ratto, E., *Time Granularity in Logical Specifications*, Proc. 6th Italian Conference on Logic Programming, Pisa, Italy, June 1991.

[Corsetti *et al.* 1991c] Corsetti, E., Crivelli, E., Mandrioli, D., Montanari, A., Morzenti, A., San Pietro, P., Ratto, E., *Dealing with Different Time Scales in Formal Specifications*; Proc. 6th International Workshop on Software Specification and Design, Como, Italy, October 1991.

[Dean and Boddy 1988] Dean, T., Boddy, M., *Reasoning about partially ordered events*; Artificial Intelligence, 36, 1988.

[Dean *et al.* 1988] Dean, T., Firby, R., Miller, D., *Hierarchical Planning involving deadlines, travel time and resources*; Computational Intelligence, 4, 1988.

[Dean 1989] Dean, T., *Using Temporal Hierarchies to Efficiently Maintain Large Temporal Databases*; Journal of ACM, 36, 4, 1989.

[Evans 1990] Evans, C., *The Macro-Event Calculus: Representing Temporal Granularity*; Proc. PRICAI, Japan 1990.

[Fum *et al.* 1989] Fum, D., Guida, G., Montanari, A., Tasso, C., *Using Levels and Viewpoints in Text Representation*; in Artificial Intelligence and Information-Control Systems of Robots - 89, North-Holland, I. Plander (Ed.), Proc. 5th International Conference, Strbske Pleso, Czechoslovakia, 6-10 November, 1989.

[Galton 1987] Galton, A., *The Logic of Occurrence*; in Temporal Logics and their applications, Galton A., (Ed.), Academic Press, 1987.

[Giunchiglia and Walsh 1989] Giunchiglia, F., Walsh, T., *Abstract Theorem Proving*; Proc. 11th IJCAI, Detroit, USA, 1989.

[Greer and McCalla 1989] Greer, J., McCalla, G., *A Computational Framework for Granularity and its Application to Educational Diagnosis*; Proc. 11th IJCAI, Detroit, USA 1989.

[EQUATOR 1991] *Formal Specification of the GRF and CRL*; CISE, FERRANTI, SYSECA (Eds.), ESPRIT Project no. 2409 EQUATOR, Deliverable D123-1, 1991.

[Hayes and Allen 1987] Hayes, P., Allen, J., *Short Time Periods*; Proc. 10th IJCAI, Milano, Italy 1987.

[Hobbs 1985] Hobbs, J., *Granularity*; Proc. 9th IJCAI, Los Angeles, USA 1985.

[Kautz and Allen 1986] Kautz, H., Allen, J., *Generalized Plan Recognition*; Proc. AAAI, 1986.

[Kowalski and Sergot 1986] Kowalski, R., Sergot, M., *A Logic-based Calculus of Events*; New Generation Computing, 4, 1986.

[Maim 1991] Maim, E., *Reasonig with Different Granularities in CRL*; Proc. IMACS '91, Dublin, Ireland, July 1991.

[Maim 1992a] Maim, E., *Uniform Event Calculus with Constructive Negation*; to appear in Artificial Intelligence, Expert Systems, and Symbolic Computing, Houstis,E.N., and Rice,J.R. (Eds.), Elsevier North-Holland, 1992 (a reprint of this paper will be sent by the author on request).

[Maim 1992b] Maim, E., *Abduction and Constraint Logic Programming*; to be presented at the Logic Programming in AI ESPRIT CompulogNet Workshop, London, 23-24 March, 1992.

[Montanari *et al.* 1991] Montanari, A., Ratto, E., Corsetti, E., Morzenti, A., *Embedding Time Granularity in Logical Specification of Real-Time Systems*; Proc. 3rd IEEE Euromicro Workshop on Real-Time Systems, Paris, France, June 1991.

[Plaisted 1981] Plaisted, D., *Theorem Proving with Abstraction*; Artificial Intelligence, 16, 1981.

[Sergot 1990] Sergot, M., *(Some Topics in) Logic Programming in AI*; Advanced School on Foundations of Logic Programming, Alghero, Italy, 1990.

[Shanahan 1990] Shanahan, M., *Representing Continuous Change in the Event Calculus*; Proc. 9th ECAI, Stockholm, Sweden 1990.

[Shanahan 1991] Shanahan, M., *Towards a Calculus for Temporal and Qualitative Reasoning*; Proc. AAAI Symposium, Stanford, 1991.

[Shoham 1988] Shoham, Y., *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence*; MIT Press, 1988.

ARCHITECTURES & SOFTWARE

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

UNIREDII: The High Performance Inference Processor for the Parallel Inference Machine PIE64

Kentaro Shimada Hanpei Koike Hidehiko Tanaka
H.Tanaka Lab., Department of Electrical Engineering,
Faculty of Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
E-Mail:{shimada,koike,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract

UNIREDII is the high performance inference processor of the parallel inference machine PIE64. It is designed for the committed choice language Fleng, and for use as an element processor of parallel machines. Its main features are: 1) a tag architecture, 2) three independent memory buses (instruction fetching, data reading, and data writing), 3) a dedicated instruction set for efficient execution of Fleng, 4) multi-context processing for reducing pipeline interlocking and overhead of inter-processor synchronization. With the multi-context processing mechanism, the internal pipeline is shared by several independent instruction streams (contexts), and which contexts are to be executed is determined cycle by cycle. So, UNIREDII acts as a pipeline-shared MIMD processor. In this paper, several architectural features and the instruction set are explained. And performance measurement results by simulation are also presented. High performance (about 1M RPS[1] with 10MHz clock) is attained, and it is shown that the multi-context processing mechanism is very effective for improved performance.

1 Introduction

Committed choice languages are designed for efficient parallel execution of logic programs, but, because of their parallel and logic semantics, high performance is hardly achieved by conventional processors which are designed for sequential and procedural languages. Therefore we designed a dedicated processor for the element processor of the parallel inference machine PIE64 [Koike and Tanaka 1988], which we are now developing, and named UNIREDII. PIE64 executes committed choice language Fleng[Nilsson and Tanaka 1988], and has sixty-four processor elements.

For design decisions in UNIREDII, we paid special attention to the following points.

1. The processor architecture should be tuned for the execution of Fleng programs.
2. It must be equipped with the features of an element processor of parallel machines.

For the first point, we designed a dedicated instruction set for executing Fleng based on the experience of developing Fleng interpreters on workstations. For the second point, we proposed the multi-context processing mechanism for reducing inter-processor synchronization, and the independent coprocessor command bus to interconnect network interface processors and a process management processor.

UNIREDII is implemented in 1.2 μ CMOS gate array and driven with 10MHz clock. This clock rate is selected because UNIREDII must synchronize with the local memory bus, which in turn synchronizes with the network hardware of PIE64.

An overview of PIE64 is given in section 2. In section 3, the architecture including the multi-context processing mechanism and instruction set of UNIREDII are described. In section 4 and 5, some simulation results are presented and discussed. Finally, we conclude this paper in section 6.

2 PIE64

PIE64 is one of several parallel inference machines which executes parallel logic programming languages such as Fleng[Nilsson and Tanaka 1988] and KL1[Kimura and Chikayama 1987]. We designed the committed choice language Fleng for PIE64 so that it is easy to implement and easy to attain high performance. PIE64 has sixty-four processor elements, which are called inference units (IUs), and two independent interconnection networks (Process Allocation Network: PAN and Data Access/Allocation Network: DAAN)[Takahashi *et al.* 1991]. These interconnection networks are implemented as circuit switching, and have a special function of broadcasting load information for automatic load balancing so that each IU can select the minimum load IU automatically.

[1]RPS: Reduction Per Second

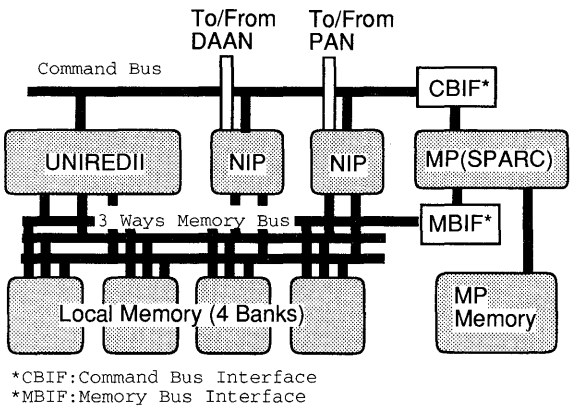


Figure 1: organization of the inference unit of PIE64

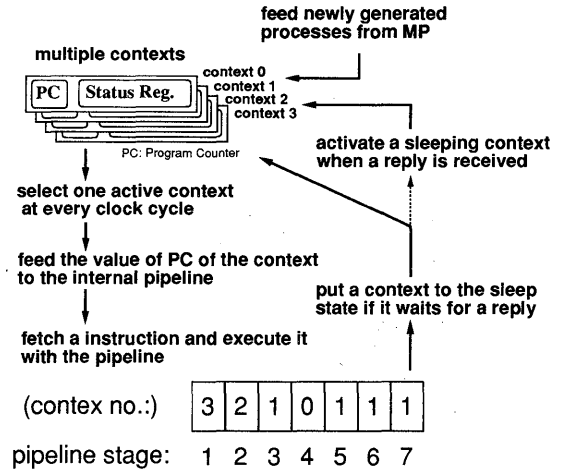


Figure 2: overview of the multi-context processing mechanism

Each IU has an inference processor UNIREDII, two[2] network interface processors(NIPs) [Shimizu *et al.* 1991], a management processor(MP), and a local memory of four banks (see figure 1). NIP, which is a dedicated processor as well as UNIREDII, performs inter-IU communicating/synchronizing functions in a form suitable for Fleng execution. The transmission throughput is 10M word per sec., namely 40MByte per sec. at one connection for each network. MP manages process scheduling, load distribution, and load balancing, and performs other functions such as system maintenance. We use a general purpose processor, SPARC, as MP to make process management flexible. Thus, in the IUs, we use functional parallelism by the three kinds of processor, where UNIREDII performs *computation*, NIP performs *communication and synchronization*, and MP performs *process management*.

In an IU, these three kinds of processors share the local memory, and access it through a three way memory bus which is driven with a 10MHz clock to synchronize with network access over NIPs. So we can get a throughput of 10M word (40MByte) per sec. each way, namely 120MByte per sec. in all. In addition, UNIREDII, NIPs, and MP communicate with each other through a coprocessor command bus using a specialized protocol for command-reply among these processors. The format of the coprocessor commands is determined with the Fleng data types taken into account.

[2]In practice, there are four network interface processor chips in one IU. Two of them act in master mode and can start the action of network connection while the other two act in slave mode and respond to the master NIP when requested.

3 UNIREDII Architecture

3.1 Overview

UNIREDII is a dedicated processor. It was designed for executing Fleng programs efficiently and to meet requirements of an element processor for parallel machines. Its main features are:

1. a tag architecture
2. three independent memory buses (instruction fetching, data reading, and data writing)
3. multi-context processing
4. a dedicated instruction set to execute Fleng programs efficiently

All instructions of UNIREDII have one word (32 bits) length, and are single-cycle instructions. Also its data types have a length of 32 bits, and consist of two mark bits for garbage collection, two tag bits, and twenty-eight bits of value part (pointer types); or two mark bits, six tag bits, and twenty-four bits of value part (constant types).

3.2 Multi-Context Processing

The internal pipeline of UNIREDII is shared by multiple instruction streams (contexts). UNIREDII executes them concurrently, and which contexts should be executed is determined cycle by cycle. In other words, UNIREDII acts as a pipeline-shared MIMD processor (see figure 2). Because Fleng is a parallel language which generates many instruction streams in parallel, we can expect to get enough instruction streams to fill the contexts of UNIREDII. Process scheduling is determined by the management processor, and UNIREDII starts a new process as one of the contexts by receiving a appropriate

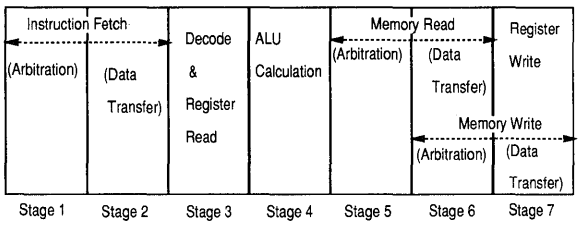


Figure 3: pipeline organization of UNIREDII

coprocessor command from MP. And when one of the contexts waits for a reply of some remote memory access, UNIREDII puts the context to sleep state and fills its pipeline with the other contexts dynamically.

Two kinds of aim of the multi-context processing exist. 1) To reduce pipeline interlocking caused by pipelined executions of the instructions (intra-processor effect). 2) To reduce the cost of process switching due to remote memory access (inter-processor effect). Especially, the second effect is a very important feature for an element processor of parallel machines.

The most remarkable point of the multi-context processing mechanism of UNIREDII is that it can continuously execute instructions of only one context while the other contexts are sleeping (as shown at the stage from 5 to 7 in the figure 2). UNIREDII has a pipeline interlocking mechanism to keep dependency among instructions of one context. Other processors which have similar mechanisms, such as HEP[Jordan 1983], MASA [Halstead and Fujita 1988], and CPC[Shimuzu *et al.* 1989], do not have pipeline interlocking mechanism and cannot execute one instruction stream in continuous cycles. Therefore they slow down dramatically as the number of executable instruction streams decrease. That is not the case of UNIREDII.

Due to the restriction of available number of gates, the number of contexts is limited to four. But that is enough to get the full effect of multi-context processing, as we show later.

3.3 Hardware Organization

3.3.1 Pipeline Organization

Figure 3 shows the internal pipeline organization of UNIREDII. The pipeline consists of seven stages. The main reason why as many stages as seven are required is that, in an IU, all memory access needs two phases, one of which is the bus arbitration phase in which the four kinds of independent accesses from UNIREDII, two NIPs, and MP are arbitrated (see figure 1), and another is the data transfer phase in which memory access is actually performed, so that the memory access time hides the bus arbitration time. Thus, at the first and the second stages, UNIREDII fetches an instruction from the local memory, decodes it and reads registers at the third stage, executes it at the fourth stage, reads data from the memory at the fifth and the sixth stages, and writes data into the memory at the sixth and the seventh stages. Also registers are written at the seventh stage. UNIREDII has thirty-two general purpose registers per context, namely 128 general purpose registers in all. By means of this pipeline organization, UNIREDII makes efficient use of its three memory buses and can execute test-and-set type instructions which require two times of memory access (one read and one write) in one cycle without any pipeline holding because the data reading and writing buses are handled by different stages. These type instructions are very important for processing elements of parallel machines.

The effectiveness of the pipeline architecture is determined by when and where the pipeline interlock occurs. As for UNIREDII, the pipeline interlock will occur when not all contexts can be executed. In that case, taking a jump (at the fourth stage) will invalidate at most three following, already-fetched instructions of the same context at the first, the second, and the third stages. And reading registers (at third stage) which are destinations of load instructions executed within three preceding cycles will cause pipeline hold. When all four contexts can be executed, no pipeline interlock occurs because instructions of the same context are not executed and registers of the same context are not read in any four continuous cycles.

3.3.2 Mechanism of Remote Memory Access

In principle, memory-accessing instructions of UNIREDII can execute remote memory access automatically. UNIREDII has a special register which holds the IU identifier of six bits, and it compares the IU identifier field of the memory address (the upper six bits of the address of twenty-eight bits) with the IU-id register when executing memory access instructions. When they are not equal, UNIREDII issues a remote memory access command to NIP instead of accessing its local memory. The result of the remote memory access is sent back as a coprocessor reply from NIP.

UNIREDII should receive the replies of the remote memory access commands correctly. For this purpose, all general purpose registers of UNIREDII have a special bit indicating that they are waiting for replies or not. When an instruction reads the register whose reply wait bit is set before the reply is received, UNIREDII cancels the instruction and puts its context to sleep. And, after receiving the reply, UNIREDII wakes the context up and re-executes the canceled instruction.

Table 1: the instruction set of UNIREDII

Dereference	derf	dereference	dfcl	dereference and check list
	dcll	dereference and check list and load car	dfcv	dereference and check vector
	dcvl	dereference and check vector and lod top	dfcc	dereference and check constant
Execute	exec	execute	exel	execute on list
	exll	execute on list and load car	exev	execute on vector
	exvl	execute on vector and load top	exct	execute on constant
Manipulate Structure	cpir	copy if remote	cprr	copy if remote with register
	cvtp	check vector top	cvtr	check vector top with register
Load / Store	ld	load	tgld	tagged load
	ldst	load and store	tlds	tagged load and store
	st	store	sudf	store undefind code
	stim	store immediate		
Active Unification	bind	bind variable	bdim	bind with immediate
	cvos	check variable order and swap		
Heap Allocation	allc	allocate		
Flow Control	jump	jump	call	call
	jcmp	jump on compare	jncp	jump on not compare
	jtag	jump on tag	jntg	jump on not tag
	jrmt	jump on remote pointer	jloc	jump on local pointer
	jcc	jump on flag condition	stop	stop
Coprocessor	cpcm	send coprocessor command		
Garbage Collection	ldsm	load and store mark	stmm	store with modified mark
	jmrk	jump on mark/stop condition		
Set	setc	set constant	sett	set mark and tag
	seta	set alternative pointer	setf	set condition flags
	clrf	clear condition flags	gfc	get flag condition
Arithmetic and Logical	add	add	adc	add with carry
	sub	subtract	sbb	subtract with borrow
	and	bit-wise and	or	bit-wise or
	xor	bit-wise exclusive or	rol	rotate left
	ror	rotate right	rcl	rotate with carry left
	rcr	rotate with carry right	shl	shift left
	shr	shift right	asr	arithmetic shift right
Management	spid	set pid register	pidt	preset pid and tag
	shp	set heap pointer	lhp	load from heap pointer
	exhp	exchange heap pointer set	stp	set scratch area pointer
	ltp	load from scratch area pointer	stf	store flags
	ldf	load from flags		

3.4 Instruction Set

Table 1 shows the instruction set of UNIREDII. We describe several notable instructions of it in the following subsections.

3.4.1 Dereference Instructions

The dereference instructions are most characteristic of the instruction set of UNIREDII. They dereference links of a variable and get the value of the variable. To dereference one link per cycle, they recursively jump to themselves when the value of the operand register of them is a pointer to a variable. In that case, they read the content of the address of the variable, write its value into the same operand register, and jump to themselves. As a result, they have dereferenced one link. By means of using this mechanism in addition to the cycle-by-cycle context switching, instructions of the other contexts can be executed even while a dereference loop is processed.

One significant point of the dereference instructions is that they have special ability for committed choice languages. That is the suspension register mechanism. In head matching of committed choice languages, a goal may suspend when a component of its arguments is an unbound variable while the corresponding component of the current clause head is not a variable. After trying all alternative clauses and committing no clauses, the goal really suspends. Therefore when a goal may suspend at one of alternative clauses, the variable which caused the suspension must be recorded to hook the goal by some suspension stack mechanisms until all alternative clause are tried. In the case of UNIREDII, the variable is recorded in the suspension register (general purpose register R30) by the dereference instructions.

There are two kinds of effect of the suspension register mechanism. First, the suspension stack, which is in memory in the case of other similar processors [Kimura and Chikayama 1987], can easily be implemented in reg-

```
append([H | T], X, Y) :- Y = [H | Z], append(T, X, Z).
append([], X, Y)      :- X = Y.

append:
        seta    $suspend, ap
        dcll    r1, $2, r4          ; [H |
$1:
        tgld    [r1 + 1], r1        ; T]
        allc    s, lst, 2, r5       ; [
        st      r4, [r5], %tmp      ; H |
        sudf    [r5 + 1], r6        ; Z]
        bind    r5, r3, r7          ; Y = [ H | Z]
        jntg    r7, udf, $check1
        mov     r6, r3              ; Z)
        exll    r1, $1, r4          ; [H |
$2:
        dfcc    r1, nil, $fail      ; []
        cvos    r2, r3, r2, r3      ; X = Y.
        bind    r2, r3, r7
        jntg    r7, udf, $check2
        succ    gtp, mp
        stop
```

Figure 4: an example of compiled codes (append)

isters so that the suspension check is speeded up. Second and more important, when the head matching is deterministic, as is often the case with real programs, once a goal suspends at one of the alternative clauses, the goal suspends after all. Therefore no suspension stack in memory is necessary in that case. The suspension register mechanism also speeds up this case.

Several combined instructions exist among the dereference instructions. The dereference-and-check-list (dfcl) instruction checks the dereferenced value to determine whether it is a pointer to a list or not, and the dereference-and-check-list-and-load-car (dcll) instruction reads the car part of the list if the dereferenced value is a pointer to a list. Similarly the dereference-and-check-vector (dfcv) instruction checks the dereferenced value to determine whether it is a pointer to a vector, and so on. These instructions are capable of a two-way jump, one for suspension and the other for pointer type check failing. The jump addresses are given by the offset value from the instruction itself and the alternative pointer register (general purpose register R28).

Another kind of combined instruction is that the execute instructions, which execute tail recursions, are combined with those dereference instructions so that they optimize the tail recursion and the consequent head matching sequence.

3.4.2 Arithmetics and Bit-wise Logic Instructions

The arithmetic and bit-wise logic instructions of UNIREDII are very similar to those of conventional processors. They exist for compiling such things as built-in arithmetic predicates. One difference between those of

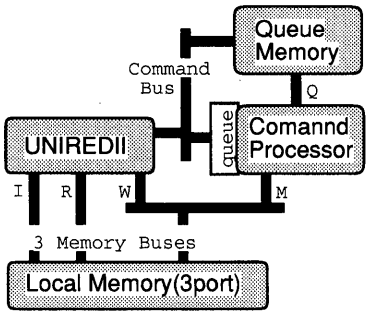


Figure 5: the simulation model used for the evaluation

Table 3: the number of clock cycles which are necessary for emulating co-processor commands issued by UNIREDII

command	description	to	reply	num.of cycles
newgoal	enqueue a new goal	MP	no	1
endreduce	end of a reduction	MP	no	1
suspend	suspend a goal	MP	no	7
deref	dereference a variable	NIP	yes	16
bind	bind a variable	NIP	yes	17
read2	read a remote list	NIP	yes	19
activates	activate a goal	NIP	no	8

UNIREDII and those of conventional processors is that those of UNIREDII check the tag part of the operands and set the tag error flag bit of the flag register according to the value of the tags[3]. Therefore there are several switches of those instructions to deal with various tag types. For example, the add.i instruction (.i switch on) adds an integer to another integer (otherwise set the tag error flag), the add.p instruction adds an integer to a pointer, and the add.b instruction adds total 32-bits to 32-bits and does not change the tag error flag.

3.4.3 An Example of Compiled Code

Now, we present an example of compiled code of UNIREDII in figure 4. It is the code compiled from a deterministic append program. In the tail recursion loop (between label $1 and $2 in the figure), there are only eight instructions. Therefore UNIREDII can execute the append program at a maximum rate of 1.25 million reductions per second with the clock rate of 10 MHz.

4 Simulation Results

4.1 Simulation Model

Figure 5 shows a simulation model used for the evalu-

[3]To simplify the hardware, there are no tag error trap mechanisms in UNIREDII.

Table 2: several aspects of the sample programs which are revealed by the simulation

program	append 100	nreverse 30	qsort 50	primes 100	8 queens
total clock cycles	1435	5427	8162	41068	656011
times of reduction	101	496	380	726	38878
times of suspension	0	29	122	103	558
num.of executed instructions	816	4858	7747	39352	647933
instructions per reduction	8.08	9.79	20.39	54.20	16.67
clock cycles per instruction	1.759	1.117	1.054	1.044	1.012

ation. We evaluated UNIREDII as a single, independent processor, and emulated the coprocessor commands issued by UNIREDII with the imaginary command processor shown in Figure 5. In a real IU of PIE64, these commands are processed by MP and NIPs. Table 3 shows the number of cycles which are necessary for emulating the commands with the command processor. As for network access commands, the number of cycles in the table is determined based on the NIP's performance from [Shimizu *et al.* 1991]. In addition, we used an independent queue memory for queuing newly spawned goals in the simulation model. This roughly corresponds to the MP memory in figure 1. The goal scheduling strategy with this queue memory is LIFO (Last-In/First-Out).

4.2 Performance with Sample Programs

First, we evaluate UNIREDII's performance under the condition that there is no remote memory access. We use, as the sample programs, append 100 (deterministic append of a list of length 100), nreverse 30 (naive reverse of a list of length 30), qsort 50 (quick sort of a list of length 50), primes 100 (generation of prime numbers up to 100), and 8 queens (the 8-queen problem). Table 2 shows some aspects of the sample programs, and figure 6 shows the performance with the sample programs. These are measured with 10 MHz clock.

As for append 100, the performance is comparatively low because, for spawning no sub-goals, the number of active contexts in the program does not exceed one and so the multi-context processing mechanism does not work. In this case, the pipeline interlock occurs frequently and therefore the performance is degraded in spite of only eight instructions in its reduction loop. Figure 7 shows the average number of active contexts about the sample programs. In the figure, more than three contexts are active in average about the other four programs. Consequently we can get enough effect of the multi-context processing in these programs.

Another example of low performance is that of primes 100 because there are no multiplier/divider units in UNIREDII and it takes long time to carry out the divisions which that program requires through integer additions and subtractions. According to table 2, there are more than fifty instructions per reduction in that program, and this is over twice as big as in other programs such as quick sort 50 and 8 queens. This is because it takes about 120 instructions to perform an integer division which is required in primes 100. For other similar programs which require multiplication and/or division of integer and/or floating point, low performance is also expected. But, because the management processor has its own FPU (floating point unit) in the IUs of PIE64, UNIREDII can pass such calculation to the MP and can concentrate on reducing goals. However, the evaluation has not been done yet.

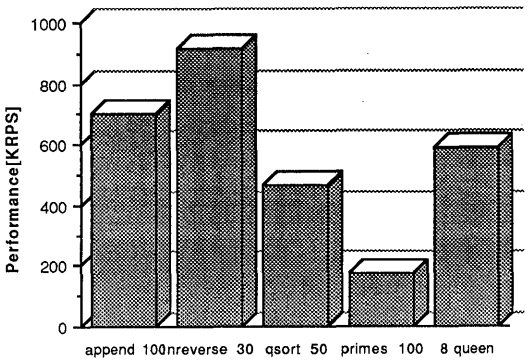


Figure 6: performance with the sample programs

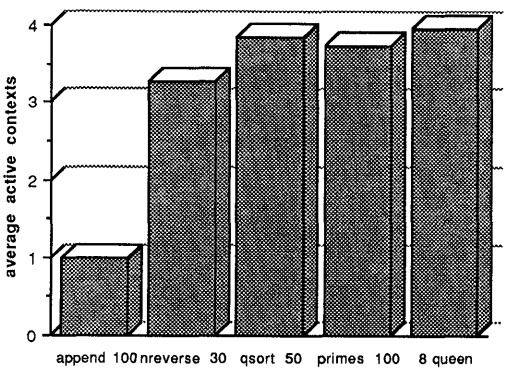


Figure 7: average number of active contexts in the sample programs

4.3 Tolerance of Remote Access Latency

To evaluate tolerance of remote memory access latency, we incorporated a pseudo-remote access mechanism in

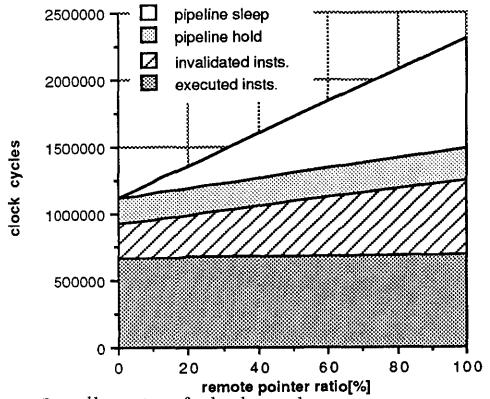


Figure 8: all sorts of clock cycles vs. remote memory access (8 queens, the maximum number of contexts = 1)

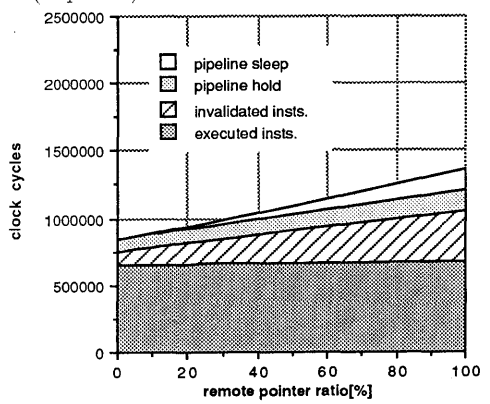


Figure 9: all sorts of clock cycles vs. remote memory access (8 queens, the maximum number of contexts = 2)

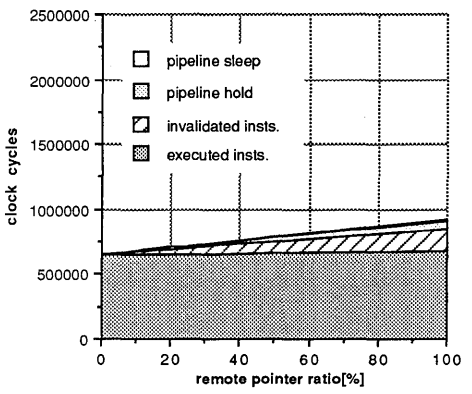


Figure 10: all sorts of clock cycles vs. remote memory access (8 queens, the maximum number of contexts = 4)

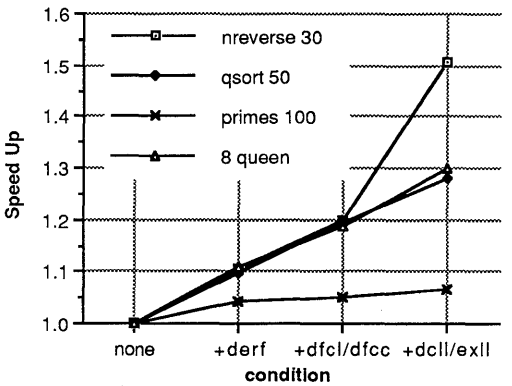


Figure 11: effects of dereference instructions

the simulator in spite of the single processor model of it as shown in figure 5. In more detail, we change the value of the IU-identifier field of the pointers included in every goal when reduction of the goal starts or resumes after suspension, with the probability which we call *remote pointer ratio*. Remote memory access commands issued by UNIREDII are emulated by the command processor shown in figure 5 with cycles listed in table 3. Under these conditions, we varied the maximum number of the contexts from one to four, and measured the clock cycles required by all sorts of the pipelined execution of instructions using the 8 queens program. Results are shown in figure 8 to 10. In these figures, the lowest part (shadowed) of the graph represents the number of executed instructions, the second part (hatched) represents the number of invalidated instructions by some jumps, the third part (lightly shadowed) the number of cycles while the internal pipeline of UNIREDII holds, and the fourth, uppermost part (white) the number of cycles while the pipeline are sleeping because, waiting for some replies, no contexts can be executed.

In figure 8, the multi-context processing mechanism of UNIREDII is not activated because the maximum number of the active contexts is set to one. Therefore the pipeline sleeping time (the white part of the graph) can not be hidden and becomes longer and longer as the remote memory access increases. Moreover, the pipeline hold time and the amount of invalidated instructions are great because the pipeline interlock occurs frequently.

In the other two figures (figure 9 and 10), the multi-context processing mechanism works and works more effectively as the number of contexts increase. The pipeline sleeping time is least in the figure 10 and the pipeline interlock (the pipeline hold and the instruction invalidation) hardly occurs in that figure. They become a little longer as the remote memory access increases because the average number of the active contexts decreases. Figure 9 shows an intermediate state between figure 8 and 10.

4.4 Effects of Dedicated Instructions

Finally, we present the effect of the dereference instructions, which are most characteristic of the instruction set of UNIREDII. Figure 11 shows the speed up about four sample programs (naive reverse 30, quick sort 50, primes 100, 8 queen) without the dereference instructions (the dereference instructions are resolved into more basic instructions), with only the basic dereference (derf)

instruction, with the dereference-and-check-list/constant (dfcl / dfcc) instruction, and with the all combined instructions such as the dereference-and-check-list-load-car / execute-on-list-load-car (dcll/exll) instruction, respectively.

In the figure, the speed up of the basic dereference instruction is about 10 % except in the primes 100 program, in which the majority of the executed instructions are arithmetic ones. In addition, the combined instructions have their effect as shown, and the total effect of these instructions is about 30% except primes 100. Therefore it can be said that the dereference instructions have a great effect.

5 Discussion

In the previous subsection, we present the effect of the dereference instructions and the combined ones. One point is that they are not such complicated instructions. In the hardware design, the instruction decoder does not include the critical path which actually determines the maximum clock rate of UNIREDII. The critical path is included in reading general purpose register file and ALU calculation. Moreover, all of the instructions of UNIREDII are single-cycle instructions because they jump to themselves recursively when they need more cycles to complete their action, as described before in section 3.4.1.

Owing to these dedicated instructions, we can compile Fleng programs so that the number of executed instructions are minimized. As the result, we can achieve high performance though the clock rate is comparatively slow, 10 MHz.

Finally, we shall mention the effect of the multi-context processing of UNIREDII. As well as reducing overhead of inter-processor synchronization, we can reduce pipeline interlock with it so that we can turn the pipeline of UNIREDII into an interlock-free one.

6 Conclusion

We have described the architecture of the inference processor UNIREDII, and evaluated some aspects of it. We got a performance of about 1 MRPS with 10MHz clock, and made certain that the multi-context processing of UNIREDII has a big effect on reducing pipeline interlocking and on reducing overhead of the remote memory access latency. In future, we will evaluate it by larger, real application programs. And, of course, we will make the real UNIREDII chip work as PIE64 system element.

Acknowledgements

We specially thank Prof. J.A.Robinson for much helpful advice. And we also thank the members of the group SIGIE in our laboratory, namely Tadashi Saito, Eiichi Takahashi, Minoru Yoshiada, Takeshi Shimizu, Yasuo Hidaka, Jun'ichi Tatemura, Hidemoto Nakada, Kei Yamamoto, Hajime Maeda, Shougo Shibauti, and Takashi Matsumoto. This work was supported by Grant-in-Aid for Specially Prompted Research (No.62065002), and is now supported by Grant-in-Aid for Encouragement of Young Scientists (No.03001269) of the Ministry of Education, Science and Culture.

References

[Jordan 1983] Jordan,H.F.: *Performance Measurements on HEP - A Pipelined MIMD Computer*, Proc. of the 10th Annual International Symposium on Computer Architecture, pp.207-212, ACM (1983)

[Halstead and Fujita 1988] Halstead,R. and Fujita,T.: *MASA:A Multithreaded Processor Architecture for Parallel Symbolic Computing*, Proc. of the 15th International Symposium of Computer Architecture, pp.443-451, IEEE (1988)

[Shimuzu *et al.* 1989] Shimizu,K., Goto,E., and Ichikawa,S.: *CPC (Cyclic Pipeline Computer) - An Architecture Suited for Josephson and Pipelined-Memory Machines*, Transactions on Computers, Vol.38, No.6, pp.825-832, IEEE (1989)

[Kimura and Chikayama 1987] Kimura,Y. and Chikayama,T.: *An Abstract KL1 Machine and Its Instruction Set* Proc. of the 1987 Symposium on Logic Programming, pp.468-477 (1987)

[Nilsson and Tanaka 1988] Nilsson,M. and Tanaka,H.: *Massively Parallel Implementation of Flat GHC on the Connection Machine*, Proc. of Fifth Generation Computer Systems 1988, pp.1031-1040, ICOT (1988)

[Koike and Tanaka 1988] Koike,H. and Tanaka,H.: *Multi-Context Processing and Data Balancing Mechanism of the Parallel Inference Machine PIE64*, Proc. of Fifth Generation Computer Systems 1988, pp.970-977, ICOT (1988)

[Takahashi *et al.* 1991] Takahashi,E., Shimizu,T., Koike,H., and Tanaka,H.: *A Study of a High Bandwidth and Low latency Interconnection Network in PIE64*, Proc. of Pacific Rim Conference on Communications, Computers and Signal Processing, pp.5-8, IEEE (1991)

[Shimizu *et al.* 1991] Shimizu,T.,Koike,H.,and Tanaka,H.: *Details of the Network Interface Processor for PIE64*, (in Japanese) SIG Reports on Computer Architecture, 87-5, IPSJ (1991)

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Hardware Implementation of Dynamic Load Balancing in the Parallel Inference Machine PIM/c

T. NAKAGAWA, N. IDO, T. TARUI, M. ASAIE and M. SUGIE
Central Research Laboratory, Hitachi Ltd.
Higashi-Koigakubo, Kokubunji, Tokyo 185, Japan

ABSTRACT

This paper proposes and evaluates the hardware implementation required for dynamic load balancing in the prototype PIM/c of the Parallel Inference Machine (PIM). In fine grain multiprocessing, dynamic load balancing is suffering from the high overhead due to the frequent access to load information. Proposed hardware can reduce the overhead by speeding up the access to the load information. In order to utilize the high locality of logic programs, PIM/c is configured along a hierarchical structure of network-connected clusters each of which is a bus-connected multiprocessor. Therefore two kinds of hardware suitable for each hierarchy are implemented for dynamic load balancing.

First, in the clusters, we propose a register with broadcast write feature. The evaluation determines the reduction of overhead due to memory polling which detects a load request. The proposed hardware reduces the execution time of logic programs by 15%.

Second, in the network, we propose the use of a shortcut path to request the value of the total load within a cluster. The evaluation shows that the overhead due to the request of that value is reduced as a result of introducing the shortcut path. The proposed hardware reduces the execution time by 50%.

The results obtained confirm that the use of hardware can reduce the high overhead of dynamic load balancing.

1. INTRODUCTION

Japan's Fifth Generation Computer project [1] has been centered around ICOT (the Institute for new generation COmputer Technology). ICOT has developed the parallel logic programming language KL1 (Kernel Language-1) [2] to describe knowledge and information processing systems. ICOT has also produced software in KL1, including the PIM operating system [3].

We are currently developing the PIM/c [4] as a KL1-based machine. A hierarchical structure of network-connected clusters each of which is a bus-connected multiprocessor is introduced to utilize high access locality of KL1 programs in PIM [5]. Use of locality could restrict the interactions to clusters of several processors and thus reduce the communications among clusters. Therefore, a double hierarchical organization is used in PIM/c.

Dynamic load balancing is one of the main research areas for PIM. As a result of the fact that logical relations are present in a KL1 program and they never define their process of execution with determinacy, dynamic load balancing must be used. For dynamic load balancing it is necessary to require load information, for example, the information about the existence of idle processors or the value of a total load within a cluster. The load information is updated and referenced by distributed processors. In other words the load information is global, therefore it has no locality.

A problem exists in that hardware for normal process execution in PIM/c is optimized to the access with locality. With this type of hardware the latency in accessing global information is large. In fine grain multiprocessing in KL1 programs, high frequency and large latency in accessing load information produces high overhead. Therefore, extensions in hardware are introduced in order to reduce the latency of load information in PIM/c.

In shared bus multiprocessors, snooping caches are known to reduce the memory latency observed by the processors [6,9]. There are two types of cache coherency protocols for rewriting shared data with

copies distributed in plural caches; invalidation-type protocols and broadcast-type protocols. The choice depends on whether it is preferable to invalidate old copies for rewriting by the same processor, or to broadcast the new data for rewriting by other processors.

Eggers [7] defined "per processor locality" as the average number of repeated write references to the same address by the same processor. For normal process execution in the KL1 system, an incremental garbage collection makes the same processor reuse the same address repeatedly for different data references [4]. Thus invalidation protocols are more suitable due to high "per processor locality".

For dynamic load balancing, broadcast protocols are preferable in order to access load information efficiently. Although protocols using both invalidation and broadcast features are known as "competitive snooping protocols [8]", the cache is insufficient to reduce the latency in accessing load information within the cluster of bus-connected multiprocessors. Thus the snooping cache in PIM/c utilizes an invalidation protocol and the implementation of broadcast feature is also considered, not for cache, but for registers to reduce the latency more efficiently.

In network-based multiprocessors, for normal process execution, it is more important to increase the throughput than to reduce the latency because the "non-busy-waiting" feature could overcome the large latency [4]. The PIM/c network unit has message queues to increase the throughput, although they produce an increase in latency. For dynamic load balancing, use of the old information may cause wasteful load dispatching. Therefore, a shortcut path to the message queues is introduced to reduce the latency in accessing load information through the network of PIM/c.

Hardware extensions in PIM/c require only a small amount of hardware because the addressable space for broadcasting is limited in the shared bus, and because the increase in the number of interconnections among clusters is less than that of a system with a special purpose network [10].

2. PIM/c HARDWARE FEATURES

PIM/c has the following hardware features:

A. Hierarchical structure of shared bus multiprocessor and network based multiprocessor.

Figure 1 shows the configuration of PIM/c. It is organized along a hierarchical structure of network-connected clusters to utilize the localities of KL1 programs. Thus, the shared bus hierarchy consists of processors combined in a cluster. Each processor has its own cache, and they share a common bus. Software simulation has proved that the common bus might be a bottle-neck. We concluded that the number of processors within a cluster should be limited to around eight, and that a two-way-interleaved common bus [11] should be possible in PIM/c.

We consider that utilizing the access locality makes it possible to reduce the amount of network hardware because of reducing the number of messages transferred among clusters. As a consequence, in PIM/c the network is connected only to cluster controllers (CC) instead of all processors in the cluster.

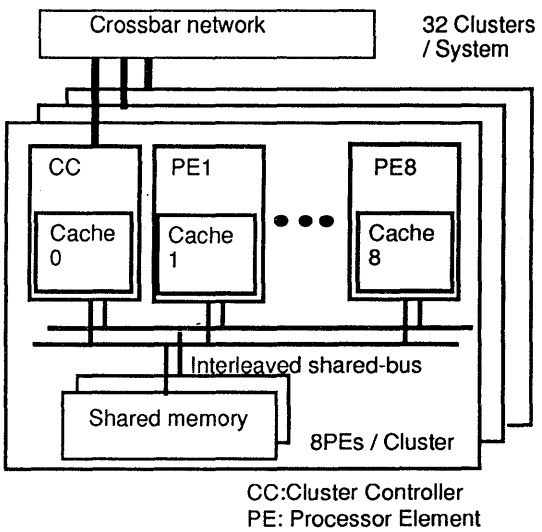


Fig. 1. The configuration of PIM/c. Each cache has a capacity of 80 Kbytes and consists of 20 byte blocks.

B. Broadcast registers in the shared bus hierarchy.

In order to reduce the access latency of load information in the shared bus hierarchy, registers with broadcast feature are introduced in PIM (Fig. 2) [12].

We denote these registers as EFR's (Event Flag Register). They have the following features:

- one-bit wide to indicate an event, and a fast detection feature for control jumps which checks the existence of events.
- feature of broadcast write; therefore, registers indicating the same request event to any processor can be written simultaneously.

The reference and jump can be done within a cycle. When using registers, there is no overhead due to cache misses. Each PIM/c processor has 16 EFRs.

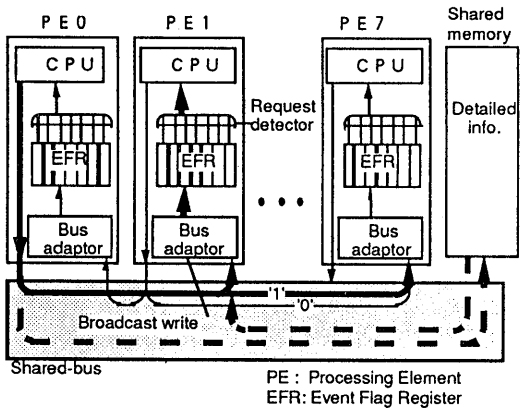


Fig. 2. Broadcast registers in the cluster. Bold lines show the propagation path of a request event to broadcast registers and the broken lines show the memory polling path without hardware support. The thin lines show the reset action of that event.

C. Shortcut path in the network hierarchy.

In order to reduce the access latency of load information in the network hierarchy, two kinds of features are introduced; a shortcut path for the specific messages (Fig. 3) [13] and the registers that hold the load information are called CIR's (Cluster Information Register). The hardware has the following features:

- a shortcut path to message queues.
- eight-bit wide registers to indicate load information in a corresponding cluster.

The register should be written with the load information by its corresponding cluster controller.

As the load information is required without waiting at message queues and without waiting for the cluster controllers to receive, specified registers can always be read in 11 cycles.

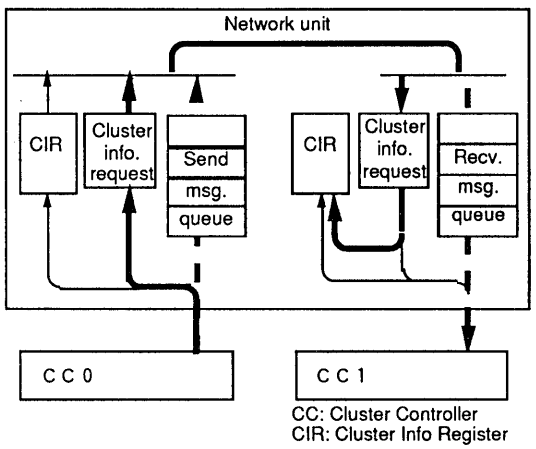


Fig. 3. Shortcut path in the network. The shortcut paths and the registers exist in the router board of the packet switching network. Broken lines show the normal path through the message queues to increase the network throughput and the bold lines show the shortcut path to bypass the queues.

3. EVALUATION STRATEGY

We defined the following two strategies to evaluate the effectiveness of the proposed load balancing hardware.

3.1 Evaluation on the Real Hardware

Real hardware was used for evaluation as the software simulation is almost impossible for the following reasons:

- The presence of the cache and the network introduce more parameters.

  There are many hardware parameters related to the internal states of the cache and the network. The common bus arbitration time, and the message packet switching time are examples. The overhead of cache misses and the network latency is important

in this evaluation. Thus, simulating the cache and network effects concurrently with processor activities would have taken a great deal of time in software simulation.

3.2 Evaluation using an Artificial Load Model

With an aim toward further improvement, we evaluated an artificial load model for the following reasons:

- to separate the effect of hardware alone.
  An evaluation independent of the specific application is necessary in order to isolate the speedup produced by the proposed hardware mechanisms.
- to separate the effect of load balancing.
  The real KL1 execution environment involves many new control sequences in addition to load balancing. For example, handling the priority of loads needs another polling action using EFR registers. The total performance depends on the usage of the proposed hardware in other control sequences.

4. EVALUATION RESULTS

We carried out the evaluation of the proposed hardware in both shared bus and network-based hierarchies.

4.1 Evaluation of broadcast registers in the shared-bus hierarchy

We carried out this evaluation by focusing on the reduction of the latency to access the information about the existence of the idle processors.

A. The load balancing scheme.

The load balancing scheme is explained below:

- Distributed load pool.
  Each processor has its own load pool in order to avoid implicit data transfers between caches due to updating a serial link in case of the generator processor of the load differs from its consumer processor using common load pool [14]. Consequently, an explicit load balancing communication for the distributed load pools is required.
- Receiver-initiated load balancing.
  The explicit load balancing communication for the distributed load pools should be initiated by fully idle processors in order to avoid wasteful dispatching. Thus the communication is request based.
- Communication with arbitrary responder.
  In order to reduce the response time without interrupting busy processors, a new type of communication, the AR (Arbitrary Responder) communication is introduced in PIM/c [12]. The request is sent to any processor which has more than one load in its load pool. In order to avoid the high overhead of context switching, every processor polls the request at intervals where the context switch overhead is low. Thus any processor which detects the request first responds to it. As the timing to detect requests differs in each PIM/c processor, this communication method is expected to reduce the response time proportionally to the number of processors in a cluster.

B. The load model.

This model reflects the following characteristics of KL1 program execution:

- Unit load.
  We denote the unit as the *reduction*. The unit is assumed to be 200 cycles in PIM/c (Fig. 4).
- Indeterminacy in the granularity of loads.
  In order to simulate "Tail Recursion Optimization" [17], we define the *goal* as consisting of an arbitrary number of reductions (1 to 16).
- Indeterminacy in the number of goals.
  In order to simulate the indeterminacy, we assume that each processor generates an arbitrary number of goals (1 to 4096).
- A high write ratio and a high share ratio.
  Accesses performed within the reductions have the

following parameters: write ratio is 0.5, share ratio is 0.5, where write ratio is defined as the ratio of write references to total memory references, and share ratio is defined as the ratio of references to shared data area to total memory references.

- A high access locality.

We define the locality as the number of successive accesses to the same address. The value is set to 4 in order to simulate free-list manipulation, which consists of allocating, instantiating, referring and deallocating a memory cell.

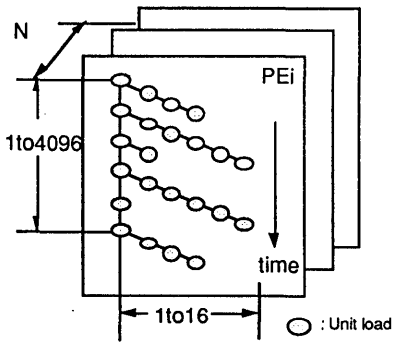


Fig. 4. A Load model with varying granularity.

C. Results of the evaluation in a cluster.

We control the initial load amount in each processor to vary load balancing conditions. According to the deviation of the initial load amounts within processors, 14 cases are simulated with an 8-processor cluster. The resulting data are the total elapsed time (T), the total idle time (I), the total wait time after requesting for load (i), the total dispatching time (t), the total reduction count (R) and the load request count (r). The total idle time includes the time spent waiting for load dispatching since requesting a load by updating a bit-map word until receiving a load by reading a non-zero value from its communication area, and the time to wait for termination of the whole program. The bit-map word is a data array in which each bit corresponds to a processor requesting load. The total dispatching time includes the time to select an idle processor by encoding the bit-map word to the address of its communication area, and the time to dispatch a load to each idle processor by updating their communication areas. The evaluation measures are i and t, and the reduction cost is defined as follows:

Reduction cost = (T - I - t) / R

Figure 5 shows the performance increase in reduction using registers. The total reduction cost and the load request count are varied in 14 simulation cases. In this figure, request ratio is introduced, which is defined as the ratio of the load request count r to the total reduction count R. The reduction cost is almost independent of the request ratio. This fact indicates that the memory-polling overhead caused by checking request occurrences is larger than the overhead due to cache misses using invalidation protocol. The speedup obtained is 15% due to the use of EFRs.

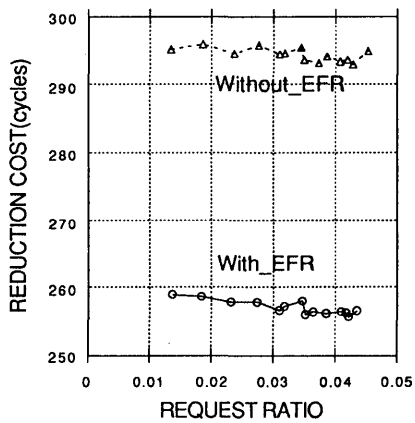


Fig. 5. The increase in speed using registers. The reduction cost is defined as the number of execution cycles per unit load. The result involves extra cycles for probing. The request ratio is defined as the number of request per reduction. Using memory polling the reduction cost is high due to the serial execution of a memory access and a branch. Using EFR, both the access and the branch can be done within a cycle. The polling is done for three kinds of events; load request, load dispatching and termination of the whole program.

Figure 6 shows the wait time i and the dispatching time t as a function of request count. It is confirmed that the use of EFR with broadcast feature reduces both the wait time and dispatching time. The use of EFR reduces the dispatching time by 20%, and reduces the wait time by 15%.

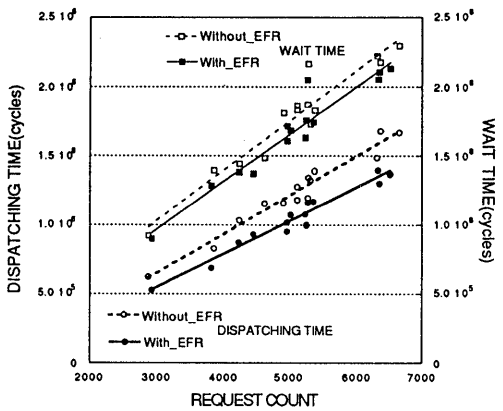


Fig. 6. The increase in speed using broadcasting. The dispatching time and the wait time increase due to the cache misses using an invalidation-type snooping cache. The use of broadcast feature eliminates the overhead due to the cache misses.

4.2. Evaluation of shortcut paths in the network-based hierarchy

We carried out this evaluation by focusing on the reduction of the latency in accessing the value of the total load in a cluster.

A. The load balancing scheme.

The load balancing scheme is described below:

- Sender-initiated load balancing.

  A study of the Multi-PSI system disclosed a problem of the receiver-initiated load balancing scheme in large-scale machines, namely that a load request contention may arise at busy processors [15]. In order to avoid this contention, an improved sender-initiated scheme, named "Smart Random Load Dispatching" [5] is efficient in reducing wasteful dispatching. In this scheme, the cluster to which goals are dispatched is determined at random and then this goal dispatch is aborted on the condition that the dispatch target has more loads in the pool than the dispatching cluster.

B. The Load model.

The load model among clusters is defined in such a way as to reflect the changes in the amount of loads in the load pool. The load model is as follows:

- An initial goal is denoted by L(16) (Fig. 7 shows L(5)).
- The execution of goal L(i) produces (i-1) subgoals, L(i-1),..., L(2), L(1). Thus, the goal L(i) has 2^{i-1} reductions.
- Each reduction takes 300 cycles to execute using network messages.
- The message length required for the load dispatching is 27 bytes long. Thus, it takes 27 cycles to send this message through the one-byte-wide network interface. The length of the message requesting the load amount is 2 bytes.

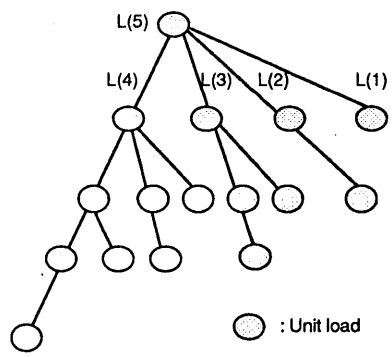


Fig. 7. A load model with floating amount of load.

C. Results of the evaluation among clusters.

We control the dispatching rate, which is defined as the ratio of all goals dispatched to other clusters to all executed goals, by changing the interval of the dispatching control. In order to determine the efficiency of load dispatching, the total elapsed time (T), the total idle time (I) and the dispatching rate (d) are measured. Differences result from the latency of load information.

Figure 8 shows the results obtained by applying the smart random load dispatching scheme to 8 cluster system without support hardware. The normalized elapsed time, which is defined as the ratio of elapsed time by 8 cluster system to elapsed time by single cluster, and the utilization of processors are plotted as a function of the dispatching rate. In order to compare the results in the two cases, we assume that the dispatching rate is controlled to be 0.2, because safe control occurs only at the upper side of the minimum point. Without the support hardware, the resulting increase in speed is approximately 3.3 in an 8-cluster system at a dispatching rate of 0.2.

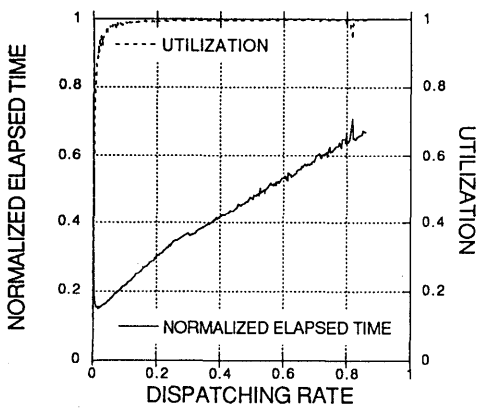


Fig. 8. Smart random dispatching without support hardware. The dispatching rate is defined as the ratio of all goals dispatched to other clusters to all executed goals. The normalized elapsed time varies considerably from 0.125 using 8 clusters connected via a network because the overhead for message handling is visible.

Figure 9 shows the results after applying the smart random load dispatching scheme with hardware support. The normalized elapsed time and the utilization of processors are plotted as a function of the dispatching rate. With the support hardware active, the processor can reduce the overhead due to requesting the load amount. The resulting increase in speed is approximately 5.5 in an 8-cluster system at a dispatching rate of 0.2.

Comparing the two results, the use of the proposed hardware halves the normalized elapsed time at 0.2 dispatching rate, where the control of dispatching rate seems to be possible.

It should be noted that the shortcut path can also be used for other load balancing schemes, including the minimum load distribution scheme [16]. These schemes will be evaluated in future work.

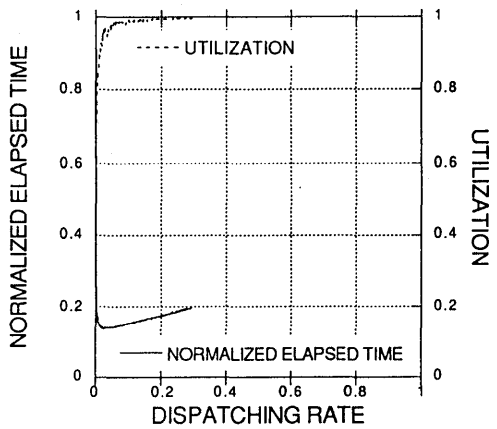


Fig. 9. Smart random dispatching with support hardware. The normalized elapsed time varies near 0.125 using 8 clusters connected via a network because the overhead for message handling is quite low.

5. CONCLUSION

Hardware for dynamic load balancing is implemented in both shared-bus and network-based multiprocessors.

We propose a register with broadcast write feature in shared-bus multiprocessors. Also, in network-based multiprocessors, the network unit uses a shortcut path. The evaluation was carried out using real hardware and an artificial load model.

The evaluation results in the shared bus hierarchy determine the overhead due to memory polling which detects a load request. The proposed hardware reduces

the execution time of logic programs by 15%.

The evaluation results in the network-based hierarchy show that the overhead due to requesting the load amount is reduced as a result of introducing the shortcut path. The proposed hardware reduces the execution time by 50%.

It is confirmed that the proposed hardware reduces the access latency of load information, and subsequently the overhead produced by dynamic load balancing.

ACKNOWLEDGEMENTS

The authors would like to thank Dr. Shun'ichi Uchida, the manager of the research department of ICOT, for his guidance and support, Dr. Kazuo Taki, chief of 1st ICOT laboratory, and Mr. Marius Hancu for helpful discussions. This research was sponsored by ICOT.

REFERENCES

[1] K. Fuchi and K. Furukawa, "The role of logic programming in the fifth generation computer project," Springer-Verlag, 1987, 1(5) pp 3-28.

[2] K. Ueda , "Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard," TR208, ICOT, 1986.

[3] T. Chikayama, H. Sato, T. Miyazaki, "Overview of the Parallel Inference Machine Operating System (PIMOS)," Proc. of the FGCS, vol.1, 1988.

[4] A. Goto, M. Sato, K. Nakajima, K. Taki, A. Matsumoto, "Overview of the Parallel Inference Machine Architecture (PIM)," Proc. of the FGCS, vol.1, 1988, pp 208-229.

[5] M. Sugie, M. Yoneyama, N. Ido, T. Tarui, "Load Dispatching Strategy on Parallel Inference Machines," Proc. of FGCS, Vol.3, 1988.

[6] J.-Archibald and J. Baer, "Cache Coherence Protocols: Evaluation using a Multiprocessor Simulation Model," ACM Trans. on Comp. Systems, Vol.4, No.4, 1986, pp 273-298.

[7] S. J. Eggers and R. H. Katz, "Evaluating the Performance of four Snooping Cache Coherency Protocols," Proc. of the 16th ISCA, 1989.

[8] A. R. Karlin, M.S. Manasse, L. Rudolph and D. D. Sleator, "Competitive Snoopy Caching," Proc. of the 27th Annual Symposium on Foundation of Computer Science, Toronto, October, 1986.

[9] A. Gupta and J. Hennessy, "Comparable Evaluation of Latency Reducing and Tolerating Techniques," Proc. of the 18th ISCA, IEEE, 1991.

[10] H. Koike and H. Tanaka, "Multi Context Processing and Data Balancing Mechanism of the Parallel Inference Machine PIE64," Proc. of FGCS, Vol.3, 1988.

[11] L. Rudolph and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," Proc. of the 11th ISCA, June, 1984.

[12] T. Nakagawa, A. Goto, T. Chikayama, "Slit-Check Features to Speedup Interprocessor Software Interruption Handling," IEICE SIG Reports, July, 1989, pp 17-24, (in Japanese).

[13] N. Ido, H. Maeda, T. Tarui, T. Nakagawa, M. Sugie, "Parallel Inference Machine PIM/c -Load Balancing Support-," the 40th Annu. Convention IPS Japan, 2L-4, (in Japanese) .

[14] M. Sato and A. Goto, "Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor," Proc. of IFIP Working Conf. on Parallel Processing, Pisa, April, 1988.

[15] M. Furuichi, K. Taki, and N. Ichiyoshi, " A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI," In Proc. of the 2nd SIGPLAN Sympo. on Principles and Practice of Parallel Programming, pp 50-59, Mar. 1990.

[16] S. Sakai, H. Koike, H. Tanaka, T. Motooka, "Interconnection network with dynamic load balancing facility," Trans. of Information Processing, Vol. 27, No. 5, pp 518-524, 1986, (in Japanese).

[17] D. H. D. Warren, " An Improved Prolog Implementation which Optimises Tail Recursion," Research paper 156, Dept. of Artificial Intelligence, Univ. of Edinburgh, Scotland, 1980.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Evaluation of the EM-4 Highly Parallel Computer using a Game Tree Searching Problem

Yuetsu KODAMA Shuichi SAKAI Yoshinori YAMAGUCHI

Electrotechnical Laboratory
1-1-4, Umezono, Tsukuba-shi, Ibaraki 305, Japan
kodama@etl.go.jp

Abstract

EM-4 is a highly parallel computer whose eventual target implementation has more than 1,000 processing elements(PEs). The EM-4 prototype consists of 80 PEs and has been fully operational at the Electrotechnical Laboratory since April 1990. EM-4 was designed to execute in parallel not only static or regular problems, but also dynamic and irregular problems. This paper presents an evaluation of the EM-4 prototype for dynamic and irregular problems. For this evaluation, we chose a checkers program as an example of the game tree searching problem. The game tree is dynamically expanded and its structure is irregular because the number and the depth of subtrees of each node depend heavily upon the status of the game. We examine effects of the load balancing by function distribution, data transfer, control of parallelism, and searching algorithms on the EM-4 prototype. The results show that the EM-4 is effective in dynamic load balancing, fine grain packet communication and high performance of instruction execution.

1 Introduction

Parallel computing has been effective for static or regular problems such as scientific computing and database systems. Parallel computing is, however, still an active research topic for dynamic or irregular problems.

EM-4 is a highly parallel computer which was developed at the Electrotechnical Laboratory in Japan. Its target applications include not only static or regular problems, but also dynamic or irregular problems. EM-4 provides special hardware for parallel computing: high data transfer rate, high data matching performance, dynamic load balancing, and high instruction execution performance.

In this paper, we evaluate the performance of EM-4 on a dynamic and irregular problem. The performance of EM-4 on some small programs such as recursive fibonacci is presented in [Kodama *et al.* 1991]. While the fibonacci program creates many function instances dynamically, it is not irregular because the tree of calling functions is a binary tree, the depth of each branch is similar to those of its neighbors, and the size of each node function is the same and small. We chose a game tree searching problem as a practical problem. This class of programs dynamically expands the game tree, and is irregular because the number of subtrees from each node of the game tree, the depth of subtrees, and the execution time of each node depends heavily upon the status of the game. Furthermore, the α-β searching algorithm is often used for game tree searching, because it cuts the evaluation of the current tree by using the evaluation of the previous tree. Tree cutting makes the program more dynamic and irregular.

This paper presents the evaluation of the EM-4 prototype using a checkers game program as an example of the game tree searching problem. We examine the effect of parallel computing on the EM-4 prototype. Section 2 presents an overview of the EM-4 and its prototype. Section 3 describes a game tree searching problem and a checkers game. Section 4 presents evaluation issues for load balancing, data transfer, control of parallelism, and searching algorithms for the checkers game. Section 5 gives an evaluation and examination of the strategies described in section 4. Section 6 concludes our results and discusses our future plans.

2 The EM-4 Highly Parallel Computer

EM-4 is a highly parallel computer whose eventual target implementation has more than 1,000 PEs[Yamaguchi *et al.* 1989, Sakai *et al.* 1989]. The EM-4 prototype consists of 80 PEs and has been fully operational since April 1990[Kodama *et al.* 1990].

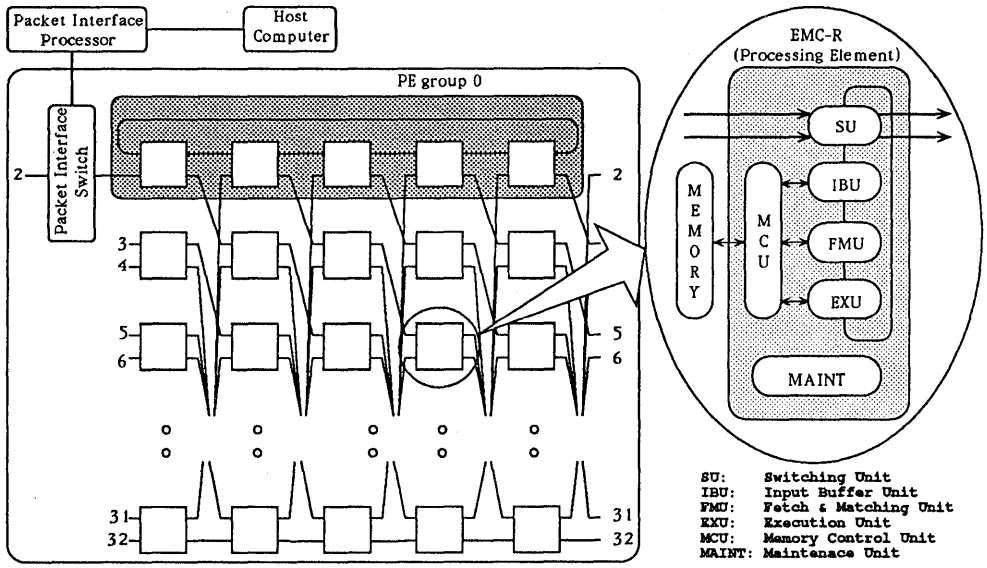


Figure 1: The organization of EM-4 Prototype

2.1 The architecture of EM-4

The organization of the EM-4 prototype is shown in Figure 1. The prototype consists of 80 PEs, and each 5 PEs are grouped and are implemented on a single PE board. The PE of the prototype is an single chip processor which is called EMC-R and is implemented in a C-MOS gate array. The PE has local memory and is connected to the other PEs through a circular omega network.

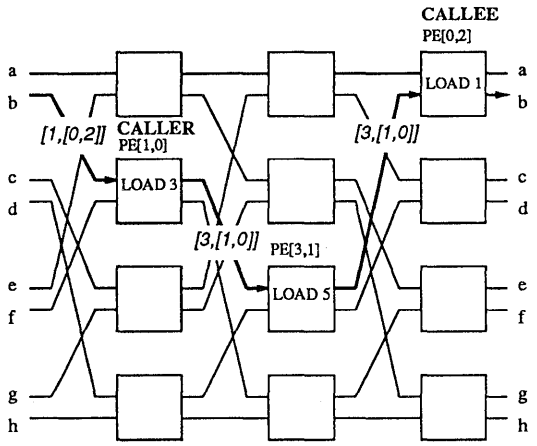
EMC-R is a RISC processor for fine grain packet-based parallel processing. EMC-R generates packets in an execution pipeline, and computation is fired by the arrival of packets. This is a dataflow mechanism, but we improved it so that it can operate on a block which consists of several instructions, executed exclusively from other instructions. This model is called the "strongly connected arc model", and the block is a strongly connected block(SCB).

When a packet arrives at a PE, the execution pipeline is fired and EMC-R executes the SCB indicated by the packet. First, EMC-R checks whether the partner of the packet has arrived. If the parter exists, it continues to execute the SCB until the end of the block. If the parter does not exist, EMC-R stores the packet data in a matching memory and waits for the next packet.

The packet size of EMC-R is two fixed words and there is only one format consisting of one address word and one data word. It can be generated in a RISC pipeline of EMC-R. During the data word is calculated in a RISC pipeline, the address word is formed in a packet generation unit when the packet output is instructed. Since the network port is only one word wide, first the address word is sent to network, and then the data word is sent. In the second clock cycle, the next instruction can be executed in parallel with data word transfer.

The circular omega network has the same structure as an omega network, except that every node of the network is connected to a PE. The network has the following features: (1) The required amount of hardware is $O(N)$, where N is the number of PEs; (2) The distance between any two PEs is $O(logN)$. The 3 by 3 packet switching unit is in a EMC-R, and a packet can be transferred to a neighboring PE independent of the instruction execution on the PE. Packets are transferred by wormhole routing, and take only $M+1$ cycles between PEs which are distance M apart if there is no network conflict.

The clock of the EMC-R runs at 12.5 MHz. The RISC pipeline can execute most instruction in one clock cycle; the peak execution performance is 12.5 MIPS. It takes two clock cycles when two operand matching fails, and takes three clock cycles when the matching succeeds. The peak synchronization performance is 2.5 Msync/s. It takes two clock cycles to transfer a packet, and the peak network packet transfer performance is 18.75 Mpacket/s. EM-4 prototype consists of 80 PEs, the peak execution performance is 1 GIPS, its peak synchronization performance is 200 Msync/s, and its peak network packet transfer performance is 1.5 Gpacket/s. EMC-R achieves a high performance in both instruction execution and packet data transfer/matching.



[LD,[GA,CA]] is the MLPE packet which shows that PE[GA,CA] has the minimum load LD

Figure 2: How to Detect the Minimum Load PE

2.2 Dynamic load balancing method

To get high performance in parallel computers, high utilization of PEs, as well as high performance of PEs are necessary. If the program has simple loop structure or static data transfer structure such as in diffusion equation applicaitons, the load of the program can be estimated and the load can be statically balanced at programming or compiling time. But, if the program is dynamic or irregular structure, static load balancing is difficult and dynamic load balancing is necessary.

In the EM-4, we implemented automatic load balancing mechanisms attached to the circular omega topology. In the circular omega network, each node has two circular paths. We use a path to group the PEs, and use another path to achieve dynamic load balancing. Suppose that a PE wants to invoke a new function. This PE will send out a special MLPE(Minimum Load PE) packet. The MLPE packet always holds the minimum load value and the PE address among the PEs which it goes through. The load of each PE is evaluated by hardware in the PE mainly based on the number of packets in the input buffer. At the starting point, the MLPE packet holds its sender's load value and its PE address; when it goes through a certain SU in the circular path, the SU compares the load value of the PE connected to it, and if the value is less than of the packet, the data in the MLPE packet will be automatically rewritten to the current PE's value; otherwise the MLPE packet keeps its value and goes to the next SU. This operation is done in one clock cycles of packet transfer. When the MLPE packet returns to the starting point, it holds the least loaded PE number and its load value. Figure 2 show this. In this figure, PE[1,0] generates an MLPE packet and, after the circulation, it obtains the least loaded PE number [0,2] and its load 1.

By this method, called the circular path load balancing, each MLPE packet scans s different groups, where s is the number of network stages. When the total number of the PEs increase, coverage of PEs by this load balancing method becomes relatively small. The efficacy of this method is reported in [Kodama *et al.* 1991].

Since it takes several cycles for the MLPE packet to return, the EM-4 resolves this latency by pre-fetching: it sends a MLPE packet in advance, allocates the new function instance on the PE specified by the returned packet of MLPE, and stores the function ID in a special register of the required PE. When a function call is necessary, the stored function ID is used and another MLPE packet is sent for the next function call. In the pre-fetch strategy, the new function ID may have not yet been stored when a function call is necessary. In this case, the pre-fetch method uses one of the other distribution methods to choose the PE.

3 Game Tree Searching Problem

We choose the checkers program as an example of a game tree searching problem in order to evaluate the EM-4 on a dynamic and irregular problem. Since the rules of checkers are very simple, the program makes it easy to characterize the parallel behavior of the program.

The rule of checkers game is as follows. Each player moves one of his pieces in turn until the player who has no pieces or moves loses. Pieces can be moved to a forward diagonal area. If there is an opponent's piece in a forward diagonal area, and the next diagonal area is empty, you must jump to the empty area and remove the enemy piece. If you can jump successively, you must jump successively. If your piece arrives at the end of the enemy area, that piece can then move in all four diagonal directions.

The Min-Max searching algorithm is the simplest algorithm for the game tree searching problem. This algorithm expands the game tree by the possible moves of each player in turn. When the game tree is expanded to a certain level, each leaf is evaluated. If the stage corresponds to your turn, the maximum node is selected; if the stage is your opponent's turn, the minimum node is selected. Although the Min-Max algorithm is simple, it is not efficient because it needs to search every branch. The α-β searching algorithm[Slagle 1971] is more efficient than the Min-

Max algorithm, because this algorithm tries to cut off the evaluation of unnecessary branches.

If the game tree is expanded in a depth-first manner, the resources required to remember the game tree are small. This expansion makes it easy to cut off the unnecessary branches, but reduces the parallelism. If the game tree is expanded in a breadth-first manner, it results in large parallelism, so this expansion is well-suited for parallel computers. However, since the number of nodes increases exponentially as a function of the depth of the tree, the resources will be exhausted quickly if the parallelism is not controlled.

4 Execution Issues of a Checkers Game

The overheads to parallelize the checkers program are the following:

1. overhead for allocating new function instances on other PEs.
2. overhead for transferring the current status of the table to other PEs.
3. idle PEs caused by an unbalanced load.
4. decline of efficiency caused by cutting branches in the α-β search.

These overheads depend upon implementation strategy decisions. The function distribution strategy effects the function allocation overhead. Packed data transfer reduces the amount of transfer data. The idle PE ratio depends upon the load balancing strategy. The searching algorithm changes the branch cutting overhead. These overheads also depend upon the control of the parallelism and the searching strategy. Each of these decisions is described in greater detail in the following subsections.

4.1 Function distribution and load balancing

Load balancing is the most important issue in achieving high performance on parallel computers. Since the checkers program requires many function instances to expand the game tree, it distributes them among the PEs in order to balance the load across the machine.

Our checkers program can distribute function calls by one of the following two strategies:

round-robin distribution Each PE independently chooses the PE which will execute the called function in a round-robin manner.

manager distribution A centralized manager PE chooses the PE which will execute the called function.

We can also combine the two methods: that is, the manager distribution can be used until a certain level in the game tree expansion, and the round-robin distribution can be used after that level. In the round-robin distribution, the load might be unbalanced at the beginning of the program. In the manager distribution, the overhead is larger than round-robin distribution because of packet communication overhead and concentration of requests.

EM-4 dynamically distributes functions according to the load of PEs by the circular path load balancing described in section 2.2. The dynamic round-robin distribution described below is the third function distribution method that we evaluated in our checkers program.

dynamic round-robin distribution A PE is dynamically chosen by the circular path load balancing method, and in the case that the MLPE packet has not returned, a PE is chosen by the round-robin distribution method.

4.2 Data transfer

Since EM-4 is a distributed-memory parallel computer, the checkers program sends the status of the table and selected moves by packets to functions on other PEs. The status of the table is represented by a 64 word array, but each word is only 4 bits. The following two transfer methods are considered in the checkers program.

unpacked transfer use packets which have data representing a position.

packed transfer use packets which have packed data representing 8 positions.

While the unpacked transfer sends eight times more packets than the packed transfer, the packed transfer needs to pack and unpack data.

4.3 Control of parallelism

Parallelism has to be controlled to both avoid exhaustion of resources, and to provide sufficient parallelism to keep all the PEs busy. To control parallelism, throttling can limit the number of the active functions. If the number of active functions exceeds a certain amount, further requests for calling functions are buffered until other functions are finished. Throttling has the possibility of deadlock.

Another way to control parallelism is to switch from breadth-first search to depth-first search at some level of the game tree, where the level can be determined either statically or dynamically. Static switching sets the level by the depth of the game tree. Dynamic switching determines the level using the load

of PEs. Breadth-first searching increases parallelism, and depth-first searching restrains parallelism.

Our checkers program uses the static switching strategy to control parallelism, because this strategy is very simple. We plan to implement the dynamic switching strategy for the checkers program in the near future.

4.4 Game tree searching algorithms

The two primary algorithms for the game tree searching problems are the Min-Max algorithm and the α-β algorithm. The Min-Max algorithm provides much parallelism in the breadth-first strategy. The α-β algorithm has high efficiency in the depth-first strategy. If the α-β algorithm is used only with the breadth-first strategy, it ignores the possibility of cutting branches, and it must search more trees than the α-β algorithm on a single processor. Since the ratio of branches cut off relative to the whole tree in the α-β algorithm increases according to the depth of the searching tree, a parallel α-β searching algorithm must be considered to increase the efficiency of branch cutting in the parallel environment.

Parallel α-β searching is complicated because of the dilemma between parallelism and efficiency of branch cutting. Another important problem is the overhead of terminating functions. Since these function instances are distributed and activated in parallel, the overhead of terminating functions is more than overhead of creating functions. This difficult trade-off is simply resolved in our checkers program by changing algorithm in breadth-first strategy and depth-first strategy. In the breadth-first strategy, we select the min-max algorithm to expand the parallelism, and in the depth-first strategy, we select the α-β algorithm to achieve the efficiency of cutting branch. We call this search "serial α-β search" in this paper. This search can be easily implemented, but the efficiency of branch cutting is less than the parallel α-β search[Oki *et al.* 1989].

To get more efficiency from branch cutting, the search that uses α-β search from the leaf of breadth-first strategy is the "partial parallel α-β search". This search algorithm is illustrated in Figure 3. In this search, depth-first search is called in parallel from the leaf of breadth-first search, but the top node(which is indicated by B in the figure) of serial depth-first search gets the α-β value from the parent node (A) every time when the child node (C) return the evaluation result, and check whether the remain branch (C') can be cut off or not. The merit of this search is that we can expect enough efficiency from branch cutting and the overhead of terminating search is nothing

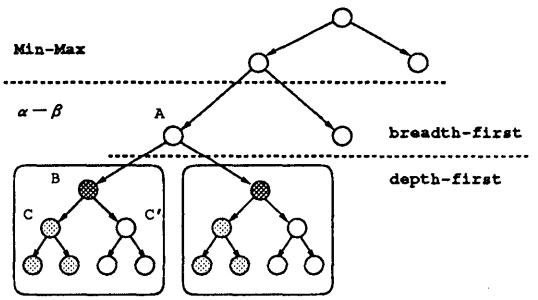


Figure 3: partial parallel search

since the child node in depth-first strategy is sequentialized.

The checkers program can use the following three searching algorithms.

Min-Max search using the Min-Max algorithm both breadth-first and depth-first.

serial α-β search using the Min-Max algorithm breadth-first, and using the α-β algorithm depth-first.

partial parallel α-β search using the Min-Max algorithm breadth-first until the last level, and using the α-β algorithm in the last level of breadth-first and then depth-first.

5 Experimental Results on the EM-4

We implemented the checkers program on the EM-4 prototype in an assembly language to evaluate the performance of the EM-4 for dynamic and irregular problems. We examine the execution issues discussed in the previous section.

5.1 Effects of function distribution and load balancing

An unbalanced workload causes idle PEs. Since the load balancing of the checkers program is performed at the function level, the function distribution strategy must be evaluated. The alternatives for the function distribution of the checkers program are the manager distribution, the round-robin distribution, the dynamic round-robin distribution, and combinations of these.

We executed the checkers program using the partial parallel α-β search using each function distribution methods. Figure 4 shows the results. We represent the speedup ratio of each distribution relative to the round-robin distribution. We executed each combination of manager distribution and round-robin distribu-

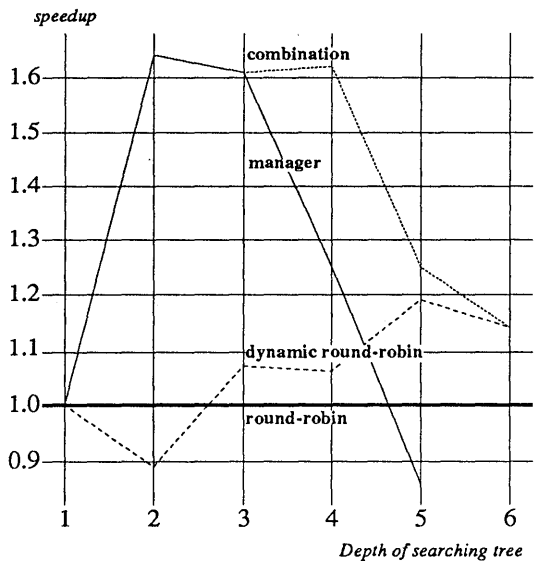


Figure 4: Effects of function distribution

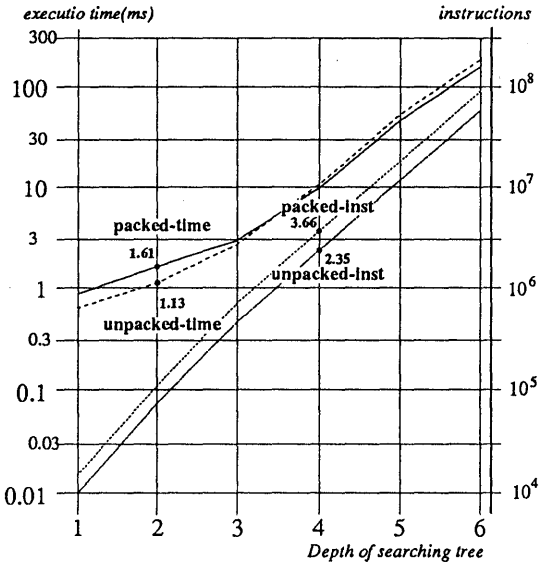


Figure 5: Comparison of data transfer

tion, and the fastest combination is shown in the figure. The combination uses the manager distribution until the third level, and thereafter uses the dynamic round-robin.

When the level of tree searching is shallow, manager distribution is better, because the manager distribution allocates functions more evenly. Since the size of each function is large relative to the whole program, the heavily loaded PE will become a bottle-neck and the program cannot achieve sufficient speed-up, even if the load is only slightly unbalanced. When the level of the search tree becomes deeper, the dynamic round-robin distribution is better, because the size of each function becomes small relative to the whole program, and a small load imbalance does not effect the execution time much. On the other hand, in the manager distribution, the requests of PE addresses for the function call concentrate on the manager PE. Because of the queue of requests, the long turnaround time of the function call makes the execution time slow. Furthermore, at the sixth level of the search tree in the manager distribution, the program cannot be executed because of overflow of the packet queue buffer.

Since the execution of the dynamic round-robin distribution is 15% faster than the round-robin distribution when the searching tree is deep, this indicates that the dynamic round-robin strategy is effective in the case that there is sufficient parallelism.

5.2 Effects of data transfer

To parallelize the program, data must be transferred between PEs, while data is only passed between memory locations in a single PE. We compared the two data transfer method, unpacked and packed. The unpacked transfer uses a packet which has data representing a position, while the packed transfer uses a packet which has packed data representing 8 positions.

Figure 5 is the results by the checker program of the partial parallel α-β search using the combination of manager and dynamic round-robin method as the function distribution. This figure shows the execution time and the total number of executed instructions of both data transfer method. Note that the execution time and the total number of the executed instructions are figured on a logarithmic scale.

In this figure, the number of executed instructions of the packed packet transfer is 50% more than the unpacked transfer for each level. The increase of the executed instructions is caused by the pack and unpack operations. When the level is shallow, the execution of the unpacked transfer is 1.5 times faster than the packed transfer. This speed-up ratio is the same as the instruction amount ratio. But when the level is deep, packed transfer is a little faster than the unpacked transfer while the instruction count of the packed transfer is larger than the unpacked transfer.

Figure 6 shows the number of active PEs and overhead PEs in both data transfer strategies. An overhead PE is a PE which is waiting for the ready of the network to send a packet or stores the packet in the memory packet buffer when the on-chip packet buffer overflows. An active PE is a PE which is neither an overhead PE nor an idle PE. At the shallow levels, the active PE ratio of both transfer strategy is low. When

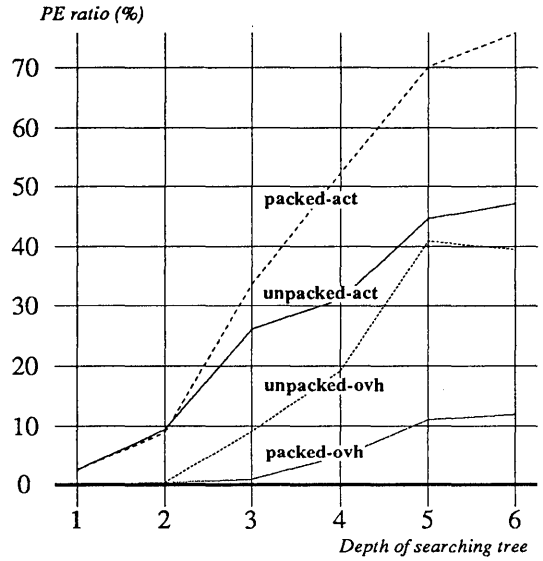


Figure 6: Examination of the active PE ratio comparing the data transfer

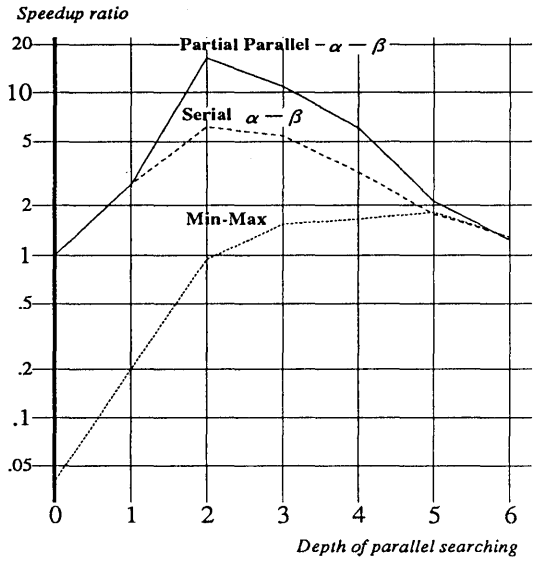


Figure 7: Effects of parallelism control

the level becomes deep, the active PE ratio of the unpacked transfer is 30% lower than the packed transfer, and the overhead PE ratio of the unpacked transfer is 30% higher than the packed transfer. This high overhead PE ratio of the unpacked transfer is the reason why it is slower than the packed transfer. Since the unpacked transfer needs to send more packets than the packed transfer, the network has many conflicts, resulting in large overhead.

Although the packed transfer shows the high ratio of the active PEs on the surface, a third of the instructions are used for packing and unpacking the packets, and the packed transfer is not so effective. Since the pipeline of the EM-4 is designed to send packets quickly, unpacked transfer is suitable for the EM-4. If there are many conflicts in the network, however, the overhead decreases the performance of sending packets. One way to reduce this overhead is to avoid the network conflicts by allocating the function locally. Since the manager and round-robin distributions does not take into account the locality between the PE which calls the function and the PE which executes the function, it increases the possibility of network conflicts. If the execution PE is selected from the neighbors of the calling PE, network conflicts do not occur as frequently. Another way to control the parallelism is by limiting the number of active functions. This is examined in detail in the next subsection.

5.3 Effects of parallelism control

While parallelism must be exploited to make the program execution faster, as mentioned before, too much parallelism causes some overhead. It is necessary to control the parallelism in order to avoid the exhaustion of resources, and to reduce the overhead of parallelization. The checkers program controls parallelism by switching the searching strategy from a breadth-first manner to a depth-first manner.

Figure 7 shows the speedup ratio to the sequential execution of the α-β search when the switchover level of the parallelism control strategy is changed. The execution uses the combination of manager and dynamic round-robin method as the function distribution strategy and the unpacked method as the data transfer strategy. Note that the X-axis represents the depth of the breadth-first searching, while these all execution search the game tree until the depth is the sixth level.

In the Min-Max search, the deeper level of parallel searching results in more parallelism, and the maximum speedup becomes 49 times. Exploiting maximum parallelism, however, does not necessarily achieve speedup. One reason is that at the sixth level, too many packets are sent and the overhead of network conflicts becomes much larger than at the shallow levels. Another reason is that excessive parallelism is just overhead such as data transfer or remote function invocation, since sufficient parallelism is exploited until the fifth level. It is sufficient to have as much parallelism as needed to activate every PE and hide the latency of remote access — excessive parallelism is not

helpful.

The serial α-β search executes fastest at the second level, and when the level is deeper the performance decreases. This is because parallel searching uses breadth-first search, and much information that could be used to cut subtrees is discarded to parallelize the program. As parallel searching gets deeper, more information is discarded. As a result, it reduces the efficiency of cutting excessive branches, and increases the number of trees to be evaluated. The partial parallel α-β is same as the serial α-β search.

5.4 Effects of searching algorithms

Figure 7 also shows the effects of searching algorithms. The execution of the Min-Max search on 80 PEs is 49 times faster than the Min-Max search on a single PE, but only 1.8 times faster than the α-β search on one PE. This shows that the Min-Max search is suitable for parallel execution, but that it is difficult to compensate for the difference of efficiency between the Min-Max search and the α-β search by parallel execution.

The α-β search is a very serial algorithm, but can achieve 16 times speedup via partial parallel α-β search, while the serial α-β search can achieve 6 times speedup. This is because the partial parallel α-β search uses the information of cutting trees at the last level of parallel searching, and the efficiency of cutting trees in the partial parallel α-β search is higher than the serial α-β search.

6 Conclusion and Future plans

To evaluate the highly parallel computer EM-4 on dynamic and irregular programs, we execute the game tree searching problem of checkers on the EM-4 prototype, which consists of 80 PEs. The effects of the strategies for load balancing, data transfer, parallelism control and searching algorithm are examined.

Our checkers program achieves 49 times speedup in the Min-Max search and 16 times speedup in the α-β search on 80 PEs system. In this execution, the combination of the manager distribution until the third level and the dynamic round-robin distribution thereafter is used as the function distribution method for load balancing, the unpacked transfer is used as the data transfer strategy, and the static switching from the breadth-first to the depth-first at the fifth level in the Min-Max search and at the second level in the α-β search is used to control parallelism.

In this evaluation, we demonstrated that the EM-4 is effective for dynamic load balancing, fine grain packet communication and high performance of instruction execution.

In the near future, we plan to implement a dynamic switching strategy which controls parallelism according to the load of neighboring PEs. We will also implement the full parallel α-β search, compare it with partial parallel α-β search, and make clear the advantages and disadvantages of each method in the EM-4 for the parallel game tree searching.

Furthermore, we are designing a higher performance parallel computer EM-5. This computer will reduce the overheads which are found in these evaluations such as network conflicts.

Acknowledgments

We wish to thank Dr. Toshitsugu Yuba, Director of the Computer Science Division, Mr. Toshio Shimada, Chief of the Computer Architecture Section for supporting this research, and the staff of the Computer Architecture Section for their fruitful discussions. Special thanks are due to Dr. Mitsuhisa Sato of the Computer Architecture Section and Mr. Andrew Shaw of MIT for their suggestions and careful reading.

References

[Slagle 1971] James R. Slagle, Artificial Intelligence: The Heuristic Programming Approach, McGraw-Hill Inc., (1971).

[Oki *et al.* 1989] H. Oki, K. Taki, S. Sei and S. Huruichi, The parallel execution and evaluation of a go problem on the multi PSI, Proc. of the Joint Symp. on Parallel Processing '89, (1989), 351-357.(in Japanese)

[Yamaguchi *et al.* 1989] Y. Yamaguchi, S. Sakai, K. Hiraki, Y. Kodama and T. Yuba, An Architectural Design of a Highly Parallel Dataflow Machine, Proc. of IFIP 89, (1989), 1155-1160.

[Sakai *et al.* 1989] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama and T. Yuba, An Architecture of a Dataflow Single Chip Processor, Proc. of ISCA 89, (1989), 46-53.

[Kodama *et al.* 1990] Y. Kodama, S. Sakai and Y. Yamaguchi, A Prototype of a Highly Parallel Dataflow Machine EM-4 and its Preliminary Evaluation, Proc. of InfoJapan 90, (1990), 291-298.

[Kodama *et al.* 1991] Y. Kodama, S. Sakai and Y. Yamaguchi, Load balancing by Function Distribution on the EM-4 Prototype, to appear in Supercomputing '91, (1991).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

OR-Parallel Speedups in a Knowledge Based System: on Muse and Aurora

Khayri A. M. Ali and Roland Karlsson

Swedish Institute of Computer Science, SICS
Box 1263, S-164 28 Kista, Sweden
khayri@sics.se and *roland@sics.se*

Abstract

The paper presents experimental results of running a knowledge based system that applies a set of rules to a circuit board (or a gate array) design and reports any design errors, on two OR-parallel Prolog systems, Muse and Aurora, implemented on a number of shared memory multiprocessor machines. The knowledge based system is written in SICStus Prolog, by the Knowledge Based Systems group at SICS in collaboration with groups from some Swedish companies, without considering parallelism. When the system was tested on Muse and Aurora, without any modifications, the OR-parallel speedups were very encouraging as a large practical application. The number of processors used in our experiment is 25 on Sequent Symmetry (S81), 37 on BBN Butterfly II (TC2000), and 70 on BBN Butterfly I (GP1000). The results obtained show that the Aurora system is much more sensitive to the machine architecture than the Muse system, and the latter is faster than the former on all the three machines used. The real speedup factors of Muse, relative to SICStus, are 24.3 on S81, 31.8 on TC2000, and 46.35 on GP1000.

1 Introduction

Two main types of parallelism can be extracted from a Prolog program. The first, AND-parallelism, utilizes possibilities for simultaneous execution of several subproblems offered by Prolog semantics. The second, OR-parallelism, utilizes possibilities for simultaneous search for multiple solutions to a single problem. This paper is concerned with two systems exploiting only the latter type of parallelism: Muse [Ali and Karlsson 1990a] and Aurora [Lusk *et al.* 1990]. Both systems support the full Prolog language with its standard semantics, and they have been implemented on a number of shared multiprocessor machines, ranging from a few processors up to around 100 processors. Both systems show good speedups, in comparison with good sequential Prolog systems, for programs with a high degree of OR-parallelism. The two systems are based on two different memory models. Aurora is based on the SRI [Warren 1987] and Muse on incremental copying of the WAM stacks [Ali and Karlsson 1990a]. The two systems are implemented by adapting the same sequential Prolog system, SICStus version 0.6. The extra overhead associated with this adaptation is low and depends on the Prolog program and the machine architecture. For a large set of benchmarks, the average extra overhead for the Muse system on one processor is around 5% on Sequent Symmetry, 8% on BBN Butterfly GP1000, and 22% on BBN Butterfly TC2000. For the Aurora system with the same set of the benchmarks, it is around 25% on Sequent Symmetry, 30% on BBN Butterfly GP1000, and 77% on BBN Butterfly TC2000. Earlier results [Ali and Karlsson 1990b, Ali and Karlsson 1990c, Ali *et al.* 1991a, Ali *et al.* 1991b] show that the Muse system is faster than the Aurora system for a large set of benchmarks and on the above mentioned machines.

In this paper we investigate the performance results of Muse and Aurora systems on those multiprocessor machines for a large practical knowledge based system [Holmgren and Orsvärn 1989, Hagert *et al.* 1988]. The knowledge based system is used to check a circuit board (or a gate array) design with respect to a set of rules. These rules may for example be imposed by the development tool, by company standards or testability requirements. The knowledge based system has been written in SICStus Prolog [Carlsson and Widén 1988], by the Knowledge Based Systems group at SICS in collaboration with groups from some Swedish companies, without considering parallelism. The gate array used in our experiment consists of 755 components. The system was tested on Muse and Aurora without any modifications. One important goal that has been achieved by Muse and Aurora systems is running Prolog programs that have OR-parallelism with almost no user annotations for getting parallel speedups.

The speedup results obtained are very good on all the machines used for the Muse system, but not for Aurora on the Butterfly machines. We found that this application has high OR-parallelism. In this paper we are going to present and discuss the results obtained from the Aurora and Muse systems on the three machines

used.

The paper is organized as follows. Section 2 briefly describes the three machines used in our experiment. Section 3 briefly describes the two OR-parallel Prolog systems, Muse and Aurora. Section 4 presents the knowledge based system. Sections 5 and 6 present and discuss the experimental results. Section 7 concludes the paper.

2 Multiprocessor Machines

The three machines used in our study are Sequent Symmetry S81, BBN Butterfly TC2000, and BBN Butterfly GP1000. Sequent Symmetry is a shared memory machine with a common bus capable of supporting up to 30 (i386) processors. Each processor has a 64-KByte cache memory. The bus supports cache coherence of shared data and its capacity is 80 MByte/sec. It presents the user with a uniform memory architecture and an equal access time to all memory.

The Butterfly GP1000 is a multiprocessor machine capable of supporting up to 128 processors. The GP1000 is made up of two subsystems, the processor nodes and the butterfly switch, which connects all nodes. A processor node consists of an MC68020 microprocessor, 4 MByte of memory and a Processor Node Controller (PNC) that manages all references. A non-local memory access across the switch takes about 5 times longer than local memory access (when there is no contention). The Butterfly switch is a multi-stage omega interconnection network. The switch on the GP1000 has a hardware supported block copy operation, which is used to implement the Muse incremental copying strategy. The peak bandwidth of the switch is 4 MBytes per second per switch path.

The Butterfly TC2000 is a similar to the GP1000 but is a newer machine capable of supporting up to 512 processors. The main differences are that the processors used in the TC2000 are the Motorola 88100s. They are an order of magnitude faster than the MC68020 and have two 16-KByte data and instruction caches. Thus in the TC2000 there is actually a three level memory hierarchy: cache memory, local memory and remote memory. Unfortunately no support is provided for cache coherence of shared data. Hence by default shared data are not cached on the TC2000. The peak bandwidth of the Butterfly switch on the TC2000 is 9.5 times faster than the Butterfly GP1000 (at 38 MBytes per second per path). The TC2000 switch does not have hardware support for block copy.

3 OR-Parallel Systems

In Muse and Aurora, OR-parallelism in a Prolog search tree is explored by a number of *workers* (processes or processors). A major problem introduced by OR-parallelism is that some variables may be simultaneously bound by workers exploring different branches of a Prolog search tree. Two different approaches have been used in Muse and Aurora systems for solving this problem. Muse uses incremental copying of the WAM stacks [Ali and Karlsson 1990a] while Aurora uses the SRI memory model [Warren 1987].

The idea of the SRI model is to extend the conventional WAM with a large binding array per worker and modify the trail to contain address-value pairs instead of just addresses. Each array is used by just one worker to store and access conditional bindings, i.e. bindings to variables which are potentially shareable. The WAM stacks are shared by all workers. The nodes of the search tree contain extra fields to enable workers to move around the tree. When a worker finishes a task, it moves over the tree to take another task. The worker starting a new task must partially reconstruct its array using the trail of the worker from which the task is taken.

The incremental copying of the WAM stacks used in Muse is based on having a number of sequential Prolog engines, each with its own local address space, and some global address space shared by all engines. Each sequential Prolog engine is a worker with its own WAM stacks. The stacks are not shared between workers. Thus, each worker has bindings associated with its current branch in its own copy of the stacks. This simple solution allows the existing sequential Prolog technology to be used without loss of efficiency. But it requires copying data (stacks) from one worker to another when a worker runs out of work. In Muse, workers incrementally copy parts of the (WAM) stacks and also share nodes with each other when a worker runs out of work. The two workers involved in copying will only copy the differing parts between the two workers states. The shared memory space stores information associated with the shared nodes on the search tree. Workers get work from shared nodes through using the normal backtracking mechanism of Prolog. Each worker having its own copy of the WAM stacks simplifies garbage collection, and caching the WAM stacks on machines, like the BBN Butterfly TC2000, that do not support cache coherence of shared data.

A node on a Prolog search tree corresponds to a Prolog choicepoint. Nodes are either *shared* or *nonshared (private)*. These nodes divide the search tree into two regions: *shared* and *private*. Each worker can be in either engine mode or in scheduler mode. In the engine mode, the worker works as a sequential Prolog system on private nodes, but is also able to respond to interrupt signals from other workers. Anytime a worker has to ac-

cess the shared region of the search tree, it switches to the scheduler mode and establishes the necessary coordination with other workers. The two main functions of a worker in the scheduler mode are to maintain the sequential semantics of Prolog and to match idle workers with the available work with minimal overhead.

The two systems, Muse and Aurora, have different working schedulers on the three machines used in our experiment. Aurora has two schedulers: the Argonne scheduler [Butler *et al.* 1988] and the Manchester scheduler [Calderwood and Szeredi 1989]. According to the reported results, the Manchester scheduler always gives better performance than the Argonne scheduler [Mudambi 1991, Szeredi 1989]. So, the Manchester scheduler will be used for Aurora in our experiment. Muse has only one scheduler [Ali and Karlsson 1990c, Ali and Karlsson 1991], so far.

The main difference between the Manchester scheduler for Aurora and the Muse scheduler is in the strategy used for dispatching work. The strategy used by the Manchester scheduler is that work is taken from the topmost node on a branch, and only one node at a time is shared. In Muse, several nodes at a time are shared and work is taken from the bottommost node on a branch. The bottommost strategy approximates the execution of sequential implementations of Prolog within a branch. Another difference between the two schedulers is in the algorithms used in the implementation of cut and side effects to maintain the standard Prolog semantics.

Many optimizations have been made of implementation of the Aurora and Muse systems on all the three machines. The only optimization that has been implemented for Muse and not for Aurora is caching the WAM stacks on the BBN Butterfly TC2000. In Aurora the WAM stacks are shared by the all workers while in Muse each worker has its own copy of the WAM stacks. Therefore, it is straightforward for Muse to make the WAM stack areas cachable whereas in Aurora it requires a complex cache coherence protocol to achieve this effect.

4 Knowledge Based System

One important process in the design of circuit boards and gate arrays is the checking of the design with respect to a set of rules. These rules may for example be imposed by the development tool, by company standards or by testability requirements. Until now, many of these rules have only been documented on paper. The check is performed manually by people who know the rules well. Increasing the number of gates in circuit boards (or in gate arrays) makes the manual check a very difficult process. Computerizing this process is very useful and may be the most reliable solution. The knowledge based systems group at SICS, in collaboration with groups from some Swedish companies, has been developing a knowledge based system that applies a set of rules to a circuit board (or a gate array) design and reports any design errors [Hagert *et al.* 1988, Holmgren and Orsvärn 1989]. The groups have developed two versions of the knowledge based system. The first version has been developed using a general purpose expert system shell while the second has been developed using SICStus Prolog. The latter, which will be used in our experiment, is more flexible and more efficient than the former. It is around 10 times faster than the first version on single processor machines. When it has been tested, without any modifications, on Muse and Aurora systems on Sequent Symmetry, the speedups obtained are linear up to 25 processors.

One reason for the high degree of OR-parallelism in this kind of application is that all of the rules applied to the circuit board (or a gate array) design are independent or could be made independent of each other. The second source of OR-parallelism is the application of each rule to all instances of a given circuit sub-assembly on the board. A circuit sub-assembly can be either a component (like *buffer, inverter, nand, and, nor, or, xor,* etc.) or a group of interconnected components. The knowledge based system mainly consists of an inference engine, design rules, and a database describing the circuit board (or the gate array). The inference engine is implemented as a metainterpreter with only 8 Prolog clauses. The gate array used in our experiment consists of 755 components (Texas gate array family TGC–100), which is described by around 10000 Prolog clauses. The design rules part with its interface to the gate array description is around 200 Prolog clauses. Eleven independent rules are used in this experiment. The metainterpreter applies the set of rules to the gate array description. For a larger gate array more OR-parallelism is expected. It should be mentioned that people who developed the knowledge based system did not at all consider parallelism, but they tried to make their system easy to maintain by writing clean code. They avoided using side effects, but they have used cuts (embedded in If_Then_Else) and findall constructs. The user interface part of this application is not included in our experiment.

Since Muse and the Aurora system are also running on larger machines, the BBN Butterfly machines, it was more natural to test the knowledge based system on those machines. The speedup results obtained differ for the Muse and the Aurora system. On 37 TC2000 processors, Muse is 31.8 times faster than SICStus, while Aurora is only 7.3 times faster than SICStus. Similarly, on 70 GP1000 processors Muse is 46.35 times faster than SICStus, while Aurora is only 6.68 times faster than SICStus. The low speedup for the Aurora system is surprising since this application is rich in OR-parallelism. Is this a scheduler problem for Aurora or an engine problem? The following two sections are going to present and

analyze the results of Muse and Aurora, in order to try to answer this question.

5 Timings and Speedups

In this section we present timing and speedup results obtained from running the knowledge based system on Muse and Aurora systems. The runtimes given in this paper are the **mean values obtained from eight runs**. On Sequent Symmetry, there is no significant difference between mean and best values, whereas on the Butterfly machines, mean values are more reliable than best values due to variations of timing results from one run to another[1]. Variations around the mean value will be shown in the graphs by a vertical line with two short horizontal lines at each end. The speedups given in this section are relative to running times of Muse on one processor on the corresponding machine. The SICStus one-processor runtime on each machine will also be presented to determine the extra overhead associated with adapting the SICStus Prolog system to the Aurora and Muse systems. Sections 5.1, 5.2, and 5.3 present those results on Sequent Symmetry, GP1000, and TC2000 machines, respectively.

5.1 Sequent Symmetry

Table 1 shows the runtimes of Aurora and Muse on Sequent Symmetry, and the ratio between them. Times are shown for 1, 5, 10, 15, 20, and 25 workers with speedups (relative to one Muse worker) given in parentheses. The SICStus runtime on one Sequent Symmetry processor is 422.39 seconds. This means that for this application and on the Sequent Symmetry machine the extra overhead associated with adapting the SICStus Prolog system to Aurora is 26.3%, and for Muse is only 1.0% (calculated from Table 1). The performance results that Table 1 illustrates are good for both systems, and Aurora timings exceed Muse timings by 25% to 26% between 1 to 25 workers. Figure 1 shows speedup curves for Muse and Aurora on Sequent Symmetry. Both systems show linear speedups with no significant variations around the mean values.

Table 1: Runtimes (in seconds) of Aurora and Muse on Symmetry, and the ratio between them.

Workers	Aurora	Muse	Aurora/Muse
1	533.69(0.80)	426.74(1.00)	1.25
5	106.87(3.99)	85.67(4.98)	1.25
10	53.58(7.96)	42.94(9.94)	1.25
15	36.06(11.8)	28.73(14.9)	1.26
20	27.22(15.7)	21.65(19.7)	1.26
25	21.83(19.5)	17.39(24.5)	1.26

[1]These variations are due mainly to switch contention.

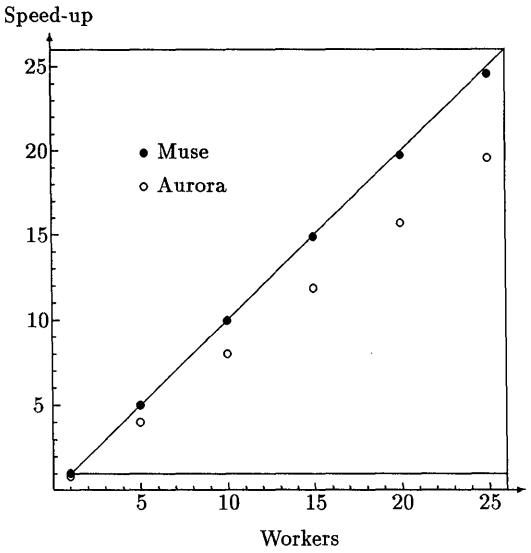


Figure 1: Speedups of Muse and Aurora on Symmetry, relative to 1 Muse worker.

5.2 BBN Butterfly GP1000

Table 2 shows the runtimes of Aurora and Muse on GP1000 for 1, 10, 20, 30, 40, 50, 60, and 70 workers. The SICStus runtime on one GP1000 node is 534.4 seconds. So, for this application and on the GP1000 machine the extra overhead associated with adapting the SICStus Prolog system to Aurora is 66%, and for Muse is only 7%. Here the performance results are good for the Muse system but not for the Aurora system. Aurora timings are longer than Muse timings by 55% to 594% between 1 to 70 workers.

Figure 2 shows speedup curves corresponding to Table 2 with variations around the mean values. The speedup curve for Aurora levels off beyond around 20 workers. On the other hand, the Muse speedup curve levels up as more workers are added.

Table 2: Runtimes (in seconds) of Aurora and Muse on GP1000, and the ratio between them.

Workers	Aurora	Muse	Aurora/Muse
1	886.4(0.65)	572.3(1.00)	1.55
10	105.3(5.44)	58.3(9.82)	1.81
20	74.1(7.72)	29.8(19.2)	2.49
30	72.7(7.88)	20.7(27.7)	3.52
40	64.3(8.91)	16.1(35.5)	3.99
50	72.4(7.90)	13.8(41.6)	5.26
60	65.7(8.71)	12.4(46.1)	5.29
70	80.0(7.15)	11.5(49.6)	6.94

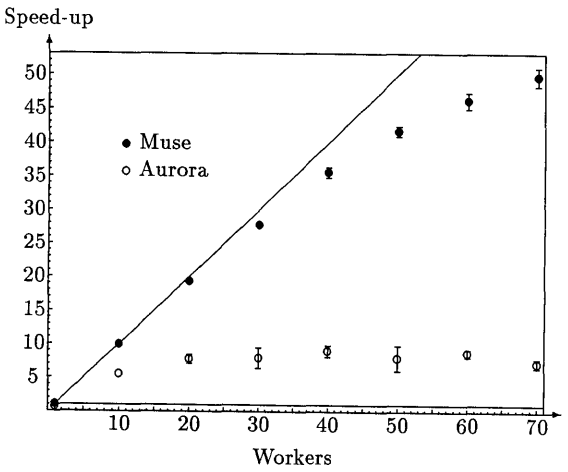


Figure 2: Speedups of Muse and Aurora on GP1000, relative to 1 Muse worker.

5.3 BBN Butterfly TC2000

Table 3 shows the performance results of Aurora and Muse on TC2000 for 1, 10, 20, 30, and 37 workers. The SICStus runtime on one TC2000 node is 100.48 seconds. Thus, for this application and on the TC2000 machine the extra overhead associated with adapting the SICStus Prolog system to Aurora is 80%, and for Muse is only

Table 3: Runtimes (in seconds) of Aurora and Muse on TC2000, and the ratio between them.

Workers	Aurora	Muse	Aurora/Muse
1	180.55(0.59)	105.97(1.00)	1.70
10	22.12(4.79)	10.81(9.80)	2.05
20	16.02(6.61)	5.56(19.1)	2.88
30	13.66(7.76)	3.93(27.0)	3.48
37	13.79(7.68)	3.29(32.2)	4.19

5%. Here also the performance results are good for the Muse system but not for the Aurora system. Aurora timings are longer than Muse timings by 70% to 319% between 1 to 37 workers.

Figure 3 shows speedup curves corresponding to Table 3. The speedup curves are similar to the corresponding ones shown in Figure 2.

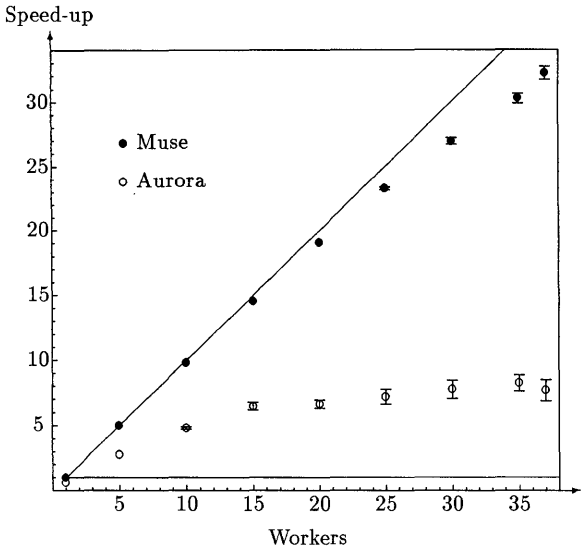


Figure 3: Speedups of Muse and Aurora on TC2000, relative to 1 Muse worker.

6 Analysis of Results

From the results presented in Section 5 we found that the Muse system shows good performance results on the three machines, whereas the Aurora system shows good results only on the Sequent Symmetry. In this section, we try to explain the reason for these results by studying the Muse and Aurora implementations on one of the Butterfly machines (TC2000). The TC2000 has better support for reading the realtime clock than the GP1000. A worker time could be divided into the following three main activities:

1. *Prolog:* time spent executing Prolog (i.e., engine time).
2. *Idle:* time spent waiting for work to be generated when there is temporarily no work available in the system.
3. *Others:* time spent in all the other activities (i.e., all scheduling activities) like spin lock, signalling other workers, performing cut, grabbing work, sharing work, looking for work, binding installation (and copying in Muse), synchronization between workers, etc.

Table 4 and Table 5 show time spent in each activity and the corresponding percentage of the total time. Results shown in Table 4 and Table 5 have been obtained from instrumented versions of Muse and Aurora on the TC2000. The times obtained from the instrumented versions are longer than those obtained from

Table 4: Time (in seconds) spent in the main activities of Muse workers on TC2000.

Muse Workers	Activity		
	Prolog	Idle	Others
1	128.36(100)	0	0
5	128.80(99.7)	0.09(0.1)	0.26(0.2)
10	129.28(99.1)	0.40(0.3)	0.71(0.5)
20	129.90(96.5)	3.56(2.6)	1.17(0.9)
30	130.32(95.4)	4.17(3.0)	2.11(1.5)

Table 5: Time (in seconds) spent in the main activities of Aurora workers on TC2000.

Aurora Workers	Activity		
	Prolog	Idle	Others
1	210.42(98.2)	0	2.36(1.1)
5	221.24(98.3)	0.19(0.1)	2.03(0.9)
10	235.34(98.1)	0.43(0.2)	2.43(1.0)
20	329.60(98.1)	1.11(0.3)	3.61(1.1)
30	412.97(94.7)	13.70(3.2)	7.64(1.8)

uninstrumented systems by around 19–27%. So, they might not be entirely accurate, but they help in indicating where most of the overhead is accrued.

Before analyzing the data in Table 4 and Table 5 we would like to make two remarks on these data. The first remark is that in the Aurora system the overhead of checking for the arrival of requests is separated from the Prolog engine time, while in the Muse system there is no such separation. This explains why there is scheduling overhead (*Others*) in the 1 worker case in Table 5 and not in Table 4. The other remark is that the figures obtained from the Aurora system do not total 100% of time, since a small fraction of the time is not allocated to any of the three activities. However, these two factors have no significant impact on the following discussion.

By careful investigation of Table 4 and Table 5 we find that the total *Prolog* time of Muse workers is almost constant with respect to the number of workers whereas the corresponding time for Aurora grows rapidly as new workers are added. We also find that the scheduling time (*Others*) in Table 5 is not very high in comparison with the corresponding time in Table 4. Similarly, the difference of *Idle* time between Muse and Aurora is not so high. So, the main reason for performance degradation in Aurora is the Prolog engine speed.

We think that the only factor that slows down the Aurora engine as more workers are added is the high access cost of non-local memory. Non-local memory access takes longer time than local memory access, and causes switch contention. Non-local memory accesses can be due to either the global Prolog tables or the WAM stacks. In Muse and Aurora systems, the global tables are partitioned into parts and each part resides in the local memory of one processor. In Aurora the WAM stacks are shared by all workers while in Muse each worker has its own copy of the WAM stacks. The global Prolog tables have been implemented similarly in the both Muse and Aurora systems. Since the Muse engine does not have any problem with the Prolog tables, the problem should lie in the sharing of the WAM stacks in Aurora, coupled with the fact that this application generates around 9.8 million conditional bindings, and executes around 1.1 million Prolog procedure calls. On the average, each procedure call generates around 9 conditional bindings. This may mean that the reason why Aurora slows down lies in the cactus stack approach, which causes a great many non-local accesses to the Prolog stacks. This results in a high amount of switch contention once over five workers. This is avoided in the Muse model, since each worker has its own copy of the WAM stacks in the processor local memory and the copy is even cachable. Unfortunately, we could not verify this hypothesis because the current Aurora implementation on the TC2000 does not provide any support for measuring the stack variables access time.

7 Conclusions

Experimental results of running a large practical knowledge based system on two OR-parallel Prolog systems, Muse and Aurora, have been presented and discussed. The number of processors used in our experiment is 25 on Sequent Symmetry (S81), 37 on BBN Butterfly II (TC2000), and 70 on BBN Butterfly I (GP1000). The knowledge based system used in our study checks a circuit board (or a gate array) design with respect to a set of rules and reports any design errors. It is written in SICStus Prolog, by the Knowledge Based Systems group at SICS in collaboration with groups from some Swedish companies, without considering parallelism. It is used in our experiment without any modifications.

The results of our experiment show that this class of applications is rich in OR-parallelism. Very good real speedups, in comparison with SICStus Prolog system, have been obtained for the Muse system on all three machines. The real speedup factors for Muse are 24.3 on 25 S81 processors, 31.8 on 37 TC2000 processors, and 46.35 on 70 GP1000 processors. The obtained real speedup factors for Aurora are lower (than for Muse) on Sequent Symmetry, and much lower on the Butterfly machines. The Aurora timings are longer than Muse timings by 25% to 26% between 1 to 25 S81 processors, 70% to 319% between 1 to 37 TC2000 processors, and 55% to 594% between 1 to 70 GP1000 processors.

The analysis of the obtained results indicates that the main reason for this great difference between Muse timing and Aurora timing (on the Butterfly machines) lies in the Prolog engine and not in the scheduler. The Aurora engine is based on the SRI memory model in

which the WAM stacks are shared by the all workers. We think that the only reason why the Aurora engine slows down as more workers are added is to be found in the large number of non-local accesses of stack variables. This results in a high amounts of switch contention as more workers are added. This is avoided in the Muse model, since each worker has its own copy of the WAM stacks in the processor local memory and even cachable in the TC2000. Unfortunately, we could not verify this hypothesis because the current Aurora implementation on the Butterfly machines does not provide any support for measuring access time of stack variables.

8 Acknowledgments

We would like to thank the Argonne National Laboratory group for allowing us to use their Sequent Symmetry and Butterfly machines. We thank Shyam Mudambi for his work on porting Muse and Aurora to the Butterfly machines. We also would like to thank Fredrik Holmgren, Klas Orsvärn and Ingvar Olsson for discussions and allowing us to use their knowledge based system.

References

[Ali and Karlsson 1990a] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, pages 129–162, Vol. 19, No. 2, April 1990.

[Ali and Karlsson 1990b] Khayri A. M. Ali and Roland Karlsson. The Muse OR-Parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776, MIT Press, October 1990.

[Ali and Karlsson 1990c] Khayri A. M. Ali and Roland Karlsson. Full Prolog and Scheduling OR-Parallelism in Muse. *International Journal of Parallel Programming*, pages 445–475, Vol. 19, No. 6, Dec. 1990.

[Ali and Karlsson 1991] Khayri A. M. Ali and Roland Karlsson. Scheduling OR-Parallelism in Muse. In *Proceedings of the 1991 International Conference on Logic Programming*, pages 807–821, Paris, June 1991.

[Ali *et al.* 1991a] Khayri A. M. Ali, Roland Karlsson and Shyam Mudambi. Performance of Muse on the BBN Butterfly TC2000. In *Proceedings of the ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, June 1991. To appear also in *Lecture Notes in Computer Science*, Springer Verlag.

[Ali *et al.* 1991b] Khayri A. M. Ali, Roland Karlsson and Shyam Mudambi. Performance of Muse on Switch-Based Multiprocessor Machines. Submitted to the *New Generation Computing Journal*, 1991.

[Butler *et al.* 1988] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605, MIT Press, August 1988.

[Calderwood and Szeredi 1989] Alan Calderwood and Péter Szeredi. Scheduling OR-parallelism in Aurora—the Manchester scheduler. In *Proceedings of the sixth International Conference on Logic Programming*, pages 419–435, MIT Press, June 1989.

[Carlsson and Widén 1988] Mats Carlsson and Johan Widén. SICStus Prolog User's Manual. SICS Research Report R88007B, October 1988.

[Hagert *et al.* 1988] G. Hagert, F. Holmgren, M. lidell and K. Orsvärn. On Methods for Developing Knowledge Systems—an Example in Electronics, Mekanresultat 88003 (in Swedish), Sveriges Mekanförbund, Box 5506, 114 85 Stockholm, 1988.

[Holmgren and Orsvärn 1989] Fredrik Holmgren and Klas Orsvärn. Towards a Domain Specific Shell for Design Rule Checking. In *Proceedings of the IFIP TC 10/WG10.2 Working Conference on the CAD Systems Using AI Techniques.* pages 221–228, Tokyo, June 6–7, 1989.

[Lusk *et al.* 1990] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora OR-parallel Prolog System. *New Generation Computing*, 7(2,3): 243–271, 1990.

[Mudambi 1991] Shyam Mudambi. Performance of Aurora on NUMA machines. In *Proceedings of the 1991 International Conference on Logic Programming*, pages 793–806, Paris, June 1991.

[Szeredi 1989] Péter Szeredi. Performance analysis of the Aurora OR-parallel Prolog System. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 713–732, MIT Press, March 1989.

[Warren 1987] David H. D. Warren. The SRI Model for OR-parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Universal Parallel Computer Architecture

William J. Dally

Artificial Intelligence Laboratory and Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
billd@ai.mit.edu

Abstract

Advances in interconnection network performance and interprocessor interaction mechanisms enable the construction of fine-grain parallel computers in which the nodes are physically small and have a small amount of memory. This class of machines has a much higher ratio of processor to memory area and hence provides greater processor throughput and memory bandwidth per unit cost relative to conventional memory-dominated machines. This paper describes the technology and architecture trends motivating fine-grain architecture and the enabling technologies of high-performance interconnection networks and low-overhead interaction mechanisms. We conclude with a discussion of our experiences with the J-Machine, a prototype fine-grain concurrent computer.

1 Introduction

Computer architecture involves balancing the capabilities of components (processors, memories, and communication facilities), organizing the connections between the components, and choosing the mechanisms that control how components interact. The top-level organization of most computer systems is similar. As shown in Figure 1, all parallel computers consist of a set of processing nodes each of which contains a processor, some memory, and a communication interface. The nodes are interconnected by a communication facility (typically a network). A sequential processor is the special case where there is only a single node and the network is used only to connect to I/O devices.

At present, the organization of processors and memories is well understood and network technology is rapidly maturing. While these components continue to evolve with improving technology and incremental architecture improvements, they do not provide significant differentiation between machines. With a convergence in machine organization, balance and mechanisms become central architectural issues and serve as the major points of differentiation.

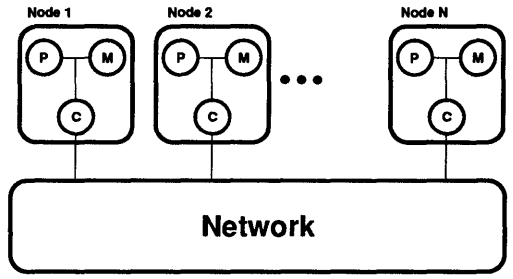


Figure 1: The structure of a parallel computer or multicomputer. All multicomputers consist of a collection of nodes connected by a network. Each node contains a processor (P), a memory (M), and a communication interface (C). Machines differ in the balance of component performance and in the mechanisms used for communication and synchronization between the nodes.

This paper explores two ideas related to balance and mechanisms. First, we propose balancing machines by cost, rather than by capacity to speed ratios. Such cost-balanced machines have a much higher ratio of processor to memory area and hence much greater processor throughput and memory bandwidth per unit cost compared to conventional machines. Cost-balanced machines are have a fine-grained physical structure. Each node is physically small and has a small amount of memory. Efficient operation with this fine-grained structure depends on high-performance communication between nodes and low overhead interaction mechanisms.

The mechanisms that control the interaction between the nodes of a parallel computer determine both the

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825, in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation, and in part by assistance from Intel Corporation.

grain-size and the programming models that can be efficiently supported. By choosing a simple, yet complete, set of primitive mechanisms, a parallel computer can support a broad range of programming models and operate at a fine grain size.

A fine-grain parallel computer with fast networks and efficient mechanisms has the potential to become a universal computer architecture in two respects. First, this class of machine has the potential to universally displace conventional (sequential and parallel) coarse-grained computers. Secondly, a simple yet efficient set of interaction mechanisms serves as the basis for a parallel computer that is universal in the sense that it runs any parallel programming system.

The remainder of this paper explores the issues of balance and mechanisms in more detail. The next section identifies trends in conventional sequential processor architecture that have led to a cost-imbalance between processors and memory. Section 3 discusses how an opportunity exists to greatly improve the performance/cost of computer systems by correcting this imbalance. The next two sections deal with the two enabling technologies: Networks (Section 4) and Mechanisms (Section 5). Together these enable fine-grain machines to give sequential performance competitive with conventional machines while greatly outperforming them on parallel applications. Our experience in building and operating a prototype fine-grain computer is described in Section 6.

2 Trends in Sequential Architecture

Two trends are present in the architecture of conventional computers:

1. The size of a processor relative to the size of its memory system is decreasing exponentially.
2. The time required for a processor to interact with an external device connected to its memory bus is increasing.

The first trend is due to an attempt to balance computer systems by ratio of processor performance (i/s) to memory capacity (bits). In 1967, Amdahl [22] suggested that a system should have 8Mbits of memory for each Mi/s of processor performance. The processor performance/size ratio (i/s $\times$ cm^2) benefits from technology improvements in both density and speed while the memory capacity/size ratio (bits/cm^2) benefits only from density improvements. Thus the processor to memory cost ratio for an Amdahl-balanced system scales inversely with speed improvements.

Let $K(67)$ denote the ratio of processor cost to memory cost for such an Amdahl-balanced system in 1967. Every Y years, the line width of the underlying semiconductor technology has halved. As a result, the area of both the processor and the memory was reduced by a factor of four [23]. At the same time, the processor speed increased by a factor of α. To keep such a system Amdahl-balanced, the capacity (and hence the size) of the memory must also be increased by α. Thus, the processor to memory ratio during year $x > 67$ is given by $K(x) = K(67)\alpha^{(67-x)/Y}$. For typical values of $\alpha = 3$ and $Y = 5$ [23], $K(92) = .004K(67)$.

The cost of a conventional machine has become largely insensitive to processor size as a result of this exponential trend in the ratio of processor to memory size. Thus, processor designers have become lavish in their use of area[1]. Costly features such as large caches, complex data paths, and complex instruction-issue logic are added even though their marginal affect on processor performance (compared to a small cache and a simple organization) is minor. As long as the size of the machine is dominated by memory, adding area to the processor has a small effect on overall size and cost.

The second trend, the increase in external interaction latency, is due to the first trend, to the increasing difference in on-chip to off-chip signal energies, and to to deepening memory hierarchies. As processors get faster and memory size increases the number of processor cycles required to access memory increases. Modern microprocessor-based computers have a latency of 5-20 cycles for a main memory access and this number is increasing. At the same time, decreasing on-chip signal energies require greater amplification to drive off-chip signals. Also, as more levels of caching are introduced, the number of cycles expended before initiating an external memory reference increases and the memory interface becomes specialized for the transfer of cache lines.

If a conventional processor is used in a parallel computer, its high external interaction latency limits its communication performance as the network must typically be accessed via the external memory interface. Whether this interface uses DMA to transfer data stored in memory (and possibly cached) or uses writes to a memory-mapped network port, each word of the message must traverse the external memory bus and the cost of initiating an external memory operation is incurred at least once. The slow external memory interface also contributes to the lack of agility in modern processors (that is, their slowness in responding to external events and switching tasks) because a great deal of processor state must be transferred to and from memory during these operations.

These trends in conventional processor architecture

[1] As a result of this lavish use of area, processor sizes have scaled slightly slower than predicted by the formula above.

make conventional processors ill-suited for use in a parallel computer. Current cost-insensitive processors are not cost effective in a machine with higher processor to memory ratio where the cost of the processor is an important factor. Their high external interaction latency severely limits their communication performance and their poor agility limits their ability to handle synchronization.

This does not mean, however, that conventional instruction set architectures (ISAs) are unsuitable for parallel computing. Rather it is the cost-insensitive design style, deep memory hierarchies, and poor agility that are the problem. As we will see in Section 5, a conventional ISA can be extended with a few instructions to provide an efficient set of parallel mechanisms.

Most importantly, the trend toward ever higher memory to processor size ratios has created an enormous opportunity for parallel computing to improve the performance/cost of computers. By adding more processors while keeping the amount of memory constant, the performance of the machine is dramatically increased with little impact on cost. The current trend, however, of building parallel computers by simply replicating workstation-sized units (increasing processors and memory proportionally) does not exploit this advantage. The processor to memory ratio must be decreased to improve efficiency. This theme is explored in more detail in the next section.

3 Balance

Balance, in the context of computer architecture, refers to the ratios of throughput, latency, and capacity of different elements of a computer. In this section we will explore the balance between processor throughput, memory capacity, and network throughput in a parallel computer. A case will be made for balancing machines based on cost[2].

Traditionally, machines have been balanced by rules of thumb such as the one due to Amdahl discussed above. However, a more economical design results if a machine is balanced based on cost. A machine is *cost-balanced* when the incremental peformance increase due to an incremental increase in the cost of each component is equal. Let each component k_i in a machine with performance P have cost c_i, then the machine is *cost-balanced* if $\partial P/\partial c_i = \partial P/\partial c_j; \forall i,j$ [7].

It is difficult to solve these balance equations because (1) no analytic function exists that relates system performance to component cost and (2) this relationship varies greatly depending on the application being run. Also, analyzing existing applications can be misleading as they have been tuned to run on particular machines and hence reflect the balance of those machines.

A workable approach is to start from the present memory-dominated system and increase the processor and network costs until they reach some fraction of total cost, for example 10%. At this point the system costs a small fraction more than a conventional system. If designed with an appropriate communication network (Section 4) and mechanisms (Section 5), it should provide sequential performance comparable to that of a conventional machine. Applications that are parallelized to take advantage of the machine can potentially speed up by the entire increase in processing cost.

To make reasonable balancing decisions, it is important to use manufacturing cost, not component price, as our measure of cost. This avoids distorting our analysis due to the widely varying pricing policies of semiconductor vendors. To simplify our analysis of cost, we will use silicon area normalized to half a minimum line width, λ, as our measure of cost [27].

First consider the issue of processor to memory balance. There are two issues: (1) how large a processor to use on each node and (2) how much memory per processor. A 64-bit processor with floating point but no cache and simple issue logic currently costs about $100M\lambda^2$, about the same as 500Kbits of DRAM, and has a performance of 50Mi/s. Making a processor larger than this gives diminishing returns in performance as heroic efforts are made to exploit instruction-level parallelism [20]. A smaller processor may improve efficiency slightly. If we are allocating 10% of our cost to processors, we will build one processor for every 5Mbits of memory – rounding up this gives one processor per MByte. In today's technology a processor of this type with 1MByte of memory can easily be integrated on a single chip. In comparison, an Amdahl-balanced machine would provide 64MBytes of memory for each processor and be packaged in 30-50 chips.

Providing a small cache memory for the processor is cost effective; however a large cache and/or a secondary cache are not. Adding a small 4KByte I-cache and D-cache requires about $16M\lambda^2$ of area and greatly boosts processor performance achieving hit rates greater than 90% on many codes [35]. Making the cache much larger or deepening the memory hierarchy would greatly increase processor area with a very small return in performance. Also, using a small co-located memory reduces processor access time to DRAM memory.

The network to memory balance is achieved in a similar manner, by adding network capability until cost is increased by a small fraction. A great deal of network performance comes at very little cost. The PC (printed-circuit) boards on which the processor-memory chips are mounted have a certain wiring capacity and the periphery of the chips can support a certain number of I/O

[2]Much of the material in this section is based on a joint work in progress with Prof. Anant Agarwal of MIT.

pads[3]. The network can make use of most of these pin and wire resources at a very small cost. The cost of the network router itself is small; a competant router can be built in less than $10M\lambda^2$ [16]. For example, a router on an integrated processor-memory chip could easily support 6 16-bit wide channels from which a 3-D network can be constructed (Section 4). Conventional PC boards and connectors can easily handle these signals.

Attempting to increase network bandwidth beyond this level becomes very expensive. To add more channel pins, the router must be moved to a separate chip or even split across several chips incurring additional overhead for communication between the chips. These chips are pad-limited and most of their area is squandered. If the amount of memory per node is increased proportionally to the cost of the network router to hold the memory to network cost ratio constant, the network bandwidth per bit of memory decreases (and the processor to memory ratio is distorted).

A computer design can be approximately cost-balanced by using technology constraints to determine the processor/memory/network ratios. A simple three step method gives a well cost-balanced system:

1. Size the processor to the knee of its performance/cost curve to get a cost effective processor.
2. Set the processor to memory ratio to allocate a fixed fraction γ (in the example above 0.1) of cost to the processor to get a machine that is within $1/1-\gamma$ of the optimium cost.
3. Holding processor and memory sizes constant, size the network to the knee of its performance/cost curve to get a cost effective network.

Machines that are cost-balanced using this method offer aggregate processor performance and local memory bandwidth that is 50 times that of an Amdahl-balanced machine per unit cost. This performance advantage will expand by a factor of α every Y years.

Why are coarse-grained Amdahl-balanced machines widespread both in uniprocessors and parallel computers? In uniprocessors, the number of processors is not a free variable. Thus the designer is driven to increase the size and cost of a single processor far past the knee of its performance/cost curve.

Existing parallel computers are driven to a coarse grain-size because (1) they are built using processors that lack appropriate mechanisms for communication and synchronization, (2) their networks are too slow to provide fast access to all memory in the machine[2], and (3) converting software to run in parallel on these machines requires considerable effort [21]. Much of the difficultly associated with (3) is due to the partitioning requied to get good performance because of 1 and 2.

For cost-balanced machines to be competitive, increasing the number of processors must (1) not substantially reduce single-processor performance and (2) must provide the potential for near-linear speedup on certain problems. To retain single-processor performance on a machine with a small amount of memory per node, the network and processor communication mechanisms must provide a single processor access to any memory location in the machine in time competitive with a main memory access in a conventional machine. Single-processor performance depends on network latency. To provide speedup on parallel applications, the processor's communication and synchronization mechanisms must provide for low-overhead interaction and the network throughput must be sufficient to support the parallel communication demands. Parallel speedup depends on throughput and agility.

The two key technologies for building cost-balanced machines are efficient networks, and processor mechanisms for communication and synchronization. The next two sections explore these technologies in more detail.

4 Network Architecture and Design

The interconnection network is the key component of a parallel computer. The network accepts messages from each processing node of a parallel computer and delivers each message to any other processing node. Latency, T, and throughput, λ_s, characterize the performance of a network. Latency is the time (s) from when the first bit of the message leaves the sending node to when the last bit of the message arrives at the receiving node. Aggregate throughput λ_{sN} is rate of message delivery (bits/s) when the network is fully loaded.

T must be kept low to achieve good performance for sequential codes and for the portions of parallel codes where the parallelism is insufficient to keep the machine busy. During these periods performance is latency-limited and execution time is proportional to T. During periods where there is abundant parallelism, performance is throughput limited. Recent developments in network technology give throughputs and latencies that approach physical and information theoretic bounds given pin and wire constraints. A detailed discussion of this technology is beyond the scope of this paper. This section briefly summarizes the major results.

An interconnection network is characterized by its topology, routing, and flow control [11]. The topology of

[3]Typical PC boards support 20wires/cm on each of 4-8 wiring layers. Typical ICs support 100pads/cm along their periphery with 20-50% of these pads reserved for power.

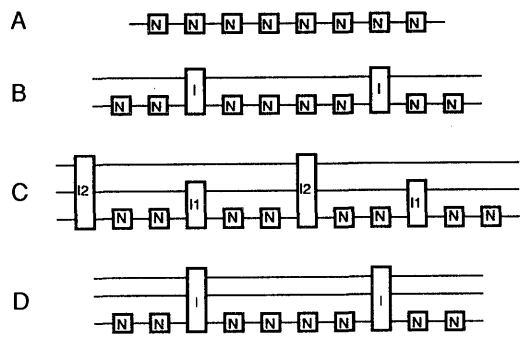


Figure 2: Insertion of express channels into a k-ary 3-cube gives performance within a small factor of physical limits : (A) One dimension of a regular k-ary 3-cube network, (B) Inserting one-level of express channels optimizes the ratio of wire to node delay for messages travelling long distances, (C) Hierarchical express channels also reduce the number of switching decisions to the minimum, $\log_q N$. (D) Adding multiple channels at each level adjusts network bisection bandwidth to maximize throughput.

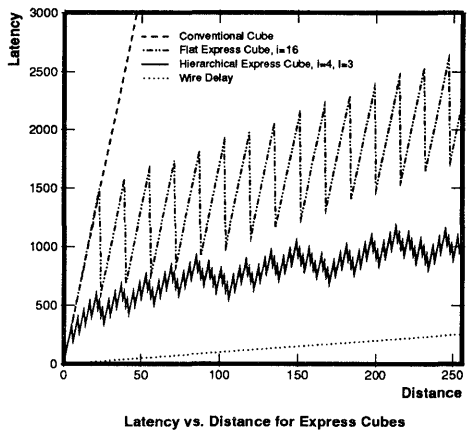


Figure 3: Latency as a function of distance for a hierarchical express channel cube with $i = 4$, $l = 3$, $\alpha = 64$, and a flat express channel cube with $i = 16$, $\alpha = 64$. In a hierarchical express channel cube latency is logarithmic for short distances and linear for long distances. The crossover occurs between $D = \alpha$ and $D = i\alpha \log_i \alpha$. The flat cube has linear delay dominated by T_n for short distances and by T_w for long distances.

a network is the arrangement of nodes and channels into a graph. Routing specifies how a packet chooses a path in this graph. Flow control deals with the allocation of channel and buffer resources to a packet as it traverses this path.

The topology strongly affects T since it determines (1) how many hops H a message must make, (2) the total wire distance D (cm) that must be traversed, and (3) the channel width W (bits) which is limited by the bisection width of the wiring media divided by the channel bisection of the network[4]. The latency seen by a single message in a network with no other traffic (zero-load latency or T_0) is directly determined by these three factors:

$$T_0 = HT_n + \frac{D}{v} + \frac{L}{Wf}. \qquad (1)$$

Where T_n is the propagation delay of a node (s), v is the signal propagation velocity[5] (cm/s), and f is the wire bandwidth (s^{-1}).

The three-dimensional express cube topology [12], a k-ary 3-cube with express channels added to skip intermediate hops (Figure 2B) when travelling large distances, can simultaneously optimize H, D, and λ_{sN} to achieve performance that is within a small fraction of physical and information-theoretic limits. The number of hops H is bounded by $log_q N$ if a q way decision is made at each step. The express cube network achieves this bound by inserting a hierarchy of interchanges into a k-ary n-cube network (Figure 2C). The wire distance, D is kept to within $2^{-1/3}$ of the physical minimum by always following a manhattan shortest path. Finally, the number of network channels can be adjusted to use all available wiring capacity (Figure 2D).

Figure 3 compares the performance of flat and hierarchical express cubes with a regular k-ary n-cube and a wire with no switching. The ratio of the delay of a node, $T-n$, to the delay of a wire between two adjacent nodes, $D(1)/v$, is denoted $\alpha = T_n v/D(1)$. The figure assumes $\alpha = 64$. The figure shows that a flat express cube decreases delay to a multiple of wire delay determined by the ratio α to interchange spacing, i. Interchange spacing is set to the square-root of the distance to balance the delay due to local channels with the delay due to express channels. The hierarchical cube with three levels ($l = 3$) permits small interchange spacing and allows local and global delays to be optimized simultaneously.

The advantages of minimum H and maximum λ_{sN} achieved by the express cube topology are important for very large networks. For smaller networks (less than 4K nodes), however, a simpler three-dimensional torus or mesh network, k-ary 3-cube, is usually more cost ef-

[4]In some small networks, W is constrained by component or module pinout and not by bisection width.

[5]Typically v is a fraction of the speed of light $0.3c \leq v \leq c$.

fective. The 3-D mesh also provides manhattan shortest paths in physical space to keep D near minimum, has a very regular structure, and uses uniformly short wires simplifying the electrical design of the network.

Three-dimensional networks are required to obtainin adequate throughput for machines larger than 256 nodes. As machines grow, the throughput per node varies inversely with the number of nodes in a row, as $N^{1/2}$ for a 2-D network and as $N^{2/3}$ for a 3-D network. 3-D networks provide adequate throughput up to 4K nodes (16 nodes per row). Beyond this point express cubes and/or careful management of locality is required. For machines of 256K or larger, express cubes become bisection-limited and locality must be exploited. No cost-effective network can scale throughput linearly with the size of the machine. Above a certain size, all networks become bisection-width limited and hence have a throughput that grows as $N^{2/3}$.

Routing, the assignment of a path to a message, determines the static load balance of a network. Most routers built to date have used deterministic routing – where the path depends only on the source and destination nodes. Deterministic routers can be made simple and fast, and deadlock avoidance becomes much easier. In particular, deterministic routing in dimension order permits the switch to be cleanly partitioned [17]. For some traffic patterns, deterministic routing results in a degradation in performance due to channel load imbalance. However, for most cases deterministic routing has proved adequate.

Several adaptive routing algorithms have been proposed [14, 4, 25] that are capable of dynamically detecting and correcting channel load imbalance. Adaptive routers also are able to route around a number of faulty nodes and channels. Most adaptive routers require much more complex logic than deterministic routers. The planar adaptive routing algorithm [4] is particularly attractive in that it retains much of the simplicity of dimension-order routing.

Flow control involves dynamically allocating buffer and channel resources to messages in the network. Most parallel computer networks use wormhole routing [8] in which buffers are allocated to messages while channels are allocated to flow-control digits or *flits*. To keep routers small and fast, channel buffers are often shorter than messages. Thus it is possible for a message to be blocked on the receiving side of a channel while part of the message remains on the transmitting side. With only a single buffer per channel, blocking a message on the transmitting side would idle the channel wasting network resources.

Virtual-channel flow control permits messages to pass blocked messages and make use of what would otherwise be idle channels [13]. By associating several buffers (virtual channels) with each physical channel and multiplexing them on demand, a network loaded with uniform traffic can operate at 90% of its peak channel capacity. In comparison, the throughput of a network with only a single buffer per node saturates at 20% to 50% of capacity depending on the topology and routing. Virtual channel flow control uses several small, independent buffers in place of a single large queue to more efficiently use valuable router storage. Figures 4 and 5 show the effect of adding virtual channels to the latency and throughput of 2-ary n-fly networks.

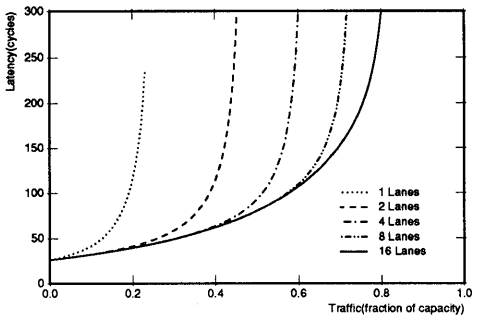


Figure 4: Latency as a function of offered traffic for a 2-ary 8-fly network with 1, 2, 4, 8, and 16 virtual channels per physical channel.

The network technology described above is able to meet the goal of providing global memory access with a latency comparable to that of a uniprocessor. Compare for example a 64-node 3-D torus with 1MByte per node with a comparably sized single processor machine with 64Mbytes. Both of these machines will fit comfortably on a desktop. Since network channels are uniformly short it is customary to operate them at twice the processor rate [10] (or more [5]). For our comparison we will use a processor rate of 50MHz and a network clock of 100MHz.

The 64-node torus requires an average of 6 hops to reach any node in the machine (HT_n=60ns). A message of six 16-bit flits (L/Wf=60ns) is sent in each direction for a read operation. The composition time of the message and the initiation of the memory access can be overlapped with this L/Wf. Thus the one-way communication time is 120ns. The memory access itself takes 100ns. Adding the reply communication time (again terminal operations are overlapped with the L/Wf time) gives a total access time of 340ns. The uniprocessor requires 1 cycle to get off chip, 2 cycles to get across a bus, and 1 cycle to initiate the memory operation (80ns total). Again the memory read itself is 100ns and the reply across the bus requires another 80ns for a total of

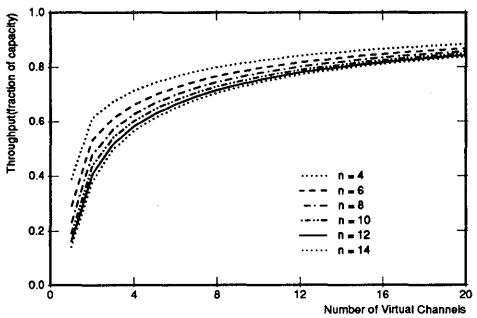


Figure 5: Throughput of 2-ary n-fly networks with virtual channels as a function of the number of virtual channels.

260ns. Thus the uniprocessor is only 80ns or 24% faster. Much of the additional delay can be attributed to the fact that the parallel computer network has more decisions to make during routing and is able to handle many messages simultaneously. While these capabilities have a slight negative affect on latency, they give a significant throughput advantage.

To see the throughput advantage, consider the problem of rotating a matrix about its center row. To perform one 64-bit move, the conventional machine requires two memory cycles or 520ns for a rate of 123Mbits/s. With an interleaved memory and a lockup-free memory interface (which few processors have) it could overlap operations to complete one every 160ns for a rate of 400Mbits/s. The parallel computer on the other hand can apply its entire bidirectional bisection bandwidth of 256 16-bit channels to the problem for a total bandwidth of 409.6GBits/s.

In summary, modern interconnection network technology gives latency comparable to conventional memory access times with throughput orders of magnitude higher. Raw network performance solves only half of the communication problem, however. To use such a network effectively requires efficient communication mechanisms.

5 Mechanisms

Mechanisms are the primitive operations provided by a computer's hardware and systems software. The abstractions that make up a programming system are built from these mechanisms [18, 9]. For example, most sequential machines provide some mechanism for a push-down stack to support the last-in-first-out (LIFO) storage allocation required by many sequential models of computation. Most machines also provide some form of memory relocation and protection to allow several processes to coexist in memory at a single time without interference. The proper set of mechanisms can provide a significant improvement in performance over a brute-force interpretation of a computational model.

Over the past 40 years, sequential von Neumann processors have evolved a set of mechanisms appropriate for supporting most sequential models of computation. It is clear, however, from efforts to build concurrent machines by wiring together many sequential processors, that these highly-evolved sequential mechanisms are not adequate to support most parallel models of computation. These mechanisms do not support synchronization of events, communication of data, or global naming of objects. As a result, these functions, inherent to any parallel model of computation, must be implemented largely in software with prohibitive overhead.

For example, most sequential machines require hundreds of instructions to create a new process or to send a message. This cost prohibits the use of fine-grain programming models where processes typically last only a few tens of instructions and messages contain only a few words. It is not hard to construct mechanisms that permit tasks to be created and messages sent in a few instruction times; however, these mechanisms are not to be found on conventional processors.

Some parallel computers have been built with mechanisms specialized for a particular model of programming, for example dataflow or parallel logic programming. However, our studies have shown that most programming models require the same basic mechanisms for communication, synchronization, and naming. More complex model-specific mechanisms can be built from the basic mechanisms with little loss in efficiency. Specializing a machine for a particular programming model limits its flexibility and range of application without any significant gain in performance. In the remainder of this section, we will examine mechanisms for communication, synchronization, and naming in turn.

Communication between two processing nodes involves the following steps:

1. *Formatting:* gathers the message contents together.
2. *Addressing:* selects the physical destination for the message.
3. *Delivery:* transports the message to the destination.
4. *Allocation:* assigns space to hold the arriving message.
5. *Buffering:* stores the message into the allocated space.

6. *Action:* carries out a sequence of operations to handle the message.

All programming models use a subset of these basic steps. A shared memory read operation, for example, uses all six steps. A read message is formatted, the address is translated, the message is delivered by the network, the message is buffered until the receiving node can process it, and finally a read is performed and reply message is sent as the action. Some models, such as synchronous message passing always send messages to preallocated storage and thus omit allocation (step 4). In some cases, no action is required to respond to a message and step 6 can be omitted.

The SEND instruction, first used in the message-driven processor [15, 16], with translation of destination addresses [19] efficiently handles the first two steps: formatting and addressing. A message is sent with a sequence of SEND instructions followed by a SENDE instruction. A SEND instruction takes a number of arguments equal to the number of read register ports (typically two) and appends its arguments to a message. A SENDE instruction is identical to the SEND except that it also signals the end of the message. The first SEND after a SENDE starts a new message. By making full use of the register bandwidth the SEND instruction reduces formatting overhead to a minimum. The alternative approaches of formatting a message (1) in memory or (2) by writing to a memory mapped network port have much lower bandwidth and higher latency.

Translation is achieved by interpreting the first word of the message stream (the first argument of the first SEND) as a virtual destination address and translating it to a physical address when a message is sent. A simple translation-lookaside buffer (TLB) efficiently performs this translation. This approach of translating virtual network addresses to physical addresses during the SEND operation permits message sends from user code to be fully protected without incurring the overhead of a sytem call (as is done on many machines today). User code is only permitted to send messages to addresses that are entered in TLB. Sending a message to any other address raises an exception.

Communication operations that do not require allocation and or remote action can use a subset of the basic mechanism. A remote write operation, for example, requires neither of these functions. Avoiding allocation and action in this case eliminates the overhead of copying the message from newly allocated storage to its final destination. The first SEND instruction of a message can specify whether allocation (.A suffix) and/or spawning a task (.S suffix) are required [19]. A SEND with no suffix would simply perform a remote write, SEND.A would allocate but not initiate a remote action, and SEND.SA would do both. The sending node treats these three SEND operations identically and simply sends along the two option bits with the message. The receiving node examines the option bits to determine whether allocation and/or action is required. If an action is required, the routine to be invoked is specified by the second word of the message.

Storage allocation and message buffering must be performed in hardware to achieve adequate performance. While approaches using stack (LIFO) or queue (FIFO) based storage are simple to implement [10], they may require copying if messages are not deallocated in order. An alternative is to allocate message buffers off a free list of fixed-sized segments [40]. Management of such a free list is simple (only a single pointer is required) and it does not restrict message lifetimes. Messages too long for the fixed-sized segments can be handled in an overflow area.

With any allocation scheme, a method for handling message buffer overflow is required. Because handling an overflow may require access to other nodes, the network must be usable even when a full buffer is causing messages to back up into the network. This is accomplished on the J-Machine by using two virtual networks [10]. The actual overflow handling may be performed in software as it is a rare event. While many strategies may be used to handle overflow, a simple one is to return overflowing messages to their senders. With this scheme each node must guarantee that it has storage to hold each message it originates until it is acknowledged.

The final step of a communication operation is to initiate a remote action by creating and dispatching a task. A task or process consists of a thread of control and an addressing environment. A thread can be created in a few clock cycles by loading a processor's IP to set the thread of control and initializing its memory management registers to alter the addressing environment. On the J-Machine, each message in the message queue is treated as a thread that is ready to run and threads are dispatched when they reach the head of the queue This dispatching on message arrival also serves as the basis of a synchronization mechanism.

Synchronization enforces an ordering of events in a program. It is used, for example, to ensure that one process writes a memory location before another reads it, to provide mutual exclusion during critical sections of code, and to require all processes to arrive at a barrier before any processes leave.

Any synchronization mechanism requires a namespace that processes use to refer to events, a method for signalling that an event is enabled, and a method for forcing a processor to wait on an event. Using tags for synchronization, as with the presence bits on the HEP [36], uses the memory address space as the synchronization namespace. This provides a large synchronization

namespace with very little cost as the memory manangement hardware is reused for this function. It also has the benefit that when signaling the availability of data, the data can be written and the event signaled in a single memory operation. Since it naturally signals the presence of data, we refer to this synchronization using tags on memory words as *data synchronization*[40].

With synchronization tags, an event is signaled by setting the tag to a particular state. A process can wait on an event by performing a synchronizing access of the location which raises an exception if the tag is not in the expected state. A synchronizing access may optionally leave the tag in a different state. Simple producer/consumer synchronization can be performed using a single state bit. In this case, the producer executes a synchronizing write which expects the tag to be empty and leaves it full. A synchronizing read which expects the location to be full and leaves it empty is performed by the consumer. If the operations proceed in order, no exceptions are raised. An attempt to read before a write or to write twice before a single read raises a synchronization exception. More involved synchronization protocols require additional states (for example to signal that a process is waiting on a location) [19].

The communication mechanism described above complements data synchronization by providing a means for a process on one node to signal an event on a remote node. In the simplest case, a message handler can perform a synchronizing read or write operation. However, it is often more efficient to move some computation to the node on which the data is resident. Consider for example the problem of adding a value to a remote location[6]. One could perform a remote synchronizing read that marks the location empty to gain exclusive access, perform the add, and then perform a remote synchronizing write. Sending a single message to invoke a handler that performs the read, add, and write on the remote node, however, reduces the time to perform the operation, the number of messages required, and the amount of time the location is locked.

Many machines have implemented some form of global barrier synchronization. For example, the Caltech Cosmic Cube [32] had four program accessible wire-or lines for this purpose. While global barrier synchronization is useful for some models, it can be emulated rapidly using communication and data synchronization. If there is sufficient slack time from when a process signals that it has reached the barrier to when it waits on the barrier, this emulation will not affect program performance. The required amount of slack time varies logarithmically with the number of processors performing the barrier. Also, the major use of barrier synchronization (inserting a barrier between code that produces a structure (e.g., array) and code that consumes the structure) is eliminated by data synchronization. By synchronizing in the data space on each individual element of the data structure, control space synchronization on the program counter between the producer and consumer is neither required nor desired. It is more efficient to allow the producer and consumer to overlap their execution subject to data dependency constraints. Barrier synchronization mechanisms also have the disadvantage that they require a separate namespace which tends to be small because of the prohibitive cost of providing many simultaneous barriers, and they consume pin and wire resources that could otherwise be used to speed up the general communication network.

The mechanism that enforces event ordering solves only half of the synchronization problem. Efficient synchronization also requires an *agile* processor that can rapidly switch processes and handle events and messages to reduce the exception handling and context switching overhead when switching processes while waiting on an event. Rapid task switching can be provided by providing multiple register sets or a named-state register set [29]. Exception handling is accelerated by specifically vectoring exceptions, providing separate registers for exception handling, and explicitly passing arguments to exception handlers [19].

6 Experience

In the Concurrent VLSI Architecture Group at MIT, we have built the J-Machine [10], a prototype fine-grain parallel computer with a high-speed network and efficient yet general communication and synchronization mechanisms. The J-Machine was built to test and evaluate our ideas on mechanisms and networks, as a proof of concept for this class of machine, and as a testbed for parallel software research. Small prototoypes have been operational since June of 1991. We expect to have a 1024-processor J-Machine on-line during the summer of 1992.

The J-Machine communication mechanism permits a node to send a message to any other node in the machine in $< 1.5\mu s$. On message arrival, a task is created and dispatched in 200ns. A translation mechanism supports a global virtual address space. These mechanisms efficiently support most proposed models of concurrent computation and allow parallelism to be exploited at a grain size of 10 operations. The hardware is an ensemble of up to 65,536 nodes each containing a 36-bit processor, 4K 36-bit words of on chip memory, 256K words of DRAM, and a router. The nodes are connected by a high-speed 3-D mesh network with deterministic dimension order routing. The J-Machine has about the grain

[6]This occurs for example when performing LU decomposition of a matrix.

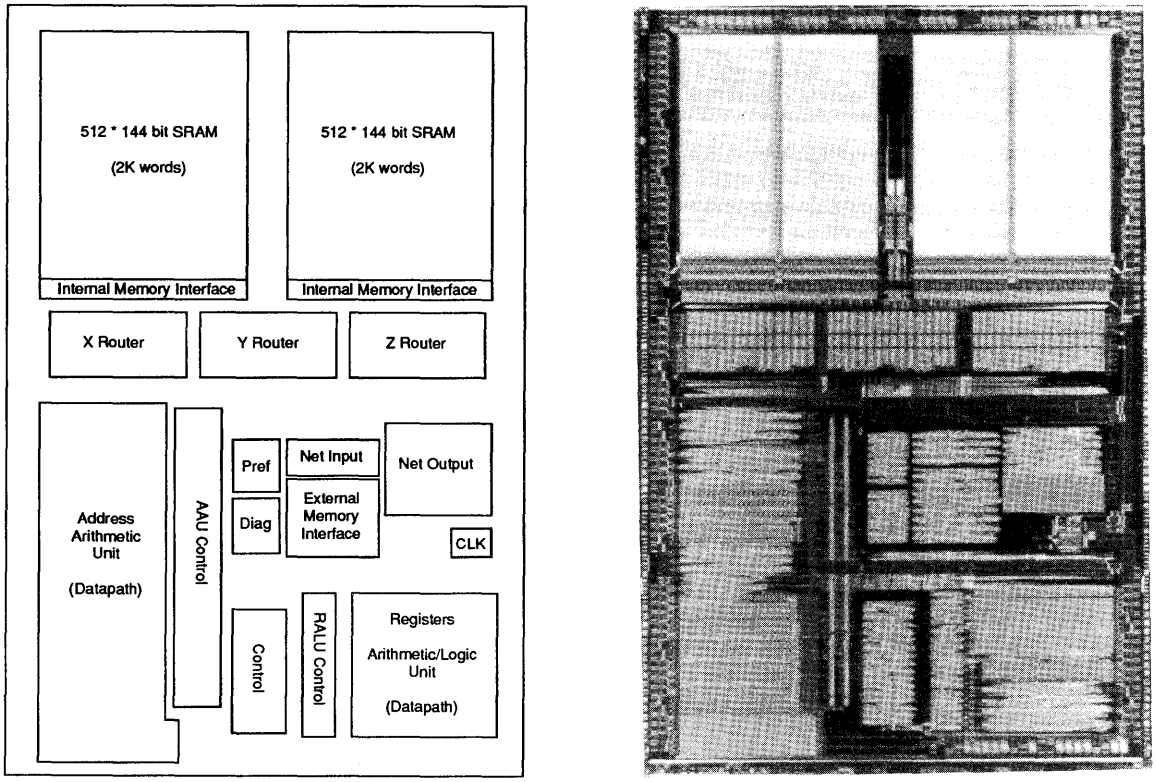


Figure 6: Floorplan and Photograph of a Message-Driven Processor chip.

size of the cost-balanced machine described in Section 3, one processor per megabyte of memory.

A photograph of the message-driven processor chip used in the J-Machine is shown in Figure 6. One of these chips combined with three external DRAM parts forms a J-Machine node. An array of 64 nodes is packaged on a single board (Figure 7). These boards are stacked and connected side-to-side to form larger J-Machines.

Three software systems are currently operational on the J-Machine. It runs Concurrent Smalltalk (CST) [24], a version of Id based on the Berkeley TAM system [37, 6], and a dialect of "C". Execution of these diverse programming systems has demonstrated the efficiency and flexibilty of the J-Machine mechanisms.

Table 1 shows the advantage of efficient mechanisms. The left column of the table lists the operations involved in performing a remote memory reference on a 1024-node parallel computer. The next two columns list the approximate number of instruction times required to perform each operation on the Intel Paragon [5] and

Operation	Paragon	J-Machine	Ideal
Send 4-Word Message	600	3	2
Network Delay	32	10	10
Buffer Allocation	20	0	0
Switch To Handle Msg	1000	10	1
Presence Test	5	0	0
Send 3-Word Return Msg	600	3	2
Network Delay	32	10	10
Buffer Allocation	20	0	0
Switch To Handle Msg	1000	3	1
Switch To Restart Task	1000	10	1
TOTAL	4309	49	27

Table 1: The time to perform a remote memory reference on the Intel Paragon, a conventional message-passing multicomputer, the J-Machine, a fine-grain parallel computer, and the time that could be achieved with current technology (Ideal). Switch refers to a task switch.

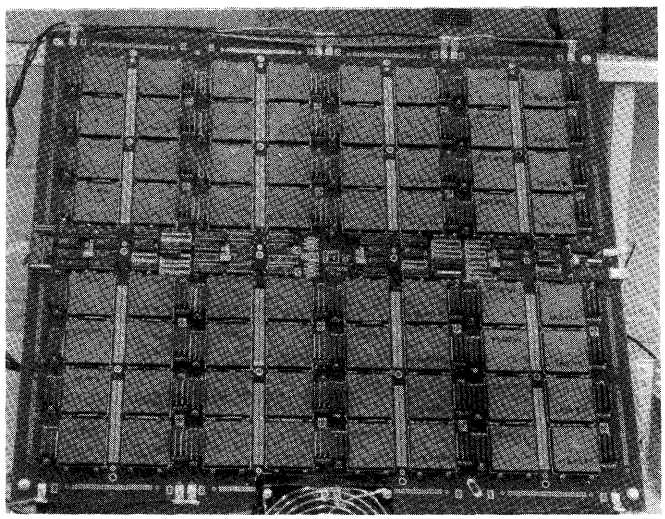


Figure 7: Photograph of a 64-node J-Machine board.

the J-Machine. Many of these times were derived from the study reported in [38]. The final column of the table shows the times that could be achieved with techniques that are currently understood.

The table shows that while both machines have fast networks the time to carry out a simple remote action is many times greater on the conventional machine. The single largest contributor is the task switching time[7]. The overhead of task switching in a conventional operating system is unacceptable in this environment. Even if the task switch time were reduced to zero, the overhead of sending a message[8] in a system where this function is handled in software is still prohibitive. End-to-end hardware support for communication is required to achieve acceptable latency.

The rightmost column represents times that could be achieved by making some minor modifications to the J-Machine. In particular, task switch time could be reduced from 10 cycles (when registers need to be saved) or 3 cycles (w/o register save) to a single cycle by providing more support for multithreading [29, 39]. The J-Machine would also benefit from more user registers, automatic destination translation on message send, being able to subset the communication operation, and a non-LIFO message buffer.

[7]The estimate of 1000 instruction times or 25μs for the i860 is extrapolated from other microprocessors and hence very generous; because of the complexity of event handling on this chip, the actual number is higher.

[8]Some receive time is also included in this number.

7 Related Work

Like the message-driven processor from which the MIT J-Machine is built, the Caltech MOSAIC [33], Intel iWARP [3], and INMOS Transputer [26] are integrated processing nodes that incorporate a processor with memory and communication on a single chip. These integrated nodes, however, lack the efficient mechanisms of the MDP and thus cannot efficiently support many different models of computation. Also, the software-routed, bit-serial Transputer network does not have adequate performance for many applications.

Many machines built for a specific model of computation have been generalizing their mechanisms. For example, the MIT Alewife machine [1], while specialized for the shared-memory model, provides an interprocessor interrupt facility that can be used for general message-passing. Being memory mapped, this operation is somewhat slower than the register-based send operation described above. Dataflow machines, which once hard-wired a particular dataflow model into the architecture [30, 34], have also been moving in the direction of general mechanisms with the EM4 [31] and *T [28].

8 Conclusion

Two enabling technologies, fast networks (Section 4) and efficient interaction mechanisms (Section 5), make it possible to build and program fine-grain parallel computers. Fine-grain machines have much less memory per processor than conventional machines because they are balanced by cost, rather than by capacity to speed ratios. Increasing the processor to memory ratio improves the processor throughput and local memory bandwidth by a factor of 50 with only a small increase in system cost.

We expect this dramatic performance/cost advantage will lead to mechanism-based fine-grain parallel computers becoming universal, replacing sequential computers in all sizes of systems from personal desktop computers to institutional supercomputers. This universal parallel computer will not happen with existing semiconductor price structures, where processor silicon is an order of magnitude more expensive per unit area than memory silicon. Cost effective fine-grain computing requires a true *jellybean* (inexpensive and plentiful) processing-node chip.

Low-latency networks enable each node in a fine grain machine to access any memory location in the machine in time competitive with a global memory access in a conventional machine. Thus, the small memory per node does not limit either the problem size that can be handled or sequential execution speed. A fine-grain machine can execute sequential programs with performance competitive with conventional machines.

High-bandwidth networks and efficient interaction mechanisms enable fine-grain computers to apply their high aggregate processor throughput and memory bandwidth with minimum overhead. Reducing interaction overhead to a few instruction times (Table 1) increases the amount of parallelism that can be economically exploited. It also simplifies programming as tasks and data structures no longer have to be grouped into large chunks to amortize large communication, synchronization, and task-switching overheads.

At MIT we have built and programmed the J-Machine to test, evaluate, and demonstrate our network and mechanisms. By running three programming systems on the machine, we have demonstrated the flexibility of its mechanisms and generated some ideas on how to improve them. The next step is to work to commercialize this technology by developing a more integrated and higher-performance processing node in today's technology and by providing bridges of compatibiliity to existing sequential software.

References

[1] Anant Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991.

[2] David Bailey. The NAS Parallel Benchmarks. Presentation given in 1991.

[3] Shekhar Borkar et al. iWARP: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–338. IEEE, November 1988.

[4] Andrew A. Chien and Jae H. Kim. Planar-Adaptive Routing: Low-cost Adaptive Networks for Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, Queensland, Australia, May 1992. IEEE.

[5] Intel Corporation. Paragon XP/S. Product Overview, 1991.

[6] David E. Culler et al. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175. ACM, April 1991.

[7] William J. Dally. Directions in Concurrent Computing. In *Proceedings of the International Conference on Computer Design*, pages 102–106. IEEE, October 1986. Conference at Port Chester, New York.

[8] William J. Dally. Wire-Efficient VLSI Multiprocessor Communication Networks. In Paul Losleban, editor, *Proceedings of Stanford Conference on Advanced Research in VLSI*, pages 391–415. MIT Press, 1987.

[9] William J. Dally. Mechanisms for Concurrent Computing. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 154–156, December 1988.

[10] William J. Dally. The J-Machine System. In Patrick Winston with Sarah A. Shellard, editor, *Artificial Intelligence at MIT: Expanding Frontiers*, chapter 21, pages 536–569. MIT Press, 1990.

[11] William J. Dally. Network and Processor Architecture for Message-Driven Computers. In Suaya and Birtwhistle, editors, *VLSI and Parallel Computation.* Morgan Kaufmann, 1990.

[12] William J. Dally. Express Cubes: Improving the Performance of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, pages 1016–1023, September 1991.

[13] William J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), March 1991.

[14] William J. Dally and Hiromichi Aoki. Adaptive Routing using Virtual Channels. *IEEE Transactions on Parallel and Distributed Computing*, 1992.

[15] William J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 189–205. IEEE, June 1987.

[16] William J. Dally et al. Design and Implementation of the Message-Driven Processor. In *Proceedings of the 1992 Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*. MIT Press, March 1992.

[17] William J. Dally and Paul Song. Design of a Self-Timed VLSI Multicomputer Communication Controller. In *Proceedings of the International Conference on Computer Design*, pages 230–234. IEEE, October 1987.

[18] William J. Dally and D. Scott Wills. Universal Mechanisms for Concurrency. In G. Goos and J. Hartmanis, editors, *Proceedings of PARLE-89*, pages 19–33. Springer-Verlag, June 1989.

[19] William J. Dally, D. Scott Wills, and Richard Lethin. Mechanisms for Parallel Computing. In *Proceedings of the NATO Advanced Study Institute on Parallel Computing on Distributed Memory Mutliprocessors*. Springer, 1991.

[20] J. A. Fisher and B. R. Rau. Instruction-Level Parallel Processing. *Science*, pages 1233–1241, September 1991.

[21] Geoffrey Fox et al. *Solving Problems on Concurrent Computers*. Prentice Hall, 1988.

[22] John L. Hennessy and Patterson David A. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, San Mateo, 1990.

[23] John L. Hennessy and Norman P. Jouppi. Computer Technology and Architecture: An Evolving Interaction. *Computer*, pages 18–29, September 1991.

[24] Waldemar Horwat, Andrew Chien, and William J. Dally. Experience with CST:Programming and Implementation. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, 1989.

[25] S. Konstantinidou and L. Snyder. Chaos router: architecture and performance. In *18th Annual Symposium on Computer Architecture*, pages 212–221, 1991.

[26] InMOS Limited. IMS T424 Reference Manual. Order Number 72 TRN 006 00, November 1984.

[27] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Mass, 1980.

[28] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. Computation Structures Group Memo 325-1, Massachusetts Institute of Technology Laboratory for Computer Science, November 15 1991.

[29] Peter R. Nuth and William J. Dally. A Mechanism for Efficient Context Switching. In *Proceedings of the International Conference on Computer Design*. IEEE, October 1991.

[30] Gregory M. Papadopoulos and David E. Culler. Monsoon: an Explicit Token-Store Architecture. In *The 17th Annual International Symposium on Computer Architecture*, pages 82–91. IEEE, 1990.

[31] S. Sakai et al. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of the 16th Annual Symposium on Computer Architecture*, pages 46–53, 1989.

[32] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[33] Charles L. Seitz et al. Submicron Systems Architecture. Semiannual Technical Report Caltech-CS-TR-90-05, Department of Computer Science, California Institute of Technology, March 15 1990.

[34] Toshio Shimada, Kei Hiraki, Kenji Nishida, and Satosi Sekiguchi. Evaluation of a Prototype Data Flow Processor of the Sigma-1 for Scientific Computations. In *13th Annual International Symposium on Computer Architecture*, pages 226–234. IEEE, June 1986.

[35] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[36] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Vol. 298 Real-Time Signal Processing IV*, pages 241–248. Denelcor, Inc., Aurora, Col, 1981.

[37] Ellen Spertus and William J. Dally. Experiments with Dataflow on a General-Purpose Parallel Computer. In *Proceedings of International Conference on Parallel Processing*, pages II231–II235, Aug 1991.

[38] Brian Totty. Experimental Analysis of Data Management for Distributed Data Structures. Master's thesis, University of Illinois, 1991.

[39] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *The 16th Annual International Symposium on Computer Architecture*, pages 273–280. IEEE Computer Society Press, 1989.

[40] D. Scott Wills. *Pi: A Parallel Architecture Interface for Multi-Model Execution*. PhD thesis, Massachusetts Institute of Technology, May 1990.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

An Automatic Translation Scheme from Prolog to the Andorra Kernel Language *

Francisco Bueno
bueno@fi.upm.es

Manuel Hermenegildo †
herme@fi.upm.es or herme@cs.utexas.edu

Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - Spain

Abstract

The Andorra family of languages (which includes the Andorra Kernel Language –AKL) is aimed, in principle, at simultaneously supporting the programming styles of Prolog and committed choice languages. On the other hand, AKL requires a somewhat detailed specification of control by the user. This could be avoided by programming in Prolog to run on AKL. However, Prolog programs cannot be executed directly on AKL. This is due to a number of factors, from more or less trivial syntactic differences to more involved issues such as the treatment of cut and making the exploitation of certain types of parallelism possible. This paper provides basic guidelines for constructing an automatic compiler of Prolog programs into AKL, which can bridge those differences. In addition to supporting Prolog, our style of translation achieves independent and-parallel execution where possible, which is relevant since this type of parallel execution preserves, through the translation, the user-perceived "complexity" of the original Prolog program.

1 Introduction

A desirable goal in logic programming language design is to support both the don't-know nondeterministic, search-oriented programming style of Prolog and the don't-care indeterministic, concurrent communicating agents programming style of committed-choice languages. Furthermore, from an implementation point of view it is interesting to be able to support the or- and independent and-parallelism often exploited in the former (e.g. [Lus88, AK90, Kal87, HG90]) as well as the dependent and-parallelism exploited in the latter (e.g. [Cra90, IMT87, HS86]). The Andorra family of languages is aimed at simultaneously supporting these two programming paradigms and their associated modes of parallel execution. The Andorra proposal in [War] (called the "basic" andorra model, on which the Andorra-I system [SCWY90] is based) defined a framework which allowed or-parallelism and also the and-parallel execution of *deterministic* goals (deterministic "stream and-parallelism"), this now being called the "Andorra Principle."

An important idea behind the choice of control in the basic Andorra model is to perform the least possible amount of computation while allowing the maximum amount of parallelism to be exploited. Another and complementary way of achieving this goal which has also been identified [HR89, HR90] is to also run in parallel *nondeterministic* goals, but provided (or while) they are independent ("independent and-parallelism" – IAP). In order to also include this type of parallelism the *Extended* Andorra Model (EAM) [War90, HJ90] defines an execution framework which allows IAP in addition to the forms of parallelism supported in the basic Andorra model. The EAM defines rules which specify a series of admissible steps of computation from each possible given state. Several rules can be admissible from a given state and this gives rise to both nondeterminism and indeterminism, and also to opportunities for parallel execution. One important issue within this framework is thus that of control: i.e. which of the admissible rules should be applied in order to achieve the most efficient execution while attaining the maximum parallelism.

Two obvious approaches to treating the above mentioned issue are to put control decisions in the hands of the programmer or to try to do this automatically by compile-time and/or run-time analysis. The Andorra Kernel Language (AKL) [HJ90, JH91], uses explicit control. In particular, AKL allows (dependent) parallel execution of determinate subgoals, as stated by the Andorra Principle, but it also allows the more general forms of parallel execution of the EAM, albeit controlled by the programmer. The specification of control is done, among other mechanisms, by positioning

*This work was funded in part by both ESPRIT project 2471 "PEPMA" and CICYT project 305.90.

†Please direct correspondence to Manuel Hermenegildo at the above address.

the goals and constraints before or after a *guard operator*, in a way that can be reminiscent of the labeling of unification as input or output (i.e. ask or tell constraints [Sar89]) in the GHC language [Ued87a]. These operators divide body clauses into two parts, the guard and the actual body. Guards are executed in independent environments and proceed unless they attempt to perform output unification, while bodies wait until guards are completely solved and goals in the body promoted. Such goals are then executed concurrently provided they are deterministic, in the spirit of the Andorra Principle. These properties give a means of control to the programmer which can be used to achieve parallel execution of general goals.

The AKL is therefore quite a powerful language. However, it does put quite a burden on the programmer in requiring certain specification of control. In particular, Prolog programs cannot always be executed directly on the AKL. This is due to a number of factors, from more or less trivial syntactic differences to more involved issues such as the treatment of cut, labeling of unification, and making the exploitation of certain types of parallelism, most notably IAP, possible without user involvement and preserving the programmer-perceived complexity of the original program.

The objective of this paper is to investigate how the above mentioned differences can be bridged, through program analysis and transformation. It points out the non-trivial problems involved in performing such a translation, and then provides solutions for these problems. Although desirable, our aim at this point is not to provide the best possible translation, which would take advantage of AKL properties to achieve a large reduction of search space, but rather to bridge the gap between Prolog and AKL in a manner that no increment in the search space is done, and also IAP can be exploited (with the important result of achieving "stability" in the frame of AKL for these cases). Building on partial translation approaches presented in [JH90, Her90] the paper presents a basic algorithm for constructing a translator from Prolog to AKL[1]. An important feature of the translation approach proposed herein is that it automatically detects and allows the parallel execution of independent goals (as well of course as or-parallelism, and the parallel execution of deterministic goals even if they are dependent as per the Andorra Principle). The execution of independent goals in parallel has the very desirable properties of preserving the program complexity perceived by the programmer [HR89]. Important requirements for such a translation are the compile-time detection of goal independence and input/output modes. This requires in general a global analysis of the program, perhaps using abstract interpretation. In the approach proposed herein heavy use will be made of our compile-time tools, developed in the context of &-Prolog [HG90]. In particular, Prolog programs are first analyzed and annotated as &-Prolog programs (thus making goal independence explicit), and then they are translated into AKL.

In the following section, the AKL control model and its rules are briefly reviewed together with some syntactic conventions. Then transformations for Prolog constructions for a basic translation are presented in section 3 and some rules for combining the AKL model with our purpose of achievement of independent parallelism are shown in section 4. Section 5 will present the analysis tools and why they are needed in the translation process. In section 6 some results are shown for the execution of a number of benchmarks automatically translated, and section 7 presents some conclusions.

[1]Ueda [Ued87b] proposed automatic translation from Prolog to a committed-choice language (GHC, in his case). However, our aim and target language are quite different.

2 The Andorra Kernel Language Revisited

In this section we present a brief overview of the AKL model of execution, in order to make the paper self-contained. The purpose is to, based on an understanding of this, extract the correct rules for a translation of Prolog which achieves the desired results. AKL and its model of execution have been fully described in [JH91, HJ90].

AKL is a language with *deep* guards. Thus, clauses are divided into two parts: the *guard* and the *body*, separated by a *guard operator*. Guard operators are: *wait* (:), *cut* (!), and *commit* (|). The following syntactical restrictions apply:

- Each clause is expected to have one and only one guard operator;
- All clauses in the definition of a predicate have to be guarded by the same guard operator. So, if any of the clauses is not guarded, the guard operator of its companions is assumed and positioned just after the clause neck.
- A wait operator is assumed, and in the above mentioned position, where no other operator can be assumed using the above mentioned rules.

Guards are regarded as part of clause selection. This means that a clause body is not entered unless head unification succeeds and its guard is completely solved. Then, execution proceeds by "expansion" of the present configuration by application of a rule of the computation model. The rules in the AKL model allow rewriting of configurations (states) leading to valid configurations from valid ones. They are fully described in [JH91], so we will simply enumerate them, providing *very informally* the concept behind the rule, rather than a precise definition:

1. *Local forking:* unfolds an atomic goal into a *choice* of all the alternatives in its definition (but without creating "copies" [2] yet of continuation goals).

2. *Nondeterminate promotion:* promotes *one* guarded goal with solved guard in a choice of several of them (i.e. copies the goal to the parent continuation, applying its constraint/substitution to it, and creates a "copy" of the continuation environment).

3. *Determinate promotion:* special case of the above when there is a *single* guarded goal in a choice if its guard is solved (no copying of the continuation environment is necessary).

4. *Failure and synchronization rules:* remove or fail configurations in the usual way.

5. *Pruning rules:* handle the effects of pruning guard operators.

6. *Distribution and bagof rules:* do the distribution of guards and the bagof operation.

These rules basically represent the allowable transitions of the EAM. The last three rules are less relevant for our purposes. In addition to these rules there are three basic control restrictions in the general computation model (meta-rules) which control the application of the above rules and which are highly relevant to our independent style translation:

- Pruning in AKL has to be *quiet*, that is, a solution for the guard of a cut or commit guarded clause may not further restrict (or constrain) variables outside its own configuration.

- Goals in the guard of a clause are completely and *locally* executed. This means that execution of guards is simultaneous but independent of the parent environment.

- Nondeterminate promotion is only admissible within a *stable subgoal* of a configuration. A goal is stable if no rule is applicable to any subgoal, and no possible changes in its environment will lead to a situation in which a rule is applicable in the goal.

As we shall soon see these three restrictions force the conditions under which translation has to be done if we want to achieve parallelism and correct pruning in the translated clauses. But first, we will illustrate the AKL execution model with a simple example:

```
partition([],_,Left,Right):- |,
        Left = [],
        Right = [].
partition([E|R],C,Left,Right):-
        E < C, |,
        Left = [E|Left1],
        partition(R,C,Left1,Right).
partition([E|R],C,Left,Right):-
        E >= C, |,
        Right = [E|Right1],
        partition(R,C,Left,Right1).
```

For a query such as `partition([2,1],3,I,D)` the initial configuration would be a choice-point with the three clauses for the predicate. Head unification would fail the first alternative (`[]=[2,1]`), but the second one would succeed (`[E|R]=[2,1],C=3,E<C`, including the guard), thus pruning the rest of the alternatives.

The single remaining alternative would then be promoted by determinate promotion, its bindings published, and execution would proceed with goals in its body. Note that this could not be done if, for example, the goal `Left=[E|Left1]` were included in the guard, as it would add constraints to the variable `I` (`I=Left`) in the external configuration, and thus pruning would have been "noisy."

On the other hand, if the clauses had no (explicit) guard operator, a wait operator would be assumed. In this case, both the second and third alternatives would succeed and only nondeterminate promotion would be applicable. If the configuration is stable, and assuming that the rightmost alternative is selected for promotion, the goal `E>=C` (i.e. `2>=3`) would be executed (and failed) *only* after promotion. After failure of this branch, determinate promotion of the remaining one would be applicable, and execution would proceed as before.

3 Translating Prolog Constructions

Having the aforementioned rules in mind, we now discuss transformation rules for translating basic Prolog constructions, disregarding any possible exploitation of IAP. Even this straightforward step is nontrivial, as we shall soon see. This is due mainly to the semantics of cut in both Prolog and AKL, cut being a guard operator in the latter. With the restrictions required for guard operators to achieve both syntactic and semantic correctness in AKL, we find problems in the following constructions:

- syntactical restrictions:
  - definitions of predicates in which a pruning clause appears,
  - clauses in which more than one cut appears;
- semantic restrictions:
  - if-then-elses, where the cut has a "local" pruning effect,
  - pruning clauses where the cut is regarded as

[2] Although we refer to "copying" throughout the paper, part of the continuation goals could in principle be shared [War90].

noisy (i.e. attempts to further restrict variables outside its scope),
- side-effects and meta-logical predicates, which should be sequentialized.

The transformations required to deal with these constructions are proposed in the following subsections. This is done mainly through examples. The aim is thus not to provide precise and formal definitions of program transformations but rather to provide the intuition behind the process of translation. In subsequent sections we will discuss other issues involved in the process of translation, such as achievement of IAP, problems in this, and its relation with the AKL stability conditions.

3.1 Direct translation

First, as all AKL clauses in a definition are forced to have the same guard operator, we have to ensure this is achieved. For example:

Example 1 Same guard operator in a definition

```
p(X,Y):- q(X), r(Y).          p(X,Y):- q(X), r(Y).
p(X,Y):- test(X), !,          p(X,Y):- pc(X,Y).
         output(Y).
p(X,Y):- s(X,Y).              pc(X,Y):- test(X), !,
                                        output(Y).
                              pc(X,Y):- s(X,Y).
```

Note that clauses before the pruning one will have an (assumed) wait operator and clauses after that one (and that one itself) will have an (assumed) cut operator. All of them but the pruning one have an empty guard. Note that, had the program not been rewritten, the rules for assuming guard operators would have put a cut operator in the first clause, which is obviously not the correct translation.

Note also, that only one guard operator is to be allowed in a clause. Therefore repeated cuts in the same body (which are otherwise strongly discouraged as a matter of style and declarativeness) have to be "folded" out using the technique sketched below:

Example 2 Single guard operator in a clause

```
p(X,Y):- test(X), !,          p(X,Y):- test(X), !,
         test(Y), !,                   foo(X,Y).
         accept(X,Y).
                              foo(X,Y):- test(Y), !,
                                         accept(X,Y).
```

Second, the AKL cut operator is regarded as a *guard* operator, and, furthermore, it has to be quiet (which is not the case in some Prolog constructions, which cannot be easily translated to AKL). One of them is local pruning, i.e. if-then-else. Indeed, an if-then-else can be viewed as a disjunction containing a cut whose scope is limited to the disjunction itself, rather than the clause in which it appears. Thus the following preprocessing can be done:

Example 3 Local pruning of if-then-else

```
p(X):- (cond(X) ->            p(X):- foo(X,Y,Z), s(Y,Z).
        q(X,Y)
      ; r(X,Z)                foo(X,Y,_):- cond(X), !,
      ), s(Y,Z).                           q(X,Y).
                              foo(X,_,Z):- r(X,Z).
```

Last but not least, we have to ensure the quietness of all AKL cuts. A cut is quiet if it does not attempt to bind variables which are seen from outside its own scope, that is, the clause where they appear. Then, if this is not the case, we have to make that binding explicit in the form of an equality constraint (a unification) and place it after the cut itself, i.e. outside the guarded part of the clause:

Example 4 Making a cut quiet

```
p(X,Y):- test(X),             p(X,Y):- test(X),
         output(Y), !.                 output(Y1), !,
p(X,Y):- s(X,Y).                      Y1=Y.
                              p(X,Y):- s(X,Y).
```

Note that knowledge of input/output modes of variables is required for performing this transformation, and that the transformation may not always be safe[3]. This will be discussed in the following subsection.

3.2 Noisiness of cut

The main difference between cut in Prolog and cut in AKL is that cut is *quiet* in AKL[4]. "Quiet" in the context of a cut means that the solution of the cut's guard is quiet, that is, it does not add constraints to variables outside the guarded goals themselves, other than those which already appear in its environment.

Indeed, a transformation such as the one proposed in example (3.1).4 can make a noisy cut quiet. What it does is to delay output unification until the guard is promoted by making it explicit in the body part of the clause. We regard a variable to be *output* in a query if execution for this query will further constrain it; a variable will be regarded as *input* if execution will depend on its state of instantiation (or constraint). In other words, a variable is an output variable in a literal if it is further instantiated by the query this literal represents, it is an input variable if it makes a difference for the execution of the literal whether the variable is instantiated or not[5]. Note that a given variable can be both input and output, or none of them.

[3]Note also that this transformation, when safe, may be of advantage as well in standard Prolog compilers in order to avoid trailing overhead.

[4]Nevertheless, a *noisy* cut has also been implemented in AKL, which we will discuss later.

[5]These definitions are similar to those independently proposed in [SCWY91], (and also in the spirit of those of Gregory [Gre85]), which describes translation techniques from Prolog to Andorra-I, an implementation of the Basic Andorra Model. Although the techniques used in such a translation have some relationship with those involved in Prolog-AKL translation, the latter requires in practice quite different techniques due to AKL being based on the

The objective of a transformation such as the one proposed is to rename apart all output variables in the head of a pruning clause, and then bind the new variables to the original ones in the body of the clause, leaving input variables untouched. In general, it is unwise to rename apart input variables since, from their own definition, this renaming would make the variable appear uninstantiated and potentially result in growth in the search space of the goals involved. This would not meet our objective of preserving the complexity of the program (and perhaps not even that of preserving its semantics). However, since a variable can be both input and output a conflict between renaming and not-renaming requirements appears in such cases. For these cases in which a variable cannot be "moved" after the cut guard operator a real noisy cut is needed. This operator exists in AKL (!!), together with a sequentialization operator, the sequential conjunction (&). It is necessary that every noisy cut be sequentialized, this to ensure that pruning would occur in the same context that it would in Prolog. Thus, every literal call to the pruning predicate has to be sequentialized to its right, and every other call to a predicate sequentialized has in turn to be also sequentialized. For this reason noisy cut is not very efficient, and thus the translation tries to minimize its use.

At this point we can summarize the action that should be taken in every case to transform the pruning clauses of a Prolog program, based on the knowledge of input/output variables, that is, whether they are "tested" or not and further instantiated or not. Here we use "noisy" to mean the transformation that defaults to the AKL noisy cut, and "move" to refer to the renaming of variables like in example (3.1).4.

Further Instantiated?	Tested?	Action
yes	yes	noisy
	no	move
	unknown	user
no	*	none
unknown	yes	user
	no	move
	unknown	user

Note that the knowledge of input/output modes in the Prolog program that is assumed in this transformation requires in general a global analysis of the program and can only be approximated, the translator having to make conservative approximations or warn the user ("user" cases above) when insufficient information is available. Note also that the "user" cases can be replaced by "noisy" cases if a non-interactive transformation is preferred. This subject will be discussed further in section 5, as well as the type of analysis required.

Extended Andorra Model (thus having to deal with the possibility of parallelism among non-deterministic goals and the stability rules) and the rather different way in which the control of the execution model (explicit in AKL and implicit in Andorra-I) is done in each language.

3.3 Synchronization of side-effects

In general, the purpose of side-effect synchronization is to prevent a side effect from being executed before other preceding (in the sense of the sequential operational semantics) side-effects or goals, in the cases when such adherence to the sequential order is desired. In our context, if side-effects are allowed within *parallel* AKL code and a behaviour of the program identical to that observable on a sequential Prolog implementation is to be preserved, then some type of synchronization code should be added to the program. In general, in order to preserve the sequential observable behaviour, side-effects can only be executed when every subgoal to their left has been executed, i.e. when they are "leftmost" in the execution tree. However, a distinction can be made between *soft* and *hard* side-effects (a side-effect is regarded to be *hard* if it could affect subsequent execution), see [DeG87] and [MH89]. This distinction allows more parallelism. It is also convenient in this context to distinguish between side-effect built-ins and side-effect procedures, i.e. those procedures that have side-effects in their clauses or call other side-effect procedures.

To achieve side-effect synchronization, various compile-time methods are possible:

- To use a chain of variables to pass a "leftmost token", taking advantage of the suspension properties of guards to suspend execution until arrival of the token [SCWY91].

- To use chains of variables as semaphores with some compact primitives that test their value. In [MH89] a solution was proposed along such lines, and its implementation discussed.

- To use a sequentialization built-in to make the side-effect and the code surrounding it wait; this primitive would be in our case the sequentialization operator "&".

In the first solution, a pair of arguments is added to the heads of relevant predicates for synchronization. Side-effects are encapsulated in clauses with a wait (:) guard containing an "ask" unification of the first argument with some known value (*token*), to be passed by the preceding side-effect upon its completion. Upon successful execution of the current side-effect the second argument is bound ("tell") to the known value and the token thus passed along. This quite elegant solution can be optimized in several cases.

The second solution can be viewed as an efficient implementation of the first one, which allows further optimization [MH89]. The logical variables which are passed to procedures in the extra arguments behave as semaphores, and synchronization primitives operate on the semaphore values.

In the third solution, every soft side-effect is synchronized to its left with the sequentialization operator "&", and every hard one both to its left and right. This sequentialization is propagated upwards to the level needed to preserve correctness. This introduces some unnecessary restrictions to the parallelism available. However, if side-effects appear close to the top of the execution tree, this may be quite a good solution.

4 Stability and Achievement of Independent And-Parallelism

In order to achieve more parallelism than that available by the translations described so far one might think of translating Prolog into AKL so that every subgoal could run in parallel unrestricted. However, this can be very inefficient and would violate the premise of preserving the results and complexity of the computation expected by the user. On the other hand, and as mentioned before, parallel execution of *independent* goals, even if they are nondeterministic, is an efficient and desirable form of parallelism and its addition motivated the development of the EAM, on which the AKL is based. Nevertheless, in AKL goals known to be independent have to be explicitly rewritten in order to make sure that they will be run in parallel. This is because of the rules that govern the (nondeterminate) promotion, that is, the stability condition on nondeterminate promotion, which will prevent these goals for being promoted if they try to bind external variables for output. Therefore, one important issue is the transformation that is needed to avoid suspension of independent goals. This is presented in section 4.1. Also, independence detection can and will be used to reduce stability checking, a potentially expensive operation.

Clearly, an important issue in this context is how stability/goal independence is detected. In the framework of the &-Prolog system we have already developed technology and the associated tools for determining independence conditions for goals and partially evaluating many of those conditions at compile-time through program analysis. Conceptual models for independent and-parallel execution have been presented and their correctness and efficiency proved [HR89]; among all we focus on the and-parallelism models proposed in [HR90, HR89]. For different but related models the reader is referred to the references in those papers. As mentioned before, in the translation process we propose to use algorithms and tools already developed in the context of &-Prolog. In this context, a series of algorithms used in the &-Prolog compiler for annotating Prolog programs have been implemented and described in [MH90]. These algorithms select goals for parallel execution and, using the sufficient rules proposed in [HR89], generate the conditions under which independence is achieved and therefore independent parallel execution ensured. The result is a transformation of a given Prolog clause into an &-Prolog clause containing parallel expressions which achieve such independent and-parallelism.

The output of this analysis is made available for the translation process in the form of an annotated &-Prolog program [HG90], i.e. the program itself expresses which goals are independent and under which conditions. These conditions are expressed in the form of if-then-elses which have the intuitive meaning of "if the conditions hold then run in parallel otherwise sequentially." The parallelism itself is made explicit by using the "&" operator to denote parallel conjunction instead of the standard sequential conjunction denoted by ","[6]. Some new issues are involved in the interaction between the conditions of these parallel expressions and other goals run in parallel concurrently, as it would be the case in AKL. These will be presented in section 4.2.

4.1 The transformation proposed

At this point the &-Prolog conditionals are regarded as input to the translator. As such, if-then-elses are preprocessed in the form mentioned in the previous sections and the remaining issue is the treatment of the parallelization operator "&". In implementing this operator we will use the AKL property that allows local and unrestricted execution of guards, i.e., goals that are encapsulated in a guard can run in parallel with goals in other guards even if they are nondeterministic. The transformation that takes advantage of this will:

- put goals known to be independent in (different) guards, and
- extract output arguments from the guards, binding them in the body part of the clauses,

the last step being required so that the execution of these goals is not suspended because of their attempting to perform output unification. With the guard encapsulation we ensure that those predicates will be executed simultaneously and independently. The following example illustrates the transformation involved:

Example 5 Encapsulation of independent subgoals

```
p(X):- (ground(X),
        indep(Y,Z) ->
          q(X,Y) & r(X,Z)
        ; q(X,Y) , r(X,Z)
       ),
       s(Y,Z).
```

```
p(X):- pp(X,Y,Z), s(Y,Z).

pp(X,Y,Z):- ground(X),
            indep(Y,Z), !,
            qp(X,Y),
            rp(X,Z).
pp(X,Y,Z):- q(X,Y), r(X,Z).

qp(X,Y):- q(X,Y1), :, Y=Y1.

rp(X,Z):- r(X,Z1), :, Z=Z1.
```

[6]Note that in AKL these operators have just the opposite meaning!.

When the condition is met, both subgoals will be tried by the local fork rule, then *both guards* will be completely and locally solved, and then, as goals are independent on X (X is ground) and no output is produced in the guard, the nondeterminate promotion rule is always applicable and all solutions will be tried in the standard cartesian product way. Thus, parallel execution is ensured for those goals that are identified as independent.

On the other hand, when the condition fails (the goals being dependent) they appear together in a body with an empty guard. This means that the guard will be immediately solved, the clause body promoted, and subgoals tried simultaneously. Then the standard stability and promotion rules will apply.

It should be noted that, as in the case of cut, and in addition to detecting goal independence, to be able to perform this transformation it is necessary to have inferred mode information regarding the predicate clauses. In section 5 techniques used in order to infer this information will be reviewed.

4.2 Cohabitation of dependent and independent and-parallelism and stability checks

When evaluating the conditions of parallel expressions at run-time within a parallel framework such as that of the AKL, they may not evaluate to the same value than during a Prolog execution. This is what we have termed in another context the *CGE-condition problem* [GSCYH91][7], and may result in a loss (or increase) of parallelism. To deal with these issues, different levels of restrictions can be placed on the translation:

- Disallow any parallel execution except for those goals found to be independent.
- Allow parallel execution only for goals not binding variables that appear in the conditions or CGE.
- Allow parallel execution outside a CGE but sequentialize before and after the conditional parallel expressions.
- Allow unrestricted parallel execution unrestricted, i.e. no sequentialization is to be done.

The first solution can be implemented by translating every conjunction as a *sequential* AKL conjunction, except those joining independent goals. This will lead to a type of execution where only goals known to be independent are run in parallel and which directly resembles that of &-Prolog [HG90]. The same search space as &-Prolog will be explored. Nondeterminate (and determinate) promotion will then be restricted to only independent and sequential goals. Thus, one very important advantage of this translation is that *no checks on stability ever need to be done*, as stability is ensured for sequential and independent execution. This is an important issue since stability checking is a potentially expensive operation (and very closely related to independence checking). Thus, in an *ideal* AKL implementation code translated as above, i.e. free of stability checks, should run with comparable efficiency to that of &-Prolog. On the other hand, the transformation loses determinate dependent and-parallelism and its desirable effect of co-routining, which could be useful in reducing search space [SCWY90].

The second solution attempts to preserve the environment in which the CGE evaluates while allowing coroutining of goals that don't affect CGE conditions and goals. Although interesting, this appears quite difficult to implement in practice as it requires very sophisticated compile-time analysis and will probably incur in run-time overheads for checking of the conditions placed in the program.

The third solution can be viewed as a relaxation of the first one to achieve some coroutining, or as an efficient (and feasible) way of partially implementing the second one. Goals before and after are allowed to execute in parallel using the Andorra Principle, but they are sequentialized just before and after a CGE. In this way CGEs evaluate in the same context as in Prolog and the same level of independent and-parallelism is achieved. This translation has the good characteristics regarding search space of the previous one. In addition, some reduction of search space due to coroutining will be achieved. However, stability checking, although reduced, cannot in general be eliminated altogether.

The fourth solution will allow every goal to run in parallel. The full EAM and AKL operational semantics (including stability) has to be preserved. The execution of goals which are unconditionally independent or depend only on groundness checks (conditionals in the parallel expressions are composed of ground/1 and indep/2 checks, as in the example of section 4.1) will be the same as in &-Prolog as eager execution of other goals cannot affect ground or empty checks [GSCYH91]. However, independence checks may fail where they wouldn't in Prolog (therefore losing this parallelism), but also succeed where they would fail in Prolog (therefore gaining this parallelism). Also, the number of parallel steps will always be the same or less as in Prolog (although different than in &-Prolog). This solution (as well as the first and second ones) appear as quite reasonable compromises and offer different trade-

[7] Note that some other problems mentioned in [GSCYH91] regarding the interaction between independent and dependent and-parallelism (in particular, the *deterministic goal problem*) are less of an issue in the proposed translation to AKL because independent goals execute in their own environments, thanks to the dynamic scoping of AKL guards. In any case, the AKL implementation is assumed to cope with all types of goal activations possible within the EAM.

offs. The current translation approach uses this fourth option, but the others should also be explored.

5 Inferring modes - Abstract Interpretation

We have mentioned in previous sections the need for inferring modes of clause variables (i.e. whether they are input or output variables) in Prolog programs. The main reason for this need is that we have to know which are the output variables in a clause in order to rename them apart and place corresponding bindings for them in the body part of the clause in both

- the pruning clauses (as shown in section 3.2), and
- the remade clauses for parallel execution (as shown in section 4.1 in example 5).

Much work has been done in global analysis of logic programs to infer run-time properties, and, in particular, modes, mostly using the technique of abstract interpretation [CC77]. A more sophisticated sort of variable binding analysis (comprising *groundness*, *aliasing*, and *freeness* information) is instrumental in the process of inferring the independence conditions for literals in a body. While not strictly needed, such an analysis is extremely useful as it allows the reduction of the number of conditions and therefore the improvement of performance by reducing run-time checking [WHD88, MH91b] (these papers provide references to the important body of other work in this area). The standard global analyzer in the &-Prolog compiler, described in [MH91b], infers groundness and variable sharing/aliasing. Since variable freeness is also needed for the AKL translator, this analyzer has been extended to use the algorithm described in [MH91a] and infer variable freeness information.

It turns out that freeness information is very useful for many reasons [MH91a]. In the translation process it is essential for determining input/output arguments. This we can show by simply expressing the information required for the table in section 3.2 in terms of information directly available from abstract interpretation. In order to do this, recall, as defined in section 3.2, that a program variable (or an argument) is output in a literal if the call to the corresponding predicate further instantiates this variable, and it is input in a literal if its state of instantiation is going to be checked in the execution of the call for that literal. With these definitions in mind the following table shows how the input or output character of variables can be decided in a good number of cases based on the information directly available from global analysis:

Before	After	Output?	Input?
ground	(ground)	no	*
free	free	no	*
	semi	yes	no
	ground	yes	no
$semi_1$	$semi_1$	no	*
	$semi_2$	yes	?
	ground	yes	?

From the table we identify cases in which it is clear that the variable is known not to be an input variable, without any further analysis (i.e. when the variable is free). Furthermore, we realize that if a variable is known not to be an output variable then it doesn't need to be renamed apart and it is not necessary to determine whether it is an input variable or not ("*" cases). Reducing the cases where knowing if a variable is input is quite useful since inferring whether a variable binding is needed or not requires additional analysis ("?" cases). This analysis seeks to decide if a variable is crucial in clause selection or checking. Note that the analysis has to be extended for every child procedure of the one being analyzed.

Finally, we would like to also mention that combining mode/type analysis (such as the one used in [SCWY91] or [Jan90]) with the accurate tracking of sharing and freeness information of [MH91a] could be very helpful in this process (improving the ability to more accurately resolve different degrees of partial instantiation such as the $semi_1/semi_2$ cases in the table above) and is part of our plans for future work.

6 Performance Timings

This section presents some results on the timing of a number of benchmarks in a prototype AKL system. The AKL versions of the programs obtained through automatic compile-time translation are compared with versions specifically written for AKL. Timings for the original Prolog versions are also included for comparison and also with the intention of identifying translation paradigms that help efficiency. With this aim in mind, the set of benchmarks has been chosen so that performance results are obtained for several different programming paradigms, and a number of different translation issues are taken into account. The results show that translation suffices in most cases, provided state-of-art analysis technology is used.

Timings[8] have been done for the Prolog program (compiled and interpreted), the AKL program obtained from automatic translation and the "hand-written-AKL" version. Execution until the first solution is obtained has been measured. Timings are an average of ten consecutive executions done after a first one (not timed) and are given in in milisseconds, rounded up to tens.

[8] SICStus 1.8 and a sequential AKL 0.0 prototype system, made available by SICS, have been used.

We briefly introduce the programming paradigms represented by each of the benchmarks used. *qsort* has been translated in two ways, one that "folds" pruning definitions, and another one that is able to "extend" the cut to all clauses; the latter showing an advantage w.r.t. the former. *sort* illustrates the advantage of being able to detect that some cuts are not noisy (as opposed to defaulting to noisy cut in every case). In fact, in this case the translated version is slightly faster than the hand-coded one.

For *money* we have used three different versions. In the first version of the program the problem is solved through extensive backtracking. In the second one the ordering of goals is improved at the Prolog level. In the third version the Prolog builtins are translated into AKL specific ones. As in *zebra* the difference with the "hand-written" version is in the use of the arithmetic predicates: addition is programmed in the hand-coded AKL version as illustrated by the *sum/3* predicate,

```
sum(X,Y,Z):- plus(X,Y,Z0), |, Z = Z0.
sum(X,Y,Z):- minus(Z,Y,X0), |, X = X0.
sum(X,Y,Z):- minus(Z,X,Y0), |, Y = Y0.
```

in which the coroutining effect provides a "constraint solving" behaviour.

Scanner is a program where AKL can take a large advantage from concurrent execution and the "determinate-first" principle, even without explicit control, and this is shown in the good performance of the translated program. On the other hand, in *triangle* and *knights* heavy use of special AKL features has been made, through hand-optimization.

	Prolog compiled	Prolog interpret.	AKL translated	AKL "hand"
qsort1	30	290	750	290
qsort	30	290	290	290
sort	20	50	870	910
money1	66,590	520,190	294,370	530
money	47,790	391,190	294,070	530
moneyb	47,790	391,190	187,920	530
zebra	8,550	43,740	10,380	1,980
scanner	1,407,450	8,838,000	540	120
triangle	3,140	7,260	152,230	11,020
knights	79,960	855,049	1,165,020	480

	Prolog compiled	Prolog interpret,	AKL translat. (encap.)	AKL translat. (direct)
qsort	30	290	290	290
matrix	50	400	610	690
hanoi	10	50	70	310
query	70	340	370	1,600
maps	90	540	140	2,240

In *matrix*, *hanoi*, *query*, and *maps* (and also *qsort*), encapsulation of different programming paradigms has been tried. The results show that encapsulating independent goals which are deterministic provides no improvement, but performance improves when they are nondeterministic. Performance also improves in the case of goals which act in producer/consumer fashion (*maps*). These results suggest that AKL control similar to that of hand-coded versions can be imposed automatically for paradigms other than independence of goals.

The automatic transformation achieves reasonably good results when compared to code specifically written for AKL, provided one takes into account that the starting point is a Prolog program with little specification of control, and it is being compared to an AKL program where control has been greatly optimized by the programmer. The examples where the largest differences show are those in which the control imposed by hand in the AKL program changes the complexity of the algorithm, generally through smart use of suspension (as in the *sum/3* predicate), something that the transformation can not yet do automatically. However, the results also show that it would obviously be desirable to extend the translation algorithms towards implementing some of the smart forms of control that can be provided by an AKL programmer.

When comparing with Prolog, both the interpreted and compiled Prolog figures should be considered, as the AKL system prototype used is somehow something in between a compiler and an interpreter. The results show that a variable performance improvement can be obtained whenever determinism is significant in the problem (this is quite spectacular in *scanner*). Also, the encapsulation transformation can help efficiency in some cases. In any case the figures are of course preliminary and a more exhaustive study should clearly be done after improvements in the translation prototype and the AKL system, and also when an actual parallel AKL system is available.

7 Conclusions

We have presented an algorithm for translating Prolog into AKL which in addition achieves independent and-parallel execution of appropriate goals. We have pointed out a series of non-trivial problems associated with such a translation and proposed solutions for them based on existing global analysis technology. We have shown how to take advantage both of the AKL execution model (the Extended Andorra Model) and the independence analysis performed in the context of &-Prolog to produce a translation that allows the exploitation of all the forms of parallelism present in AKL (dependent-and, independent-and, and or-parallelism) while offering the user the familiar Prolog (or, in general, logic with minimal control) view (and debugging ease!). Most importantly, this is done while preserving or improving the user-perceived complexity of the program. The transformation is relevant even in the case of a sequential AKL implementation since the reduction of stability checking which follows from knowledge

of goal independence can already be of significant advantage, given the expected cost of stability tests. In the case of a parallel AKL implementation the transformation amounts to a form of automatic parallelization and search space reducing implementation for Prolog programs which exploits the EAM, and imposes a particular form of control on it.

A *sequential* AKL implementation is already being developed at SICS with a first prototype already running. The translator itself is also being implemented and a preliminary version is already integrated with the &-Prolog system compilation tools. The combination has been tested and some sample programs executed successfully on AKL, and compared with their specific AKL counterparts. Further work is expected in the translator as better translation algorithms are developed to take more specific advantage of the AKL control facilities, in particular coroutining, in more accurately detecting input and output variables, in adapting the algorithms to possible evolutions of the AKL, in evaluating the performance of the translated programs with respect to Prolog, and in the formal proof of the correctness of the transformation and its preservation of user expected computation size, the latter point being supported already in part by the basic results on independent and-parallelism.

Acknowledgements

The authors would like to thank Seif Haridi, Sverker Jansson, Johan Montelius, and Mats Carlsson of SICS, and David H.D. Warren, Vitor Santos Costa, and Gopal Gupta of U. of Bristol for many useful discussions. Also thanks to SICS for making the prototype AKL implementation available for experimentation. This work has been performed in the context of the ESPRIT "PEPMA" project and has greatly benefited from discussions with other members of the partner institutions, most significantly from SICS, U. of Bristol, and U.P. Madrid.

References

[AK90] K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*. MIT Press, October 1990.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin. of Programming Languages*, pages 238–252, 1977.

[Cra90] Jim Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *1990 North American Conference on Logic Programming*. MIT Press, 1990.

[DeG87] D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

[Gre85] S. Gregory. *Design, Application and Implementation of a Parallel Logic Programming Language*. PhD thesis, Imperial College of Science and Technology, London, England, 1985.

[GSCYH91] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: A Model Intergrating Dependent-, Independent-, and Or-parallelism. Technical report, University of Bristol, March 1991.

[Her90] M. Hermenegildo. Compile-time Analysis Requirements for the Extended Andorra Model. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[HJ90] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*. MIT Press, June 1990.

[HR89] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[HR90] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[HS86] A. Houri and E. Shapiro. A sequential abstract machine for flat concurrent prolog. Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel, July 1986.

[IMT87] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Fourth International Conference on Logic Programming*, pages 257–275. University of Melbourne, MIT Press, May 1987.

[Jan90] G. Janssens. *Deriving Run-time Properties of Logic Programs by means of Abstract Interpretation.* PhD thesis, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, March 1990.

[JH90] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. Technical Report PEPMA Project, SICS, Box 1263, S-164 28 KISTA, Sweden, November 1990. Forthcoming.

[JH91] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[Kal87] L. Kale. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming*, pages 616–632. Melbourne, Australia, May 1987.

[Lus88] E. Lusk et. al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems.* Tokyo, November 1988.

[MH89] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *1989 International Conference on Logic Programming.* MIT Press, June 1989.

[MH90] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[MH91a] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming.* MIT Press, June 1991.

[MH91b] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 1991. To appear (also published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, Aug 1990).

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon, Pittsburgh, 1989. School of Computer Science.

[SCWY90] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, April 1990.

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[Ued87a] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[Ued87b] K. Ueda. Making Exhaustive Search Programs Deterministic. *New Generation Computing*, 5(1):29–44, 1987.

[War] D. H. D. Warren. The Andorra Principle. Presented at Gigalips workshop, 1987. Unpublished.

[War90] D. H. D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

[WHD88] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming.* MIT Press, August 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Recomputation based Implementations of And-Or Parallel Prolog

Gopal Gupta†
Department of Computer Science
Box 30001, Dept. 3CU,
New Mexico State University
Las Cruces, NM 88003-0001
gupta@nmsu.edu

Manuel V. Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid
28660-Boadilla del Monte, Madrid, SPAIN
herme@fi.upm.es

Abstract

We argue that in order to exploit both Independent And- and Or-parallelism in Prolog programs there is advantage in recomputing some of the independent goals, as opposed to all their solutions being reused. We present an abstract model, called the Composition-Tree, for representing and-or parallelism in Prolog Programs. The Composition-tree closely mirrors sequential Prolog execution by recomputing some independent goals rather than fully re-using them. We also outline two environment representation techniques for And-Or parallel execution of *full* Prolog based on the Composition-tree model abstraction. We argue that these techniques have advantages over earlier proposals for exploiting and-or parallelism in Prolog.

1. Introduction

One of the features of logic programming languages that make them attractive is that they allow implicit parallel execution of programs. There are three main forms of parallelism present in logic programs: or-parallelism, Independent And-parallelism and Dependent and-parallelism. In this paper we restrict ourselves to Or-parallelism and Independent and-parallelism. There have been numerous proposals for exploiting or-parallelism in logic programs [AK90, HC87, LW90, W84, W87, etc.]‡ and quite a few for exploiting independent and-parallelism [H86, LK88, etc.]. Models have also been proposed to exploit both or-parallelism and independent and-parallelism in a single framework [BK88, GJ89, RK89]. It is the latter aspect of combining independent and- and or-parallelism that this paper addresses.

One aspect which most models that have been proposed (and some implemented) so far for combining or-parallelism and independent and-parallelism have in common is that they have either considered only pure logic programs (pure Prolog), e.g. [RK89, GJ89], or, alternatively, modified the language to separate parts of the program that contain extra-logical predicates (such as cuts and side-effects) from those that contain purely logical predicates, then allowing parallel execution only in parts containing purely logical predicates [RS87, BK88]. In the former case practical Prolog programs cannot be executed since most such programs use extra-logical features. The latter approach has a number of disadvantages: first, it requires programmers to divide the program into sequential and parallel parts themselves. As a result of this, parallelism is not exploited completely implicitly since some programmer intervention is required. This also rules out the possibility of taking "dusty decks" of existing Prolog programs and running them in parallel. In addition, some parallelism may also be lost since parts of the program that contain side-effects may also actually be the parts that contain parallelism. It has been shown that or-parallelism and independent and-parallelism can be exploited in *full* Prolog completely implicitly (for example, in the Aurora and Muse Systems [HC88, LWH90, AK91], and in the &-Prolog system [HG90, MH89, CC89]). We argue that the same can be done for systems that combine independent and-and or-parallelism and that will be one of the design objectives of the approach presented in this paper.†

The paper thus describes a general approach for

† Much of this work was done while the first author was a Research Associate in David H.D. Warren's group at the University of Bristol.

‡ See [GJ90] for a systematic analysis of the various models.

† Due to length limitations the actual techniques for incorporating side effects in and-or parallel systems in order to execute full Prolog are presented in a separate report [GS91]. However, the model presented in this paper has been designed with this issue in mind, i.e., having as one of the objectives that the inclusion of side effects be facilitated.

combined exploitation of independent and- and or-parallelism in full Prolog. We present an abstract model of and-or parallelism for logic programs which mirrors sequential Prolog execution more closely, essentially by recomputing some independent goals (those that Prolog recomputes) rather than re-using them, and show the advantages of this approach. Our presentation is then two-pronged, in that we propose two alternative efficient environment representation techniques to support the model: paged binding arrays and stack copying. Using the concept of teams of processors‡, we also briefly discuss issues such as scheduling and memory management.

The environment representation techniques proposed are extensions of techniques designed for purely or-parallel systems—specifically the Aurora [LW90] and Muse [AK90] systems. The method for encoding independent and-parallelism is taken from purely independent and-parallel systems—specifically the &-Prolog system [HG90]: we use the parallel conjunction operator "&" to signify parallel execution of the goals separated by this operator and Conditional Graph Expressions (CGEs) [HN86,H86]§. Hence our model can be viewed as a combination of the &-Prolog system and a purely or-parallel system such as Aurora or Muse—in the presence of only independent and-parallelism our model behaves *exactly* like &-Prolog while in the presence of only or-parallelism it behaves *exactly* like the Aurora or Muse systems, depending on the environment representation technique chosen.

The rest of the paper is organised as follows: Section 2 describes or-parallelism and independent and-parallelism in Prolog programs. Section 3 presents arguments for favouring recomputation of some independent and-parallel goals over their complete reuse. Section 4 then presents an abstract model called the Composition-tree for representing and-or parallel execution of Prolog with recomputation. Section 5 deals with environment representation issues in the Composition-tree: section 5.1 presents a comparison of environment representation techniques based on whether there is *sharing* or *non-sharing*; section 5.2 presents an extension of the Binding Arrays method, an environment representation technique based on sharing; while section 5.3 presents another technique, based on non-sharing, which employs *stack-copying*. Finally, section 6 presents our conclusions. We assume that the reader is familiar to some extent with Binding Arrays [W84, W87], the Aurora and Muse Systems [LWH90, AK90], and the &-Prolog system [HG90], as well as with some aspects of sequential Prolog implementation.

2. Or- and Independent And-parallelism

Or-parallelism arises when more than one rule defines some relation and a procedure call unifies with more than one rule head in that relation—the corresponding bodies can then be executed in or-parallel fashion. Or-parallelism is thus a way of efficiently searching for solutions to a goal, by exploring alternative solutions in parallel. It corresponds to the parallel exploration of the branches of the proof tree. Or-parallelism has successfully been exploited in full Prolog in the Aurora [LWH90] and the Muse [AK90] systems both of which have shown very good speed up results over a range of problems.

Informally, *Independent And-parallelism* arises when more than one goal is present in the query or in the body of a procedure, and the run-time bindings for the variables in these goals are such that two or more goals are *independent* of one another. In general, independent and-parallelism includes the parallel execution of any set of goals in a resolvent, provided they meet some independence condition. Independent and-parallelism is thus a way of speeding up a problem by executing its subproblems in parallel. One way for goals to be independent is that they don't share any variable at run-time (*strict* independence [HR90]†). This can be ensured by checking that their resulting argument terms after applying the bindings of the variables are either variable-free (i.e., *ground*) or have non-intersecting sets of variables. Independent and-parallelism has been successfully exploited in the &-Prolog system [HG90]. Independent and-parallelism is expressed in the &-Prolog system through the parallel conjunction operator "&", which will also be used in this paper. For syntactic brevity we will also use &-Prolog's Conditional Graph Expressions (CGEs), which are of the form

$$(condition \Rightarrow goal_1 \ \& \ goal_2 \ \& \ \ldots \ \& \ goal_n \ )$$

meaning, using the standard Prolog if-then-else construct,

$$(condition \rightarrow goal_1 \ \& \ldots \& \ goal_n \ ; \ goal_1, \ldots, \ goal_n)$$

‡ We refer to the working "agents" of the system –the "workers" of Aurora and Muse and "agents" of &-Prolog– simply as processors, under the assumption that the term will generally represent processes mapped onto actual processors in an actual implementation.

§ Note that CGEs and & operators can be introduced automatically in the program at compile time [MH89a] using abstract interpretation and thus the programmer is not burdened with the parallelization task.

† There is a more general concept of independence, *non-strict* independence [HR90], for which the same results (the model presented in this paper included) apply. However, the rest of the presentation in this section will refer for simplicity, and without loss of generality, to strict independence.

i.e., that, if *condition* is true, goals $goal_1 \ldots goal_n$ are to be evaluated in parallel, otherwise they are to be evaluated sequentially. The *condition* can obviously be any prolog goal but is normally a conjunction of special builtins which include *ground*/1, which checks whether its argument has become a ground term at run-time, or *independent*/2, which checks whether its two arguments are such at run-time that they don't have any variable in common, or the constant *true* meaning that $goal_1 \ldots goal_n$ can be evaluated in parallel unconditionally. It is possible to generate parallel conjunctions and or CGEs automatically and quite successfully at compile-time using abstract interpretation [MH89]. Thus, exploitation of independent and-parallelism in &-Prolog is completely implicit (although user annotation is also allowed).

There have been a number of attempts to exploit or- and independent and-parallelism together in a single framework [GJ89, RK89, WR87, etc.], however, and as mentioned earlier, they either don't support the full Prolog language, or require user intervention. Also, in general these systems advocate *solution sharing* which, as will be argued in the following section, stands in the way of supporting full Prolog.

3. Recomputation *vs* Reuse

In the presence of both and- and or-parallelism in logic programs, it is possible to avoid recomputing certain goals. This has been termed as solution sharing [GJ89, G91a]. For example, consider two independent goals `a(X)`, `b(Y)`, each of which has multiple solutions. Assuming that all solutions to the program are desired, the most efficient way to execute this goal would be to execute `a` and `b` in their entirety and combine their solutions (possibly incrementally) through a join [BK88, GJ89, RK89]. However, to solve the above goal in this way one needs to be sure that the set of solutions for `a` and `b` are *static* (i.e., if either goal is executed multiple times, then each invocation produces an identical set of solutions). Unfortunately, this can hold true only if clauses for `a` and `b` are pure logic programs. If side-effects are present (as is usually the case with Prolog programs), then the set of solutions for these goals may not be static. For example, consider the case where, within `b`, the value of a variable is read from the standard input and then some action taken which depends on the value read. The solutions for `b` may be different for every invocation of `b` (where each invocation corresponds to a different solution of `a`), even if the goal is completely independent of the others. Hence solution sharing would yield wrong results in such a case. The simple solution of sequentializing such and-parallel computations results in loss of too much and-parallelism, because if `a(X)`, `b(Y)` falls in the scope of some other goal, which is being executed in and-parallel, then that goal has to be sequentialized too, and we have to carry on this sequentialization process right up to the top level query. If, however, the goals are recomputed then this sequentialization can be avoided, and parallelism exploited even in the presence of cuts and side-effects [GS91].

Hence, there is a strong argument for recomputing non-deterministic and-parallel goals, especially, if they are not pure, and even more so if we want to support Prolog as the user language†. Additionally, recent simulations of and-or parallelism [SH91] show that typical Prolog programs perform very little recomputation, thus providing further evidence that the amount of computation saved by a system which avoids recomputation may be quite small in practice. Presumably this behaviour is due to the fact that Prolog programmers, aware of the selection and computation rules of Prolog, order literals in ways which result in efficient search which minimises the recomputation of goals. Note that the use of full or partial recomputation can never produce any slowdown with respect to Prolog since Prolog itself uses full recomputation.

Recomputation of independent goals was first proposed in the context of &-Prolog.‡. It is obviously also used in Aurora and Muse (since, performing no goal independence analysis, no possibility of sharing arises) and has made these three systems quite capable of supporting full Prolog. Recomputation in the context of and-or parallelism has also been proposed in [SH91]§. The argument there was basically one of ease of simulation and, it was argued, of implementation (being a simulation study no precise implementation approach was given). Here we add the important argument of being able to support full Prolog, provide an abstract representation of the corresponding execution tree, and outline two efficient implementation approaches.

4. And-Or Composition Tree

The most common way to express and- and or-

† There is a third possibility as well: to recompute those independent and-parallel goals that have side-effects and share those that don't. Since the techniques for implementing solution sharing are in the literature and techniques for implementing solution recomputation are presented herein such an approach would represent a –perhaps non-trivial– combination of the given methods.

‡ In the case of &-Prolog there are even further arguments in favour of recomputation, related to management of a single binding environment and memory economy.

§ The idea of recomputation is referred to as "or-under-and" in [SH91].

parallelism in logic programs is through the traditional concept of and-or trees, i.e. trees consisting of or-nodes and and-nodes. Or-nodes represent multiple clause heads matching a goal while and-nodes represent multiple subgoals in the body of a clause being executed in and-parallel. Since in the model presented herein we are representing and-parallelism via parallel conjunctions, our and-nodes will represent such conjunctions. Thus, given a clause `q :- (true => a & b)`, and assuming that `a` and `b` have 3 solutions each (to be executed in or-parallel form) and the query is `?- q`, then the corresponding and-or tree would appear as shown in figure 1.

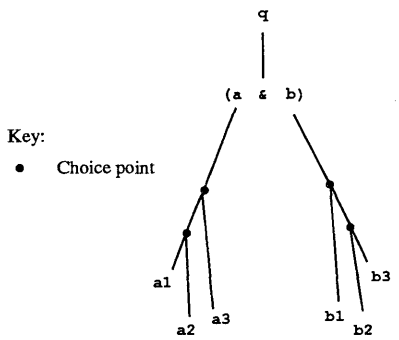


Figure 1: And-Or Tree

One problem with such a traditional and-or tree is that bindings made by different alternatives of `a` are not visible to different alternatives of `b`, and vice-versa, and hence the correct environment has to be created before the continuation goal of the parallel conjunction can be executed. Creation of the proper environments requires a global operation, for example, *Binding Array loading* in AO-WAM [GJ89, G91a], the complex dereferencing scheme of PEPSys [BK88], or the "global forking" operation of the Extended Andorra Model [W90]. To eliminate this possible source of overhead in our model, we extend the traditional and-or tree so that the various or-parallel environments that simultaneously exist are always separate.

The extension essentially uses the idea of recomputing independent goals of a parallel conjunction of &-Prolog [HG90] (and Prolog!). Thus, for every alternative of `a`, the goal `b` is computed in its entirety. Each separate combination of `a` and `b` is represented by what we term as a *composition* node (c-node for brevity). Thus, each composition node in the tree corresponds to a different solution for the parallel conjunction, i.e., a different "continuation". Thus the extended tree, called the *Composition-tree* (C-tree for brevity), for the above query might appear as shown in figure 2—for each alternative of the and-parallel goal `a`, goal `b` is entirely recomputed (in fact, the tree could contain up to 9 c-nodes, one for each combination of solutions of `a` and `b`). To represent the fact that a parallel conjunction can have multiple solutions we add a branch point (choice point) before the different composition nodes. Note that c-nodes and branch points serve purposes very similar to the Parcall frames and markers of the RAP-WAM [H86, HG90]. The C-tree can represent or- and independent and-parallelism quite naturally—execution of goals in a c-node gives rise to independent and-parallelism while parallel execution of untried alternatives gives rise to or-parallelism.†.

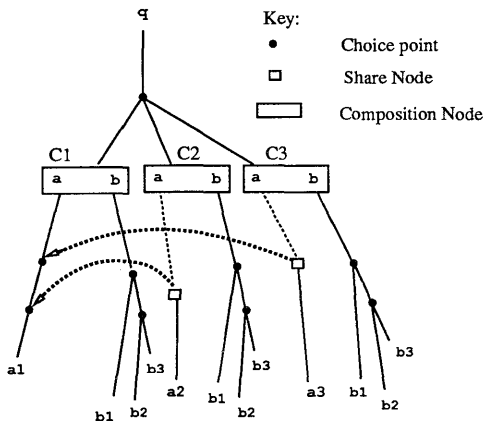


Figure 2: Composition Tree

Notice the topological similarity of the C-tree with the purely or-parallel tree shown in figure 3 for the program above. Essentially, branches that are "shared" in the purely or-parallel tree (i.e. that are "common", even though different binding environments may still have to be maintained –we will refer to such branches and regions for simplicity simply as "shared") are also shared in the C-tree. This sharing is represented by means of a *share-node*, which has a pointer to the shared branch and a pointer to the composition node where that branch is needed (figure 2). Due to sharing the subtrees of some independent and-parallel goals maybe spread out across different composition nodes. Thus, the subtree of goal `a` is spread out over c-nodes C1, C2 and C3 in the C-tree of figure 2, the total amount of program-related work being essentially maintained.

† In fact, a graphical tool capable of representing this tree has shown itself to be quite useful for implementors and users of independent and- and or-parallel systems [CG91].

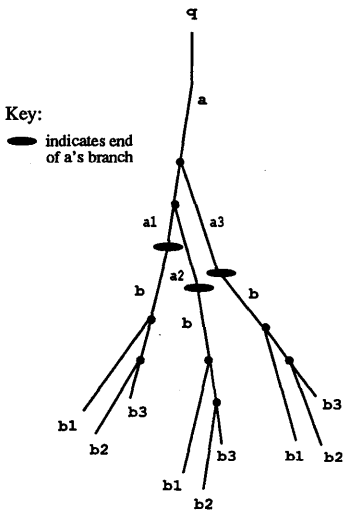


Figure 3: Or-Parallel Tree

4.1 And-Or Parallelism & Teams of Processors

We will present some of the implementation isuues from the point of view of extending an or-parallel system to support independent and-parallelism. When a purely or-parallel model is extended to exploit independent and-parallelism then the following problem arises: at the end of independent and-parallel computation, all participating processors should see all the bindings created by each other. However, this is completely opposite to what is needed for or-parallelism where processors working in or-parallel should not see the (conditional) bindings created by each other. Thus, the requirements of or-parallelism and independent and-parallelism seem anti-thetical to each other. The solutions that have been proposed range from updating the environment at the time independent and-parallel computations are combined [RK89, GJ89] to having a complex dereferencing scheme [BK88]. All of these operations have their cost.

We contend that this cost can be eliminated by organising the processors into teams. Independent and-parallelism is exploited among processors within a team while or-parallelism is exploited among teams. Thus a processor within a team would behave like a processor in a purely and-parallel system while all the processors in a given team would collectively behave like a processor in a purely or-parallel system. This entails that all processors within each team share the data structures that are used to maintain the separate or-parallel environments. For example, if binding arrays are being used to represent multiple or-parallel environments, then only one binding array should exist per team, so that the whole environment is visible to each member processor of the team. In copying is used, then processors in the team share the copy. Note that in the limit case there will be only one processor per team. Also note that despite the team arrangement a processor is free to migrate to another team as long as it is not the only one left in the team. Although a fixed assignment of processors to teams is possible a flexible scheme appears preferable. This will be discussed in more detail in section 4.3. The concept of teams of processors has been successfully used in the Andorra-I system [SW91], which extends an or-parallel system to accommodate dependent and-parallelism.

4.2. C-tree & And-Or Parallelism

The concept of organising processors into teams also meshes very well with C-trees. A team can work on a c-node in the C-tree—each of its member processors working on one of the independent and-parallel goal in that c-node. We illustrate this by means of an example. Consider the query corresponding to the and-or tree of figure 1. Suppose we have 6 processors P1, P2, ..., P6, grouped into 3 teams of 2 processors each. Let us suppose P1 and P2 are in team 1, P3 and P4 in team 2, and P5 and P6 in team 3. We illustrate how the C-tree shown in figure 2 would be created.

Execution commences by processor P1 of team 1 picking up the query q and executing it. Execution continues like normal sequential execution until the parallel conjunction is encountered, at which point a choice point node is created to keep track of the information about the different solutions that the parallel conjunction might generate. A c-node is then created (node C1 in figure 2). The parallel conjunction consists of two and-parallel goals a and b, of which a is picked up by processor P1, while b is made available for and-parallel execution. The goal b is subsequently picked up by processor P2, teammate of processor P1. Processor P1 and P2 execute the parallel conjunction in and-parallel producing solutions a1 and b1 respectively. In the process they leave choice points behind. Since we allow or-parallelism below and-parallel goals, these untried alternatives can be processed in or-parallel by other teams. Thus the second team, consisting of P3 and P4 picks up the untried alternative corresponding to a2, and the third team, consisting of P5 and P6, picks up the untried alternative corresponding to a3. Both these teams create a new c-node, and restart the execution of and-parallel goal b (the goal to the right of goal a): the first processor in each team (P3 and P5, respectively) executes the alternative for a, while the second processor in each team (P4 and P6, respectively) executes the restarted goal b. Thus, there are

3 copies of b executing, one for each alternative of a. Note that the nodes in the subtree of a, between c-node C1 and the choice points from where untried alternatives were picked, are "shared" among different teams (in the same sense as the nodes above the parallel conjunction are—different binding environments still have to be maintained).

Since there are only three teams, the untried alternatives of b have to be executed by backtracking. In the C-tree, backtracking always takes place from the right to mimic Prolog's behaviour—goals to the right are completely explored before a processor can backtrack inside a goal to the left. Thus, if we had only one team with 2 processors, then only one composition node would actually need to be created, and all solutions would be found via backtracking, exactly as in &-Prolog, where only one copy of the Parcall frame exists [H86, HG90]. On the other hand if we had 5 teams of 2 processors each, then the C-tree could appear as shown in fig 4. In figure 4, the 2 extra teams steal the untried alternatives of goal b in c-node C3, This results in 2 new c-nodes being created, C4 and C5 and the subtree of goal b in c-node C3 being spread across c-nodes C3, C4 and C5. The topologically equivalent purely or-parallel tree of this C-tree is still the one shown in figure 3. The most important point to note is that new c-nodes get created only if there are resources to execute that c-node in parallel. Thus, the number of c-nodes in a C-tree can vary depending on the availability of processors.

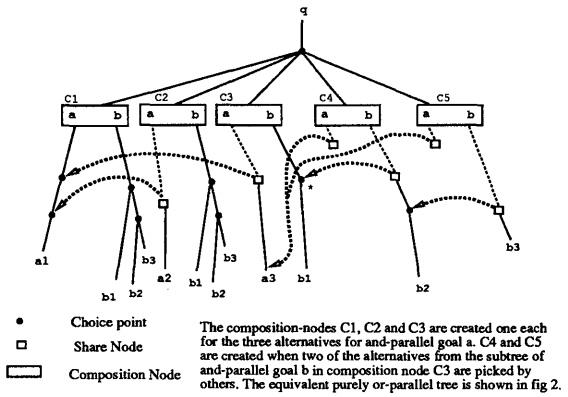


Figure 4: C-tree for 5 Teams

It might appear that intelligent backtracking, that accompanies independent and-parallelism in &-Prolog, is absent in our abstract and-or parallel C-tree model. This is because if b were to completely fail, then this failure will be replicated in each of the three copies of b. We can incorporate intelligent backtracking by stipulating that an untried alternative be stolen from a choice point, which falls in the scope of a parallel conjunction, only after at least one solution has been found for each goal in that parallel conjunction. Thus, c-nodes C2, C3, C4 and C5 (fig 4) will be created only after the first team (consisting of P1 and P2) succeeds in finding solutions a1 and b1 respectively. In this situation if b were to fail, then the c-node C1 will fail, resulting in the failure of the whole parallel conjunction.

4.3. Processor Scheduling

Since our abstract model of C-trees is dependent upon the number of processors available, some of the processor scheduling issues can be determined at an abstract level, without going into the details of a concrete realization of the C-trees. As mentioned earlier, teams of processors are used to carry out or-parallel work while individual processors within a team perform and-parallel work. Since and-parallel work is shared within a team, a processor can in principle steal and-parallel work only from members of its own team. Or-parallel work is shared at the level of teams, thus only an idle team can steal an untried alternative from a choice point. An idle processor will first look for and-parallel work in its own team. If no and-parallel work is found, it can decide to migrate to another team where there is work, provided it is not the last remaining processor in that team. If no such team exists it can start a new team of its own, perhaps with idle processors of other teams, and the new team can steal or-parallel work. One has to carefully balance the number of teams and the number of processors in each team, to fully exploit all the and- and or-parallelism available in a given Prolog program†.

5. Environment Representation

So far we have described and-or parallel execution with recomputation at an abstract level. We have not addressed the crucial problem of environment representation in the C-tree. In this section we discuss how to extend the Binding Arrays (BA) method [W84,W87] and the Stack-copying [AK90] methods to solve this problem. These extensions enable a team of processors to share a single BA without wasting too much space.

5.1 Sharing *vs* Non-Sharing

In an earlier paper [GJ90] we argued that environment representation schemes that have constant-time task creation and constant-time access to variables, but non-constant time task-switching, are superior to those

† Some of the 'flexible scheduling' techniques that have been developed for the Andorra-I system [D91] can be directly adapted for optimal distribution of or- and and-parallel work.

methods which have non-constant time task creation or non-constant time variable-access. The reason being that the number of task-creation operations and the number of variable-access operations are dependent on the program, while the number of task-switches can be controlled by the implementor by carefully designing the work-scheduler.

The schemes that have constant-time task creation and variable-access can be further subdivided into those that physically share the execution tree, such as Binding Arrays scheme [W84, W87, LW90] and Versions Vectors [HC87] scheme, and those that do not, such as MUSE [AK90] and Delphi [CA88]. Both these kinds of schemes have their advantages. The advantage of non-sharing schemes such as Muse and Delphi are that less synchronization is needed in general since each processor has its own copy of the tree and thus there is less parallel overhead [AK90]. This also means that they can be implemented on non-shared memory machines more efficiently. However, operations that may require synchronization and voluntary suspension such as side effects, cuts and speculative scheduling are more overhead prone to implement. When an or-parallel system reaches a side effect which is in a non-leftmost or-branch, it has two choices: (i) it can suspend the current branch and switch to some other node where there is work available, the suspended branch would be woken up when it becomes leftmost; or (ii) it can busy-wait at the current branch until it becomes left most. In case (i) an or-parallel system that does not share the execution tree, such as Muse, will have to save its current execution stack in a scratch memory-area since switching to a new node means that the current stack would be overwritten due to copying of the branches corresponding to the new node. Even if modern sophisticated multiprocessor Operating Systems may allow some memory-saving optimizations, a substantial memory overhead may still be present†. The same holds for case (ii), where a modern OS may manage to avoid busy-waiting, but at the cost of extra memory.

The essential conclusion is that for some applications (those that require processors to synchronize often due to presence of a large number of side-effects and cuts) environment representation schemes which share the or-tree are better, and for some other applications (those that require processors to synchronize less often) schemes which maintain an independent or-tree per processor are better. With this observation in mind we have extended both types of environment representation schemes to accommodate independent and-parallelism with recomputation of goals. We first describe an extension of the Binding Arrays scheme, and then an extension of the stack-copying technique. Due to space limitations the essence of both approaches will be presented rather than specifying them in detail as full models, which is left as future work.

5.2. Environment Representation using BAs

Recall that in the binding-array method [W84, W87] an offset-counter is maintained for each branch of the or-parallel tree for assigning offsets to conditional variables (CVs)† that arise in that branch. The 2 main properties of the BA method for or-parallelism are the following:

(i) The offset of a conditional variable is fixed for its entire life.

(ii) The offsets of two consecutive conditional variables in an or-branch are also consecutive.

The implication of these two properties is that conditional variables get allocated space consecutively in the binding array of a given processor, resulting in optimum space usage in the BA. This is important because a large number of conditional variables might need to be created at runtime‡.

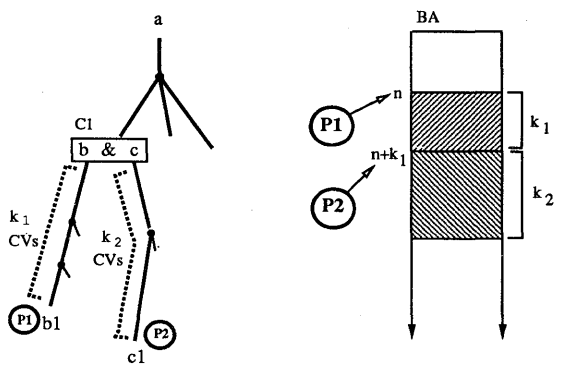


Fig (i): Part of a C-tree

Figure (ii): Optimal Space Allocation in the BA

Figure 5: BAs and Independent And-Parallelism

In the presence of independent and-parallel goals, each of which has multiple solutions, maintaining contiguity in the BA can be a problem, especially if processors are allowed (via backtracking or or-parallelism) to search for these multiple solutions. Consider a goal with a parallel conjunction: `a, (true => b & c), d`. A part of its C-tree is shown in figure 5(i) (the figure

† Experimental results show that processors may voluntarily suspend as much as 10 to 100s of times for large sized programs [SI91].

† Conditional variables are variables that receive different bindings in different environments [GJ90].

‡ For instance, in Aurora [LW90] about 1Mb of space is allocated for each BA.

also shows the number of conditional variables that are created in different parts of the tree). If b and c are executed in independent and-parallel by two different processors P1 and P2, then assuming that both have private binding arrays of their own, all the conditional variables created in branch b-b1 would be allocated space in BA of P1 and those created in branch of c-c1 would be allocated space in BA of P2. Likewise conditional bindings created in b would be recorded in BA of P1 and those in c would be recorded in BA of P2. Before P1 or P2 can continue with d after finding solutions b1 and c1, their binding arrays will have to be merged somehow. In the AO-WAM [GJ89, G91a] the approach taken was that one of P1 or P2 would execute d after updating its Binding Array with conditional bindings made in the other branch (known as the the BA loading operation). The problem with the BA loading operation is that it acts as a sequential bottleneck which can delay the execution of d, and reduce speedups. To get rid of the BA loading overhead we can have a common binding array for P1 and P2, so that once P1 and P2 finish execution of b and c, one of them immediately begins execution of d since all conditional bindings needed would already be there in the common BA. This is consistent with our discussion in section 4.1 about having teams of processors where all processors in a team would share a common binding array.

However, if processors in a team share a binding array, then backtracking can cause inefficient usage of space, because it can create large unused holes in the BA. This is because processors in a team, that are working on different independent and-parallel branches, will allocate offsets in the binding array concurrently. The exact number of offsets needed by each branch cannot be allocated in advance in the binding array because the number of conditional variables that will arise in a branch cannot be determined a priori. Thus, the offsets of independent and-branches will overlap: for example, the offsets of k_1 CVs in branch b1 will be intermingled with those of k_2 CVs in branch c1. Due to overlapping offsets, recovery of these offsets, when a processor backtracks, requires tremendous book-keeping. Alternatively, if no book-keeping is done, it leads to large amount of wasted space that becomes unusable for subsequent offsets (see [GS92, G91, G91a] for more details).

5.2.1. Paged Binding Array

To solve the above problem we divide the binding array into *fixed sized segments*. Each conditional variable is bound to a pair consisting of a segment number and an offset within the segment. An auxiliary array keeps track of the mapping between the segment number and its starting location in the binding array. Dereferencing CVs now involves double indirection: given a conditional variable bound to $\langle i, o \rangle$, the starting address of its segment in the BA is first found from location i of the auxiliary array, and then the value at offset o from that address is accessed. A set of CVs that have been allocated space in the same logical segment (i.e. CVs which have common i) can reside in any physical page in the BA, as long as the starting address of that physical page is recorded in the ith slot in the auxiliary array. Note the similarity of this scheme to memory management using paging in Operating Systems, hence the name Paged Binding Array (PBA)†. Thus a segment is identical to a page and the auxiliary array is essentially the same as a page table. The auxiliary and the binding array are common to all the processors in a team. From now on we will refer to the BA as the Paged Binding Array (PBA), the auxiliary array as the Page Table (PT), and our model of and-or parallel execution as the PBA model‡.

Every time execution of an and-parallel goal in a parallel conjunction is started by a processor, or the current page in the PBA being used by that processor for allocating CVs becomes full, a *page-marker node* containing a unique integer id i is pushed onto the trail-stack. The unique integer id is obtained from a shared counter (called a pt_counter). There is one such counter per team. A new page is requested from the PBA, and the starting address of the new page is recorded in the ith location of the Page Table. i is referred to as the page number of the new page. Each processor in a team maintains an offset-counter, which is used to assign offsets to CVs within a page. When a new page is obtained by a processor, the offset-counter is reset. Conditional variables are bound to the pair <i, o>, where i is the page number, and o is the value of the offset-counter, which indicates the offset at which the value of the CV would be recorded in the page. Every time a conditional variable is bound to such a pair, the offset counter o is incremented. If the value of o becomes greater than K, the fixed page size, a new page is requested and new page-marker node is pushed.

† Thanks to David H. D. Warren for pointing out this similarity.

‡ A paged binding array has also been used in the ElipSys system of ECRC [VX91], but for entirely different reasons. In ElipSys, when a choice point is reached the BA is replicated for each new branch. To reduce the overhead of replication, the BA is paged. Pages of the BA are copied in the children branches on demand, by using a "copy-on-write" strategy. In ElipSys, unlike our model, paging is not necessitated by independent and-parallelism.

A list of free pages in the PBA is maintained separately (as a linked list). When a new page is requested, the page at the head of the list is returned. When a page is freed by a processor, it is inserted in the free-list. The free-list is kept ordered so that pages higher up in the PBA occur before those that are lower down. This way it is always guaranteed that space at the top of the PBA would be used first, resulting in optimum space usage of space in the PBA.

While selecting or-parallel work, if the untried alternative that is selected is not in the scope of any parallel conjunction, then task-switching is more or less like in purely or-parallel system (such as Aurora), modulo allocation/deallocation of pages in the PBA. If, however, the untried alternative that is selected is in the and-parallel goal **g** of a parallel conjunction, then the team updates its PBA with all the conditional bindings created in the branches corresponding to goals which are to the left of **g**. Conditional bindings created in **g** above the choice point are also installed. Goals to the right of **g** are restarted and made available to other member processors in the team for and-parallel execution. Notice that if a C-tree is folded into an or-parallel tree according to the relationship shown in figures 2 and 3, then the behaviour of (and the number of conditional bindings installed/deinstalled during) task switching would closely follow that of a purely or-parallel system such as Aurora, if the same scheduling order is followed.

Note that the paged binding array technique is a generalization of the environment representation technique of AO-WAM [GJ89, G91a], hence some of the optimizations [GJ90a] developed for the AO-WAM, to reduce the number of conditional bindings to installed/deinstalled during task-switching, will also apply to the PBA model. Lastly, seniority of conditional variables, which needs to be known so that "older" variables never point to "younger ones", can be easily determined with the help of the $<i,\ o>$ pair. Older variables will have a smaller value of i; and if i is the same, then a smaller value of o.

More details on Paged Binding Arrays can be found in [GS92, G91].

5.3. The Stack Copying Approach

An alternative approach to represent multiple environments in the C-tree is to use explicit *stack-copying*. Rather than sharing parts of the tree, the shared branches can be explicitly copied, using techniques similar to those employed by the MUSE system [AK90].

To briefly summarize the MUSE approach, whenever a processor P1 wants to share work with another processor P2 it selects an untried alternative from one of the choice points in P2's stack. It then copies the entire stack of P2, backtracks up to that choice point to undo all the conditional bindings made below that choice point, and then continues with the execution of the untried alternative. In this approach, provided there is a mechanism for copying stacks, the only cells that need to be shared during execution are those corresponding to the choice points. Execution is otherwise completely independent (modulo side-effect synchronization) in each branch and identical to sequential execution.

If we consider the presence of and-parallelism in addition to or-parallelism, then, depending on the actual types of parallelism appearing in the program and the nesting relation between them, a number of relevant cases can be distinguished. The simplest two cases are of course those where the execution is purely or-parallel or purely and-parallel. Trivially, in these situations standard MUSE and &-Prolog execution respectively applies, modulo the memory management issues, which will be dealt with in section 5.3.2.

Of the cases when both and- and or-parallelism are present in the execution, the simpler one represents executions where and-parallelism appears "under" or-parallelism but not conversely (i.e. no or-parallelism appears below c-nodes). In this case, and again modulo memory management issues, or-parallel execution can still continue as in Muse while and-parallel execution can continue like &-Prolog (or in any other local way. The only or-parallel branches which can be picked up appear then above any and-parallel node in the tree. The process of picking up such branches would be identical to that described above for MUSE.

In the presence of or-parallelism under and-parallelism the situation becomes slightly more complicated. In that case, an important issue is carefully deciding which portions of the stacks to copy. When an untried alternative is picked from a choice-point, the portions that are copied are precisely those that have been labelled as "shared" in the C-tree. Note that these will be precisely those branches that will also be copied in an equivalent (purely or-parallel) MUSE execution. In addition, precisely those branches will be recomputed that are also recomputed in an equivalent (purely and-parallel) &-Prolog execution.

Consider the case when a processor selects an untried alternative from a choice point created during execution of a goal g_j in the body of a goal which occurs after a parallel conjunction where there has been and-parallelism above the the selected alternative, but all the forks are finished. Then not only will it have to copy

all the stack segments in the branch from the root to the parallel conjunction, but also the portions of stacks corresponding to all the forks inside the parallel conjunction and those of the goals between the end of the parallel conjunction and g_j. All these segments have in principle to be copied because the untried alternative may have access to variables in all of them and may modify such variables.

On the other hand, if a processor selects an untried alternative from a choice point created during execution of a goal g_i inside a parallel conjunction, then it will have to copy all the stack segments in the branch from the root to the parallel conjunction, and it will also have to copy the stack segments corresponding to the goals $g_1 \ldots g_{i-1}$ (i.e. goals to the left). The stack segments up to the parallel conjunction need to be copied because each different alternative within the g_is might produce a different binding for a variable, X, defined in an ancestor goal of the parallel conjunction. The stack segments corresponding to goals g_1 through g_{i-1} have to be copied because the different alternatives for the goals following the parallel conjunction might bind a variable defined in one of the goals $g_1 \ldots g_{i-1}$ differently.

5.3.1. Execution with Stack Copying

We now illustrate by means of a simple example how or-parallelism can be exploited in non deterministic and-parallel goals through stack copying. Consider the tree shown in figure 1 that is generated as a result of executing a query q containing the parallel conjunction (true => a(X) & b(Y)). For the purpose of illustration we assume that there is an unbounded number of processors, P1 ... Pn.

Execution begins with processor P1 executing the top level query q. When it encounters the parallel conjunction, it picks the subgoal a for execution, leaving b for some other processor. Let's assume that Processor P2 picks up goal b for execution (figure 6.(i)). As execution continues P1 finds solution a1 for a, generating 2 choice points along the way. Likewise, P2 finds solution b1 for b.

Since we also allow for full or-parallelism within and-parallel goals, a processor can steal the untried alternative in the choice point created during execution of a by P1. Let us assume that processor P3 steals this alternative, and sets itself up for executing it. To do so it copies the stack of processor P1 up to the choice point (the copied part of the stack is shown by the dotted line; see index at the bottom of figure 6), simulates failure to remove conditional bindings made below the choice point, and restarts the goals to its right (i.e. the goal b). Processor P4 picks up the restarted goal b and finds a solution b1 for it. In the meantime, P3 finds the solution a2 for a (see figure 6.(ii)). Note that before P3 can commence with the execution of the untried alternative and P4 can execute the restarted goal b, they have to make sure that any conditional bindings made by P2 while executing b have also been removed. This is done by P3 (or P4) getting a copy of the trail stack of P2 and resetting all the variables that appear in it.

Like processor P3, processor P5 steals the untried alternative from the second choice point for a, copies the stack from P1 and restarts b, which is picked up by processor P6. As in MUSE, the actual choice point frame is shared to prevent the untried alternative in the second choice point from being executed twice (once through P1 and once through P3). Eventually, P5 finds the solution a3 for a and P6 finds the solution b1 for b.

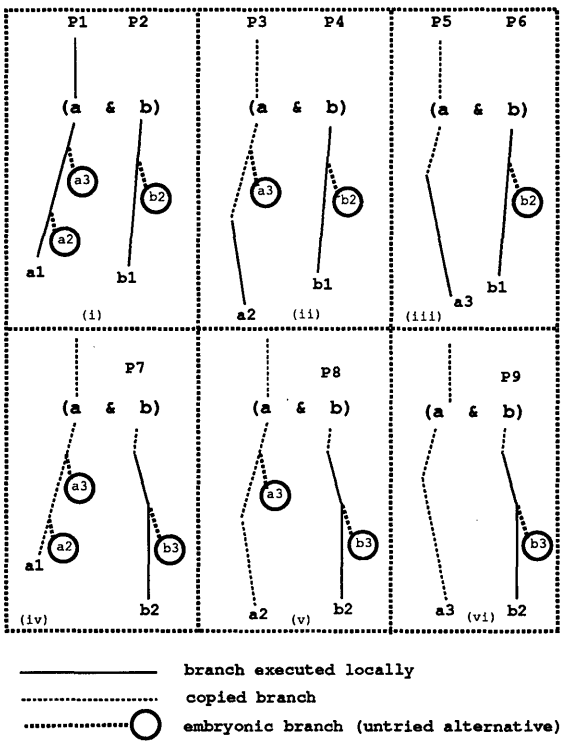


Figure 6: Parallel Execution with Stack Copying

Note that now 3 copies of b are being executed, one for each solution of a. The process of finding the solution b1 for b leaves a choice point behind. The untried alternative in this choice point can be picked up for execution by another processor. This is indeed what is done by processors P7, P8 and P9 for each copy of b that is executing. These processors copy the stack of P2, P4 and P6, respectively, up to the choice point.

The stack segments corresponding to goal a are also copied (figures 6.(iv), 6.(v), 6.(vi)) from processors P1, P3 and P5, respectively. The processors P7, P8 and P9 then proceed to find the solution b2 for b.

Execution of the alternative corresponding to the solution b2 in the three copies of b produces another choice-point. The untried alternatives from these choice points can be picked up by other idle teams in a manner similar to that for the previous alternative of b (not shown in figure 6). Note that if there were no processors available to steal the alternative (corresponding to solution b3) from b then this solution would have been found by processors P7, P8 and P9 (in the respective copies of b that they are executing) through backtracking as in &-Prolog. The same would apply if no processors were available to steal the alternative from b corresponding to solution b2.

5.3.2. Managing the Address Space

While copying stack segments we have to make sure that pointers in copied portions do not need relocation. In Muse this is ensured by having a physically separate but logically identical memory spaces for each of the processors [AK90]. In the presence of and-parallelism and teams of processors a more sophisticated approach has to be taken.

All processors in a team share the same logical address space. If there are n processors in the team the address space is divided up into m memory segments ($m \geq n$). The memory segments are numbered from 1 to m. Each processor allocates its heap, local stacks, trail etc. in one of the segments (this also implies that the maximum no. of processors that a team can have is m). Each team has its own independent logical address space, identical to the address space of all other teams. Also, each team has an identical number of segments. Processors are allowed to switch teams so long as there is a memory segment available for them to allocate their stacks in the address space of the other team.

Consider the scenario where a choice point, which is not in the scope of any parallel conjunction, is picked up by a team Tq from the execution tree of another team Tp. Let x be the memory segment number in which this choice point lies. The root of the Prolog execution tree must also lie in memory segment x since the stacks of a processor cannot extend into another memory segment in the address space. Tq will copy the stack from the xth memory segment of Tp into its own xth memory segment. Since the logical address space of each team is identical and is divided into identical segments, no pointer relocation would be needed. Failure is then simulated and the execution of the untried alternative of the stolen choice point begun. In fact, the copying of stacks can be done incrementally as in MUSE [AK90] (other optimizations in MUSE to save copying should apply equally well to our model, and are left as future work).

Now consider the more interesting scenario where a choice point, which lies within the scope of a parallel conjunction, is picked up by a processor in a team Tq from another team Tp. Let this parallel conjunction be the CGE ($true \Rightarrow g_1 \& \ldots \& g_n$) and let g_i be the goal in the parallel conjunction whose sub-tree contains the stolen choice point. Tq needs to copy the stack segments corresponding to the computation from the root up to the parallel conjunction and the stack segments corresponding to the goals g_1 through g_i. Let us assume these stack segments lie in memory segments of team Tp and are numbered $x_1, \ldots, x_k$. They will be copied into the memory segments numbered $x_1, \ldots, x_k$ of team Tq. Again, this copying can be incremental. Failure would then be simulated on g_i. We also need to remove the conditional bindings made during the execution of the goal $g_{i+1} \ldots g_n$ by team Tp. Let $x_{k+1} \ldots x_l$ be the memory segments where $g_{i+1} \ldots g_n$ are executing in team Tp. We copy the trail stacks of these segments and reinitialize (i.e. mark unbound) all variables that appear in them. The copied trail stacks can then be discarded. Once removal of conditional bindings is done the execution of the untried alternative of the stolen choice point is begun. The execution of the goals $g_{i+1} \ldots g_n$ is restarted and these can be executed by other processors which are members of the team. Note that the copied stack segments occupy the same memory segments as the original stack segments. The restarted goals can however be executed in any of the memory segments.

An elaborate description of the stack-copying approach, with techniques for supporting side-effects, various optimizations that can be performed to improve efficiency, and implementation details are left as future work. Preliminary details can be found in [GH91].

6. Conclusions & Comparison with Other Work

In this paper, we presented a high-level approach capable of exploiting both independent and-parallelism and or-parallelism in an efficient way. In order to find all solutions to a conjunction of non-deterministic and-parallel goals in our approach some goals are explicitly recomputed as in Prolog. This is unlike in other and-or parallel systems where such goals are shared. This allows our scheme to incorporate side-effects and to support Prolog as the user language more easily and simplifies other implementation issues.

In the context of this approach we also presented two techniques for environment representation in the presence of independent and-parallelism which are extensions of highly successful environment representation techniques for supporting or-parallelism. The first technique, based on Binding Arrays [W84, W87], and termed Paged Binding Array technique, yields a system which can be viewed as a direct combination of the Aurora [LW90] and &-Prolog [HG90] systems. The second technique based on stack copying [AK90] yields a system which can be viewed as a direct combination of the MUSE [AK90] and &-Prolog systems. If an input program has only or-parallelism, then the system based on Paged Binding Arrays (resp. Stack copying) will behave *exactly* like Aurora (resp. Muse). If a program has only independent and-parallelism the two models will behave *exactly* like &-Prolog (except that conditional bindings would be allocated in the binding array in the system based on Paged Binding Arrays). Our approach can also support the extra-logical features of Prolog (such as cuts and side-effects) transparently [GS91], something which doesn't appear to be possible in other independent-and/or parallel models [BK88, GJ89, RK89]. Control in the models is quite simple, due to recomputation of independent goals. Memory management is also relatively simpler. We firmly believe that the approach, in its two versions of Paged Binding Array and stack copying can be implemented very efficiently, and indeed their implementation is scheduled to begin shortly. The implementation techniques described in this paper can be used for even those models that have dependent and-parallelism, such as Prometheus [SK92], and IDIOM (with recomputation) [GY91]. They can also be extended to implement the Extended Andorra Model [W90].

Acknowledgements

Thanks to Vítor Santos Costa for his numerous comments on this paper, and to Raéd Sindaha and Tony Beaumont for answering many questions about Aurora and its schedulers. The research presented in this paper has benefited from discussions with Kish Shen, Khayri Ali, Roland Carlsson, and David H.D. Warren, all of whom we would like to thank. This research was supported by U.K. Science and Engineering Research Council grant GR/F 27420, ESPRIT project PEPMA and CICYT project TIC90-1105-CE.

References

[AK90] K. Ali, R. Karlsson, "The Muse Or-parallel Prolog Model and its Performance," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press.

[AK91] K. Ali, R. Karlsson, "Full Prolog and Scheduling Or-parallelism in Muse," To appear in *International Journal of Parallel Programming*, 1991.

[BK88] Uri Baron, et. al., "The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results," In *Proceedings of FGCS '88*, Tokyo, pp. 841-850.

[CA88] W. F. Clocksin and H. Alshawi, "A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors," In *New Generation Computing, 5(1988),* 361-376.

[CC89] S-E. Chang and Y.P. Chiang, "Restricted And-Parallelism Model with Side Effects," Proceedings of North American Conference on Logic Programming, 1989, MIT Press, pp. 350-368.

[CG91] M. Carro, L. Gomez, and M. Hermenegildo, "VISANDOR: A Tool for Visualizing And-/Or-parallelism in Logic Programs," Technical Report, U. of Madrid (UPM), Madrid-Spain, 1991.

[D91] I. Dutra, "Flexible Scheduling in the Andorra-I System," In *Proc. ICLP'91 Workshop on Parallel Logic Prog.*, Springer Verlag, LNCS 569, Dec. 1991.

[G91] G. Gupta, "Paged Binding Array: Environment Representation in And-Or Parallel Prolog," Technical Report TR-91-24, Department of Computer Science, University of Bristol, Oct. 1991.

[G91a] G. Gupta, "And-Or Parallel Execution of Logic Programs on Shared Memory Multiprocessors," Ph.D. Thesis, University of North Carolina at Chapel Hill, Nov. 1991.

[GS92] G. Gupta, V. Santos Costa, "And-Or Parallel Execution of full Prolog based on Paged Binding Arrays," To appear in Proceedings of Parallel Languages and Architectures Europe (PARLE '92), June 1992.

[GH91] G. Gupta and M. Hermenegildo, "ACE: And/Or-parallel Copying-based Execution of Logic Programs," Technical Report TR-91-25, Department of Computer Science, University of Bristol, Oct. 1991. Also in Springer Verlag LNCS 569, Dec. '91.

[GJ89] G. Gupta and B. Jayaraman, "Compiled And-Or Parallel Execution of Logic Programs," In *Proceedings of the North American Conference on Logic Programming '89*, MIT Press, pp. 332-349.

[GJ90] G. Gupta and B. Jayaraman, "On Criteria for Or-Parallel Execution Models of Logic Programs," In *Proceedings of the North Amer-*

ican Conference on Logic Programming '90, MIT Press, pp. 604-623.

[GJ90a] G. Gupta and B. Jayaraman, "Optimizing And-Or Parallel Implementations," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, pp. 737-756.

[GS91] G. Gupta, V. Santos-Costa, "Cut and Side Effects in And-Or Parallel Prolog," Technical Report TR-91-26, Department of Computer Science, University of Bristol, Oct. 1991.

[GY91] G. Gupta, V. Santos Costa, R. Yang, M. Hermenegildo, "IDIOM: A Model for Integrating Dependent-and, Independent-and and Or-parallelism," In *Proc. Int'l. Logic Programming Symposium '91*, MIT Press, Oct. 1991.

[H86] M. V. Hermenegildo, "An Abstract Machine for Restricted And Parallel Execution of Logic Programs". *3rd International Conference on Logic Programming*, London, 1986.

[HG90] M. V. Hermenegildo, K.J. Greene, "&-Prolog and its performance: Exploiting Independent And-Parallelism," In *Proceedings of the 7th International Conference on Logic Programming*, 1990, pp. 253-268.

[HN86] M. V. Hermenegildo and R. I. Nasr, "Efficient Implementation of backtracking in AND-parallelism", *3rd International Conference on Logic Programming*, London, 1986.

[HC87] B. Hausman, et. al., "Or-Parallel Prolog Made Efficient on Shared Memory Multiprocessors," in *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco, CA.

[HC88] B. Hausman, A. Ciepielewski, and A. Calderwood, "Cut and Side-Effects in Or-Parallel Prolog," In *International Conference on Fifth Generation Computer Systems*, Tokyo, Nov. 88, pp. 831-840.

[LK88] Y-J. Lin and V. Kumar, "AND-parallel execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results", in *Fifth International Logic Programming Conference*, Seattle, WA.

[LW90] E. Lusk, D.H.D. Warren, S. Haridi et. al. "The Aurora Or-Prolog System", In *New Generation Computing*, Vol. 7, No. 2,3, 1990 pp. 243-273.

[MH89] K. Muthukumar and M. Hermenegildo, "Complete and Efficient Methods for Supporting Side-effects in Independent/Restricted And-Parallelism," In *Proc. of ICLP*, 1989.

[MH89a] K. Muthukumar, M. V. Hermenegildo, "Determination of Variable Dependence Information through Abstract Interpretation," In *Proc. of NACLP '89*, MIT Press.

[RS87] M. Ratcliffe, J-C Syre, "A Parallel Logic Programming Language for PEPSys" In *Proceedings of IJCAI '87*, Milan, pp. 48-55.

[RK89] B. Ramkumar and L. V. Kalé, "Compiled Execution of the REDUCE-OR Process Model," In *Proc. of NACLP '89*, MIT Press, pp. 313-331.

[S91] R. Sindaha, "The Dharma Scheduler — Definitive Scheduling in Aurora on Multiprocessor Architecture," Technical Report, Department of Computer Science, University of Bristol, forthcoming.

[S89] P. Szeredi, "Performance Analysis of the Aurora Or-parallel Prolog System," In *Proc. of NACLP*, MIT Press, 1989, pp. 713-732.

[SH91] K. Shen and M. Hermenegildo, "A Simulation Study of Or- and Independent And-Parallelism," In *Proc. Int'l. Logic Programming Symposium '91*, MIT Press, Oct. 1991.

[SI91] R. Sindaha, Personal Communication, Sep. 1991.

[SK92] K. Shen, "Studies of And-Or Parallelism in Prolog," Ph.D. thesis, Cambridge University, 1992, forthcoming.

[SW91] V. Santos Costa, D. H. D. Warren, R. Yang, "Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism," In *Proceedings of Principles & Practice of Parallel Programming*, Apr. '91, pp. 83-93.

[VX91] A. Véron, J. Xu, et. al., "Virtual Memory Support for Parallel Logic Programming Systems," In *PARLE'91*, Springer Verlag, LNCS 506, 1991.

[W84] D. S. Warren, "Efficient Prolog Memory Management for Flexible Control Strategies," In *The 1984 Int. Symp. on Logic Prog.*, Atlantic City, pp. 198-202.

[W87] D. H. D. Warren, "The SRI-model for Or-Parallel execution of Prolog – Abstract Design and Implementation Issues," *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco.

[W90] D.H.D. Warren, "Extended Andorra Model with Implicit Control" Talk given at Workshop on Parallel Logic Programming, 7th ICLP, Eilat.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Estimating the Inherent Parallelism in Prolog Programs

David C. Sehr * Laxmikant V. Kalé †
University of Illinois at Urbana-Champaign

Abstract

In this paper we describe a system for compile time instrumentation of Prolog programs to estimate the amount of inherent parallelism. Using this information we can determine the maximum speedup obtainable through OR- and AND/OR-parallel execution. We present the results of instrumenting a number of common benchmark programs, and draw some conclusions from their execution.

1 Introduction

In this paper we describe a method for timing Prolog programs by instrumenting the source code. The resulting program is run sequentially to estimate the sequential and best possible OR parallel execution times. This method is then extended to give the best possible AND/OR parallel execution time. Our instrumentation does not drastically reduce efficiency, and we present the results of a number of programs.

Our AND parallelism estimation method is based upon the work of by Kumar [1988] in estimating the inherent parallelism in Fortran programs. His method augments the source program with a timestamp for each data item d, which is updated each time d is written. In order to honor dependences, each computation that reads d can begin no earlier than the time recorded in d's timestamp. The largest timestamp computed by such an augmented program is the *optimal parallel time* for the original program. This time can be used to evaluate how well a given implementation exploits parallelism.

This paper comprises six sections. The remainder of the first presents some terminology. The second describes measuring the amount of OR parallelism in a Prolog program. The third section extends this method to include AND parallelism. The fourth presents the timing methods used for several builtin predicates. The fifth section gives the results of our technique on the UCB Benchmarks. The last section presents some conclusions and suggests some future work.

1.1 Terminology

A prolog program consists of a *top-level query* and a set of *clauses*. The top-level query is a sequence of *literals*; we shall also use the term *query* to refer to any arbitrary sequence of literals. A literal is an *atom* or a *compound term* consisting of a *predicate name* and a list of *subterms* or *arguments*. Each subterm is an atom or a compound term. The number of subterms of a compound term is its *arity*. A clause has a *head literal* and zero or more *body literals*. A clause with no body literals is a *fact*; others are *rules*. Clauses are grouped into *procedures* by the predicate name and arity of their head literals. The rest of this paper assumes some working knowledge of Prolog's execution strategy.

For our timings we model a program's execution as traversal of its *OR tree* (*SLD tree*). Each node in an OR tree is labeled by a *query*. The first literal of the query at node N is the *literal at* N. The label of the root is the top-level query[1]. Each child N of a node M is produced by unifying a clause C's head with the literal L at M. N's query is formed by replacing L in M's query by the body of C. The left-to-right order of such children is the order of the clauses in the source program. A leaf node with an empty query is a *success*. Sequential Prolog systems traverse this tree depth-first and left to right.

*Center for Supercomputing Research and Development, 305 Talbot Laboratory, 104 S. Wright St., Urbana, IL 61801, USA. (sehr@csrd.uiuc.edu) This work was supported by Air Force Office of Scientific Research grant AFOSR 90-0044 and a grant from the IBM Corporation to CSRD.

†Department of Computer Science, Digital Computer Laboratory, 1304 W. Springfield Ave., Urbana, IL 61801, USA. (kale@cs.uiuc.edu) This work was supported in part by NSF grant NSF-CCR-89-02496.

[1]Which may have appeared in the source program, or may have been typed by the user at the read-evaluate-print prompt.

2 Sequential and OR time

The most efficient OR parallel implementations of Prolog to date [Warren 1987, Ali 1990] have been based upon the Warren Abstract Machine (WAM) [Warren 1983]. Because of this, we compute critical path timings in number of WAM instructions executed. The number of instructions is an approximation to execution time, since each type of WAM instruction takes a slightly different time. Variations in execution time come mainly from two sources: argument unification and backward execution. The former comes from the **get_value** and **unify_value** instructions, whose costs depend on the size of the terms they unify, which can be substantial. We address this by making the cost of these instructions the number of unification steps they perform. Backward execution comes from instruction failure and may perform significant bookkeeping changes, especially for deep backtracking. Different WAM implementations, particularly parallel ones, have differing costs for backward execution. In the measurements presented here we have assumed zero backward execution cost, but other cost assumptions can be used.

The execution time of a program has two components. The literal L at a node N in the OR tree is a call to a procedure p. Calling p consists of setting up L's calling arguments by a sequence of **put** instructions and performing the call by a **call** or **execute** instruction. The execution time of this sequence is a statically computable time $t_p(L)$ for L, which we approximate by the number of **put** instructions plus one.

Executing a called procedure consists of trying clauses in succession. If C is being tried for the call, the call arguments are ***unified*** with the arguments of C's head literal H. This is done by **get** and **unify** instructions and takes a time $t_u(H)$. In general the execution time of these instructions cannot be estimated at compile time, so this head unification is performed by calls to run-time routines for the corresponding WAM instructions. $t_u(H)$ is the sum of the times computed by these routines.

To represent execution times the OR tree is given two new labels. First, each node N is labeled with the time $t_p(L)$ for the literal L at N. Second, each edge (N, M) is labeled with $t_u(H)$, where H is the head literal of the clause C applied to produce node M. The program's all-solutions sequential execution time is the sum of the all t_p's and t_u's in the tree's processed region[2].

[2]Predicates such as cut may prevent traversal of parts of the tree.

```
fib(0,1).
fib(1,1).
fib(N,F) :-
    N > 1,
    N1 is N - 1,
    fib(N1,F1),
    N2 is N - 2,
    fib(N2,F2),
    F is F1 + F2.
```

Figure 1: A program to be timed

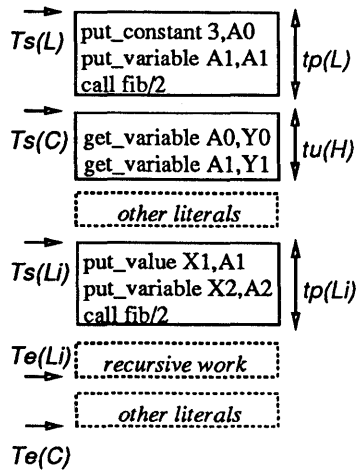


Figure 2: Execution of a timed literal L

2.1 Pure Prolog

Finding the minimum OR parallel time requires finding the *critical path* in the OR tree. For a *pure* Prolog program this is done by summing the t_p's and t_u's from the root to each leaf. The critical path has the largest such sum. Programs containing builtins such as **read**, **setof**, **recorda**, and **assert** require timing in sequential order. We first describe the method for pure programs and extend it to handle these predicates below.

Figure 1 shows a program to be instrumented, and Figure 2 shows its execution. The time at which literal L is to be processed is denoted by $T_s(L)$. If L is at the root of the OR tree, then $T_s(L) = 0$. Otherwise $T_s(L)$ is the time the preceding computation finished. Execution of L begins with the **puts** and **call**, which take time $t_p(L)$, as we noted above. Thus the earliest time any clause can be tried for L is $T_s(L)+t_p(L)$. This is the *start time* $T_s(C)$ for every clause C applied for L, since all are tried in OR parallel. Head unification for

C begins at $T_s(C)$ and is done by **get** (and **unify**) instructions. If successful, this completes at time $T_s(C) + t_u(H)$. If C is a fact, then the end time is $T_e(C) + 1$, where the 1 is for the **proceed** instruction.

If C is a rule, each literal L_i is processed as L was, begins at time $T_s(L_i)$, and ends successful execution at time $T_e(L_i)$. The first body literal begins at time $T_s(L_1) = T_s(C) + t_u(H)$. If the call from L_i is successful and returns at time $T_e(L_i)$, then the next literal L_{i+1} starts at time $T_s(L_{i+1}) = T_e(L_i)$. This continues until the last literal L_n completes at time $T_e(L_n)$, which is also the finish time $T_e(C)$ for C.

The time for a success is $T_e(L)$ for the last literal L in the top-level query. The time for a failed instruction in C is $T_s(C)$ plus the portion of $t_u(H)$ computed before the failure. Most builtins are given a cost of one, and builtin failure takes the same time as a successful call does.

The system maintains a global critical path time $T_{\max}$. Whenever a library routine performing head unification fails at time T_f, it examines $T_{\max}$, and stores the larger of the two times as the new $T_{\max}$. The library routine that computes $T_s(C)$ also updates $T_{\max}$, and the top-level query is modified to update it as well.

Figure 3 shows the timed version of Figure 1. Each clause has two new arguments, **Ts** and **Te**, and head unification is performed by routines such as **get_constant** and **get_variable**. These routines perform the corresponding WAM operation and update the critical path time. The first two clauses are facts, so the end time is computed by an **update_time** literal for the **proceed** instruction.

The third clause is a rule, so each body literal L has a preceding **update_time** literal. If L refers to a user-defined predicate this literal computes $T_s(L) + t_p(L)$ for use as the start time for the call. If L refers to a builtin predicates (except those in Section 4) the **update_time** literal adds $t_p(L)$, plus one for the builtin's execution time, and uses this as the *end time* for L.

Each clause also has an initial *index* literal that enables last call optimization. Moving head unifications to the body made indexing impossible, so this literal is added to perform first argument indexing. If this is not done, last call optimization rarely works. This literal appears sufficient for last call optimization with the Sicstus compiler.

```
fib(A,B,Ts,Te) :-
    (A == 0 ; var(A)),
    get_constant(A,0, Ts, Tu1),
    get_constant(B,1, Tu1, Tu2),
    update_time(Tu2, 1, Te).

fib(A,B,Ts,Te) :-
    (A == 1 ; var(A)),
    get_constant(A,1, Ts, Tu1),
    get_constant(B,1, Tu1, Tu2),
    update_time(Tu2, 1, Te).

fib(A,B,Ts,Te) :-
    get_variable(A,N, Ts, Tu1),
    get_variable(B,F, Tu1, Tneck),
    update_time(Tneck, 4, Te1),
    N > 1,
    update_time(Te1, 6, Te2),
    N1 is N - 1,
    update_time(Te2, 3, Ts3),
    fib(N1, F1, Ts3, Te3),
    update_time(Te3, 6, Te4),
    N2 is N - 2,
    update_time(Te4, 3, Ts5),
    fib(N2, F2, Ts5, Te5),
    update_time(Te5, 6, Te),
    F is F1 + F2.
```

Figure 3: Program after instrumentation

3 Adding AND parallelism

The critical path time determines the best possible OR parallel execution time. Often segments of a branch can execute simultaneously, and doing so would reduce that critical path time. This is AND parallel execution, and unlike OR parallelism, it requires testing for dependences even in pure Prolog programs. In this section we describe the application of Kumar's [1988] techniques for Fortran to estimate the best AND/OR parallel execution time. The method we describe extends his work to deal with the dynamic data structures and aliasing present in Prolog. We believe this framework has the advantage over other methods [Shen 1986, Tick 1987] of allowing us to extend it to measure critical path times in programs with user parallelism.

A program's dependences can only be exactly determined at execution time, since one execution may have a dependence while another does not. A compiler, to ensure legal execution, must assume a

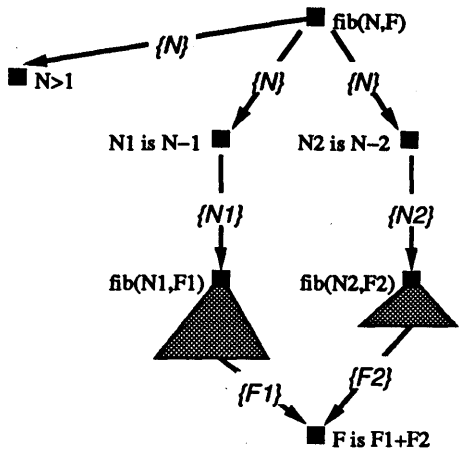


Figure 4: A dependence graph

dependence exists unless it can be proven not to. Because of this, compilers often infer many more dependences than are actually present in the program. Another use of the method we propose is to compute exact dependences to test the effectiveness of dependence tests.

There are a number of AND parallel execution models that differ in their treatment of the dynamic nature of dependences. The approaches range from dependence graphs that are static [Kalé 1987, Chang *et al* 1985, Wise 1986] to partly dynamic (conditional) [DeGroot 1984, Hermenegildo 1988] to completely dynamic [Conery and Kibler 1985]. Kal'e [1984] notes that in some rare situations it may be beneficial to evaluate dependent literals in parallel. His Reduce-Or Process Model allows for dependent AND parallelism, but his implementation [Ramkumar and Kalé 1989] supports only independent AND parallelism. Epilog [Wise 1986] also permits dependent AND parallelism, but provides a primitive (`CAND`) to curtail it. The model we have developed includes dynamic, independent AND parallelism, with a strict sequential ordering on dependent literals. We are only able to present the results here for independent AND parallel execution, though, because of a problem in the Prolog system used to execute the instrumented programs. In the future we hope to report the timings for the more general approach.

3.1 Dependences

The third clause in Figure 1 contains six body literals that might potentially execute in parallel. The arguments of the > builtin must both be numeric expressions, so to execute correctly the argument `N` to `fib` must be an integer. Because neither writes `N`, the two `is` goals can execute independently. Each reads `N` and produces a binding for `N1` or `N2`, the values of `N` for the recursive instances. Since all `fib` clauses read `N`, the recursive calls can only begin after their corresponding `is`. The final `is` literal requires the value of both `F1` and `F2`, so the two `fib` calls must precede the final `is`. There need be no other ordering between literals.

Figure 4 shows the *dependence graph* for the clause. There is a node for the initial call to `fib` and a node for each body literal. Recursive computations are represented by shaded areas. An arc between two nodes represents a *dependence*, or that the node at the tail must precede the node at the head of the arc. Dependence arcs are labeled with the variables causing them. Such a variable v causes a dependence δ in one of two ways. First, if the node at the tail of δ binds v, and and v is read at the head, then there is a *data dependence*. Second, if the node at the head of δ binds v and the node at the tail reads v using a metalogical predicate (`var`, `write`, etc.), then there is an *anti-dependence*. Anti-dependences arise when a literal succeeds with a variable v unbound and would fail or produce incorrect output because v is subsequently bound.

3.2 Shadow terms

Dependences are detected at run time by *shadow terms*. Each term t has a shadow term $\psi(t)$ associated with it, which mirrors t's structure. The shadow of an atomic term is the atom a. The shadow term of a compound term $t = f(t_1, \ldots, t_n)$ is $s(\psi(t_1), \ldots, \psi(t_n))$, where $\psi(t_i)$ is the shadow for t_i.

A variable must be bound for a dependence to exist, so the shadow term for a variable keeps the binding times for that variable (there can be multiple bindings, since some may be variable-to-variable). The shadow of an unbound variable is unbound. If v is bound to any term t at time T by a `get_variable` or `unify_variable` instruction, the shadow variable $\psi(v)$ is dereferenced and the final variable is bound to the structure $w(\psi(t), T)$. The same operation is performed if v is bound to a non-variable term t by any other instruction. If v is bound to another variable v' by any other instruction at time T, an alias has been created. The two shadows reflect this by dereferencing both $\psi(v)$ and $\psi(v')$ and binding the final variables of both to the term $w(\psi'(v), T)$, where $\psi'(v)$ is a new unbound

```
fib(A, B, Sa, Sb, Ts, Te) :-
    (A == 0 ; var(A)),
    get_constant(A,0, Sa, Ts, Tu1),
    get_constant(B,1, Sb, Tu1, Tu2),
    update_times(Tu2, 1, Te).

fib(A, B, Sa, Sb, Ts, Te) :-
    (A == 1 ; var(A)),
    get_constant(A,1, Sa, Ts, Tu1),
    get_constant(B,1, Sb, Tu1, Tu2),
    update_times(Tu2, 1, Te).

fib(A, B, Sa, Sb, Ts, Te) :-
    get_variable(A,N, Sa, Sn, Ts, Tu1),
    get_variable(B,F, Sb, Sf, Tu1, Tu),
    max_shadow_time(Tu, [Sn], Tt1),
    update_time(Tt1, 4, Te1),
    N > 1,
    max_shadow_time(Tu, [Sn1,Sn], Tt2),
    update_time(Tt2, 6, Te2),
    N1 is N - 1,
    set_shadows([Sn1],[N1],Te2),
    update_time(Tu, 3, Ts3),
    fib(N1, F1, Sn1, Sf1, Ts3, Te3),
    max_shadow_time(Tu, [Sn2,Sn], Tt4),
    update_time(Tt4, 6, Te4),
    N2 is N - 2,
    set_shadows([Sn2], [N2], Te4),
    update_time(Tu, 3, Ts5),
    fib(N2, F2, Sn2, Sf2, Ts5, Te5),
    max_shadow_time(Tu, [Sf,Sf1,Sf2], Tt6),
    update_time(Tt6, 6, Te6),
    F is F1+F2,
    set_shadows([Sf], [F], Te6),
    max([Te1,Te2,Te3,Te4,Te5,Te6], Te).
```

Figure 5: AND/OR instrumented program

variable. If v is examined by a meta-logical builtin at time T, $\psi(v)$ is dereferenced, and the final variable is bound to $\mathrm{m}(\psi'(v), T)$, where $\psi'(v)$ is a new unbound variable.

3.3 Dependences with shadow terms

Figure 5 shows `fib` after instrumentation for AND/OR parallelism. Each variable `V` in a clause has a shadow variable `Sv`, and each head argument has a shadow argument. The end time for a clause is the largest end time for any literal in that clause, as if each literal starts immediately after head unification and suspends until its dependences are satisfied. In Figure 5 the end time is shown as computed by a `max` literal at the end of the clause. This is for clarity of presentation only, because this would inhibit last call optimization. In the real version a current maximum is passed to each body literal in succession.

The head unification routines now include shadow variables as arguments, since it is in these instructions that dependences in user-defined predicates are enforced. These routines previously computed their finish time only from the start time and the cost of the instruction. Now there is the possibility that the instruction must wait until the shadow time for a variable causing a dependence before performing the unification. Hence the completion time is computed by performing the unification and keeping a current time. Whenever a term is referenced the current time becomes the maximum of the current time and the timestamp. The unification is then performed and the current time incremented.

Two other predicates enforce dependences involving builtins. The first, `max_shadow_time`, computes the earliest time the builtin's arguments are available[3] from the latest time in the arguments' shadows. This enforces data dependences that have the builtin as their head. The builtin's end time is computed by `update_time`, as before. The second predicate, `set_shadows`, builds shadows for changes to the arguments of a builtin. Shadows are built for those arguments that are bound or are examined by meta-logicals, and they are constructed from the variable bindings after execution. This handles builtins at the tail of a dependence. For some builtins such as `=..` this can be fairly complex.

4 Builtin predicates

Prolog has several types of builtin predicates, each with a different set of effects on critical path timing. We have already noted that meta-logical builtins (`var`, `write`, etc.) can cause anti-dependences. In this section we describe four other kinds of predicates and methods for timing each of them.

4.1 Predicates involving call

There are four predicates that implicitly use the meta-logical builtin `call`. They are `bagof`, `setof`, `not`, and `\+`. Timing these predicates requires two

[3] This predicate is also used to enforce independent-AND parallel execution, by making every user predicate strict

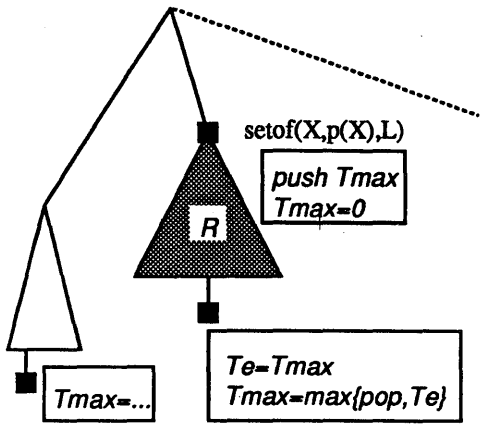


Figure 6: Processing setof

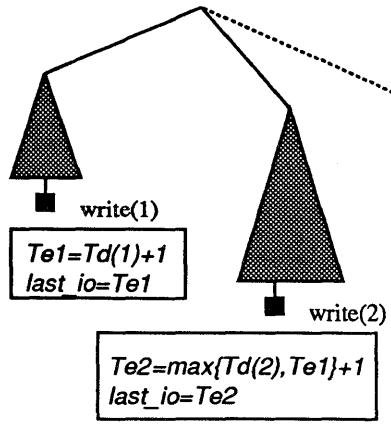


Figure 7: Processing the input/output predicates

kinds of special handling. First, since `call`'s arguments may be constructed at run time, instrumentation is done at run time. This is done by including the the instrumentation program in the timed program. Second, `setof`, `bagof`, `not` and `\+` traverse an entire OR tree, so their finish times are related to the longest path in that tree. A stack of maximum times is used with nested calls to these predicates to collect a subtree's maximum time. For `setof` and `bagof` we also add one for each solution for the cost of building the returned list.

Figure 6 shows the processing of a call to `setof` that computes all the solutions for the `p(X)` in region R and collects them in a list `L`. Since it traverses the whole OR tree R required to compute `p(X)`, `setof`'s finish time is the longest completion time in R. The maximum time is maintained by `update_time` in the global variable `Tmax`[4]. Since there may be a previous maximum time greater than the largest completion time in R, `Tmax` is pushed on a stack and the start time for the `setof` is used as `Tmax`. R is traversed and the maximum time is stored in `Tmax`, as always. The return time for `setof`, `Te` is `Tmax`. At the end of `setof` `Tmax` is set to the maximum of the stack value and `Te`, so again `Tmax` contains the global largest time.

4.2 Read and write

Neither `setof` nor pure Prolog cause dependences between branches in the OR tree. The input/output predicates (`read`, `write`, etc.) cause cross-branch dependences, since the observable order of input/output needs to conform to Prolog's left-to-right order. Figure 7 depicts the execution of a program with two `writes`, w_1 and w_2. Data dependence would permit each `write` to start when its arguments were ready (times $T_d(1)$ and $T_d(2)$ respectively) were it not for the order of output. w_1 must write its output before w_2, so to determine when input or output can be done we maintain a global variable `last_io`. In this example, w_2 cannot write its output until $\max\{T_d(2), \texttt{last\_io}\}$. Writes cost one time unit, so w_2 can start no earlier than $\max\{T)d(2), T_d(1)+1\}$. In the instrumented version each input/output predicate is preceded by a literal that updates `last_io`.

4.3 Recorda and recorded

Prolog also has the builtins `recorda`, `recorded`, and `erase` to manipulate an internal database. Parallel accesses to relations in the database must appear to preserve the sequential execution order. Accesses to different database relations do not affect one another, so this order is only within a relation. It is not necessary to serialize accesses to each relation to preserve the appearance of sequential access order. All we need is to guarantee that read accesses to an element by `recorded` occur after the write access that placed that element there, and that write accesses (`recordas` and `erases`) are ordered. The former is enforced by pairing each item placed in the database with its insertion time. Accesses by `recorded` wait until the maximum of the data dependence time T_d and the element's insertion time. The write order is enforced by labeling each relation with a `last_modify` that is updated

[4] In the implementation of our system the maximum time, along with a parallelism histogram, is maintained by several C routines accessed through a foreign function interface, but this is done only for the sake of efficiency.

Program Name	Serial WAM Instr.	OR Parallel Speedup	AND/OR Parallel Speedup
chat_parser	1014791	257	1596
crypt	31787	58	114
divide10	207	1	2
fast_mu	8899	9.1	10.7
flatten	5218	1.25	2.37
log10	119	1	1.2
meta_qsort	38675	2.1	3.7
mu	5925	16.7	17.7
nand	180145	5.4	14.3
nreverse	4460	1	1
ops8	163	1.04	2.8
poly10	307177	1.1	76.3
prover	7159	4.5	14.2
qsort	5770	1.3	1.5
queens8	33821	26.4	69.3
query	17271	243	480
reducer	279220	2	3.3
serialise	3199	1.4	1.9
tak	1431202	1.1	686
times10	207	1	1.9
unify	29490	1.6	3.5
zebra	261858	453	482

Table 1: Instrumented benchmark times

just like `last_io`.

4.4 Assert and retract

Prolog also allows **assert** and **retract** to modify the program at run time. These predicates are timed by the method for **call** and that for the internal database. The former is because the **asserted** clause can be constructed at run time, and hence the instrumentation must be done then. The latter is because predicates modified at run time must obey the access rules for database updates. The write-write (**assert** and **retract**) order is enforced by updating the **last_modify** for the predicate. The read-write ordering is maintained by adding a first literal to each **asserted** clause that records when it was added. This is used to determine the earliest time a read (a **clause** builtin or call to the modified predicate) can execute.

5 Analysis of programs

Table 1 presents the results obtained by instrumenting 23 of the University of California at Berkeley's UCB benchmarks. These programs range over a variety of sizes and purposes. There are several interesting facts to observe from these programs. First, David H. D. Warren's assertion [Warren 1987] that OR parallelism was likely to produce significant speedups on a range of programs appears to be borne out. Several programs achieved small speedups from OR parallelism, mostly due to shallow backtracking (e.g flatten, ops8, poly10, qsort, tak, unify). Improved indexing would probably eliminate most of this OR parallelism. A number of programs exhibited essentially no OR parallelism (e.g. divide10, log10, nreverse, times10).

In general, independent AND parallel execution improved the performance of programs already speeded up by OR parallel execution by a small factor (1-6). These programs have all shown reasonable speedups in real OR parallel systems[Szeredi 1989]. Our results show that there is plenty of parallelism in several of these programs to extend to much larger machines (e.g. consider chat_parser, query and zebra). Those with smaller speedups may profit from the introduction of independent AND parallelism.

Of the programs that were mostly OR-sequential, the majority get very small speedup by applying independent AND parallel execution. For divide10, log10, and times10, this is because the AND parallel sub-problems are very unbalanced; that is, one sub-problem is much larger than the other. For nreverse, the reason is that independent AND parallel execution is not able to execute the two body goals of nreverse in parallel. It is a recurrence, and is hence completely sequential. This can be addressed by replacing the algorithm or applying a parallel recurrence solver.

The best results for independent AND parallelism come from poly10 and tak. In both cases these give rise to fairly large numbers of independent subcomputations. In the case of tak, the branching factor is approximately three and the calling depth is large, so a large speedup is obtained. Qsort on a well-chosen input list with a better partition routine should be able to obtain similar results.

These results are just the beginning of understanding the parallelizability of programs, as we would like information on the more general AND and other sorts of parallelism. However, they can tell us something about how much speedup we can reasonably expect from parallel models. Moreover, examining these programs to see where dependences occur should help in designing restruc-

turing transformations.

6 Conclusions

The amount of OR and AND/OR parallelism in a Prolog program can be effectively measured by sequentially executing an instrumented version of that program. The timings obtained this way give a best-possible speedup under two different parallelism models, and can be used for a number of purposes. First, they can be used to evaluate the ability of a parallel execution model to exploit parallelism. These results can suggest areas of improvement for such models. We intend to instrument a number of programs for this purpose.

With some relatively simple extensions this technique can measure the amount of a number of lower-level program characteristics. Among these are unification parallelism, backtracking properties, aliasing, data dependences, and dereference costs.

Prolog can also be extended with predicates for source-level parallelism. With proper timing methods, this instrumentation method can be used to evaluate restructuring transformations for Prolog. The instrumentation system we described has been extended with such predicates and we have begun to evaluate transformations. In the future we will describe these extensions to the instrumentation method as well as the results of our restructuring transformations.

Acknowledgments

The authors would like to thank David Padua for his many useful suggestions about this work.

References

[Ali 1990] Khayri Ali. The muse or-parallel prolog model and its performance. In *Proceedings of the 1990 North American Logic Programming Conference*, pages 757–776, 1990.

[Chang *et al* 1985] J. Chang, A. M. Despain, and D. DeGroot. And-parallelism of logic programs based on a static data dependency analysis. In *Proceedings of Compcon 85*, 1985.

[Conery and Kibler 1985] J.S. Conery and D.F. Kibler. And parallelism and nondeterminism in logic programs. *New Generation Computing*, 3:43–70, 1985.

[DeGroot 1984] D. DeGroot. Restricted and-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478. North Holland, 1984.

[Hermenegildo 1988] M. V. Hermenegildo. *Independent AND-Parallel Prolog and its Architecture.* Kluwer Academic Publishers, 1988.

[Kalé 1984] Laxmikant V. Kalé. *Parallel Architectures for Problem Solving.* PhD thesis, State University of New York at Stony Brook, 1985.

[Kalé 1987] Laxmikant V. Kalé. Parallel execution of logic programs: the reduce-or process model. In *Proceedings of the International Conference on Logic Programming*, pages 616–632, May 1987.

[Kumar 1988] Manoj Kumar. Measuring parallelism in computation intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9), September 1988.

[Ramkumar and Kalé 1989] B. Ramkumar and L.V. Kalé. Compiled execution of the reduce-or process model on multiprocessors. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 313–331, October 1989.

[Shen 1986] Kish Shen. An investigation of the argonne model of or-parallel prolog. Master's thesis, University of Manchester, 1986.

[Szeredi 1989] Peter Szeredi. Performance analysis of the aurora or-parallel prolog system. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 713–732, 1989.

[Tick 1987] Evan Tick. *Studies in Prolog Architectures.* PhD thesis, Stanford University, June 1987.

[Warren 1983] David H. D. Warren. An abstract prolog instruction set. Technical report, SRI International, October 1983. Technical Note 309.

[Warren 1987] David H.D. Warren. The sri model for or parallel execution of prolog — abstract design and implementation. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–103. September 1987.

[Wise 1986] Michael Wise. *Prolog Multiprocessors.* Prentice-Hall International Publishers, 1986.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Implementing Streams on Parallel Machines with Distributed Memory

†Koichi Konishi †Tsutomu Maruyama †Akihiko Konagaya
‡Kaoru Yoshida ‡Takashi Chikayama

Abstract

Stream-based concurrent object-oriented programming languages (SCOOL) to date have been typically implemented in concurrent logic programming languages (CLL). However, CLLs have two drawbacks when used to implement message streams on parallel machines with distributed memory. One is the lack of restriction on the number of readers of a shared variable. The other is a cascaded buffer representation of streams. These require many interprocessor communications, which can be avoided by language systems designed specially for SCOOLs. The authors have been developing such a language system named A'UM-90 for A'UM, a SCOOL with highly abstract stream communication. This paper presents the optimized method used in A'UM-90 to implement streams on distributed memory. A stream is represented by a message queue, which migrates to its reader's processor after the processor becomes known. The improvement from using this method is estimated in terms of the number of required interprocessor communication, and is demonstrated by the result of a preliminary evaluation.

1 Introduction

One natural use of concurrent logic programming languages(CLLs) is to implement the Actor or object-oriented programming models. In a CLL, it is easy to specify objects running concurrently, communicating with one another by messages sent in streams[Shapiro and Takeuchi 1983]. Message streams in CLLs are especially useful, as they provide flexibility and modularity, and facilitates the exploitation of parallelism; they allow dynamic re-configuration of communication channels, while each object knows little about the partners with whom it is communicating.

To support this style of programming, a number of languages have been proposed ([Furukawa et al. 1984] [Kahn et al. 1986] [Yoshida and Chikayama 1988] [Saraswat et al. 1990]). We call these languages stream-based concurrent object-oriented languages(SCOOL).

Most research on SCOOLs to date has been focused on providing excellent expressibility. While SCOOLs have been implemented in CLLs, to our knowledge, no language system dedicated for SCOOLs has been implemented.

A dedicated system for SCOOL can be much more efficient than those implemented in CLLs when the abstraction and other information in programs are fully exploited. The authors have been developing such a dedicated system for a kind of SCOOL, A'UM. The system is named A'UM-90, and is targeted for multiprocessor systems with distributed memory.

In this paper, some drawbacks of CLLs as implementation languages for stream communications are discussed, then it is shown how A'UM's well-regulated abstract streams can be efficiently implemented. A brief description of such an implementation is given, its improvement over a CLL implementation is estimated, and the results of a preliminary evaluation are given.

The next section describes the implementation of objects and stream communication in CLLs. Section 3 introduces SCOOLs as natural descendants of CLLs. Section 4 explains why CLLs are inadequate for implementing streams. Section 5 describes A'UM and A'UM-90 briefly. Section 6 describes the implementation of stream communication in A'UM-90 and its costs. Section 7 shows some results of evaluation. The last section gives conclusion.

2 Objects in CLL

Stream-based concurrent object-oriented programming languages have evolved from efforts to embody the Actor or object-oriented programming models in CLLs[Shapiro and Takeuchi 1983]. This style of pro-

†NEC Corporation
4-1-1, Miyazaki, Miyamae-ku, Kawasaki, Kanagawa 216, Japan
{konishi, maruyama, konagaya}@csl.cl.nec.co.jp

‡Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, Japan
{yoshida, chikayama}@icot.or.jp

```
object([message(Arguments) | In], State) :-
            true |
            method(Arguments, State, NewState),
            object(In, NewState).
```

Figure 1: A clause representing an object

gramming has the virtues of object-oriented programming such as modularity and natural parallelism in an extended way[Kahn et al. 1989]. For example, an object implemented in a CLL may have multiple input ports, and communication ports can be transferred between processes. Moreover, it can send messages before the destination is determined. In this chapter, an implementation of object-oriented programming in a CLL is briefly described.

Many CLLs (FCP, FGHC, Fleng, Oc, Strand, etc.) have been proposed to date. We use FGHC[Ueda 1985] in the following explanation.

Figure 1 shows a typical example of representing an object in FGHC. The behavior of an object is defined by a number of clauses similar to the one above. Given these clauses, a goal named `object` represents the state of an object at a certain moment. The first argument is a shared variable used as a communication port, from which the object receives messages. The second argument is the internal state of the object.

When another goal sharing the variable with the first goal assigns a term `[message(Actuals) | Rest]` to the variable, the above clause can be selected, and `Rest` becomes shared by the two goals. `Actuals` are bound to `Arguments`, and the body of the clause is executed.

A goal named `method` performs most of the actual work, creating new states and assigning it to `NewState`. A new `object` goal is created with `Rest` as the first argument and `NewState` the second. Thus, an object, or a process, is represented by the recurring creation of goals with altered states.

Communication ports are represented by variables shared by two goals. One goal emits a message by assigning a structure containing a message and a new variable. When the other goal receives the message by successfully matching itself with a head of a clause, the new variable becomes shared, to be used as a new port. By repeating this procedure, these goals can communicate as many messages as required, one after another. The connection is closed when a structure containing no variable is assigned. Communication in this style is called *stream communication.*

Basically, stream communication is one-to-one as described above. However, several streams of messages can easily be merged into one by a simple process. A merger should have several ports representing the input streams to be merged and one more for the output. It receives a message from one of its input ports and forwards it to the output port.

Many types of mergers with varying policies can be devised. A merger of one type might receive from an arbitrary port, utilizing the non-determinism in clause selection of the CLL. A merger of another type might concentrate on one port until the connection through it is closed, then it might move on to another port. We call the former type a *merger*, and the latter an *appender*, because it effectively appends streams one after another.

3 SCOOL

Programming objects in a CLL has several obvious drawbacks. First, the implementation of stream communication is explicitly described in the program. Streams are explicitly formed using messages and a variable, and many to one communications are implemented with merger processes. Programmers must make sure that the same conventions are used throughout their programs. Secondly, contentions are apt to happen, due to the lack of restriction on multiple writers to a variable. Lastly, the verbosity, in particular manipulation of internal states, is excessive. It is cumbersome to provide all the details of communication.

Many SCOOLs have been proposed to remove these drawbacks ([Furukawa et al. 1984] [Kahn et al. 1986] [Yoshida and Chikayama 1988] [Saraswat et al. 1990]). These languages have a form for class definition, introduced to make a concise description of object behavior possible. Stream communication is denoted by dedicated expressions, with its implementation removed from programs.

To our knowledge, all SCOOLs have been implemented in CLLs. It is natural and efficient to use CLLs for this purpose, but is problematic with respect to the resulting system's performance. CLL systems can not provide a thoroughly object-oriented view efficiently, such as integers operated on by messages. Another problem is implementing stream communication on a multiprocessor system with distributed memory. We focus on the latter problem, and explain the inadequacies of CLLs in the next section.

4 Problems in implementing streams in CLLs

Stream communication, and more generally asynchronous communication, uses message buffers to store pending messages. In distributed memory multiprocessor systems, accessing a message buffer requires interprocessor communications(IPC), unless both the accessing process and the buffer are on the same processor.

While a single IPC suffices to write a message into a

buffer on a remote processor, reading a message requires two: a request and a reply. Placing the buffer on the reader's processor can save one IPC for each message communicated through the buffer.

However, it's difficult for CLL systems to place the buffer on the reader's processor. CLL systems use a shared variable as a message buffer, and they can't tell the readers of a variable from the writers. In addition, there may be multiple readers for a variable. In that case, there is a relatively small advantage in saving IPCs for only one reader among many.

Moreover, the number of IPCs required would not be reduced even if the buffer is placed on the reader's processor. In a CLL, streams are represented as a sequence of message buffers, and the writer only knows the last one. When it becomes full, a new buffer is appended to the sequence, and if it is created on the reader's processor, the address must be propagated to the writer. This costs an additional IPC for every message sent.

Since CLL systems may not place shared variables on the reader's processor, implementing these streams in CLLs results in costly remote reads, repeated for every buffer.

The argument so far prompts the development of a dedicated system for SCOOLs. A'UM-90 is such a system for A'UM, a SCOOL that thoroughly integrates streams into its specification. The next section describes A'UM and gives an overview of A'UM-90.

5 A'UM and A'UM-90

5.1 Behavior of Objects

All A'UM objects run concurrently. They keep internal states called *slots*, and execute methods according to the messages they receive.

The class an object belongs to defines its behavior. A class definition has the following form, which includes the declaration of the class name, the classes it inherits from, slot names (local state) and definitions of its methods.

```
class class_name.
        super_class_decl
        slot_decl
        method_defs
end.
```

An object receives messages from only one stream, called its *interface*. An object is referenced by connecting a stream to its interface. Streams connected to the object later on will be merged into the interface.

A method is defined by the following form.

selector -> *actions*.

where *selector* is the method's name, and *actions* specify the operations it performs.

The only operations methods are allowed to perform are connecting a stream to another, creating an object, and sending a message to a stream.

5.2 Streams in A'UM

Stream communication in A'UM is highly abstract, providing safe communications and the notion of channels. Directed variables prevent contentions for a stream. The semantics of variables are enhanced so that they denote a set of confluent streams called a channel, a more general concept than a stream.

All variables in A'UM have a stream as their value. The role of streams in A'UM is similar to pointers in Lisp; streams are the sole way of referencing objects.

5.2.1 Operations on Streams

A stream is a sequence of messages, directed to a certain receiver. A message sent to a stream is placed at the end of the stream. Sending is expressed simply by juxtaposing a stream and a message, as follows.

stream message

Connection of two streams are denoted by the following syntax.

receiver = *stream*

This means that all messages sent to *stream* flow into *receiver*.

Closing a stream indicates that no more messages will be sent through it. Closing is always performed automatically, when a stream is discarded.

In addition, messages arriving at an object's interface stream are consumed exclusively by that object. This operation is also performed automatically.

5.2.2 Directed Streams

Stream connection is asymmetric; a stream may only be connected to another stream once, but many other streams may be connected to it. In order to assure at compile-time that streams are connected only once, references to a stream are classified into two types, called *directions*. An *inlet* is a reference to a stream from which messages flow; an *outlet* is another kind of reference in which messages are sent[1]. The single connection of a stream is assured by the restrictions requiring that a stream has only one inlet and that the right hand value of a *connect* expression be an inlet.

Inlets and outlets are distinguished syntactically. Variables referencing inlets are denoted with a variable name with ^ prepended to it, e.g. ^X. Slots holding inlets and

[1]They are named from an object's point of view.

```
class account.
    out balance.
:init ->        0 = !balance.
:deposit(^Amount) ->
        !balance + Amount = !balance.
:withdraw(^Amount, ^Ack) ->
        (Amount < !balance) ? (
        :'true ->
                !balance - Amount = !balance.
        :'false ->
                Ack :overdrawn(!balance).
        ).
:balance(!balance) -> .
end.
```

Figure 2: Bank account

outlets are written as slot names preceded by @ and by !, respectively. Expressions have a value whose direction is determined according to their kind. Messages are distinguished by the directions of their arguments as well as their number, and the message's name.

5.2.3 Channel Abstraction

Two types of stream confluence, namely mergers and appenders have special support in the language. As mentioned earlier, a merger performs non-deterministic merging, and an appender connects streams one after another in a specified order.

A channel is a tree formed of these confluences of streams. Variables represent a channel of a particular form, consisting of an appender and an arbitrary number of mergers. All outputs of the mergers are connected to inputs of the appender.

For a variable named `Foo`, `^Foo` is an inlet of the root stream of the channel. `Foo$1`, `Foo$2`, `Foo$3`, and so on, are leaf streams. `Foo` is equivalent to `Foo$1`. They are appended into the root in the order of their number. When there are many expressions having the same number, the streams they denote are merged before being appended.

Using channels reduces the description of mergers and appenders in programs, which would be indecipherable otherwise.

5.3 An Example Program

Figure 2 is an example A'UM program defining a class for a bank account.

Arguments in a message are connected with values of the expressions in the selector corresponding to the message. For example, `:deposit` receives an outlet and connects `^Amount` to it. `:balance` receives an inlet and connects it to the value of `!balance`.

A binary expression is a macro form. It expands into a *send* expression, which sends to the left hand value a message with two arguments, the right hand value and an inlet of a new stream. The name of the message is determined according to the operator. A macro form evaluates into an outlet of the new stream. Thus, `!balance + Amount` are expanded into `!balance :add(Amount, ^Result)`, with `Result` as its value.

exp `?` `( ... )` is an anonymous class definition, which is used to represent a conditional behavior. Either of the methods `:'true` or `:'false` is executed by the instance of the anonymous class, according to the result of `Amount < !balance`.

5.4 An outline of A'UM-90

A'UM-90 is an A'UM language system, independent of any CLL. It provides efficient stream communication on a distributed memory multiprocessor system. Moving stream data structures to their reader's processor saves many IPCs, which are otherwise required in stream communication.

A'UM-90 manages coarse-grained processes. Specifically, a process executes an instance of a user-defined class.

An A'UM-90 system consists of a compiler and an emulator. The compiler generates code for an abstract-machine designed for the system, and the emulator executes the code.

Two different types of platform have been used. One is a Sequent Symmetry with 16 processors, and the other is a number of Sun Sparc Stations communicating by Ethernet. Although a Symmetry has shared memory, we used it as a distributed memory machine. We used a small part of the memory to implement message communication, and divided the rest among processors.

6 Implementation of Streams in A'UM-90

The implementation described here fully utilizes information on stream abstraction and message flow direction available in A'UM programs. Although the delivery of messages is somewhat delayed, the number of IPCs required is significantly reduced, when many messages are sent through a long cascade of streams. Moreover, the delay is eliminated in many cases by various subtle optimization methods.

6.1 Streams

A stream is represented by a structure consisting of a message queue, a pointer to its receiver, and a reference count. The reference count is necessary for detecting

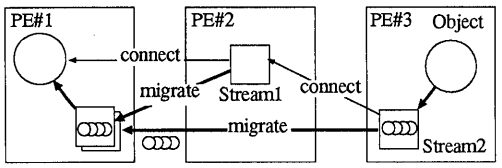


Figure 3: Stream location

closed streams and for implementing the appenders correctly. The structure is named *M node*, where M stands for merging. A merger is simply represented as an M node having more than one pointer referring to it. An appender is represented by a structure consisting of an M node and a pointer to the following stream. The structure is named *A node*.

With these structures, implementing operations on streams within a processor is straightforward. Sending a message is simply queuing it. Connecting a stream to a receiver is making the pointer in the stream point to the receiver and increasing the reference count of the receiver. When a stream is closed, its reference count is decreased. Receiving a message is just dequeuing it.

6.2 Location of Streams

As argued in a previous section, a stream should be placed on its receiver's processor in order to decrease the number of IPCs. However, when a stream is created, its receiver is still unknown. So we place it on the processor local to its creater at its creation, and let it migrate later to the receiver's processor(see Figure 3).

Since it is always an object that ultimately receives messages sent to a stream, the stream migrates to the object's processor. When the stream is directly connected to the object, it migrates immediately. If it is connected to an intermediate stream, it waits until the intermediate stream migrates.

Suppose that an address of a stream in a processor is announced to an object in another processor and that the stream has not yet migrated. If the object sends messages to the stream, two series of IPCs occur, one for sending them to the stream, and another for the migration process of the stream. We eliminate the former series by putting the messages into a new stream created on the same processor as the sending object and connecting the new stream to the original.

With this strategy, and assuming that objects do not migrate, all messages except those used for implementing the strategy are transferred between processors at most once. In the next section, a more detailed description of the stream migration is given.

6.3 Migration Procedure

In the following description, all streams are supposed to reside in different processors until they move. Operations within a processor are trivial, and are assumed to cost much less than ones involving IPCs. It is also supposed that streams are connected in a processor other than that of the receiving object. Otherwise, the migration procedure is so simple to become identical with an ordinary sending without migration.

1. A stream is placed on the same processor as its creator object.

2. When the stream is connected, a control message named *where* is sent to the specified receiver. The control message has a pointer to the stream and a tag showing the type of the stream, i.e., either an M node or an A node.

3. The *where* causes the following actions according to the type of the receiver:

   a stream before its migration handles the control message as if it is an ordinary message. That is, it is put into the receiver's queue. It will be transferred again when the receiver eventually migrates, and will be forwarded to another receiver, which should cause the following case.

   an object or a stream after its migration creates a new node of the type indicated by the tag in the control message, and reports the address of the new node by a control message named *here* to the stream waiting for the reply. When the type of the immigrant and the receiver is the same, the receiver creates no new node, and reports its own address.

4. When the stream receives the *here*, it migrates to the specified new residence, in one of the following manners according to its type:

   M node It sends all messages in its queue to the new residence. If it hasn't been closed yet, it leaves in the former residence a pointer forwarding to the new location. The original residence will be reclaimed when it is closed.

   A node In addition to the procedure for the M node, the stream to be appended to the migrating one is connected to the same receiver at the moment when this A node is closed. That is, a new *where* with a pointer to the stream is sent to the receiver.

6.4 Migration Cost

Each stream creates a *where*. It is transferred between processors twice, once when the stream is connected, and once when its receiver migrates. The second transfer doesn't happen if the receiver is an already moved stream or an object. Suppose a channel connected to an object consists of n streams, and of which n_d are connected directly to the object, then the number of IPCs for *where* is $n + (n - n_d)$.

A *here* is created in correspondence with a *where*, and is transferred between processors once. For all *here*'s , n IPCs occur.

Migration brings about no transfer of control messages, so the number of IPCs required for migration is $n + (n - n_d) + n = 3n - n_d$.

Closing a stream requires another kind of control message. We call it *close*. Each stream sends its reader one *close* when closed. This adds up to n *close*'s requiring n IPCs.

Ordinary messages are transferred between processors always once. If there are m ordinary messages to be sent, then, in total,

$$(3n - n_d) + m + n$$

transfers between processors occur.

How many IPCs occur for stream communication if streams don't move? Neither of *where* and *here* are created. A *close* is still created for a stream. The number of times ordinary messages and *close*'s are transferred depends on the structure of the channel.

A channel is a tree having streams as its nodes. Suppose the i-th node receives m_i messages, and its depth is d_i, where a depth of a node is number of streams in the path from the leaf to the root. For example, the depth of a leaf directly connected to an object is 2. Then messages sent to the i-th leaf is transferred $d_i - 1$ times, and the total number of transfers will be:

$$\sum_{i=1}^{n}(d_i - 1)(m_i + 1)$$

The condition when it requires less IPCs to implement stream communication with migrating streams than without them is:

$$\sum_{i=1}^{n}(d_i - 1)(m_i + 1) > (3n - n_d) + m + n$$

This can be rewritten as:

$$\sum_{i=1}^{n}(d_i - 2)(m_i + 1) > 3n - n_d$$

Since d_i can not be smaller than 2, $d_i - 2$ never becomes negative. The next term $m_i + 1$ is the number of messages sent from a node, including a *close*. The last term $3n - n_d$ is the number of control messages used to move all streams.

The above condition says that if the channel has some intermediate nodes between the root and leaves, and more than a certain number of messages are sent through them, then stream migration is beneficial. Conversely, if all streams in a channel are directly connected to an object, or too few messages are sent, streams should not be moved. The next section discusses some optimization based on detecting those cases.

6.5 Further Optimization

The left-hand side of the above condition becomes zero when all streams are directly connected to an object. When connecting a stream, it is detected at run-time that the receiver is an object; pointers are tagged to indicate the type of the pointed structure. By not moving those streams, the right-hand side is also decreased to zero when the left-hand becomes zero.

When less than two messages are sent through a stream, the stream does not migrate, i.e. it does not send out a *where*. More detailed analysis shows that *two* is the least number to make stream migration beneficial.

In addition, various minor optimization methods are applied to reduce the delay of the first message's delivery. For example, the first message is sent with a *where*, packed together in one IPC, if it is available when the *where* is sent out. When a *where* is received by a stream that only bridges two other streams, receiving no ordinary messages, it immediately forwards this *where* instead of sending out a new one. Such a stream can be distinguished by checking its reference count when it receives a *where*.

7 Evaluation

In order to evaluate performance of the implementation described in the sections so far, we measured the following three values:

- Delay time
- IPC load
- Total elapsed time for entire execution of a program

As a control, we measured against an A'UM-90 system which does not migrates streams. We call this system NO_WHERE, and the system that performs the migration WHERE in the following sections.

Programs used in the measurement of delay time and IPC load form a linear channel, a long chain of streams without any branches, and send along the channel. Figure 5 shows the objects' configuration. Each PE creates one stream on itself. When the PE receives a message *connect*, it connects its stream to the next lower stream

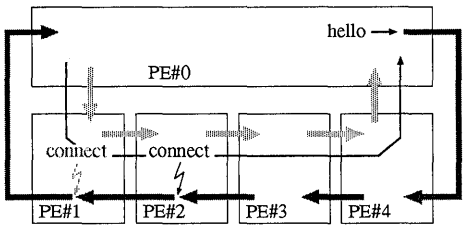


Figure 4: Objects' configuration

on another PE. Also, the first PE releases several messages named *hello* at its stream.

The *connect* circulates around the PEs, one at a time, through a channel different from that thorough which *hello*s flow. Two programs which differ in direction of the circulation were used. We call one of them DOWNSTREAM, in which a *connect* flows in the same direction as *hello*s, and the other UPSTREAM, in which a *connect* flows against *hello*s. The *connect* in Figure 5 is flowing UPSTREAM.

The time was measured from after the release of the *hello*s and a *connect* until the arrival of the last *hello*.

7.1 Delay time

Figure 6 shows the result of the delay time measurement, sending up to ten messages down a channel of length ten.

The values are elapsed time measured on an unloaded Sequent Symmetry, using 10 PEs. They includes CPU time and idle time during which PEs were waiting for messages.

In the DOWNSTREAM case, delay time in the WHERE is longer than in the NO_WHERE by at most 1000 msec, as expected. In the UPSTREAM case, however, messages arrive earlier in the WHERE than in the NO_WHERE by 200 msec. The reason for this reversal is that the migration of streams took place concurrently with the circulation of the *connect* in the WHERE. After the *connect* reached the uppermost PE, *hello*s were sent directly to their final receiver in the WHERE, while, in the NO_WHERE, they flowed through every PE having a part of the channel.

From these results, we can expect that the difference in the delay time of the WHERE and the NO_WHERE would be smaller than 1000 msec when the connections of a channel's constituent streams occur in a varying order.

Also, note that the delay time for the first message in the WHERE is much smaller than those for the later messages. This results from the optimization, mentioned at the section 6.5, of sending a *where* and the first message together whenever possible.

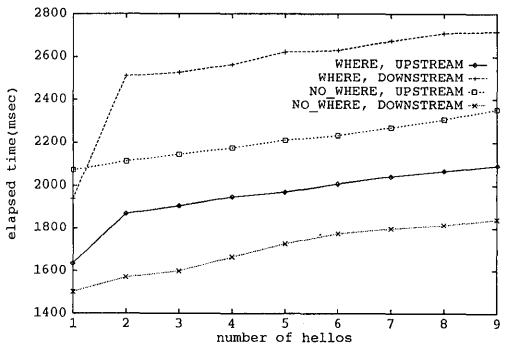


Figure 5: Delay time

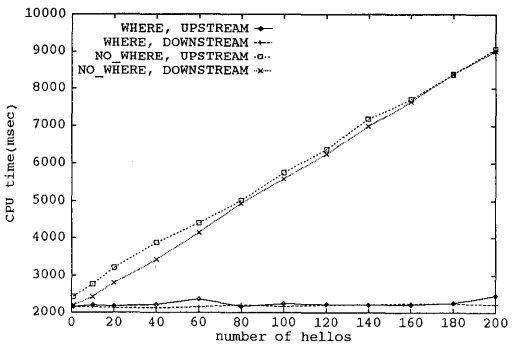


Figure 6: IPC load

7.2 IPC load

Figure 7 shows the result of the IPC load measurement, sending up to 200 messages down a channel of length 500. The values are CPU time measured on an unloaded Sequent Symmetry, using 10 PEs. The results confirm that the IPC load in the NO_WHERE eventually becomes much larger than that in the WHERE as the number of released messages grows.

7.3 Total elapsed time

Figure 8 and Figure 9 shows results of measurements using a program PRIME, which enumerates prime numbers by the generate-and-test method. The graphs in Figure 8 are obtained from 10 PEs in a Symmetry, and those in Figure 9 are from isolated Ethernet network consisting of two Sun Sparc Stations. The top two graphs in each figure are elapsed time, the next two are average total CPU time for a PE, and the other one is CPU time for a PE, spent only for processing other than IPC. The last one is estimated from CPU time for execution using 1 PE, divided by the number of PEs, i.e., 10.

The graphs for elapsed time shows that the WHERE

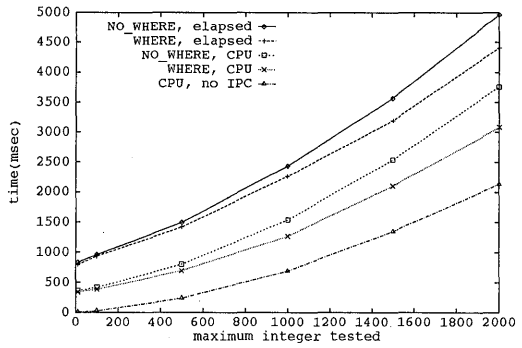


Figure 7: PRIME on shared memory

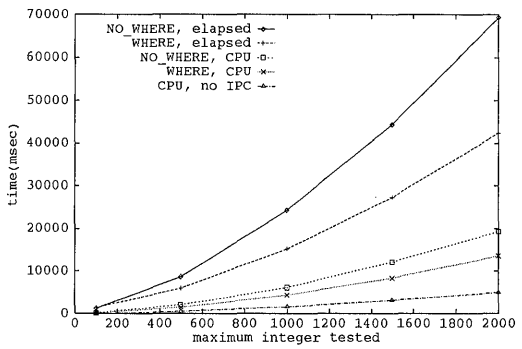


Figure 8: PRIME on Ethernet

is faster than the NO_WHERE. On a Symmetry, the entire speedup can be explained by decrease of CPU time. There is up to 40% improvement in CPU time spent for IPC, which can be read from the difference between total and non-IPC portion of CPU time. On Ethernet, the speedup is much larger than the decrease of CPU time, due to much slower communication.

8 Conclusion

Streams in CLLs are difficult to implement efficiently for two reasons:

1. Message buffers are not always placed on their readers' processor, because an arbitrary number of readers are allowed for a buffer. Therefore, interprocessor reading from the buffer takes place with two IPCs, instead of one required for writing into it.

2. A stream is represented by cascaded message buffers, which CLLs don't treat as a single body. Consequently, even if these buffers are placed on their reader's processor, their address has to be repeatedly sent to their writer.

This is not the case for A'UM. A'UM has abstract stream communication, whose implementation is left as the language systems' responsibility. In addition, every stream is restricted to have only one reader. So streams in A'UM can be more efficiently implemented than ones in CLLs.

An A'UM-90 moves a stream to its reader's processor, and saves about half of the IPCs required in CLLs. In spite of the migration, it deliver the first message through the stream with small delay. A prime number generator program runs up to 40 % faster in an A'UM-90 than in the system does not migrate streams.

While the optimization method given in this paper tries to reduce the number of IPCs for a given distribution of objects, it is also important to find the best distribution of objects. Of course, those methods have to balance the amount of IPCs and the parallelism exploitation.

Acknowledgments

We thank Shinji Yanagida and Toshio Tange of NEC Scientific Information System Development for developing the A'UM-90 abstract-machine emulator.

References

[Furukawa et al. 1984] K. Furukawa, A. Takeuchi, S. Kunifuji, H. Yasukawa, M. Ohki, K. Ueda, *Mandala: A Logic Based Knowledge Programming System*, Proc. FGCS'84, November 1984.

[Kahn et al. 1986] K. Kahn, E. D. Tribble, M. S. Miller, D. G. Bobrow, *Objects in Concurrent Logic Programming Languages*, Proc. OOPSLA'86, September, 1986.

[Kahn et al. 1989] K. Kahn, *Objects - a fresh look*, Proc. Third European Conf. on Object-Oriented Programming, Cambridge University Press, July 1989.

[Saraswat et al. 1990] V. A. Saraswat, K. Kahn, J. Levy, *Janus: A step towards distributed constraint programming*, North American Logic Programming Conference, October 1990.

[Shapiro and Takeuchi 1983] E. Shapiro, A. Takeuchi, *Object-oriented Programming in Concurrent Prolog*, New Generation Computing, 1, 1983.

[Ueda 1985] K. Ueda, *Guarded Horn Clauses*, Technical Report TR-103, ICOT, June 1985.

[Yoshida and Chikayama 1988] K. Yoshida, T. Chikayama, *A'UM: A Stream-Based Object-Oriented Language*, Proc. FGCS'88, November 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Message-Oriented Parallel Implementation of Moded Flat GHC

Kazunori Ueda

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
ueda@icot.or.jp

Masao Morita

Mitsubishi Research Institute
3-6, Otemachi 2-chome, Chiyoda-ku, Tokyo 100, Japan
morita@asdal.mri.co.jp

Abstract

We proposed in [Ueda and Morita 1990] a new, *message-oriented* implementation technique for Moded Flat GHC that compiled unification for data transfer into message passing. The technique was based on constraint-based program analysis, and significantly improved the performance of programs that used goals and streams to implement reconfigurable data structures. In this paper we discuss how the technique can be parallelized. We focus on a method for shared-memory multiprocessors, called the *shared-goal method*, though a different method could be used for distributed-memory multiprocessors. Unlike other parallel implementations of concurrent logic languages which we call *process-oriented*, the unit of parallel execution is not an individual goal but a chain of message sends caused successively by an initial message send. Parallelism comes from the existence of different chains of message sends that can be executed independently or in a pipelined manner. Mutual exclusion based on busy waiting and on message buffering controls access to individual, shared goals. Typical goals allow *last-send optimization*, the message-oriented counterpart of last-call optimization. We are building an experimental implementation on Sequent Symmetry. In spite of the simple scheduling currently adopted, preliminary evaluation shows good parallel speedup and good absolute performance for concurrent operations on binary process trees.

1. Introduction

Concurrent processes can be used both for programming computation and for programming storage. The latter aspect can be exploited in concurrent logic programming to program reconfigurable data structures using the following analogy,

records	$\longleftrightarrow$	(body) goals
pointers	$\longleftrightarrow$	streams (implemented by lists)

where a (concurrent) process is said to be *implemented* by a multiset of goals.

```
nt([],                  _, _, L,R) :- true |
    L=[], R=[].
nt([search(K,V)|Cs],K, V1,L,R) :- true |
    V=V1, nt(Cs,K,V1,L,R).
nt([search(K,V)|Cs],K1,V1,L,R) :- K<K1 |
    L=[search(K,V)|L1], nt(Cs,K1,V1,L1,R).
nt([search(K,V)|Cs],K1,V1,L,R) :- K>K1 |
    R=[search(K,V)|R1], nt(Cs,K1,V1,L,R1).
nt([update(K,V)|Cs],K, _, L,R) :- true |
    nt(Cs,K,V,L,R).
nt([update(K,V)|Cs],K1,V1,L,R) :- K<K1 |
    L=[update(K,V)|L1], nt(Cs,K1,V1,L1,R).
nt([update(K,V)|Cs],K1,V1,L,R) :- K>K1 |
    R=[update(K,V)|R1], nt(Cs,K1,V1,L,R1).

t([]              ) :- true | true.
t([search(_,V)|Cs]) :- true |
    V=undefined, t(Cs).
t([update(K,V)|Cs]) :- true |
    nt(Cs,K,V,L,R), t(L), t(R).
```

Program 1. A GHC program defining binary search trees as processes

An advantage of using processes for this purpose is that it allows implementations to exploit parallelism between operations on the storage. For instance, a search operation on a binary search tree (Program 1), given as a message in the interface stream, can enter the tree soon after the previous operation has passed the root of the tree. Programmers do not have to worry about mutual exclusion, which is taken care of by the implementation. This suggests that the programming of reconfigurable data structures can be an important application of concurrent logic languages. (The verbosity of Program 1 is a separate issue which is out of the scope of this paper.)

Processes as storage are almost always suspending, but should respond quickly when messages are sent. However, most implementations of concurrent logic languages have not been tuned for processes with this characteristic. In our earlier paper [Ueda and Morita 1990], we proposed *message-oriented* scheduling of goals for sequential implementation, which optimizes goals that suspend and resume

frequently. Although our primary goal was to optimize storage-intensive (or more generally, demand-driven) programs, the proposed technique worked quite well also for computation-intensive programs that did not use one-to-many communication. However, how to utilize the technique in parallel implementation was yet to be studied.

Parallelization of message-oriented scheduling can be quite different from parallelization of ordinary, *process-oriented* scheduling. An obvious way of parallelizing process-oriented scheduling is to execute different goals on different processors. In message-oriented scheduling, the basic idea should be to execute different message sends on different processors, but many problems must be solved as to the mapping of computation to processors, mutual exclusion, and so on. This paper reports the initial study on the subject.

The rest of the paper is organized as follows: Section 2 reviews Moded Flat GHC, the subset of GHC we are going to implement. Section 3 reviews message-oriented scheduling for sequential implementation. Section 4 discusses how to parallelize message-oriented scheduling. Of the two possible methods suggested, Section 5 focuses on the shared-goal method suitable for shared-memory multiprocessors and discusses design issues in more detail. Section 6 shows the result of preliminary performance evaluation. The readers are assumed to be familiar with concurrent logic languages [Shapiro 1989].

2. Moded Flat GHC and Constraint-Based Program Analysis

Moded Flat GHC [Ueda and Morita 1990] is a subset of GHC that introduces a *mode system* for the compile-time global analysis of dataflow caused by unification. Unification executed in clause bodies can cause bidirectional dataflow in general, but mode analysis tries to guarantee that it is assignment to an uninstantiated variable effectively and does not fail (except due to occur check).

Our experience with GHC and KL1 [Ueda and Chikayama 1990] has shown that the full functionality of bidirectional unification is seldom used and that programs using it can be rewritten rather easily (if not automatically) to programs using unification as assignment. These languages are indeed used as general-purpose concurrent languages, which means that it is very important to optimize basic operations such as unification and to obtain machine codes close to those obtained from procedural languages.

For global compile-time analysis to be practical, it is highly desirable that individual program modules can be analyzed separately in such a way that the results can be merged later. The mode system of Moded Flat GHC is thus constraint-based; the mode of a whole program can be determined by accumulating the mode constraints obtained separately from the syntactic analysis of each program clause. Another advantage of the constraint-based system is that it allows programmers to *declare* some of the mode constraints, in which case the analysis works as mode checking as well as mode inference.

The modularity of the analysis was brought by the rather strong assumption of the mode system: whether the function symbol at some position (possibly deep in a data structure) of a goal g is determined by g or by other goals running concurrently is determined solely by that position specified by a *path*, which is defined as follows. Let *Pred* be the set of predicate symbols and *Fun* the set of function symbols. For each $p \in Pred$ with the arity n_p, let N_p be the set $\{1, 2, \ldots, n_p\}$. N_f is defined similarly for each $f \in Fun$. Now the sets of *paths* P_t (for terms) and P_a (for atoms) are defined using disjoint union as:

$$P_t = (\sum_{f \in Fun} N_f)^*, \quad P_a = (\sum_{p \in Pred} N_p) \times P_t.$$

An element of P_a can be written as a string $\langle p, i \rangle \langle f_1, j_1 \rangle \ldots \langle f_n, j_n \rangle$, that is, it records the predicate and the function symbols on the way as well as the argument positions selected. A mode is a function from P_a to the set $\{in, out\}$, which means that it assigns either of *in* or *out* to every possible position of every possible instance of every possible goal. Whether some position is *in* or *out* can depend on the predicate and function symbols on the path down to that position. The function can be partial, because the mode values of many uninteresting positions that will not come to exist can be left undefined.

Mode analysis checks if every variable generated in the course of execution will have exactly one *out* occurrence (occurrence at an *out* position) that can determine its top-level value, by accumulating constraints between the mode values of different paths.

Constraint-based analysis can be applied to analyzing other properties of programs as well. For instance, if we can assume that streams and non-stream data structures do not occur at the same position of different goals, we can try to classify all the positions into

(1) those whose top-level values are limited to the list constructors (*cons* and *nil*) and
(2) those whose top-level values are limited to symbols other than the list constructors,

which is the simplest kind of type inference. Other applications include the static identification of 'single-reference' positions, namely positions whose values are not read by more than one goal and hence can be discarded or destructively updated after use. This could replace the MRB (multiple-reference bit) scheme [Chikayama and Kimura 1987], a runtime scheme

adopted in current KL1 implementations for the same purpose.

3. Message-Oriented (Sequential) Implementation

In a process-oriented sequential implementation of concurrent logic languages, goals ready for execution are put in a queue (or a stack or a deque, depending on the scheduling). Once a goal is taken from the queue, it is reduced as many times as possible, using last-call optimization, until it suspends or it is swapped out. A suspended goal is hooked on the uninstantiated variable(s) that caused suspension, and when one of the variables is instantiated, it is put back into the queue.

Message-oriented implementation has much in common with process-oriented implementation, but differs in the treatment of stream communication: It compiles the generation of stream elements into procedure calls to the consumer of the stream. A stream is an unbounded buffer of messages in principle, but message-oriented implementation tries to reduce the overhead of buffering and unbuffering by transferring control and messages simultaneously to the consumer whenever possible. To this end, it tries to schedule goals so that whenever the producer of a stream sends a message, the consumer is suspending on the stream and is ready to handle the message. Of course, this is not always possible because we can write a program in which a stream *must* act as a buffer; messages are buffered when the consumer is not ready to handle incoming messages.

Process-oriented implementation tries to achieve good performance by reducing the frequency of costly goal switching and taking advantage of last-call optimization. Message-oriented implementation tries to reduce the cost of each goal switching operation and the cost of data transfer between goals.

Suppose two goals, p and q, are connected by a stream s and p is going to send a message to q that is suspending on s. Message-oriented implementation represents s as a two-field *communication cell* that points to (1) the instruction in q's code from which the processing of q is to be resumed and (2) q's goal record containing its arguments (Fig. 1). (Throughout the paper, we assume that a suspended goal will resume its execution from the instruction following the one that caused suspension, not from the first instruction of the predicate.) To send a message m, p first loads m on a hardware register called the *communication register*, changes the current goal to the one pointed to by the communication cell of s, and calls the code pointed to by the communication cell of s. The goal q gets m from the communication register and may send other messages in its turn. Control returns to p when all the message sends caused directly or indirectly by m

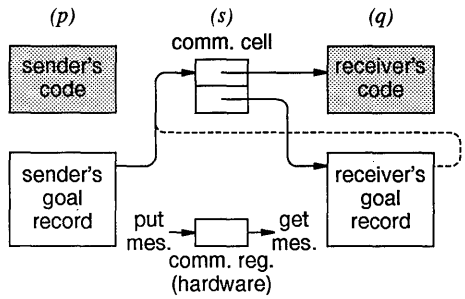


Fig. 1. Immediate message send

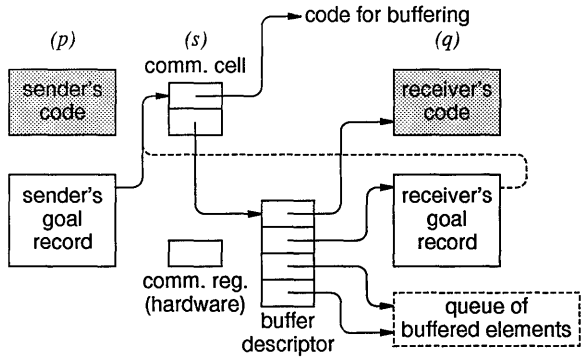


Fig. 2. Buffered message send

have been processed. However, if m is the last message which p can send out immediately (i.e., without waiting for further incoming messages), control need not return to p but can go directly to the goal that has outstanding message sends. This is called *last-send optimization*, which we shall see in Section 5.4 in more detail.

We have observed in GHC/KL1 programming that the dominant form of interprocess communication is one-to-one stream communication. It therefore deserves special treatment, even though other forms of communication such as broadcasting and multicasting become a little more expensive. One-to-many communication is done either by the repeated sending of messages or by using non-stream data structures.

Techniques mentioned in Section 2 are used to analyze which positions of a predicate and which variables in a program are used for streams and to distinguish between the sender and the receiver(s) of messages.

When a stream must buffer messages, the communication cell representing the stream points to the code for buffering and the descriptor of a buffer. The old entries of the communication cell are saved in the descriptor (Fig. 2). In general, a stream must buffer incoming messages when the receiver goal is not ready to handle them. The following are the possible reasons [Ueda and Morita 1990]:

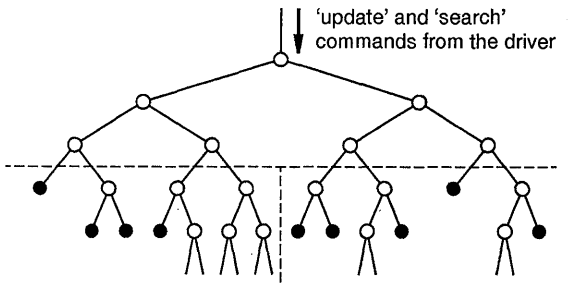


Fig. 3. Binary search tree as a process

(1) (selective message receiving) The receiver is waiting for a message from other input streams.
(2) The receiver is suspending on non-stream data (possibly the contents of messages).
(3) The sender of a message may run ahead of the receiver.
(4) When the receiver r belongs to a circular process structure, a message m sent by r may possibly arrive at r itself or may cause another message to be sent back to r. However, unless m has been sent by last-send optimization, r is not ready to receive it.

The receiver examines the buffer when the reason for the buffering disappears, and handles messages (if any) in it.

Process-oriented implementation often caches (part of) a goal record on hardware registers, but this should not be done in message-oriented implementation in which process switching takes place frequently.

4. Parallelization

How can we exploit parallelism from message-oriented implementation? Two quite different methods can be considered:

Distributed-goal method. Different processors take charge of different goals, and each processor handles messages sent to the goals it is taking charge of. Consider a binary search tree represented using goals and streams (Fig. 3) and suppose three processors take charge of the three different portions of the tree. Each processor performs message-oriented processing within its own portion, while message transfer between portions is compiled into inter-processor communication with buffering.

Shared-goal method. All processors share all the goals. There is a global, output-restricted deque [Knuth 1973] of outstanding work to be done in parallel, from which an idle processor gets a new job. The job is usually to execute a non-unification body goal or to send a message, the latter being the result of compiling a unification body goal involving streams. The message send will usually cause the reduction of a suspended goal. If the reduction generates another unification goal that has been compiled into a message send, it can be performed by the same processor. Thus a chain of message sends is formed, and different chains of message sends can be performed in parallel as long as they do not interfere with each other. In the binary tree example, different processors will take care of different operations sent to the root. A tree operation may cause subsequent message sends inside the tree, but they should be performed by the same processor because there is no parallelism within each tree operation.

Unlike the shared-goal method, the distributed-goal method can be applied to distributed-memory multiprocessors as well as shared-memory ones to improve the throughput of message handling. On shared-memory multiprocessors, however, the shared-goal method is more advantageous in terms of latency (i.e., responses to messages), because (1) it performs no inter-processor communication within a chain of message sends and (2) good load balancing can be attained easily. The shared-goal method requires a locking protocol for goals as will be discussed in Section 5.1, but it enables more tightly-coupled parallel processing that covers a wider range of applications. Because of its greater technical interest, the rest of the paper is focused on the shared-goal method.

5. Shared-Goal Implementation

In this section, we discuss important technicalities in implementing the shared-goal method. We explain the method and the intermediate code mainly by examples. Space limitations do not allow the full description of the implementation, though we had to solve a number of subtle problems related to concurrency control.

5.1 Locking of Goals

Consider a goal `p(Xs,Ys)` defined by the following single clause:

```
p([A|Xs1],Ys) :- true |
      Ys=[A|Ys1], p(Xs1,Ys1).
```

In the shared-goal method, different messages in the input stream `Xs` may be handled by different processors that share the goal `p(Xs,Ys)`. Any processor sending a message must therefore try to lock the goal record (placed in the shared memory) of the receiver first and obtain the grant of exclusive access to it. The receiver must remain locked until it sends a message through `Ys` and restores the dormant state.

The locking operation is important in the following respect as well: In message-oriented implementation, the order of the elements in a stream is not represented

spatially as a list structure but as the chronological order of message sends. The locking protocol must therefore make sure that when two messages, α and β, are sent in this order to `p(Xs,Ys)`, they are sent to the receiver of `Ys` in the same order. This is guaranteed by locking the receiver of `Ys` *before* `p(Xs,Ys)` is unlocked.

5.2 Busy Wait vs. Suspension

How should a processor trying to send a message wait until the receiver goal is unlocked? The two extreme possibilities are (1) to spin (busy-wait) until unlocked and (2) to give up (suspend) the sending immediately and do some other work, leaving a notice to the receiver that it has a message to receive. We must take the following observations into account here:

(a) The time each reduction takes, namely the time required for a resumed goal to restore the dormant state, is usually short (several tens of CISC instructions, say), though it can be considerably long sometimes.

(b) As explained in Section 5.1, a processor may lock more than one goal temporarily upon reduction. This means that busy wait may cause deadlock when goals and streams form a circular structure.

Because busy wait incurs much smaller overhead than suspension, Observation (a) suggests that the processor should spin for a period of time within which most goals can perform one reduction. However, it should suspend finally because of (b).

Upon suspension, a buffer is prepared as in Fig. 2, and the unsent message is put in it. Subsequent messages go to the buffer until the receiver has processed all the messages in the buffer and has removed the buffer. As is evident from Fig. 2, no overhead is incurred to check if the message is going to the buffer or to the receiver. The receiver could notice the existence of outstanding messages by checking its input streams upon each reduction, but it incurs overhead to (normal) programs which do not require buffering. So we have chosen to avoid this overhead by letting the *sender* spawn and schedule a special routine, called the *retransmitter* of the messages, when it creates a buffer. The retransmitter is executed asynchronously with the receiver. When executed, it tests if the receiver has been unlocked, in which case it sends the first message in the buffer and re-schedules itself.

For the shared resources other than goals (such as logic variables and the global deque), mutual exclusion should be attained by busy wait, because access to them takes a short period of time. On the other hand, synchronization on the values of non-stream variables (due to the semantics of GHC) should be implemented using suspension as usual.

5.3 Scheduling

Shared-goal implementation exploits parallelism between different chains of message sends that do not interfere with each other. For instance, a binary search tree (Fig. 3) can process different operations on it in a pipelined manner, as long as there is no dependence between the operations (e.g., the key of a search operation depending on the result of the previous search operation). When there is dependency, however, parallel execution can even lower the performance because of synchronization overhead.

Another example for which parallelism does not help is a demand-driven generator of prime numbers which is made up of cascaded goals for filtering out the multiples of prime numbers. The topmost goal receiving a new demand from outside filters out the multiples of the prime computed in response to the last demand. However, until the last demand has almost been processed, the topmost goal doesn't know what prime's multiples should be filtered out, and hence will be blocked.

These considerations suggest that in order to avoid ineffective parallelism, it is most realistic to let programmers specify which chains of message sends should be done in parallel with others. The simple method we are using currently is to have (1) a global deque for the work to be executed in parallel by idle processors and (2) one local stack for each processor for the work to be executed sequentially by the current processor. Each processor obtains a job from the global deque when its local stack is empty. We use a global deque rather than a global stack because, if the retransmitter of a buffer fails to send a message, it must go to the tail of the deque so it may not be retried soon.

Each job in a stack/deque is uniformly represented as a pair $\langle code, env \rangle$, where *code* is the job's entry/resumption point and *env* is its environment. The job is usually to start the execution of a goal or to resume the execution of a clause body. In these cases, *env* points to the goal record on which *code* should work. When the job is to retransmit buffered messages, *env* points to the communication cell pointing to the buffer.

When a clause body has several message sends to be executed in parallel, they will not put in the deque separately. Instead, the current processor executing the clause body performs the first send (and any sends caused by that send), putting the rest of the work to the deque after the first send succeeds in locking the receiver. Then an idle processor will get the rest of the work and perform the second message send (and any sends caused by that send), putting the rest of the rest back to the deque. This procedure is to guarantee the order of messages sent through a single stream by different processors. Suppose two messages, α and β, are sent by a goal like `Xs=[`α`,`β`|Xs1]`. Then we have to make sure that the processor trying to send β will

not lock the receiver of Xs before the processor trying to send α has done so.

5.4 Reduction

This section outlines what a typical goal should do during one reduction, where by 'typical' we mean goals that can be reduced by receiving one message. As an example, consider the distributor of messages defined as follows,

```
p([A|Xs],Ys,Zs) :- true |
     Ys=[A|Ys1], Zs=[A|Zs1], p(Xs,Ys1,Zs1).
```

where we assume A is known, by program analysis or declaration, to be a non-stream datum. (Otherwise a somewhat more complex procedure is necessary, because the three occurrences of A will be used for one-to-two communication.) The intermediate code for above program is:

```
entry(p/3)
  rcv_value(A1)
  get_cr(A4)
  send_call(A2)
  put_cr(A4)
  send_call(A3)  } or send_jmp(A3).
  execute        }
```

The Ai's are entries of the goal record of the goal being executed, which contain the arguments of the goal and temporary variables. Other programs may use Xi's, which are (possibly virtual) general registers local to each processor, and GAi's, which are the arguments of a new goal being created. The label entry(p/3) indicates the initial entry point of the predicate p with three arguments.

The instruction rcv_value(A1) waits for a message from the input stream on the first argument. If messages are already buffered, it takes the first one and puts it on the communication register. A retransmitter of the buffer is put on the deque if more messages exist; otherwise the buffer is made to disappear (Section 5.7). If no messages are buffered, which is expected to be most probable, rcv_value unlocks the goal record, and suspends until a message arrives. In either case, the instruction records the address of the next instruction in the communication cell (or, if the communication cell points to a buffer, in the buffer descriptor). *The goal is usually suspending at this instruction.*

The instruction get_cr(A4) saves into the goal record the message in the communication register, which the previous rcv_value(A1) has received. Then send_call(A2) sends the message in the communication register through the second stream. The instruction send_call(A2) tries to lock the receiver of the second stream and if successful, transfers control to the receiver. If the receiver is busy for a certain period of time or it isn't busy but is not ready to handle the message, the message is buffered. The instruction send_call does not unlock the current goal record. When control eventually returns, put_cr(A4) restores the communication register and send_call(A3) sends the next message.

When control returns again, execute performs the recursive call by going back to the entry point of the predicate p. Then the rcv_value(A1) instruction will either find no buffered messages or find some. In the former case, rcv_value(A1) obviously suspends. In the latter case, a retransmitter of the buffer must have been scheduled, and so rcv_value(A1) can suspend until the retransmitter sends a message. Moreover, the resumption address of the rcv_value(A1) instruction has been recorded by its previous execution. Thus in either case, execute effectively does nothing but unlocking the current goal. This is why last-send optimization can replace the last two instructions into a single instruction, send_jmp(A3).

The instruction send_jmp(A3) locks the receiver of the third stream, unlocks the current goal, and transfers control to the receiver without stacking the return address. Last-send optimization enables the current goal to receive the next message earlier and allows the pipelined processing of message sends. Note that with last-send optimization, the rcv_value(A1) instruction will be executed only once when the goal starts execution. The instructions executed for each incoming message are those from get_cr(A4) through send_jmp(A3).

The above instruction sequence performs the two message sends sequentially. However, a variant of send_call called send_fork stacks the return address on the global deque instead of the local stack, allowing the continuation to be processed in parallel. Note that send_fork leaves the continuation to another processor rather than the message send itself for the reason explained in Section 5.3.

We have established a code generation scheme for general cases including the spawning and the termination of goals (Section 5.5), explicit control of message buffering (Section 5.6), and suspension on non-stream variables. Several optimization techniques have been developed as well, for instance for goals whose input streams are known to carry messages of limited forms (e.g., non-root nodes of a binary search tree (Fig. 3)). Finally, we note that although process-oriented scheduling and message-oriented scheduling differ in the flow of control, they are quite compatible in the sense that an implementation can use both in running a single program. Our experimental implementation has actually been made by modifying a process-oriented implementation.

5.5 An Example

Here we give the intermediate code of a naïve reverse

```
The program:  (1) nreverse([H|T],O) :- true | append(O1,[H],O), nreverse(T,O1).
              (2) nreverse([],   O) :- true | O=[].
              (3) append([I|J],K,L) :- true | L=[I|M], append(J,K,M).
              (4) append([],   K,L) :- true | K=L.

entry(nreverse/2)
  rcv_value(A1)              receive a message from the 1st arg
                             (the program is usually waiting for incoming messages here)
  check_not_eos(101)         if the message is eos then collect the current comm. cell and goto 101
  get_cr(X3)                 save the message H in the comm. reg. to the register of the current PE
  commit                     Clause 1 is selected (no operation)
  put_cc(X4)                 create a comm. cell with a buffer
  push_value(X3)             put the message H into the buffer
  push_eos                   put eos into the buffer
  g_setup(append/3,3)        create a goal record for 3 args and record the name
  put_value(A2,GA3)          set the 3rd arg of append to O
  put_value(X4,GA2)          set the 2nd arg of append to [H]
  put_com_variable(A2,GA1)   create a locked variable O1 and set the 2nd arg of nreverse and the
                             1st arg of append to the pointer to O1,
                             assuming that append will turn O1 into a comm. cell soon
  g_call                     execute append until it suspends
  return                     unlock the current goal and do the job on the local stack top
label(101)
  commit                     Clause 2 is selected (no operation)
  send_call(A2)              send eos in the comm. reg. to the receiver of O
  proceed                    deallocate the goal record and return

entry(append/3)
  deref(A3)                  dereference the 3rd arg L
  rcv_value(A1)              receive a message from the 1st arg.
  check_not_eos(102)         if the message is eos then collect the current comm. cell and goto 102
  commit                     Clause 3 is selected (no operation)
  sendn_jmp(A3)              send the received message to the receiver of L, where
                             'n' means that the instruction assumes that L has been dereferenced
label(102)
  commit                     Clause 4 is selected (no operation)
  send_unify_jmp(A2,A3)      make sure that messages sent through K are
                             forwarded to the receiver of L, and return
```

Fig. 4. Intermediate code for naïve reverse

program (Fig. 4). In order for the code to be almost self-explanatory, some comments are appropriate here.

Suppose the messages $m_1, \ldots, m_n$ are sent to the goal `nreverse(In,Out)` through `In`, followed by the *eos* (end-of-stream) message indicating that the stream is closed. The `nreverse` goal generates one suspended `append` goal for each m_i, creating the structure in Fig. 5. The ith `append` has as its second argument a buffer with two messages, m_i and *eos*. The final *eos* message to `nreverse` causes the second clause to forward the *eos* to the most recent `append` goal holding m_n. The `append` holding m_n, in response, lets different (if available) processors send the two buffered messages m_n and *eos* to the `append` holding m_{n-1}. The message m_n is transferred all the way to the `append` holding m_1 and appears in `Out`. The following *eos* causes the next `append` goal to send m_{n-1} and another *eos*.

The performance of `nreverse` hinges on how fast each `append` goal can transfer messages. For each incoming message, an `append` goal checks if the message is not *eos* and then transfers both the message and control to the receiver of the output stream. The message remains on the communication register and need not be loaded or stored.

The `send_unify_jmp(`r_1`,`r_2`)` instruction is used for the unification of two streams. Arrangements are made so that next time a message is sent through r_1, the sender is made to point directly to the communication cell of r_2. If the stream r_1 has a buffer (which is the case with `nreverse`), the above redirection is made to happen after all the contents of the buffer are sent to the receiver of r_2.

It is worth noting that the multiway merging of streams can transfer messages as efficiently as `append`.

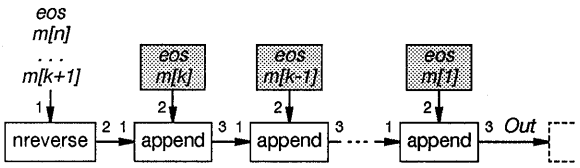


Fig. 5. Process structure being created by `nreverse([`$m_1, \ldots, m_n$`],Out)`

5.6 Buffering

As discussed in Section 5.2, the producer of a stream s creates a buffer when the receiver is locked for a long time. However, this is a rather unusual situation; a buffer is usually created by s's *receiver* when it remains unready to handle incoming messages after it has unlocked itself. Here we re-examine the four reasons of buffering in Section 3:

(1) *Selective message receiving.* This happens, for instance, in a program that merges two sorted streams of integers into a single sorted stream:

```
omerge([A|X1],[B|Y1],Z) :- A< B |
      Z=[A|Z1], omerge(X1,[B|Y1],Z1).
omerge([A|X1],[B|Y1],Z) :- A>=B |
      Z=[B|Z1], omerge([A|X1],Y1,Z1).
```

Two numbers, one from each input stream, are necessary for a reduction. Suppose the first number `A` arrives through the first stream. Then the goal `omerge` checks if the second stream has a buffered value. Since it doesn't, the goal cannot be reduced. So it records `A` in the goal record and changes the first stream to a buffer, because it has to wait for another number `B` to come through the second stream. Suppose `B`($>$ `A`) arrives and the first clause is selected. Then the second stream should become a buffer and `B` will be put back. The first stream, now being a buffer, is checked and a retransmitter is stacked if it contains an element; otherwise the buffer is made to disappear. Finally `A` is sent to the receiver of the third stream. The above procedure is admittedly complex, but this program is indeed one of the hardest ones to execute in a message-oriented manner. A simpler example of selective message receiving appears in the `append` program in Section 5.5; its second input stream buffers messages until the non-recursive clause is selected.

(2) *Suspension on non-stream data.* The most likely case is suspension on the content of a message (e.g., the first argument of an `update` message to a binary search tree). When a goal receives from a stream s a message that is not sufficiently instantiated for reduction, it changes s to a buffer and puts the message back to it. A retransmitter is hooked on the uninstantiated variable(s) that caused suspension, which will be invoked when any of them are instantiated.

(3) *The sender of a stream running ahead of the receiver.* It is not always possible to guarantee that the sender of a stream does not send a message before the receiver commences execution, though the scheduling policy tries to avoid such a situation. The simplest solution to this problem is to initialize each stream to an empty buffer. However, creating and collecting a buffer incurs certain overhead, while a buffer created for the above reason will receive no messages in most cases. So the current scheme defers the creation of a real buffer until a message is sent. Moreover, when the message is guaranteed to be received soon, the `put_com_variable` instruction (Fig. 4) is generated and lets the sender busy-wait until the receiver executes `rcv_value`.

(4) *Circular process structure.* When the receiver sends more than one message in response to an incoming message, sequential implementation must buffer subsequent incoming messages until the last message is sent out. In parallel implementation, the same effect is automatically achieved by the lock of the goal record, and hence the explicit control of buffering is not necessary.

The retransmission of a buffer created due to the reason (1) or (3) is explicitly controlled by the receiver. When a buffer is created due to the reason (2) or by the sender of a stream, a retransmitter of the buffer is scheduled asynchronously with the receiver.

5.7 Mutual Exclusion of Communication Cells

The two fields of a communication cell representing a stream may be updated both by the sender and the receiver of the stream. For instance, the sender may create a buffer and connect it to the cell when the receiver is locked for a certain period of time. The receiver may set or update the cell by the `rcv_value` instruction, may create or remove a buffer for the cell when buffering becomes necessary or unnecessary, may execute `send_unify_jmp` and connect the stream to another, and may move or delete the goal record of its own.

This of course calls for some method of mutual exclusion for communication cells. The simplest solution would be to lock a communication cell whenever updating or reading it, but locking both a goal record and a communication cell for each message send would be too costly. It is highly desirable that an ordinary message send, which reads but does not update a communication cell, need not lock the communication cell.

However, without locking upon reading, the following sequence can happen and inconsistency arises:

(1) the sender follows the pointer in the second field (the environment) of the communication cell,
(2) the receiver starts and completes the updating of the communication cell (under an appropriate locking protocol), and then

Table 1. Performance Evaluation (in seconds)

Language	Processing	binary process tree (5000 operations) (search)	(update)	naïve reverse (1000 elements)
GHC	1 PE (no locking)	1.25	1.83	2.23 (225 kRPS)*
	1 PE	1.38	2.10	3.27 (154 kRPS)
	2 PEs	0.78	1.15	2.43 (207 kRPS)
	3 PEs	0.55	0.81	1.71 (294 kRPS)
	4 PEs	0.44	0.63	1.33 (377 kRPS)
	5 PEs	0.36	0.53	1.10 (456 kRPS)
	6 PEs	0.33	0.46	0.96 (523 kRPS)
	7 PEs	0.33	0.39	0.85 (591 kRPS)
	8 PEs	0.33	0.36	0.77 (652 kRPS)
C (recursion)	`cc -O`	0.71	0.72	
C (iteration)	`cc -O`	0.32	0.35	

(* kilo Reductions Per Second)

(3) the sender locks the (wrong) record r (the goal record for the receiver or a buffer for the communication cell) obtained in Step (1) and calls the code pointed to by the first field (the code) of the updated communication cell.

This can be avoided by not letting the receiver update the second field of the communication cell. The receiver instead stores into the record r the pointer p to the right record. The receiver accordingly sets the first field of the communication cell to the pointer to a code sequence (to be called by the sender in Step (3)) that notifies the sender of the existence of the pointer p.

The sender can now access the right record pointed to by p via the wrong record r, but it is still desirable that p is finally written into the second field of the communication cell so that the right record can be accessed directly next time. This update of the communication cell must be done before the sender is unlocked and the control is completely transferred to the receiver.

For this purpose, we take advantage of the fact that the 1-byte lock of a record can take states other than 'locked' and 'unlocked'. When the lock of a record has one of these other states, a special routine corresponding to that state runs before the goal record of the sender is unlocked. This feature is being used for updating the second field of a communication cell safely.

6. An Experimental System and Its Performance

We have almost finished the initial version of the abstract machine instruction set for the shared-goal method. An experimental runtime system for performance evaluation has been developed on Sequent Symmetry, a shared-memory parallel computer with 20MHz 80386's. The system is written in an assembly language and C, and the abstract machine instructions are expanded into native codes automatically by a loader. A compiler from Moded Flat GHC to the intermediate code is yet to be developed.

The current system employs a simple scheme of parallel execution as described in Section 5.3. When the system runs with more than one processor, one of them acts as a master processor and the others as slaves. They act in the same manner while the global deque is non-empty. When the master fails to obtain a new job from the deque, it tries to detect termination and exceptions such as stack overflow. The current system does not care about perpetually suspended goals; they are treated just like garbage cells in Lisp. A slight overhead of counting the number of goals in the system will be necessary to detect perpetually suspended goals [Inamura and Onishi 1990] and/or to feature the *shoen* construct of KL1 [Ueda and Chikayama 1990], but it should scarcely affect the result of performance evaluation described below.

Locking of shared resources, namely logic variables, goal records, communication cells, the global deque, etc., is done using the xchg (exchange) instruction as usual.

Using Program 1, we measured (1) the processing time of 5000 `update` operations with random keys given to an empty binary tree and (2) the processing time of 5000 `search` operations (with the same sequence of keys) to the resulting tree with 4777 nodes. The number of processors was changed from 1 to 8. For the one-processor case, a version without locking/unlocking operations was tested as well. The numbers include the execution time of the driver that sends messages to the tree. The result was compared with two versions of (sequential) C programs using records and pointers, one using recursion and the other using iteration. The performance of `nreverse` (Fig. 4) was measured as well. The results are shown in Table 1.

The results show good (if not ideal) parallel speedup, though for `search` operations on a binary tree, the performance is finally bounded by the sequen-

tial nature of the driver and the root node. Access contention on the global deque can be another cause of overhead. Note, however, that the two examples are indeed harder to execute in parallel than running independent processes in parallel, because different chains of message sends share goals. Note also that the binary tree with 4777 nodes is not very deep.

The binary tree program run with 4 processors outperformed the optimized recursive C program. The iterative C program was more than twice as fast as the recursive one and was comparable to the GHC program run with 8 processors. The comparison, however, would have been more preferable to parallel GHC if a larger tree had been used.

The overhead of locking/unlocking was about 30% in `nreverse` and about 10% in the binary tree program. Since `nreverse` is one of the fastest programs in terms of the kRPS value, we can conclude that the overhead of locking/unlocking is reasonably small on average even if we lock such small entities as individual goals.

As for space efficiency, the essential difference between our implementation and C implementations is that GHC goal records have pointers to input streams while C records do not consume memory by being pointed to. The difference comes from the expressive power of streams; unlike pointers, streams can be unified together and can buffer messages implicitly.

One may suspect that message-oriented implementation suffers from poor locality in general. This is true for *data* locality, because a single message chain can visit many goals. However, streams in process-oriented implementation cannot enjoy very good locality either, because a tail-recursive goal can generate a long list of messages. Both process-oriented and message-oriented implementations enjoy good *instruction* locality for the binary tree program and `nreverse`.

Comparison of performance between a message-oriented implementation and a process-oriented implementation was reported in [Ueda and Morita 1990] for the one-processor case.

7. Conclusions and Future Works

The main contribution of this paper is that message-oriented implementation of Moded Flat GHC was shown to benefit from small-grain, tightly-coupled parallelism on shared-memory multiprocessors. Furthermore, the result of preliminary evaluation shows that the absolute performance is good enough to be compared with procedural programs.

These results suggest that the programming of reconfigurable storage structures that allow concurrent access can be a realistic application of Moded Flat GHC. Programmers need not worry about mutual exclusion necessitated by parallelization, because it is achieved automatically at the implementation level. In procedural languages, parallelization may well require major rewriting of programs. To our knowledge, how to deal with reconfigurable storage structures efficiently in non-procedural languages without side effects has not been studied in depth.

We have not yet fully studied language constructs and their implementation for more minute control over parallel execution. The current scheme for the control of parallelism is a simple extension to the sequential system; it worked well for the benchmark programs used, but will not be powerful enough to be able to tune the performance of large programs. We need a notion of priority that should be somewhat different from the priority construct in KL1 designed for process-oriented parallel execution. The notion of fairness may have to be reconsidered also. KL1 provides the *shoen* (manor) construct as well, which is the unit of execution control, exception handling and resource consumption control. How to adapt the *shoen* construct to message-oriented implementation is another research topic.

Acknowledgments

The authors are indebted to the anonymous referees for helpful comments.

References

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura, Multiple Reference Management in Flat GHC. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 276–293.

[Inamura and Onishi 1990] Y. Inamura and S. Onishi, A Detection Algorithm of Perpetual Suspension in KL1. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 18–30.

[Knuth 1973] D. E. Knuth, *The Art of Computer Programming, Vol. 1 (2nd ed.)*. Addison-Wesley, Reading, MA, 1973.

[Shapiro 1989] Shapiro, E., The Family of Concurrent Logic Programming Languages. *Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.

[Ueda and Morita 1990] K. Ueda and M. Morita, A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 3–17. A revised, extended version to appear in *New Generation Computing*.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (Dec., 1990), pp. 494–500.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Towards an Efficient Compile-Time Granularity Analysis Algorithm

X. Zhong, E. Tick, S. Duvvuru,
L. Hansen, A. V. S. Sastry and R. Sundararajan
Dept. of Computer Science
University of Oregon
Eugene, OR 97403

Abstract

We present a new granularity analysis scheme for concurrent logic programs. The main idea is that, instead of trying to estimate costs of goals precisely, we provide a compile-time analysis method which can efficiently and precisely estimate *relative* costs of active goals given the cost of a goal at runtime. This is achieved by estimating the cost relationship between an active goal and its subgoals at compile time, based on the call graph of the program. *Iteration parameters* are introduced to handle recursive procedures. We show that the method accurately estimates cost, for some simple benchmark programs. Compared with methods in the literature, our scheme has several advantages: it is applicable to any program, it gives a more precise cost estimation than static methods, and it has lighter runtime overheads than absolute estimation methods.

1 Introduction

The importance of grain sizes of tasks in a parallel computation has been well recognized [6, 5, 7]. In practice, the overhead to execute small grain tasks in parallel may well offset the speedup gained. Therefore, it is important to estimate the costs of the execution of tasks so that at runtime, tasks can be scheduled to execute sequentially or in parallel to achieve the maximal speedup.

Granularity analysis can be done at compile time or runtime or even both [7]. The compile-time approach estimates costs by statically analyzing program structure. The program is partitioned statically and the partitioning scheme is independent of runtime parameters. Costs of most tasks, however, are not known until parameters are instantiated at runtime and therefore, the compile-time approach may result in inaccurate estimates. The runtime approach, on the other hand, delays the cost estimation until execution and can therefore make more accurate estimates. However, the overhead to estimate costs is usually too large to achieve efficient speedup, and therefore the approach is usually infeasible. The most promising approach is to try to get as much cost estimation information as possible at compile time and make the overhead of runtime scheduling very slight. Such approach has been taken by Tick [10], Debray *et al.* [2], and King and Soper [4]. In this paper, we adopt this strategy.

A method for the granularity analysis of concurrent logic programs is proposed. Although the method can be well applied to other languages, such as functional languages, in this paper, we discuss the method only in the context of concurrent logic programs. The key observation behind this method is that task spawning in many concurrent logic program language implementations, such as Flat Guarded Horn Clauses (FGHC) [13], depends only on the *relative* costs of tasks. If the compile-time analysis can provide simple and precise cost relationships between an active goal and its subgoals, then the runtime scheduler can efficiently estimate the costs of the subgoals based on the cost of the active goal. The method achieves this by estimating, at compile time, the cost relationship based on the call graph and the introduction of iteration parameters. We show that for common benchmark programs, the method gives correct estimates.

2 Motivations

Compile-time granularity analysis is difficult because most of the information needed, such as size of a data structure and number of loop iterations, are not known until runtime. Sarkar [7] used a profiling method to get the frequency of recursive and nonrecursive function calls for a functional language. His method is simple and does not have runtime overheads, but can give only a rough estimate of the actual granularity.

In the logic programming community, Tick [10] first proposed a method to estimate weights of procedures by analyzing the call graph of a program. The method, as refined by Debray [1], derives the call graph of the program, and then combines procedures which are mutually recursive with each other into a single cluster (i.e., a strongly connected component in the call graph). Thus the call graph is converted into an acyclic graph. Procedures in a cluster are assigned the same weight

which is the sum of the weights of the cluster's children (the weights of leaf nodes are one, by definition). This method has very low runtime overhead; however, goal weights are estimated statically and thus cannot capture the dynamic change of weights at runtime. This problem is especially severe for recursive (or mutually recursive) procedures.

As an example of the method, consider the naive-reverse procedure in Figure 1. (The clauses in the `nrev/2` program do not have guards, i.e., only head unification is responsible for commit.) Examining the call graph, we find that the algorithm assigns a weight of one to `append/3` (it is a leaf), and a weight of two to `nrev/2` (one plus the weight of its child). Such weights are associated with *every* procedure invocation and thus cannot accurately reflect execute time.

Debray *et al.* [2] presented a compile-time method to derive costs of predicates. The cost of a predicate is assumed to depend solely on its input argument sizes. Relationships between input and output argument sizes in predicates are first derived based on so-called data dependency graphs and then recurrence equations of cost functions of predicates are set up. These equations are then solved at compile time to derive closed forms (functions) for the cost of predicates and their input argument sizes, together with the closed forms (functions) between the output and input argument sizes. Such cost and argument size functions can be evaluated at runtime to estimate costs of goals. A similar approach was also proposed by King and Soper [4]. Such approaches represent a trend toward precise estimation. For `nrev/2`, Debray's method gives $\text{Cost}_{nrev}(n) = 0.5n^2 + 1.5n + 1$, where n is the size of the input argument. This function can then be inserted into the runtime scheduler. Whenever `nrev/2` is invoked, the cost function is evaluated, which obviously requires the value n, the size of its first argument. If the cost is bigger than some preselected overhead threshold, the goal is executed in parallel; otherwise, it is executed sequentially.

The method described suffers from several drawbacks (see [11] for further discussion). First, there may be considerable runtime overhead to keep track of argument sizes, which are essential for the cost estimation at runtime. Furthermore, the sizes of the initial input arguments have to be given by users or estimated by the program when the program begins to execute. Second, within the umbrella of argument sizes, different metrics may be used, e.g., list length, term depth, and the value of an integer argument. It is unclear (from [2, 4]) how to correctly choose metrics which are relevant for a given predicate. Third, the resultant recurrence equations for size relationships and cost relationships can be fairly complicated.

It is therefore worth remedying the drawbacks of the above two approaches. It is also clear that there is a tradeoff between precise estimation and runtime overhead. In fact, Tick's approach and Debray's approach represent two extremes in the granularity estimation spectrum. Our intention here is to design a middle-of-the-spectrum method: fairly accurate estimation, applicable to any procedures, without incurring too much runtime overhead.

3 Overview of the Approach

We argue here, as in our earlier work, that it is sufficient to estimate only *relative* costs of goals. This is especially true for an on-demand runtime scheduler [8]. Therefore, it is important to capture the cost *changes* of a subgoal and a goal, but not necessarily the "absolute" granularity. Obviously the costs of subgoals of a parent goal are always less than the cost of the parent goal, and the sum of costs of the subgoals (plus some constant overhead) is equal to the cost of the parent goal. The challenging problem here is how to distribute the cost of the parent goal to its subgoals properly, especially for a recursive call. For instance, consider the naive reverse procedure `nrev/2` again. Suppose goal `nrev([1,2,3,4],R)` is invoked (i.e., clause two is invoked) and the cost of this query is given, what are the costs of `nrev([2,3,4],R1)` and `append(R1,[1],R)`?

It is clear that the correct cost distribution depends on the runtime state of the program. For example, the percentage of cost distributed to `nrev([1,2,3,4],R)` (i.e., as one of the subgoals of `nrev([1,2,3,4,5],T)` will be different from that of cost distributed to `nrev([1,2],R)`. To capture the runtime state, we introduce an *iteration parameter* to model the runtime state, and we associate an iteration parameter with every active goal. Since the cost of a goal depends solely on its entry runtime state, its cost is a function of its iteration parameter. Several intuitive heuristics are used to capture the relations between the iteration parameter of a parent goal and those of its children goals. To have a simple and efficient algorithm, only the AND/OR call graph of the program, which is slightly different from the standard call graph, is considered to obtain these iteration relationships. Such relations are then used in the derivation of recurrence equations of cost functions of an active goal and its subgoals. The recurrence equations are derived simply based on the above observation, i.e., the cost of an active goal is equal to the summation of the costs of its subgoals.

We then proceed to solve these recurrence equations for cost functions bottom up, first for the leaf nodes of the modified AND/OR call graph, which can be obtained in a similar way in Tick's modified algorithm by clustering those mutually recursive nodes together in the AND/OR call graph of the program (see Section 2). After we obtain all the cost functions, cost distribution functions are derived as follows. Suppose the cost of an

active goal is given, we first solve for its iteration parameter based on the cost function derived. Once the iteration parameter is solved, costs of its subgoals, which are functions of their iteration parameters, can be derived based on the assumption that these iteration parameters have relationships with the iteration parameter of their parent, which are given by the heuristics. This gives the cost distribution functions desired for the subgoals.

To recap, our compile-time granularity analysis procedure consists of the following steps:

1. Form the call graph of the program and cluster mutually recursive nodes of the modified AND/OR call graph.
2. Associate each procedure (node) in the call graph with an iteration parameter and use heuristics to derive the iteration parameter relations.
3. Form recurrence equations for the cost functions of goals and subgoals.
4. Proceed bottom up in the modified AND-OR call graph to derive cost functions.
5. Solve for iteration parameters and then derive cost distribution functions for each predicate.

4 Deriving Cost Relationships

4.1 Cost Functions and Recurrence Equations

To derive the cost relationships for a program, we use a graph G (called an AND/OR call graph) to capture the program structure. Formally, G is a triple (N, E, A), where N is a set of procedures denoted as $\{p_1, p_2, \ldots, p_n\}$ and E is a set of pair nodes such that $(p_1, p_2) \in E$ if and only if p_2 appears as one of the subgoals in one of the clauses of p_1. Notice that there might be multiple edges (p_1, p_2) because p_1 might call p_2 in multiple clauses. A is a partition of the multiple-edge set E such that (p_1, p_2) and (p_1, p_3) are in one element of A if and only if p_2 and p_3 are in the body of the same clause whose head is p_1. Intuitively, A denotes what subgoals are AND processes. After applying A to edges leaving out a node, edges are partitioned into clusters which correspond to clauses and these clauses are themselves OR processes. Figure 2 shows an example, where the OR branches are labeled with a bar, and AND branches are unmarked. Leaf facts (terminal clauses) are denoted as empty nodes.

As in [1], we modify G so that we can cluster all those recursive and mutually recursive procedures together and form a directed acyclic graph (DAG). This is achieved by traversing G and finding all strongly-connected components. In this traversing, the difference between AND and OR nodes is immaterial, and we simply discard the partition A. A procedure is recursive if and only if the procedure is in a strongly-connected component. After nodes are clustered in a strongly-connected component in G, we form a DAG G', whose nodes are those strongly-connected components of G and edges are simply the collections of the edges in G. This step can be accomplished by an efficient algorithm proposed by Tarjan [9].

The cost of an active goal p is determined by two factors: its entry runtime state s during the program execution and the structure of the program. We use an integer n, called the *iteration parameter*, to approximately represent state s. Intuitively, n can be viewed as an encoding of a program runtime state. Formally, let $\mathcal{S}$ be the set of program runtime states, M be a mapping from $\mathcal{S}$ to the set of natural numbers N such that $M(s) = n$ for $s \in \mathcal{S}$. It is easy to see that the cost of p is a function of its iteration parameter n. It is also clear that the iteration parameter of a subgoal of p is a function of n. Hereafter, suppose p_{ij} is the j^{th} subgoal in the i^{th} clause of p. We use $I_{ij}(n)$ to represent the iteration parameter of p_{ij}. The problem of how to determine function I_{ij} will be discussed in Section 4.2.

To model the structure of the program, we use the AND/OR call graph G as an approximation. In other words, we ignore the attributes of the data, such as size and dependencies. We first derive recurrence equations of cost functions between a procedure p and its subgoals by looking at G. Let $\text{Cost}_p(n)$ denote the cost of p. Three cases arise in this derivation:

> **Case 1:** p is a leaf node of G' which is non-recursive. This includes cases where that p is a built-in predicate. In this case, we simply assign a constant c as $\text{Cost}_p(n)$. c is the cost to execute p. For instance such cost can be chosen as the number of machine instructions in p.

For the next two cases, we consider non-leaf nodes p, with the following clauses (OR processes),

$$\begin{aligned} C_1 &: p \;\; :- \;\; p_{11}, \ldots, p_{1n_1}. \\ C_2 &: p \;\; :- \;\; p_{21}, \ldots, p_{2n_2}. \\ &\cdots \\ C_k &: p \;\; :- \;\; p_{k1}, \ldots, p_{kn_k}. \end{aligned}$$

Let the cost of each clause be $\text{Cost}_{C_j}(n)$ for $1 \leq j \leq k$. We now distinguish whether or not p is recursive.

> **Case 2:** p is not recursive and not mutually recursive with any other procedures. We can easily see that

$$\text{Cost}_p(n) \leq \sum_{j=1}^{k} \text{Cost}_{C_j}(n). \quad (1)$$

Conservatively, we approximate $\text{Cost}_p(n)$ as the right-hand side of the above inequality. Notice that in a committed-choice language, the summation in the above inequality can be changed to the maximum (i.e., max) function. However this increases the difficulty of the algebraic manipulation of the resultant recurrence equations (see [11] for example) and we prefer to use the summation as an approximation.

Case 3: p is recursive or mutually recursive. In this case, we must be careful in the approximation, since minor changes in the recurrence equations can give rise to very different estimation. This can be seen for `split` in `qsort` example in Section 2.

To be more precise, we first observe that some clauses are the "boundary clauses," that is, they serve as the termination of the recursion. The other clauses, whose bodies have some goals which are mutually recursive with p, are the only clauses which will be effective for the recursion. Without loss of generality, we assume for $j > u$, C_j are all those "mutually recursive" clauses. For a nonzero iteration parameter n (i.e., $n > 0$), we take the average costs of these clauses as an approximation:

$$\text{Cost}_p(n) = \frac{1}{k-u} \sum_{j=u+1}^{k} \text{Cost}_{C_j}(n) \quad (2)$$

and for $n = 0$, we take the sum of the costs of those "boundary clauses" as the boundary condition of $\text{Cost}_p(n)$:

$$\text{Cost}_p(0) = \sum_{j=1}^{u} \text{Cost}_{C_j}(0).$$

The above estimation only gives the relations between cost of p and those of its clauses. The cost of clause C_j can be estimated as

$$\text{Cost}_{C_j}(n) = \text{CHead}_j + \sum_{m=1}^{n_j} \text{Cost}_{p_{jm}}(I_{jm}(n)) \quad (3)$$

where CHead_j is a constant denoting the cost for head unification of clause C_j and $I_{jm}(n)$ is the iteration parameter for the m^{th} body goal. Substituting Equation 3 back into Equation 1 or 2 gives us the recurrence equations for cost functions of predicates.

4.2 Iteration Parameters

There are several intuitions behind the introduction of the iteration parameter. As we mentioned above, iteration parameter n represents an encoding of a program runtime state as a positive integer. In fact, this type of encoding has been used extensively in program verification, e.g., [3], especially in the proof of loop termination. A loop $\mathcal{L}$ terminates if and only it is possible to choose a function M which always maps the runtime states of $\mathcal{L}$ to nonnegative integers such that M monotonically decreases for each iteration of $\mathcal{L}$. Such encoding also makes it possible to solve the problem that once the cost of an active goal is given, its iteration parameter can be obtained. This parameter can be used to derive costs of its subgoals (provided the iteration-parameter functions I_m are given), which in turn give the cost distribution functions.

Admittedly, the encoding of program states may be fairly complicated. Hence, to precisely determine the iteration-parameter functions for subgoals will be complicated too. In fact, this problem is statically undecidable since this is as complicated as to precisely determine the program runtime behavior at compile time. Fortunately, in practice, most programs exhibit regular control structures that can be captured by some intuitive heuristics.

To determine the iteration-parameter functions, we first observe that there is a simple conservative rule: for a recursive body goal p, when it recursively calls itself back again, the iteration parameter must have been decreased by one (if the recursion terminates). This is similar to the loop termination argument. Therefore, as an approximation, we can use $I_m(n) = n - 1$ as a *conservative estimation* for a subgoal p_{im} which happens to be p (self-recursive). Other heuristics are listed as follows:

§1. For a body goal p_{im} whose predicate only occurs in the body once and it is not mutually recursive with p (i.e., not in a strongly-connected component of p), $I_{im}(n) = n$.

§2. If p_{im} is mutually recursive with p and its predicate only occurs once in the body, $I_{im}(n) = n - 1$.

§3. If p_{im} is mutually recursive with p and its predicate occurs l times in the body, where $l > 1$, $I_{im}(n) = n/l$ (this is integer division, i.e., the floor function).

The intuitions behind these heuristics are simple. Heuristic §1 represents the case where a goal does not invoke its parent. In almost all programs, this goal will process information supplied by the parent, thus the it-

eration parameter remains unmodified. Heuristic §2 is based on the previous conservative principle. Heuristic §3 is based on the intuition that the iteration is divided evenly for multiple callees. Notice for the situation in heuristic §3, we can also use our conservative principle. However, we avoid use of the conservative principle, if possible, because the resultant estimation of $\text{Cost}_p(n)$ may be an exponential function of n, which, for most practical programs, is not correct.

These heuristics have been derived from experimentation with a number of programs, placing a premium on the *simplicity* of $I(n)$. A partial summary of these results is given in Section 6. A remaining goal of future research is to further justify these heuristics with larger programs, and derive alternatives.

4.3 An Example: Quicksort

After we have determined the iteration-parameter functions, we have a system of recurrence equations for cost functions. These system of recurrence equations can be solved in a bottom-up manner in the modified graph G'. The problem of systematically solving these recurrence equations in general is discussed in [11]. Here, we consider a complete example for the `qsort/2` program given in Figure 2.

The boundary condition for $\text{Cost}_{qsort}(n)$ is that $\text{Cost}_{qsort}(0)$ is equal to the constant execution cost d_1 of `qsort/2` clause one. The following recurrence equations are derived:

$$\begin{aligned} \text{Cost}_{qsort}(0) &= d_1 \\ \text{Cost}_{qsort}(n) &= \text{Cost}_{C_2} \end{aligned}$$

With Heuristic §3, we have

$$\text{Cost}_{C_2} = d_2 + \text{Cost}_{split}(n) + 2\text{Cost}_{qsort}(n/2)$$

where d_2 is the constant cost for the head unification of the second clause of `qsort/2`.

Similarly, the recurrence equations for $\text{Cost}_{split}(n)$ are

$$\begin{aligned} \text{Cost}_{split}(0) &= d_3 \\ \text{Cost}_{split}(n) &= (\text{Cost}_{C_2} + \text{Cost}_{C_3})/2 \end{aligned}$$

Furthermore,

$$\begin{aligned} \text{Cost}_{C_2} &= \text{Cost}_{C_3} \\ &= d_4 + \text{Cost}_{split}(n-1) \end{aligned}$$

where d_4 is the constant cost for the head unification of the second (and the third) clause of `split`. We first solve the recurrence equations for `split`, which is in the lower level in G' and and then solve the recurrence equations for `qsort`. This gives us $\text{Cost}_{split}(n) = d_3 + d_4 n$ which can be approximated as $d_4 n$ and $\text{Cost}_{qsort}(n) = d_1 + d_2 \log n + d_4 n \log n$, which is the well known average complexity of `qsort`.

Finally, it should be noted that it is necessary to distinguish between the recursive and nonrecursive clauses here and take the average of the recursive clause costs as an approximation. If we simply take the summation of all clause costs together as the approximation of the cost function, both cost functions for `split` and `qsort` would be exponential, which are not correct. More precisely, if the summation of all costs of clauses of `split` is taken as $\text{Cost}_{split}(n)$, we will have

$$\text{Cost}_{split}(n) = d_3 + 2(d_4 + \text{Cost}_{split}(n-1))$$

The solution of $\text{Cost}_{split}(n)$ is an exponential function, which is not correct.

5 Distributing Costs

So far, we have derived cost functions of the iteration parameter for each procedure. However, to know the cost of a procedure, we need to first know the value of its iteration parameter. This, as pointed out in our introduction, may require too much overhead. We notice that, in most scheduling policies (such as on-demand scheduling), only *relative* costs are needed. This can be relatively easily achieved in our theory since cost functions only have a single parameter (iteration parameter).

To derive cost distributing formulae for a given procedure and its body goals, the first step is to solve for the iteration parameter n in Equation 3 assuming that $\text{Cost}_p(n)$ is given at runtime as C_p. Assuming that clause i is invoked in runtime, we approximate $\text{Cost}_{C_i}(n)$ as C_p and solve Equation 3 for n. Let $n = F(C_p)$ be the symbolic solution, which depends on the runtime value of $\text{Cost}_p(n)$ (i.e., C_p), we can easily derive costs of its subgoals of clause i as we can simply substitute n with $F(C_p)$ in $\text{Cost}_{p_{im}}(I_{im}(n))$, which gives rise to the cost distributing functions we need to derive at compile time.

Let's reconsider the `nrev/2` procedure. The cost equations are derived as follows:

$$\begin{aligned} \text{Cost}_{nrev}(n) &= \text{Cost}_{nrev}(n-1) + \text{Cost}_{append}(n) \\ \text{Cost}_{nrev}(0) &= c_1 \\ \text{Cost}_{append}(n) &= \text{Cost}_{append}(n-1) + C_a \\ \text{Cost}_{append}(0) &= c_2 \end{aligned}$$

We can easily derive the closed forms for these two cost functions as $\text{Cost}_{append}(n) = n \times C_a + c_2$ which can be approximated as $C_a \times n$, and $Cost_{nrev}(n) \doteq C_a \times n^2/2$. Now, given the $\text{Cost}_{nrev}(n)$ as C_r, we solve for n and have $n = \sqrt{\frac{2C_r}{C_a}}$. Hence, we have $\text{Cost}_{nrev}(n-1) =$

$C_a(\sqrt{\frac{2C_r}{C_a}} - 1)^2/2$ and $\text{Cost}_{append}(n) = C_a\sqrt{\frac{2C_r}{C_a}}$. These are the desired cost distributing functions.

It should be pointed out that in some cases, it is not necessary to first derive the cost functions and then derive the cost distributing functions since we can simply derive the cost distributing scheme directly from the cost recurrence equations. For example, consider the Fibonacci function, where the cost equations are

$$\begin{aligned} \text{Cost}_{fib}(n) &= C_f + 2 \times \text{Cost}_{fib}(n/2) \\ \text{Cost}_{fib}(0) &= C_1 \end{aligned}$$

Without actually deriving the cost functions of $\text{Cost}_{fib}(n)$, we can simply derive the cost distributing relationship from the first equation as $\text{Cost}_{fib}(n/2) = (\text{Cost}_{fib}(n) - C_f)/2$.

Also note that at compile time, the cost distributing functions should be simplified as much as possible to reduce the runtime overhead. It is even worthwhile sacrificing precision to get a simpler function. Therefore, a conservative approach should be used to derive the upper bound of the cost functions. In fact, we can further simplify the cost function derived in the following way. If the cost function is of a polynomial form such as $c_0n^k + c_1n^{k-1} + \ldots c_k$, we simplify it as kc_0n^k and if the cost function is of several exponential components such as $c_1a^n + c_2b^n$ where $b > a$, we simplify it as $(c_1 + c_2)b^n$. This will simplify the solution of the iteration parameter and the cost distributing function and hence simplify the evaluation of them at runtime.

5.1 Runtime Goal Management

The above cost relationship estimation is well suited for a runtime scheduler which adopts an on-demand scheduling policy (e.g., [8]), where PEs maintain a local queue for active goals and once a PE becomes idle, it requests a goal from other PEs. A simple way to distribute a goal to a requesting PE is to migrate an active goal in the queue. The scheduler should adopt a policy to decide which goal is going to be sent. It is obvious that the candidate goal should have the maximal grain size among those goals in the queue. Hence, we can use a priority queue where weights of goals are their grain sizes (or costs). The priority is that the bigger the costs are, the higher priority they get. Because the scheduler only needs to know the relative costs, we can always assume the weight of the initial goal is some fixed, big-enough number. Based on this initial cost and the cost distributing formulae derived at compile time, every time a new clause is invoked, the scheduler derives the relative costs of body goals. The body goals are then enqueued into the priority queue based on their costs.

Some bookkeeping problems arise from this approach. First, even though we can simplify the cost distributing functions at compile time to some extent, the runtime overhead may still be large, since for each procedure invocation, the scheduler has to calculate the weights of the body goals. One solution to this problem is to let the scheduler keep track of a modulo counter and when the content of the counter is not zero, the scheduler simply lets the costs of the body goals be the same as that of their parent. Once the content of the counter becomes zero, the cost-distributing functions are used. If we can choose an appropriate counting period, this method is reasonable (one counter increment has less overhead than the evaluation of the cost estimate).

Another problem in this approach is that for long-running programs, costs may become negative, i.e., the initial weight is not large enough. Since we require only relative costs, a solution is to reset all costs (including those in the queue, and in suspended goals), when some cost becomes too small. Cost resetting requires the incremental overhead of testing to determine when to reset.

As stated above, we need to choose the initial cost as big as possible. However, this can introduce an anomaly for our relative cost scheme. To see this, consider the `nrev` example again. Suppose that the initial query is `nrev([1,...,50])`. The correct query cost is approximately $50 \times 50 = 2500$. The correct cost of its immediate `append` goal is approximately 49, and the correct cost of one of its leaf descendant goals `nrev([])` is one (the head unification cost). If we choose the initial cost as a big number, say 10^6, then the corresponding iteration parameter is 10^3. This will give the cost of `nrev([])` as $(10^3 - 50)^2$ which is bigger than the estimated cost of the initial `append` goal (only around 10^3). In other words, this gives an incorrect relationship between goals near the very top and near the very bottom of the proof tree.

For this particular example, the problem could be finessed by precomputing the "correct" initial value of the iteration parameter: exactly equal to the weight of the query. However, in general, a correct initial estimation is not always possible, and when it *is* possible, its computation incurs too much overhead. All compile-time granularity estimation schemes must make this trade-off. Fortunately, in our scheme, the problem is not as serious as it first appears. For initial goals with sufficiently large cost, our scheme is still able to give correct relative cost estimation for sufficiently large goals which are not close to leaves of the execution call graph. This can be seen in the `nrev` example, where the relative costs among `nrev([2,...,50]` through `nrev([42,...,50])`, and the initial `append` are still correct in our scheme. Correct estimation for the large goals (those near the root of the proof tree) is more important than that for small goals (those near the leaves) because the load balance of the system is largely dependent on those big goals, and so is performance.

Heuristic	Applicable	Correct	Percentage
§1	24	21	87.5%
§2	29	26	89.6%
§3	4	2	50.0%
all	32	27	84.7%

Table 1: Statistics for Benchmark Programs

Heuristic	Applicable	Correct	Percentage
§1	64	57	89.1%
§2	49	55	87.3%
§3	6	4	66.7%
all	111	101	91.0%

Table 2: Statistics for a Compiler Front End

6 Empirical Results: Justifying the Heuristics

We applied our three heuristics and the cost estimation formulae to two classes of programs. The first class includes nine widely used benchmark programs [12], containing 32 procedures. The second class consists of 111 procedures comprising the front-end of the Monaco FGHC compiler. The results are summarized in Table 1 and Table 2. For each heuristic, the tables show the number of procedures for which the heuristic is applicable (by the syntactic rules given in Section 4.2), and the number for which the heuristic is correctly estimates complexity. The row labeled "all" gives the total number of procedures analyzed. Since more than one heuristic may be applicable in a single procedure, the total number of procedures may be less than the sum of the previous rows.

From the tables, we see that §1 and §2 apply most frequently. This indicates that most procedures are linear recursive (i.e., have a single recursive body goal) which can be estimated correctly by our scheme. The relatively low percentage of §3 correctness is because the benchmarks are biased towards procedures with exponential time complexity, whereas §3 usually gives polynomial time complexity.

Analysis of the benchmarks indicated two major anomalies in the heuristics. Although §1 may apply, a procedure may distribute a little work (say, the head of a list) to one body goal and the rest of the work (say, the tail of the list) to another goal. This cannot be captured by §1, which essentially treats the head and tail of the list as equal, i.e., a binary tree. A correct cost analysis needs to explore the data structures of the program.

For recursive procedures, §3 can capture only the fixed-degree divide & conquer programming paradigm. However, the compiler benchmark contained procedures which recursively traverse a list (or vector) and the degree of the divide & conquer dynamically depends on the number of top-level elements in the list (or vector). In this situation, the procedure may have to loop on the top level while recursively traversing down for each element (which may be tree structures). Again, this presents inherent difficulty for our scheme because we take the call graph as the sole input information for the program to be analyzed.

To summarize, our statistics show that our scheme achieves a fairly high percentage of correct estimation. However, we need to apply multiply-recursive heuristics §2 and §3 with more finesse. Further quantitative performance studies of the algorithm's utility are presented in Tick and Zhong [11]. Those multiprocessor simulation results quantify the advantage of dynamically scheduling tasks with the granularity information.

7 Conclusions and Future Work

We have proposed a new method to estimate the relative costs of procedure execution for a concurrent language. The method is similar to Tick's static scheme [10], but gives a more accurate estimation and reflects runtime weight changes. This is achieved by the introduction of an iteration parameter which is used to model recursions.

Our method is based on the idea that it is not the absolute cost, but rather the relative cost that matters for an on-demand goal scheduling policy. Our method is also amenable to implementation. First, our method can be applied to any program. Second, the resultant recurrence equations can be solved systematically. In comparison, it is unclear how to fully mechanically implement the schemes proposed in [2, 4]. Nonetheless, our method may result in an inaccurate estimation for some cases. This is because we use only the call graph to model the program structure, not the data. We admit that further static analysis of program structure such as argument-size relationships can give more precise estimations.

Future work in granularity analysis includes the development of a more systematic and precise method to solve the derived recurrence equations. It is also necessary to examine this method for more practical programs, performing benchmark testing on a multiprocessor to show the utility of the method.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with funding from Sequent Computer Systems Inc. The authors wish to thank S. Debray and the anonymous referees for their helpful criticism.

REFERENCES

[1] S. K. Debray. A Remark on Tick's Algorithm for Compile-Time Granularity Analysis. *Logic Programming Newsletter*, 3(1):9–10, 1989.

[2] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[3] D. Gries. *Science of Programming.* Springer-Verlag, 1989.

[4] A. King and P. Soper. Granularity Control for Concurrent Logic Programs. In *International Computer Conference*, Turkey, 1990.

[5] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, pages 23–32, January 1988.

[6] C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32:1073–1978, 1989.

[7] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, Cambridge MA, 1989.

[8] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

[9] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics, Philadelphia PA, 1983.

[10] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. *New Generation Computing*, 7(2):325–337, January 1990.

[11] E. Tick and X. Zhong. A Compile-Time Granularity Analysis Algorithm and its Performance Evaluation. *Journal of Parallel and Distributed Computing*, submitted to special issue.

[12] E. Tick. *Parallel Logic Programming.* MIT Press, Cambridge MA, 1991.

[13] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA, 1987.

```
nrev([],R) :- R=[].
nrev([H|T],R) :- nrev(T,R1), append(R1,[H],R).

append([],L,A) :- A=L.
append([H|T],L,A) :- A=[H|A1], append(T,L,A1).
```

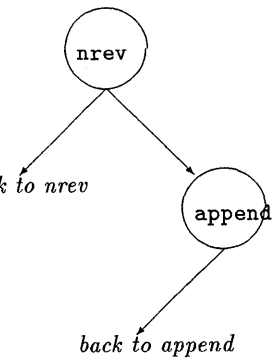


Figure 1: Naive Reverse and its Call Graph

```
qsort([], S) :- S=[].
qsort([M|T],S) :-
    split(T,M,S,L),
    qsort(S,SS),
    qsort(L,LS),
    append(SS,LS,S).

split([],   M,S,L) :- S=[], L=[].
split([H|T],M,S,L) :- H < M  |
    S=[H|TS], split(T,M,TS,L).
split([H|T],M,S,L) :- H >= M |
    L=[H|TL], split(T,M,S,TL).
```

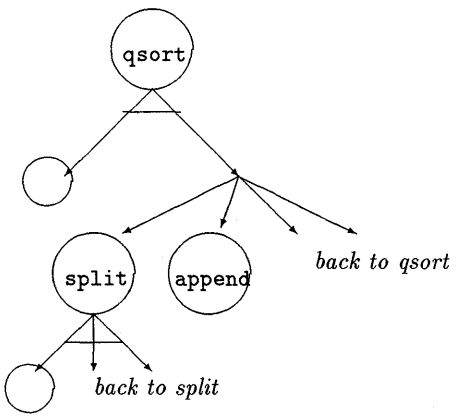


Figure 2: Quick Sort: FGHC Source Code and the AND/OR Call Graph

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Providing Iteration and Concurrency in Logic Programs through Bounded Quantifications

Jonas Barklund and Håkan Millroth, UPMAIL
Computing Science Dept., Uppsala University,
Box 520, S-751 20 Uppsala, Sweden
E-mail: jonas@csd.uu.se or hakanm@csd.uu.se

Abstract

Programs operating on inductively defined data structures, such as lists, are naturally defined by recursive programs. Millroth has recently shown how many such programs can be transformed or compiled to iterative programs operating on arrays. The transformed programs can be run more efficiently than the original programs, particularly on parallel computers.

The paper proposes the introduction of 'bounded quantifications' in logic programming languages. These formulas offer a natural way to express programs operating on arrays and other 'indexable' data structures. 'Bounded quantifications' are similar to 'array comprehensions' in functional languages such as Haskell. They are inherently concurrent and can be run efficiently on sequential computers as well as on various classes of parallel computers.

1 PROCESSING DATA STRUCTURES

There are two principal ways of building a data structure in a logic program.

A1. Use a recursive relation which defines explicitly the contents of a finite part of the data structure and then uses itself recursively to define the rest of the data structure.

B1. Express directly the contents of each element of the data structure, preferrably through an 'indexing' of the elements of the data structure.

Correspondingly there are two principal ways of traversing a data structure in a logic program.

A2. Use a recursive relation which examines explicitly the contents of a finite part of the data structure and then uses itself recursively to traverse the rest of the data structure.

B2. Access directly the contents of each element of the data structure, preferrably through an 'indexing' of the elements of the data structure.

(There is, of course, an obvious duality between these operations.)

Method A is often natural when one uses inductively defined data structures, including lists, trees, etc. Method B is often natural when one uses data structures whose elements can be indexed. Some data structures, most importantly lists, fall in both categories and which method is most natural depends on the context.

2 RECURSION

We can broadly classify recursive programs in 'conjunctive' and 'disjunctive' programs (some are a mixture). The former category use recursion to compute a conjunction, like the following *lessall* program.[1]

$lessall(A, [B|X]) \leftarrow A < B \wedge lessall(A, X).$
$lessall(A, []).$

A formula $lessall(A, [B_1, B_2, \ldots, B_n])$ reduces to the finite conjunction

$$A < B_1 \wedge A < B_2 \wedge \cdots \wedge A < B_n$$

which could be expressed more briefly as

$$\forall i\{1 \leq i \leq n \rightarrow A < B_i\}.$$

This reduction can be performed at compile time, except that the value of n is the length of the list actually supplied to the program. Such a program can be run efficiently as an iteration on a sequential computer.

The latter category uses recursion to compute a disjunction, for example the member program.

$member(A, [B|X]) \leftarrow A = B.$
$member(A, [B|X]) \leftarrow member(A, X).$

A formula $member(A, [B_1, B_2, \ldots, B_n])$ reduces to the finite disjunction

$$A = B_1 \vee A = B_2 \vee \cdots \vee A = B_n$$

[1] Our language consists (initially) of clauses whose bodies may contain comjunctions, disjunctions and negations. We assume "Herbrand" equality except for arithmetic expressions and array elements. All examples can be easily translated into Prolog or Gödel (Hill & Lloyd, 1991).

which could, in turn, be expressed more briefly as

$$\exists i\{1 \leq i \leq n \wedge A = B_i\}$$

which can, similarly, be run efficiently. Millroth's compilation method (1990, 1991), based on Tärnlund's Reform inference system (1992) transforms 'conjunctive' and 'disjunctive' recursive programs to the iterative programs above.

2.1 Concurrency

The conjunction, or disjunction, in a logic program can be interpreted as a concurrent operator, as in AND-parallel and OR-parallel logic programming systems. This does not yield sufficient concurrency for running recursive programs efficiently on parallel computers. Even using a concurrent connective, work is only initiated on one 'recursion level' in each step. This implies a linear run time which can be approximated by an expression $An+B$ (where A is the overhead for each recursion level, n is the recursion depth and B is the time spent in each recursion level). The number of literals in a recursive clause is typically much smaller than the depth of the recursion. For recursive programs with simple bodies, such as *lessall* or *member*, the An term will always dominate Only for small recursion depths and complex bodies will the B term be significant.

Recursive programs transformed by Millroth's method have a much larger potential to run efficiently on parallel computers. The iterative programs can be run in parallel on n processors unless prohibited by data dependencies etc. Techniques for parallelizing this kind of iterations have been developed for, and applied to, FORTRAN programs for some time.

3 EXPLICIT QUANTIFICATION

It is possible to build arrays and other indexable data structures, or express relations over them using recursive programs. It is often more natural to use a universal or existential quantification over the members of the data structure.

We may express the *lessall* relation over arrays as

$$lessall(A, X) \leftarrow \forall B \forall I\{X[I] = B \rightarrow A < B\},$$

provided that the the value of the expression $X[I]$ is the Ith element of the array X.

We may express reversal of the elements in an array:

$$\begin{aligned} &reverse(X_1, X_2) \leftarrow \\ &\quad size(0, X_1, L) \wedge size(0, X_2, L) \wedge \\ &\quad \forall A \forall I\{X_1[I] = A \rightarrow X_2[L - I - 1] = A\}. \end{aligned}$$

(Our notation assumes that the expression $L - I - 1$ is evaluated and replaced by its value. We also assume that array indices are zero based. Finally, we let $size(D, X, S)$ express that the size of the array X in dimension D is S.)

We may express one generation of Conway's game of Life:

$$\begin{aligned} &step(G_1, G_2) \leftarrow \\ &\quad size(0, G_1, S_0) \wedge size(0, Q, S_0) \wedge size(0, G_2, S_0) \wedge \\ &\quad size(1, G_1, S_1) \wedge size(1, Q, S_1) \wedge size(1, G_2, S_1) \wedge \\ &\quad \forall I \forall J\{Q[I, J] = G_1[I - 1 \bmod S_0, J - 1 \bmod S_1] + \\ &\qquad\qquad G_1[I - 1 \bmod S_0, J] + \\ &\qquad\qquad G_1[I - 1 \bmod S_0, J + 1 \bmod S_1] + \\ &\qquad\qquad G_1[I, J - 1 \bmod S_1] + \\ &\qquad\qquad G_1[I, J + 1 \bmod S_1] + \\ &\qquad\qquad G_1[I + 1 \bmod S_0, J - 1 \bmod S_1] + \\ &\qquad\qquad G_1[I + 1 \bmod S_0, J] + \\ &\qquad\qquad G_1[I + 1 \bmod S_0, J + 1 \bmod S_1] \rightarrow \\ &\qquad (Q[I, J] < 2 \wedge G_2[I, J] = 0 \vee \\ &\qquad\ Q[I, J] = 2 \wedge G_2[I, J] = 1 \vee \\ &\qquad\ Q[I, J] = 3 \wedge G_2[I, J] = 1 \vee \\ &\qquad\ Q[I, J] > 3 \wedge G_2[I, J] = 0)\}. \end{aligned}$$

We can also present a simple example of the use of explicit existential quantifiers. The problem is to find the position I in a array X of some element which is smaller than a given value A.

$$small(I, X, A) \leftarrow \exists J\{X[J] = B \rightarrow B < A \wedge J = I\}.$$

In all these examples we have quantified over the elements of an indexable data structure. There are other useful relations which can be expressed naturally in this way, and run efficiently. Specifically we want to include all quantifications over the elements of a finite set, whose members are 'obvious'. Below we will be somewhat more precise what this means.

4 BOUNDED QUANTIFICATION

Consider those universally quantified formulas which are instances of the schema

$$\forall x\{\Theta[x] \rightarrow \Phi[x]\}$$

where Θ is a formula which is "obviously" true for only a finite number of values of x, denoted by, say, $\{c_0, c_1, \ldots, c_{k-1}\}$. In this case the quantification is clearly equivalent to the finite conjunction

$$\begin{aligned} &(\Theta[c_0] \rightarrow \Phi[c_0]) \wedge \\ &(\Theta[c_1] \rightarrow \Phi[c_1]) \wedge \cdots \wedge \\ &(\Theta[c_{k-1}] \rightarrow \Phi[c_{k-1}]) \end{aligned}$$

which is, by the definition of Θ, equivalent to

$$\Phi[c_0] \wedge \Phi[c_1] \wedge \cdots \wedge \Phi[c_{k-1}].$$

Similarly, a formula which is an instance of the schema $\exists x\{\Theta[x] \wedge \Phi[x]\}$ is under the same assumptions equivalent to

$$\Phi[c_0] \vee \Phi[c_1] \vee \cdots \vee \Phi[c_{k-1}].$$

We propose to

1. identify a set of formulas which always are true for only a finite number of objects, we call them *range formulas*,

2. make a system which recognizes those instances of the schema above where Θ is a range-formula, we call them *bounded quantifications*, and

3. interpret bounded quantifications concurrently. The conjuncts obtained from a bounded quantification may be run in any order, even simultaneously, provided that any data dependencies (arising, e.g., from numerical expressions) are satisfied.

Since a range formula is required to hold for a finite number of objects, it is possible to enumerate them (as we have indeed done above with $\{c_0, c_1, \ldots, c_{k-1}\}$). It will become apparent from examples below that it is very useful to have range formulas relate each object with a unique integer in $\{0, 1, \ldots, k-1\}$.

In the following sections we will first identify a few useful range formulas and then show how to run bounded quantifications efficiently on sequential and parallel computers.

5 RANGE FORMULAS

The following is an incomplete set of interesting range formulas.

5.1 Array and "structure" elements

As we have seen, it is useful to quantify over all elements of a data structure. In an array, each element is associated with a unique integer in the range, say, $\{0, 1, \ldots, n\}$. We could, for example let $X[I] = E$ (where X is an array, I is a variable and E is a term) be a range formula and the *lessall* and *reverse* programs above are examples of its use. It may be difficult to write a compiler which recognizes precisely this use of an equality as a range formula. One solution would be to predefine, say, the predicate symbol *elt* by

$$elt(I, X, E) \leftarrow X[I] = E.$$

and only recognize predications on the form $elt(\cdot, \cdot, \cdot)$ as range formulas.

5.2 Integer ranges

An obviously useful range formula would be one which is true for the first k integers ($[0, k-1]$). Again, the formula $0 \leq X \wedge X < K$ expresses exactly that relation, but for practical reasons it may be wise to define the binary predication *cardinal*(X, K) to stand for the binary relation which is true whenever $0 \leq X < K$. Note that the enumeration in this case coincides with the objects themselves.

Note, moreover, that it is trivial to obtain a range formula which is true for all integers in an arbitrary range $[I, J]$ using the binary *cardinal* predicate.

5.3 Enumerable types

A logic programming language with types is likely to contain "enumerable" types, for example, finite sets of distinct constants. One may wish to consider any predication, whose predicate symbol coincides with the name of such a type, a range relation. For example, suppose that colour is a type with the elements *spades*, *hearts*, *clubs*, and *diamonds* (in that order). Then *colour*(I, X) is a range formula which is true if and only if I is 0 and X is *spades*, I is 1 and X is *hearts*, I is 2 and X is *clubs*, or I is 3 and X is *diamonds*.

Note that in this view an enumerable type of K elements is isomorphic with the integer range $[0, K-1]$, so it does not really add anything to the language as such.[2]

5.4 List elements and list suffixes

Lists are usually operated upon by recursively defined programs. Still, there are occasionally reasons for expressing programs through bounded quantifications. We propose two range formulas involving lists. The first associates every element of some list with its (zero-based) position in the list. The second enumerates every (not necessarily proper) suffix of some list (with the list itself being suffix 0). We propose to recognize the predication *member*(I, L, X) as a range formula which is true if and only if X is the Ith element of the list L.

The predication *suffix*(I, L, X) is a range formula which is true if and only if X is the Ith suffix of the list L. Note that if the length of L is K and [] denotes an empty list, then *suffix*$(0, L, L)$ and *suffix*$(K, L, [])$ are true formulas. (Since Prolog has no occur check, a programmer in that language could apply these predicates to cyclic "terms". We leave the behaviour in such a case undefined.)

5.5 Finite sets

Given that finite sets are provided as a data structure it would make sense to have range formulas for sets (e.g., membership), as has been suggested by Omodeo (personal communication). This is an interesting proposal, but is is difficult to represent arbitrary sets efficiently in a way that allows the elements to be enumerated. Multisets (bags) are easier to implement, but these are, on the other hand, quite similar to lists, except that the order in which elements occur is irrelevant.

6 SEQUENTIAL ITERATION

Consider a bounded quantification $\forall x\{\Theta[x] \rightarrow \Phi[x]\}$, such that $\Theta[x]$ is true when (and only when) the value of x is one of $\{c_0, c_1, \ldots, c_{k-1}\}$. We may run the conjuncts $\Phi[c_0] \wedge \Phi[c_1] \wedge \cdots \wedge \Phi[c_{k-1}]$ in any order, provided that any data dependencies are satisfied.

[2] They do, however, seem to make programs easier to understand and debug.

We consider now a bounded quantification without dependencies. Running it on a sequential computer is straightforward: translate the quantified formula into an iteration which evaluates, in sequence, the formulas $\Phi[c_0], \Phi[c_1], \ldots, \Phi[c_{k-1}]$.

Since the compiler knows in advance about the possible range formulas, it may generate specialized code for each kind of range formula. For example, if the range formula $\Theta[x]$ is $member(I, X, L)$ then we can illustrate the resulting code as

```
allocate_environment;
y = deref(l);
while (y != NIL)
{
    x = deref(y->head);
    code for Φ[x];
    y = deref(y->tail);
}
deallocate_environment;
```

using a C-style notation. (Note that we ignore the enumeration of the list elements in this example.) Assuming that the implementation is based on WAM (Warren, 1983) the "code for $\Phi[x]$" may introduce choice points (and thus be unable to deallocate environments) if there are alternative solutions for $\Phi[x]$.

In the important case that the proof for $\Phi[x]$ is deterministic, every pass through the loop will begin in the same environment. This is more efficient than the corresponding recursive computation in Prolog (under WAM) which will allocate and deallocate an environment for each recursive call. Most implementations will also refer to the symbol table when making the recursive call. That is somewhat less efficient than the (conditional) jump performed at the end of a loop. We predict that together these improvements will result in substantial savings, particularly when proofs are deterministic, the bodies of recursive clauses are small and recursion is deep. Meier also notes these advantages when compiling some recursive programs as iterations (1991).

7 PARALLEL ITERATION

On sequential computers bounded quantification, when at all appropriate, is likely to offer significant improvements over the corresponding recursive programs, run in the usual way. The potential speed-ups on parallel computers are still more dramatic.

Consider the conjunction

$$\Phi[c_0] \wedge \Phi[c_1] \wedge \cdots \wedge \Phi[c_{k-1}]$$

obtained from a bounded quantification $\forall x\{\Theta[x] \rightarrow \Phi[x]\}$. Since we may run the conjuncts in any order, we may also run them all in parallel (similarly for disjunctions), provided that we add synchronization to satisfy dependencies.

7.1 Running deterministic programs

There are several methods for running deterministic iterations in parallel; these ideas have successfully applied to FORTRAN programs for a long time. The following is one of the simplest. If there are k processors, numbered from 0 to $k-1$, simply let processor i evaluate $\Phi[c_i]$, for each i, $0 \leq i < k$. If there are fewer than k processors, say k' processors, simulate k processors by letting processor i evaluate $\Phi[c_j]$, for each j, $1 \leq j < k$, such that j modulo k' is i. If the computation of each $\Phi[c_i]$ is deterministic, then this is quite straightforward.

7.2 Running nondeterministic programs

Suppose that the formula Φ is such that there is a choice of two or more potential proofs for some conjunct $\Phi[c_i]$. If no two conjuncts $\Phi[c_i]$ and $\Phi[c_j]$, $i \neq j$, share any variables, then we have independent parallelism in which backtracking is 'local' and easily implemented, cf., e.g., DeGroot (1984).

This is a special case of the more general situation in which one can compute the variable assignments satisfying each conjunct independently of each other. For example, the conjuncts may share a variable, whose value at runtime is an array, and only access distinct elements of it. In general it is not possible to verify this condition statically so some run time tests will be necessary.

Consider the other case: that the free variables of conjuncts interact in such a way that it is not possible to compute variable assignments independently for each conjunct. In that case the corresponding recursive program, if run in the usual way using depth-first search of the proof tree, has to perform deep backtracking to earlier recursion levels. When investigating this class of programs we have noted that they occur surprisingly infrequently. Running such programs often leads to a combinatorial explosion of potential proofs which is only feasible when backtracking over a few recursion levels. The programs also do not behave nicely when running on, e.g., WAM. They tend to consume stack space rapidly if choice information prevents environments from being deallocated.

The problem of simultaneously finding variable assignments for a set of non-independent and non-deterministic conjuncts is also very difficult. Earlier research on backtracking in AND-parallel logic programming systems by, e.g., Conery (1987) confirms this claim.

Our current position is therefore to refuse to run in parallel any bounded quantification for which we cannot show statically, or at least with simple run time tests, that the conjuncts are independent. In the context of AND-parallel logic programming systems, DeGroot among others have investigated appropriate run time tests for independency. Note that the overhead for such tests is lower in our context. One test (say, for determining whether a free variable in a bounded quantification is instantiated at run time) is sufficient for starting

arbitrarily many independent computations.

By applying these requirements also when running bounded quantifications on sequential processors it is guaranteed that the stack size when starting the proof of each conjunct will be constant.

8 SIMD AND MIMD PARALLEL COMPUTERS

We believe that bounded quantifications will run efficiently on both SIMD and MIMD parallel computers. When the bodies of bounded quantifications are simple and no backtracking is needed inside them, the capabilities of SIMD parallel computers are sufficient. It seems that most programs belong, or can be made to belong, to this class.

For those programs which do more complicated processing in the bodies of bounded quantifications, e.g., backtracking, not all processors of a SIMD parallel computer will be active simultaneously. This will reduce the efficiency of such a computer, while it may still be possible to fully utilize a MIMD parallel computer.

9 OTHER OPERATIONS

We think it is also beneficial to predefine certain useful operations, such as reductions and 'scans' over lists and arrays. Such operations will make it easy to eliminate many parallelization problems with variables shared between conjuncts in bounded universal quantifications.

For example, this is a program which computes the inner product S of two arrays X and Y.

$$
\begin{array}{l}
i\_p(X, Y, S) \leftarrow \\
\quad size(0, X, Z) \wedge size(0, Y, Z) \wedge size(0, T, Z) \wedge \\
\quad \forall I \forall Q\{Y[I] = Q \rightarrow T[I] = X[I] \times Q\} \wedge \\
\quad reduce(+, T, S).
\end{array}
$$

The arrays X, Y and T are shared between all conjuncts but they all access distinct elements of the arrays. (The variable Q was only introduced to maintain the standard form of bounded quantifications. It seems convenient and possible to relax the syntax to recognize expressions such as $\forall I\{T[I] = X[I] \times Y[I]\}$ as bounded quantifications, which is certainly even more elegant.

Sometimes the partial sums are also needed in the computation. In this case it is useful to compute a 'scan' with plus over an array. The result is an array of the same length but where each element contains the sum of all preceding elements in the first array.

10 FURTHER EXAMPLES

We now turn to a few more examples written using bounded quantifications. In the authors' opinion these formulas express at a high level the essentials of the algorithms they implement. In some cases they contain formulas reminiscent of what would be (informally expressed) loop invariants when programming in another language.

10.1 Factorial

The following program computes the factorial of N. The program shows the use of the *cardinal* range formula.

$$
\begin{array}{l}
factorial(N, F) \leftarrow \\
\quad size(0, T, N) \wedge \\
\quad \forall I\{cardinal(I, N) \rightarrow T[I] = I + 1\} \wedge \\
\quad reduce(\times, T, F).
\end{array}
$$

10.2 Fibonacci

The following program computes the Nth Fibonacci number. The program is remarkable in being both simple and efficient, since it does not recompute any Fibonacci numbers. Similar effects have been accomplished using 'memo' relations and 'bottom-up' resolution, etc., but this solution appears both simple, elegant and semantically impeccable.

$$
\begin{array}{l}
fibonacci(N, F) \leftarrow \\
\quad size(0, T, N + 1) \wedge \\
\quad \forall I\{cardinal(I, N) \rightarrow \\
\qquad I = 0 \wedge T[I] = 1 \vee \\
\qquad I = 1 \wedge T[I] = 1 \vee \\
\qquad I > 1 \wedge T[I] = T[I - 1] + T[I - 2]\} \wedge \\
\quad F = T[N - 1].
\end{array}
$$

10.3 Finding roots in oriented forests

Suppose that the array P represents an oriented tree.[3] Each element of P contains the index of the parent of some node; roots contain their own index. The following program returns a new array in which each element points immediately to the root of its forest. This is an example of a parallel-prefix algorithm and it also illustrates how bounded quantifications and recursion can be used together.

$$
\begin{array}{l}
find(P, P) \leftarrow \forall I\{P[I] = P[I] \rightarrow P[I] = P[P[I]]\}. \\
find(P_0, P) \leftarrow \\
\quad \forall I \forall J\{P_0[I] = J \rightarrow (J = P_0[J] \wedge P_1[I] = J \vee \\
\qquad\qquad J \neq P_0[J] \wedge P_1[I] = P_0[J])\} \wedge \\
\quad find(P_1, P).
\end{array}
$$

10.4 Matrix transposition

The following little program transposes a matrix.

$$
\begin{array}{l}
trans(M_1, M_2) \leftarrow \\
\quad size(0, M_1, A) \wedge size(1, M_1, B) \wedge \\
\quad size(0, M_2, B) \wedge size(1, M_2, A) \wedge \\
\quad \forall I \forall J \forall Q\{M_1[I, J] = Q \rightarrow M_2[J, I] = Q\}.
\end{array}
$$

[3]Recall that an oriented tree is a "directed graph with a specified node R such that: each node $N \neq R$ is the initial node of exactly on arc; R is the initial node of no arc; R is a root in the sense that for each node $N \neq R$ there is an oriented path from N to R" (Knuth, 1968).

10.5 Numerical integration

The following program computes an approximation to the integral

$$\int_a^b f(x)\,dx$$

using Simpson's method (a quadrature method). In the program below we let A and B be the limits, N the number of intervals and I the resulting approximation of the integral. We assume that the relation $r(X,Y)$ holds if and only if $f(X) = Y$, where f is the function being integrated.

$$\begin{aligned}
&intsimp(A,B,N,I) \leftarrow \\
&\quad W = (B-A)/N \wedge \\
&\quad size(0,G,2\times N+1) \wedge size(0,Z,N) \wedge \\
&\quad \forall I \forall Y\{G[I] = Y \rightarrow r(A + I \times W/2, Y)\} \wedge \\
&\quad \forall I \forall S\{Z[I] = S \rightarrow \\
&\qquad\qquad S = W \times (G[2\times I] + \\
&\qquad\qquad\qquad 4 \times G[2\times I+1] + \\
&\qquad\qquad\qquad G[2\times I+2])/6\} \wedge \\
&\quad reduce(+,Z,I).
\end{aligned}$$

The array G is set up to contain the $2 \times N + 1$ values $f(a)$, $f(a + w/2)$, $f(a + w)$, ..., $f(b - w)$, $f(b - w/2)$, $f(b)$. These values are used to compute the area for each of the intervals, stored in Z. Finally the sum of the areas is computed.

10.6 Linear regression

This is an example of a more involved numeric computation, adopted from Press *et al.* (1989). The problem is to fit a set of n data points (x_i, y_i), $0 \le i < n$, to a straight line defined by the equation $y = A + Bx$. We assume that the uncertainty σ_i associated with each item y_i is known, and that all x_i (values of the dependent variable) are known exactly.

Let us first define the following sums.

$$S = \sum_{i=0}^{n-1} \frac{1}{\sigma_i^2} \qquad S_x = \sum_{i=0}^{n-1} \frac{x_i}{\sigma_i^2} \qquad S_y = \sum_{i=0}^{n-1} \frac{y_i}{\sigma_i^2}$$
$$S_{xx} = \sum_{i=0}^{n-1} \frac{x_i x_i}{\sigma_i^2} \qquad S_{xy} = \sum_{i=0}^{n-1} \frac{x_i y_i}{\sigma_i^2}$$

The coefficients A and B of the equation above can now be computed as

$$\begin{aligned}
\Delta &= SS_{xx} - S_x S_x \\
A &= \frac{S_{xx}S_y - S_x S_{xy}}{\Delta} \\
B &= \frac{SS_{xy} - S_x S_y}{\Delta}
\end{aligned}$$

The following program computes A and B from three arrays X, Y and U.

$$\begin{aligned}
&linear_regression(X,Y,U,A,B) \leftarrow \\
&\quad size(0,X,N) \wedge size(0,Y,N) \wedge size(0,U,N) \wedge \\
&\quad size(0,Z,N) \wedge size(0,Z_x,N) \wedge size(0,Z_y,N) \wedge \\
&\quad size(0,Z_{xx},N) \wedge size(0,Z_{xy},N) \wedge \\
&\quad \forall I\{cardinal(I,N) \rightarrow \\
&\qquad Z[I] = 1/(U[I]\times U[I]) \wedge \\
&\qquad Z_x[I] = X[I]/(U[I]\times U[I]) \wedge \\
&\qquad Z_y[I] = Y[I]/(U[I]\times U[I]) \wedge \\
&\qquad Z_{xx}[I] = (X[I]\times X[I])/(U[I]\times U[I]) \wedge \\
&\qquad Z_{xy}[I] = (X[I]\times Y[I])/(U[I]\times U[I])\} \wedge \\
&\quad reduce(+,Z,S) \wedge \\
&\quad reduce(+,Z_x,S_x) \wedge reduce(+,Z_y,S_y) \wedge \\
&\quad reduce(+,Z_{xx},S_{xx}) \wedge reduce(+,Z_{xy},S_{xy}) \wedge \\
&\quad Delta = S\times S_{xx} - S_x \times S_x \wedge \\
&\quad A = (S_{xx}\times S_y - S_x \times S_{xy})/Delta \wedge \\
&\quad B = (S\times S_{xy} - S_x \times S_y)/Delta.
\end{aligned}$$

It is obvious that this program can be run in $O(\log n)$ time using n processors, dominated by the reductions. The bounded quantification which computes the intermediate arrays Z, Z_x, Z_y, Z_{xx} and Z_{xy} runs in constant time using n processors.

11 LIST EXAMPLES

The following two examples are present simply to show that it is possible to express also list algorithms using bounded quantifications, although the recursive programs are usually more elegant.

11.1 Lessall

The *lessall* program for lists is of course very similar to the array program (this makes it easy to change the data structure).

$$lessall(A,L) \leftarrow \forall B \forall I\{member(I,L,B) \rightarrow A < B\}.$$

11.2 Partition

The *partition* program, finally, is an example of a program which is much clearer when expressed recursively. We intend that $partition(X,A,L,H)$ be true if and only if L contains exactly those of elements of X which are less than or equal to A, and H contains exactly those which are greater than A. The partition predicate is usually part of an implementation of Hoare's Quicksort algorithm. Here is the recursive program:

$$\begin{aligned}
&partition([],A,[],[]). \\
&partition([B|X],A,L,[B|H]) \leftarrow \\
&\quad A \le B \wedge partition(X,A,L,H). \\
&partition([B|X],A,[B|L],H) \leftarrow \\
&\quad A > B \wedge partition(X,A,L,H).
\end{aligned}$$

In the following program which uses bounded quantifications, we have tried to keep some of the structure of the recursive program.

$$\begin{aligned}
&partition(X,A,L,H) \leftarrow \\
&\quad \forall F_X \forall Z \forall I\{suffix(I,X,F_X) \rightarrow \\
&\qquad\qquad member(I,S_L,L) \wedge \\
&\qquad\qquad member(I,S_H,H) \wedge \\
&\qquad\qquad part(F_X,L,H,A,S_L,S_H)\} \wedge
\end{aligned}$$

$$
\begin{array}{l}
\quad member(1, S_L, L) \wedge member(1, S_H, H). \\
part([], [], [], A, S_L, S_H). \\
part([B|X], L, H, A, S_L, S_H) \leftarrow \\
\quad J = I + 1 \wedge \\
\quad member(J, S_L, L_1) \wedge member(J, S_H, H_1) \wedge \\
\quad (A \leq B \wedge L = L_1 \wedge H = [B|H_1] \vee \\
\quad A > B \wedge L = [B|L_1] \wedge H = H_1).
\end{array}
$$

The program computes two lists of lists S_L and S_H which are scans of partitions on X, picking out those elements which are less than or greater than A, respectively.

12 NESTED BOUNDED QUANTIFICATIONS

Consider a bounded quantification whose body is another bounded quantification:

$$\forall x\{\Theta_1[x] \rightarrow \forall y\{\Theta_2[y] \rightarrow \Phi[x, y]\}\}.$$

Provided that $\Theta_1[x]$ is true for any x in $\{c_0, c_1, \ldots, c_{k-1}\}$, and that similarly $\Theta_2[y]$ is true for any y in $\{d_0, d_1, \ldots, d_{\ell-1}\}$, the nested bounded quantification is equivalent to the $k \times \ell$ element conjunction

$$
\begin{array}{llll}
\Phi[c_0, d_0] & \wedge \Phi[c_0, d_1] & \wedge \cdots & \wedge \Phi[c_0, d_{\ell-1}] \\
\wedge \Phi[c_1, d_0] & \wedge \Phi[c_1, d_1] & \wedge \cdots & \wedge \Phi[c_1, d_{\ell-1}] \\
\vdots & \vdots & \ddots & \vdots \\
\wedge \Phi[c_{k-1}, d_0] & \wedge \Phi[c_{k-1}, d_1] & \wedge \cdots & \wedge \Phi[c_{k-1}, d_{\ell-1}]
\end{array}
$$

As before, provided that all data dependencies are satisfied, all these conjuncts can be run simultaneously.

13 TOLERATING DEPENDENCIES

In all examples shown above the computations of the conjuncts obtained from a bounded quantification have been independent. Therefore the conjuncts could be computed in any order, for example in parallel.

There are interesting computations where the resulting conjuncts are dependent. Consider, for example, the following program (adapted from a program by Anderson & Hudak [1990]) which defines an $n \times n$ matrix A through a recurrence.

$$
\begin{array}{l}
rec(A) \leftarrow \\
\quad size(0, A, N) \wedge size(1, A, N) \wedge \\
\quad \forall I \forall J\{A[I, J] = X \rightarrow \\
\qquad\qquad I = 1 \wedge X = 1 \vee \\
\qquad\qquad I > 1 \wedge J = 1 \wedge X = 1 \vee \\
\qquad\qquad I > 1 \wedge J > 1 \wedge \\
\qquad\qquad X = A[I-1, J] + \\
\qquad\qquad\quad A[I-1, J-1] + \\
\qquad\qquad\quad A[I, J-1]\}.
\end{array}
$$

This program requires a co-routining implementation of bounded quantification to run on a sequential computer or synchronization to run on a parallel computer. We are currently investigating whether automatic generation of synchronization/co-routining code is sufficient or if the programmer should be allowed to annotate the program, for example, through read-only variables (Shapiro, 1983).

14 RELATED WORK

We noted above that M. Meier has suggested (1991) how to compile some tail recursive (conjunctive as well as disjunctive) programs to iterative programs on top of WAM.

Several authors, e.g., Lloyd & Topor (1984) and Sato & Tamaki (1989), have discussed methods for running logic programs with arbitrary formulas in bodies. Our method only covers a limited extension of Horn clauses.

14.1 Array Comprehensions

It is obvious that there are similarities between arrays and bounded quantifications on one side, and the array comprehensions proposed for the Haskell language (Hudak & Wadler, 1990) on the other. Both concepts aim to express the contents of an array, or the relationship between several arrays, declaratively.

It appears to us, as with most functional programming language concepts, that when they are at all appropriate they offer a more compact and occasionally more elegant notation. For example, the factorial program above could have been expressed more easily if an expression describing the temporary array T could have been written immediately.

However, when the relationship between the elements of more than one array are to be described, the bounded quantifications appear to be more comprehensive.

Array comprehensions are, in general, evaluated by lazy computation. This can be thought of as a degenerated form of concurrency which suspends part of a computation until it is known that it must be performed. We do not think lazy computation is necessary, provided unification with the "logical variable" and a more general form of concurrency.

Futures (Halstead, 1985) are yet another way of giving names for values which are yet to be fully computed.

14.2 Nova Prolog

The ideas presented above originated as a generalization of the language Nova Prolog (Barklund & Millroth, 1988).[4] Here, however, it is appropriate to present Nova Prolog as a language embodying a subset of bounded quantifications. The subset is chosen to obtain a language tailored specifically for massively parallel SIMD computers, such as the Connection Machine. More specifically, we assume that we can store some data structures in such a way that processor i has particularly efficient access to the ith element of each data structure. We say that those data structures are distributed.

[4]Nova Prolog relates to Prolog in much the same way as *LISP (by Thinking Machines Corp.) relates to Common LISP and C* (also by Thinking Machines Corp.) to C. That is, it is a sequential programming langauge extended with a distributed data structure and a control structure for expressing computations over each element on the data structure.

We currently limit the distributed data structures to be compound terms; in fact only those compund terms whose function symbol is *pterm* and whose arity is some fixed value. We shall call them 'pterms.' (This is to help a compiler distinguish distributed data structures from other compound terms.)

Since pterms are the only distributed data structures and they are compound terms, the only range formula we need is $arg(i, t, x)$.[5] We have chosen a syntax for bounded quantifications which makes it possible to combine the range formula with the quantification of variables. In Nova Prolog a formula

$$A_1 : T_1, A_2 : T_2, \ldots, A_n : T_n \,//\, \Phi[self],$$

where T is a pterm, is called a 'parall' and has the same meaning as the bounded quantification

$$\forall I \forall A_1 \forall A_2 \cdots \forall A_n (arg(I, T_1, A_1) \rightarrow arg(I, T_2, A_2) \wedge \cdots \wedge arg(I, T_n, A_n) \wedge \Phi[I]),$$

namely that Φ is true for every corresponding element A_i of T_i, $1 \leq i \leq n$. We can see that in Nova Prolog the 'index' I is implicit and is denoted by the constant symbol *self* in the body Φ.

All examples above for array computations can be translated into Nova Prolog. We have recently implemented parts of Nova Prolog in *LISP (Blanck, 1991).

15 CONCLUSION AND FUTURE WORK

We have defined bounded quantifications, a new construct for logic programming languages. We have discussed how they can be efficiently implemented on sequential and parallel computers. They offer clarity as well as efficiency and we propose that language designers and implementors consider including them in implementations of, e.g., Prolog, Gödel and KL1.

A natural continuation of this work is to verify experimentally that bounded quantifications can be implemented efficiently in sequential and concurrent languages, and on sequential and parallel computers. It is also important to investigate how data dependencies and other synchronization considerations can be handled, when bounded quantifications are interpreted concurrently.

REFERENCES

Anderson, S. & Hudak, P., 1990, Compilation of Haskell Array Comprehensions for Scientific Computing. In *Proc. SIGPLAN '90 Conf. on Programming Language Design and Implementation.* ACM Press, New York, N.Y.

Barklund, J. & Millroth, H., 1988, Nova Prolog. *UPMAIL Tech. Rep. 52.* Computing Science Dept., Uppsala University.

Blanck, J., 1991, Abstrakt maskin för Nova Prolog. Internal report. Computing Science Dept., Uppsala University.

DeGroot, D., 1984, Restricted And-Parallelism. In *Proc. Intl. Conf. on Fifth Generation Comp. Systems 1984*, pp. 471–8. North-Holland, Amsterdam.

Halstead, R., 1985, Multilisp—a Language for Concurrent Symbolic Computation. *ACM TOPLAS, 2,* 501–38.

Hill, P. M. & Lloyd, J. W., 1991, The Gödel Report. *Tech. Rep. 91-02.* Computer Science Dept., University of Bristol.

Hudak, P. & Wadler, P., 1990, Report on the Programming Language Haskell. *Tech. Rep. YALEU/DCS/RR-777.* Dept. of Computer Science, Yale Univ.

Knuth, D. E., 1968, *The Art of Computer Programming.* Volume 1 / Fundamental Algorithms. Reading, Mass.

Lloyd, J. W. & Topor, R. W., 1984, Making Prolog more Expressive. *J. Logic Programming, 1,* 225–40.

Meier, M., 1991, Recursion vs. Iteration in Prolog. In *Proc. 8th Intl. Conf. on Logic Programming* (ed. K. Furukawa), pp. 157–69. MIT Press, Cambridge, Mass.

Millroth, H., 1990, Reforming Compilation of Logic Programs. Ph.D. thesis. *Uppsala Theses in Computing Science 10.* Computing Science Dept., Uppsala University. (A summary will appear in the next item.)

Millroth, H., 1991, Reforming Compilation of Logic Programs. In *Proc. 1991 Intl. Logic Programming Symp.* (ed. V. Saraswat, K. Ueda). MIT Press, Cambridge, Mass.

Press, W. H. *et al.*, 1989, *Numerical Recipes.* The Art of Scientific Computing. Cambridge Univ. Press, Cambridge, U.K.

Sato, T. & Tamaki, H., 1989, First Order Compiler: a Deterministic Logic Program Synthesis Algorithm. *J. Symbolic Computation, 8,* 605–27.

Shapiro, E., 1983, A Subset of Concurrent Prolog and Its Interpreter. *Technical Report TR-003.* ICOT, Tokyo.

Tärnlund, S.-Å., 1992, Reform. In *Massively Parallel Reasoning Systems* (ed. J. A. Robinson). To be published by MIT Press, Cambridge, Mass.

Warren, D. H. D., 1983, An Abstract Prolog Instruction Set. *SRI Tech. Note 309.* SRI International, Menlo Park, Calif.

[5] The difference from the *elt* predicate we proposed earlier is that *arg* operates on compund terms, rather than arrays, and that indexing is one-based. This is of course related to the use of the arg predicate in Prolog.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

An Implementation for a Higher Level Logic Programming Language

Anthony S.K. Cheng* Ross A. Paterson†
Software Verification Research Centre
Department of Computer Science
The University of Queensland
4072, Australia

Abstract

For representing high level knowledge, such as the mathematical knowledge used in interactive theorem provers and verification systems, it is desirable to extend Prolog's concept of data object. A basic reason is that Prolog data objects—Herbrand objects—are terms of a minimal object language, which does not include its own object variables, or quantification over those variables.

Qu-Prolog (Quantifier Prolog) is an extended logic programming concept which takes as its data objects, object terms which may include free or bound occurrences of object variables and arbitrary quantifiers to bind those variables. Qu-Prolog is unique in allowing its data objects to include free occurrences of object variables.

In this paper the design of the abstract machine for Qu-Prolog is given. The underlying design of the machine reflects the extended data objects and Qu-Prolog's unification algorithm.

1 Introduction

The extended logic programming language Qu-Prolog (Quantifier Prolog) [Cheng *et al.* 1991, Paterson and Hazel 1990, Paterson and Staples 1988, Staples *et al.* 1988a, Staples *et al.* 1988b] has been designed to provide improved support for language processing applications such as interactive proof systems. Its main feature is that it supports higher level symbolic data types than does Prolog. In particular, the data objects which Qu-Prolog reasons about are terms of a full first order logic syntax, which includes both object level variables and arbitrary bindings of object level variables.

The language λProlog [Miller and Nadathur 1986], which extends Prolog with typed *lambda*-terms, may also be used for these purposes. Qu-Prolog is weaker, in that its terms correspond to second-order *lambda*-terms; substitution is provided, but not application of terms to terms. However, in Qu-Prolog, as in traditional notation, term variables may refer to open terms, raising further questions of whether an object level variable occurs free in a term, or whether two object level variables are distinct.

The Qu-Prolog Abstract Machine (QuAM) [Cheng and Paterson 1990] is designed as the target for compilation of the logic programming language Qu-Prolog. QuAM is developed from the Warren Abstract Machine (WAM). New mechanisms are introduced to handle quantified terms and substitutions and flexible programming in Qu-Prolog. This paper presents the basic structure of the language and describes its implementation.

The main features of Qu-Prolog are described in section 2. In section 3, unification is extended to Qu-Prolog terms. The design of QuAM is given in section 4. Some examples are given in section 5. It is assumed that the reader has some knowledge of the design of WAM [Aït-Kaci 1990, Warren 1983] and the compilation of logic programming languages.

2 Qu-Prolog – the Language

Qu-Prolog has Prolog as a subset, and uses Edinburgh Prolog syntax for constants and structures, and for ordinary variables which are intended to range over arbitrary object level terms. These variables will be referred to as *meta* variables, in recognition of the meta level status of the Qu-Prolog language relative to the object language. In addition, Qu-Prolog introduces syntax to represent object level variables and quantifiers, as follows.

Qu-Prolog has other features not described here. These include persistent variables, which are used to manage incomplete information in the database. For a description of persistent variables and their implementation, see [Cheng and Robinson 1991].

2.1 Object Variables

Since object level variables are simply part of the object level syntax, it might seem natural to name them at the Qu-Prolog (meta) level by constants. Instead, Qu-Prolog refers to object level variables only by a type of Qu-Prolog (meta) level variable, called object-var variables. The semantics of object-var variables is that they range over object level variables. The success of this approach reflects the common intuition that object level variables are interchangeable.

*e-mail: chena@cs.uq.oz.au
†present address: Department of Computing, Imperial College, London SW7.

The phrase 'object variable' is commonly used to abbreviate 'object-var variable' since it has no other use in describing Qu-Prolog syntax. For an occasional reference to a variable of the object language, the phrase 'object level variable' will be used.

Qu-Prolog object variables have the same lexical conventions as constants. In order to distinguish them, object variable notations must be declared by `object_var/1`. The declaration convention is that an explicit declaration of an object variable name also implicitly declares all variant names derived by appending an underscore followed by a positive integer. The standard library declares the atoms `x`, `y` and `z` as object variables.

As each object variable is intended to range over all object level variables, it is important to know whether two object variables denote different object level variable. This information can be supplied implicitly or by explicit use of the predicate `distinct_from/2`. For example, `x distinct_from y` asserts that `x` and `y` do not denote the same object level variable. By default, all object variables occurring in the same clause/query are distinct from each other.

Remark: In fact Qu-Prolog makes internal use of some meta level constants representing object level variables. These terms, called *local* object variables, are mentioned below but they are not discussed here in detail. Their key role is as 'new' variables, for use when changing bound variables. This newness is implemented by a convention that they are excluded from instantiations of user accessible meta variables and object-var variables.

2.2 Quantifiers

Qu-Prolog can reason about object level terms which include arbitrary quantifiers, in much the same way that Prolog can reason about terms which include arbitrary function symbols. The user declares quantifier notations as needed. Thus it is possible to have representations of $\int$ for integral calculus as well as $\forall, \exists$ for first order logic.

Distinct quantifier notations in Qu-Prolog represent distinct object level quantifiers. Qu-Prolog uses the traditional prefix notation for quantified terms. Quantifiers are declared explicitly by executing

$$op(Precedence, quant, Q)$$

where Q is the representation for the quantifier; Q must have the same lexical structure as a Prolog constant.

2.3 Substitutions

Throughout logical reasoning, the need for substitutions arises naturally. Qu-Prolog directly supports parallel substitution for free occurrences of object level variables.

The syntax for substitutions in Qu-Prolog is

$$[t_1/x_1, \ldots, t_n/x_n] * term$$

where $x_1, \ldots, x_n$ are object variables and $t_1, \ldots, t_n$ are arbitrary Qu-Prolog terms.

Qu-Prolog substitutions are evaluated at unification time, in accordance with the standard concept of correct substitution into quantified terms, which substitutes only for free occurrences of variables and which changes bound variables to avoid capture of free variables from the substituted terms. For a term $s_1 * \ldots * s_n * y$ where $s_1, \ldots, s_n$ is a sequence of substitutions, the substitutions are applied from right to left. That is, s_n is applied to y first. The effect of applying a substitution to a term can be observed with this example:

$$s * [t_1/x_1, \ldots, t_n/x_n] * y.$$

After applying the rightmost substitution, the result will be:

- $s * t_i$ if for some $i = 1, \ldots, n, x_i = y$, or
- $s * y$ if for all $i = 1, \ldots, n, x_i$ *distinct_from* y.

It is also possible that there is insufficient information at a particular stage to determine which of these cases applies. In that case evaluation of the substitution will be delayed. That may lead to delaying of unification subproblems, perhaps extending beyond the current unification call.

As well as substitutions appearing in user inputs, the system can generate substitutions via unification. For example, the problem *lambda x A = lambda y B* has the solution $A = [x/y] * B$.

2.4 Example

As a small example, we give a λ-calculus evaluator in Qu-Prolog. The terms of the λ-calculus are transcribed directly, except that we use the infix constructor `@` for application. First, we declare the quantifier `lambda` and the application operator:

```
?- op(700, quant, lambda).
?- op(600, yfx, @).
```

Now the following predicate defines the structure of λ-terms:

```
lambda_term(x).
lambda_term(A@B) :-
        lambda_term(A),
        lambda_term(B).
lambda_term(lambda x A) :-
        lambda_term(A).
```

For example, the following are λ-terms:

```
x
lambda x x
(lambda x x)@y
lambda x (x@y)
```

Note that λ-terms may contain free object variables.

Now we can define a single-step reduction predicate on λ-terms:

```
?- op(800, xfx, =>).
(lambda x A)@B => [B/x]*A.
A@B => C@B :- A => C.
A@B => A@C :- B => C.
lambda x A => lambda x B :- A => B.
```

The first clause is the well-known β-rule. The others allow rewrites anywhere in the expression. If desired, we could also add the η-rule:

```
lambda x A@x => A :- x not_free_in A.
```

The full reduction relation in the usual reflexive, transitive closure of the single-step reduction predicate:

```
?- op(800, xfx, =>*).
A =>* C :- A => B, !, B =>* C.
A =>* A.
```

3 Unification

Qu-Prolog extends Prolog unification to cover the new data objects in the language. Two terms are unified if they are equivalent up to changes of bound variables (α equivalent). Since unification for Prolog terms is not changed (except that Qu-Prolog includes occurs checking), our discussion will concentrate on the new features.

Because Qu-Prolog unification is more difficult than ordinary unification—it is not decidable, but semi-decidable [Paterson 1989]—we often encounter sub-problems which cannot be solved at that point in the computation, but we may be able to make further progress on them later. Such sub-problems are delayed, waiting for a relevant variable (or variables) to be instantiated, at which point they are re-attempted. If the sub-problems remain unsolved at the end of query solution, they are displayed as part of the answer. This approach has proved practical in our implementation.

We have also found it useful to delay sub-problems to avoid branching. As a simple example, consider the unification problem $[X/y]*Z = c$, where c is a constant (a similar situation arises with structures). The unification can succeed in one of two ways:

- Imitation: $Z = c$. Here the substitution has a null effect on Z.
- Projection: $Z = y$ and $X = c$.

Hence it is impossible to determine a unique most general unifier. Rather than branch the unification problem, Qu-Prolog delays it until the binding of Z is known.

3.1 Object Variables

Since an object variable is intended to range over object level variables, and since object variables are the only Qu-Prolog terms of this type, an object variable can be instantiated only to another object variable. Further, unification fails if the object variables denote distinct object level variables. Also, whenever a meta variable is unified with an object variable, the meta variable is bound to the object variable.

3.2 Quantifiers

To motivate the treatment of unification for quantified terms, consider

$$lambda\ x\ x \quad = \quad lambda\ y\ y$$

Intuitively, the two terms are unifiable without instantiation of x or y, because the terms are the same up to change of bound variable. To unify x and y would be incorrect: the two terms are α equivalent even if x and y denote distinct object level variables. Hence during quantifier unification, Qu-Prolog uses substitution to rename the bound variables to a common bound variable. The bound variable must not appear in the unified terms. This is where the local object variables mentioned previously are used. In general, a problem of the form $q\ x\ t = q\ y\ t'$ is reduced to

$$[\nu/x]*t \quad = \quad [\nu/y]*t'$$

for some new local object variable ν, and unification continues. Here is how the approach applies to the example, (ν is a local object variable).

$$\begin{aligned} lambda\ x\ x &= lambda\ y\ y \\ lambda\ \nu\ [\nu/x]*x &= lambda\ \nu\ [\nu/y]*y \\ [\nu/x]*x &= [\nu/y]*y \\ \nu &= \nu \\ (success) & \end{aligned}$$

A substitution containing local object variables, when applied to a meta variable, may be removed by a rule called *inversion*: a problem of the form $[\nu/x]*X = t$ is reduced to the two problems

$$X \quad = \quad [x/\nu]*t,\ x\ not\_free\_in\ t$$

For example, we have the following reduction:

$$\begin{aligned} lambda\ x\ A &= lambda\ y\ y \\ [\nu/x]*A &= [\nu/y]*y \\ A &= [x/\nu]*[\nu/y]*y, \\ & \quad x\ not\_free\_in\ [\nu/y]*y \\ A &= x,\ x\ not\_free\_in\ \nu \\ A &= x \end{aligned}$$

Unification produces the answer $A = x$.

As a further example, consider

$$lambda\ x\ A = lambda\ y\ x$$

Since x does occur free on the right and cannot occur free on the left, this unification problem should fail. In Qu-Prolog unification, that failure is detected when, at the time of calculation of $A = [x/\nu]*[\nu/y]*x$, the constraint $x\ not\_free\_in\ [\nu/y]*x$ is generated and tested; and after substitution evaluation, the test fails.

Such *not_free_in* constraints may be delayed if they cannot be immediately decided. For example, the unification problem

$$lambda\ x\ A \quad = \quad lambda\ y\ [x/z]*Z$$

gives the solution

$$A = [x/\nu] * [\nu/y] * [x/z] * Z$$
$$provided:$$
$$x\ not\_free\_in\ [\nu/y] * [x/z] * Z$$

In the absence of further information about Z, the *not_free_in* test must be delayed.

3.3 Occurs Checking

Unlike Prolog, occurs checking is included as standard in Qu-Prolog unification. However, it is not always possible to determine whether a variable occurs in the final form of a term. For example, it is impossible to determine whether X occurs in the term $[X/y] * Z$ without knowing more information about Z. If Z is bound to y, X occurs in the term. On the other hand, if Z is bound to a constant c, X does not occur.

Thus, if we are considering a sub-problem of the form $X = t$, we cannot always reduce the problem. We use two conservative syntactic conditions:

- If X occurs in t outside of any substitution, and t is not of the form $s * X$, the unification fails, for the X must appear in t no matter how other variables are instantiated.
- If X does not appear in t, including substitutions, X is instantiated to t.

If neither of these conditions is met, the unification sub-problem must be delayed, pending further instantiation of X.

4 The Qu-Prolog Abstract Machine

One of the design criteria for QuAM is that the efficiency of ordinary Prolog queries within Qu-Prolog must be maintained wherever possible. Thus, most of the features of WAM are retained and the description below will concentrate on the differences between QuAM and WAM. The current implementation of QuAM differs from the present description in that it uses an experimental representation for structures, intended for future enhancements to the Qu-Prolog language with higher-order predicates and multiple-place quantifiers. The present paper focuses on other aspects of the machine, so we omit these details here, assuming a WAM-like representation of structures. Because of the difference of the representation of the structures, no performance evaluation will be given. A description of the current implementation can be found in [Cheng and Paterson 1990].

4.1 Data Objects

Unbound Variables

Because of the association with delayed problems described below, the representation of a self reference cell for unbound variables as in WAM is inapplicable. A data cell with a VARIABLE tag is used to indicate an unbound variable in Qu-Prolog. The value field of the data cell contains a pointer to a list of delayed problems associated with the variable (Figure 1). Although the representation of variables is different to WAM, the classification into temporary and permanent variables, the age determining method and the rules of binding a variable are retained.

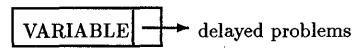


Figure 1: An Unbound Qu-Prolog Variable

The REFERENCE tag is retained to indicate that one variable is bound to another one. When two heap variables are bound together, the one created more recently points to the one created earlier on the heap. The delayed problems from the younger one are appended to those of the older one.

Unbound Object Variables

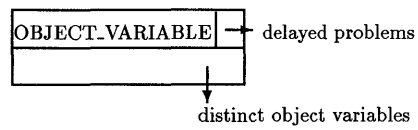


Figure 2: An Unbound Object Variable

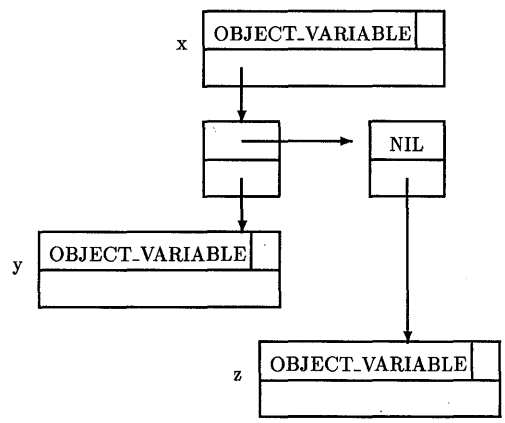


Figure 3: x *distinct_from* y and x *distinct_from* z

A separate tag OBJECT_VARIABLE is given to the object variables to distinguish its function from the variables. The value field has the same purpose as the value field in variables. The second cell in an object variable points to a list of object variables from which it is distinguished (Figures 2, 3). Rather than record all object variables in the distinctness list, an ALL_DISTINCT tag is placed in this cell for local object variables.

The classification method, the binding rules and the age determining method used for variables is also applied to object variables.

The OBJECT_REFERENCE tag indicates that an object variable is bound to another object variable. When two object variables are bound together, the distinctness information from both object variables are merged together and placed in the older object variable and the delayed problems will be woken up.

Quantified Terms

Qu-Prolog currently allows 1 place quantifiers (*i.e.* quantifiers with one bound variable) only. To represent quantified terms in Qu-Prolog, a tag QUANTIFIER is introduced, analogous to the STRUCTURE tag of the WAM. Such a value points to a three contiguous cells, containing the quantifier atom, a reference to the bound object variable, and the body of the quantified term (Figure 4).

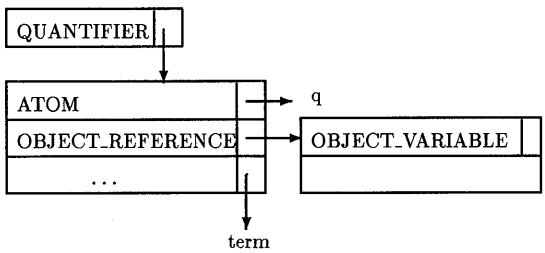


Figure 4: Quantified Term q x term

Substitution Operators

In QuAM, an application of one or more substitutions to a term is represented as a data cell, marked with a SUBSTITUTION_OPERATOR tag and pointing to a pair of cells. The first cell contains a pointer to the list of substitutions, while the second is a data cell denoting the term (Figure 5). The list of substitutions is stored in reverse order, with the innermost substitution at the front, to simplify evaluation.

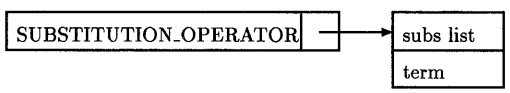


Figure 5: *sub* $*$ *term*

An ordinary parallel substitution is represented as a data cell with the property tag, containing a pointer to a pair of cells. The first of the cells is a pointer to the parallel substitution, while the second represents the rest of the substitution list. A parallel substitution involving n pairs of object variables and terms is represented as a block of $2n + 1$ cells; the first contains the size of the substitution, while the renaming $2n$ cells refer to the object variables and terms. Again the substitution pairs are stored in the reverse order for easy evaluation (Figure 6).

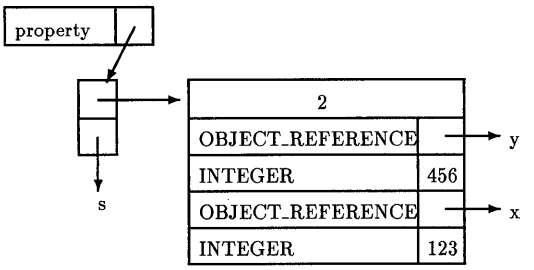


Figure 6: A Parallel Substitution $s * [123/x, 456/y]$

Each substitution list contains a marker describing the property of the substitution list. It is used during unification to assist the determination of whether or not the unification can be solved by projection. In general, a problem of the form $s * A = t$, where t is a constant, structure, quantified term or object variable, can always be reduced by imitation. If s is known not to contain any terms of the same top-level structure as t, then the problem cannot be solved by projection. Thus branching is eliminated and we can proceed by imitation. Otherwise, the unification problem will be delayed to avoid branching. In most cases, the whole substitution list must be examined in order to eliminate projection. In special cases, the marker will contain enough information to make a complete search unnecessary.

It is also convenient to know if a substitution list consists solely of renamings generated by quantifier unification, as such a list can be safely inverted. Thus, each substitution list is marked as one of:

- invertible: the substitution list consists solely of renamings.
- object variables only: the substitution list is not invertible, but its range contains only object variables.
- others: the range of the substitution list contains constants, structures, quantifiers or meta variables.

4.2 Data Areas

QuAM supports the same data areas as in WAM. The heap provides space to store data objects as well as the distinctness information and linking cells required for delayed problems. The local stack holds choice points and environments. The choice points are enlarged to reflect the extra data areas and registers.

Because the delayed problems list and distinctness information must be reset to their previous value upon backtracking, the method of trailing (*i.e.* resetting the address to null) used in WAM is inapplicable. Each entry in the trail is extended to be a pair of addresses and

previous values to provide extra information for backtracking.

In addition to the standard WAM data areas, a delayed problems stack that holds any delayed problem generated during unification is provided. Apart from containing pointers to the arguments for the delayed problem, it has a type tag and a solved tag. The type tag indicates whether the delayed problem is a unification or a *not_free_in* problem (Figure 7). The solved tag is set whenever the problem is solved.

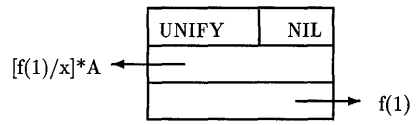


Figure 7: Delayed Problem: $[f(1)/x] * A = f(1)$

When a query is solved, any delayed problem that remains is printed as a constraint to the solution. Storing the delayed problems in a separate area allows fast access to the problems when the solution is printed.

4.3 Registers

There are a few extra registers used in QuAM:

- the top of the delayed problems stack,
- a list of formerly delayed problems that have been woken up,
- The *substitution pointer* register points to the entry in the parallel substitution where the next component is to be added.

As well as the X registers, there is an associated set of registers, known as the XS (X substitution) registers, which hold the substitution of a term when the substitution and the term of a SUBSTITUTION_OPERATOR data cell are broken up during dereference. This procedure enables the substitution to be passed from the outer structure to the inner terms effectively.

Because each Y register is one data cell in size, and an OBJECT_VARIABLE is two cells in size, a Y register cannot hold an OBJECT_VARIABLE directly, and instead contains a reference to an OBJECT_VARIABLE in the heap.

4.4 Instruction Set

For each new data object provided in QuAM, there are *put* and *get* instructions to build and unify the data object. The new instructions are:

put_object_variable X_i
: Create a new object variable on the heap, and place a reference to it in X_i.

get_object_variable X_i X_j
: Copy the object variable reference in X_j into X_i.

put_object_value X_i X_j
: Copy the object variable reference in X_i into X_j.

get_object_value X_i X_j
: Unify XS_j, X_j with the object variable referenced by X_i.

put_quantifier q X_i X_j X_k
: Construct a quantified term, with quantifier q, bound object variable X_i and body X_j, and place a reference to it in X_k.

get_quantifier q X_i X_j X_k
: Match the term in XS_k, X_k with a quantified term, with quantifier q and bound object variable X_i. The body is placed in XS_j, X_j.

In each of the last two instructions, the register X_i must have been previously set to an object variable.

Note that some of these instructions use the XS registers, while others ignore them, expecting any substitution to be incorporated into the term in the X register. Thus during head matching substitutions are conveniently accessible in the substitution registers, allowing efficient dereferencing, and sharing of substitutions. However, if such a value is to be a sub-term, its substitution (if any) must be re-incorporated into the term.

There is a set of *put* instructions to build substitutions, but no corresponding set of *get* instructions. This is because a substitution occurring in the head must be built in the same way as if it had occurred in the body, and then the substituted term must be unified with the corresponding head argument (or component). The instructions available are:

put_subs_operator X_i X_j
: Combine XS_j and X_j into a SUBSTITUTION_OPERATOR, and place a reference to it in X_i.

put_empty_subs X_i
: Set XS_i to an empty substitution.

put_parallel_subs n X_i
: Prepend a parallel substitution, consisting of n pairs (each supplied with the next instruction), to XS_i.

put_parallel_subs_pair X_i X_j
: Add a pair, substituting X_j for the object variable referred to by X_i, to the parallel substitution currently under construction.

put_subs X_i X_j
: Transfer a substitution from XS_i to XS_j.

set_object_property X_i
: Set the property tag on XS_i to "object variables only".

determine_property X_i
: determine the property tag of XS_i.

The only new procedural instructions are:

do_delayed_problems
: Solve any woken delayed problems. This instruction is executed after the head has been matched.

not_free_in
: Perform a *not_free_in* test during quantifier unification.

4.5 Dereference

Because of the presence of substitution, additional operations are included into the `dereference` algorithm. The substitutions are evaluated during `dereference` whenever possible. Given an object variable, the substitution will map the object variable to its value. Depending on the type of the data object encountered in the term, `dereference` also simplifies the substitution before returning.

5 Examples

A number of small examples are given here to highlight the design differences between QuAM and WAM.

5.1 Quantified Terms

Quantified terms are constructed in a similar fashion to the unary structures, except for the object variable. The following sequence of instructions shows how a quantified term `lambda x x` is built in register X_1:

```
put_object_variable X0                % x
put_quantifier lambda X0 X0 X1
```

Matching a quantified term is slightly more complicated than structure matching. Apart from matching the term from outside in (*i.e.* match the quantifier before matching the body), it must establish that the bound variable of the quantified term in the head does not occur freely in the body of the quantified term from the query. Thus, a `not_free_in` instruction must be executed before the quantifier matching is performed. The following instructions match the argument X_0 with the head argument `(lambda x A)@B`:

```
get_structure @/2 X0
unify_variable X2              % lambda x A
unify_variable X0                % B
put_object_variable X3           % x
put_empty_subs X3
not_free_in X3 X2
get_quantifier lambda X3 X2 X2  % A
```

5.2 Substitutions

QuAM is designed to create substitutions independent of the term. The term is created before the substitution. The example `[a / x, b / y] * A` illustrates this property.

```
put_variable X0 X0            % A
put_empty_subs X0
put_object_variable X1        % y
put_atom 'b' X2
put_object_variable X3        % x
put_atom 'a' X4
put_parallel_subs 2 X0        % * A
put_parallel_subs_pair X1 X2 % [b/y] * A
put_parallel_subs_pair X3 X4 % [a/x,b/y]*A
determine_property X0
```

If the substitution is nested inside another term, an extra step is needed. A SUBSTITUTION_OPERATOR data object is created to group the substitution and its associated term together. To construct the term `f([a/x, b/y] * A)`, the following additional instructions are required:

```
put_subs_operator X0 X0  % group together
put_structure f/1 X1
unify_value X0
```

Whenever a substitution is associated with a term in the head, that term together with the substitution will be built by *put* instructions and general unification will be called. For example, consider the following clause from the λ-calculus evaluator:

```
(lambda x A)@B => [A/x]*B.
```

In section 5.1 above, we gave the translation of the matching of the first argument, leaving `x` in X_3, `A` in X_2 and `B` in X_0. The following instructions match the second argument (in X_1):

```
put_subs_operator X0 X0 % group together B
put_subs_operator X2 X2 % group together A
put_empty_subs X0              % *B
put_parallel_subs X0 1         % *B
put_parallel_subs_pair X3 X2  % [A/x]
determine_property X0
get_value X0 X1  % unify with the argument
```

Note that `A` and `B` must both be combined with their substitutions, if any. In the case of `A`, this allows the value to fit into a cell in the substitution pair. In the case of `B`, the substitution must be incorporated into the value, and the substitution register set to empty, so that the new substitution will be outside any existing substitutions.

If the substitution is nested within another term, the outer term is matched by the *get* instructions, while the substitution is built and unified with the appropriate component.

6 Conclusions

QuAM has been implemented in C under the SUN 4 environment. The compiler was initially implemented in NU-Prolog [Naish 1986], and subsequently transferred to Qu-Prolog, which includes Prolog as a subset.

Qu-Prolog, including the extensions and features mentioned here, has been motivated particularly by the need to rapidly prototype interactive proof systems, and currently it is the implementation language for a substantial experimental proof system [Robinson and Tang 1991].

Acknowledgements

John Staples, Peter Robinson, Gerard Ellis and Dan Hazel have made substantial contributions to the design and implementation of QuAM. This research was supported by the Australian Research Council.

References

[Aït-Kaci 1990] H. Aït-Kaci, The WAM: a (Real) Tutorial, Report No. 5, Paris Research Laboratory (PRL), France, 1990.

[Cheng and Robinson 1991] A.S.K. Cheng and P.J. Robinson, An Implementation for Persistent Variables in Qu-Prolog 3.0, Software Verification Research Centre, Department of Computer Science, University of Queensland, 1991.

[Cheng and Paterson 1990] A.S.K. Cheng and R.A. Paterson, The Qu-Prolog Abstract Machine, Technical Report No. 149, Key Centre for Software Technology, Department of Computer Science, University of Queensland, February 1990.

[Cheng *et al.* 1991] A.S.K. Cheng, P.J. Robinson and J. Staples, Higher Level Meta Programming in Qu-Prolog 3.0, *Proc. of 8th International Conference on Logic Programming*, Paris, June 1991.

[Miller and Nadathur 1986] D.A. Miller and G. Nadathur, Higher-order Logic Programming, *Proc. of 3rd International Conference on Logic Programming*, London, July 1986.

[Naish 1986] L. Naish, Negation and Quantifiers in NU-Prolog, *Proc. of 3rd International Conference on Logic Programming*, London, July 1986.

[Paterson 1989] R.A. Paterson, Unification of Schemes of Quantified Terms, Technical Report No. 154, Key Centre for Software Technology, Department of Computer Science, University of Queensland, Dec. 1989.

[Paterson and Hazel 1990] R.A. Paterson and D. Hazel, Qu-Prolog 3.0 – Reference Manual, Technical Report No. 195, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1990.

[Paterson and Staples 1988] R.A. Paterson and J. Staples, A General Theory of Unification and Solution of Constraints, Technical Report No. 90, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1988.

[Staples *et al.* 1988a] J. Staples, P.J. Robinson, R.A. Paterson, R.A. Hagen, A.J. Craddock and P.C. Wallis, Qu-Prolog: an Extended Prolog for Meta Level Programming, *Proc. of the Workshop on Meta Programming in Logic Programming*, University of Bristol, June 1988.

[Staples *et al.* 1988b] J. Staples, R.A. Paterson, P.J. Robinson and G.R. Ellis, Qu-Prolog: Higher Level Symbolic Computation, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1988.

[Robinson and Tang 1991] P.J. Robinson and T.G. Tang, The Demonstration Interactive Theorem Prover: Demo2.1, Technical Report 91-4, Software Verification Research Centre, University of Queensland, September 1991.

[Warren 1983] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, 1983.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Implementing Prolog Extensions : a Parallel Inference Machine

Jean-Marc Alliot* (alliot@irit.fr)
Andreas Herzig† (herzig@irit.fr)
Mamede Lima-Marques‡ (mamede@irit.fr)

Institut de Recherche en Informatique de Toulouse
118 Route de Narbonne
31062 Toulouse cedex, France

Abstract

We present in this paper a general inference machine for building a large class of meta-interpreters. In particular, this machine is suitable for implementing extensions of Prolog with non-classical logics. We give the description of the abstract machine model and an implementation of this machine in a fast language (ADA), along with a discussion on why and how parallelism can easily increase speed, with numerical results of sequential and parallel implementation.

1 Introduction

In order to get closer to human reasoning, computer systems, and especially logic programming systems, have to deal with various concepts such as time, belief, knowledge, contexts, etc... Prolog is just what is needed to handle the Horn clause fragment of first order logic, but what about non-classical logics? Just suppose we want to represent in Prolog time, knowledge, hypotheses, or two of them at the same time; or to organize our program in modules, to have equational theories, to treat fuzzy predicates or clauses. All these cases need different ways of computing a new goal from an existing one.

Theoretical solutions have been found for each of the enumerated cases, and particular extensions of Prolog have been proposed in this sense in the literature. Examples are [BK82], [GL82], Tokio [FKTMO86], N-PROLOG [GR84], Context Extension [MP88], Templog [Bau89], Temporal Prolog [Sak89], and [Sak87].

For all these solutions it is possible to write specific meta-interpreters in Prolog that implement these non-classical systems ([SS86]). But there are disadvantages of a meta-interpreter: lower speed and compilation notoriously inefficient. If we want to go a step further, and to write proper extensions of Prolog, then the problem is that costs for that are relatively high (because for each case we will lead to write a new extension), and we are bound to specific domains: we can only do temporal reasoning, but not reasoning about knowledge (and what if we want to add modules?).

Our aim is to define a framework wherein a *superuser* can create easily "his" extension of Prolog. This framework should be as general as possible. Hence, we must provide a general methodology to implement non-classical logics.

There are four basic assumptions on which our frame is built:

1. to keep as a base the *fundamental logic programming mechanisms* that are backward chaining, depth first strategy, backtracking, and unification,
2. to *parametrize the inference step*: it is the superuser who specifies how to compute the new goal from a given one, and he specifies it in a logic form.
3. to be able to *rewrite goals.*
4. to *select clauses "by hand".*

Points (2) and (3) postulate a more flexible way of computing goals than that of Prolog, where first a clause is selected from the program, then the Robinson unification algorithm is applied to the clause and the head of the goal, and finally a new goal is produced.

Point (4) introduces a further flexibility: the superuser may select clauses that do not unify exactly with the current goal, but just "resemble" it in some sense. Even more, if the current goal contains enough information to produce the next goal, or if we just want to simplify a goal or to reorder literals we don't need to select a fact clause at all.

The assumptions (1) and (2) were at base of the development of a meta-level inference system called MOLOG [FdC86], [ABFdC+86], [BFdCH88],

*Supported by the Centre d'Etudes de la Navigation Aérienne, France
†Supported by the Medlar Esprit Project
‡Supported by CAPES - Brasil

[Esp87b], [Esp87a]. The inference machine that is presented in this paper is a complete rewriting of MOLOG realizing assumption (4). It has been developped at IRIT ([Bri87] and [AG88]).

A formal specification of the inference mechanism called TIM : *Toulouse Inference Machine*, together with various examples, has been published in [BHLM91]. Here, in this paper, we present the TARSKI : *Toulouse Abstract Reasoning System for Knowledge Inference*, which is an abstract machine in which the inference mechanism can be implemented. In the preliminary version of this work nothing has been said about abstract machine and implementation, and the specifications are being defined more clearly now.

TARSKI was designed to implement parallelism (see sections 6 and 7). For example, for a given definite fact and goal clauses, more than one rule is possible. In this case it is possible to use a different processor for each rule. The parallel machine was developped and differents solutions was be done.

2 Horn clauses

The base of the language is that of Prolog. That language can (but need not) be enriched with *context operators* if one wants to mechanize non-classical logics.

Characteristically, non-classical logics possess symbols with a particular behaviour. These symbols are

- either classical connectors with modified semantics (e.g. intuitionist, minimal, relevant, paraconsistent logics)
- or new connectors called context operators (*necessary* and *possible* in modal, *knows* in epistemic, *always* in temporal, *if* in conditional logics).

Example In epistemic logics, the context operators are *knows* and *comp*, and

> *knows(a):P* means that agent a knows that P
>
> *comp(a):P* means that it is compatible with a's knowledge that P

Hence inference engines for non-classical logics must reckon for the particular behaviour of some given symbols. These properties will be handled by built-in features of the inference engine.

The *conditio sine qua non* for logic programming languages is that they possess an implicational symbol to which a procedural sense can be given. To define a programming language it's less important if this is material implication or not, but it's rather the dynamic aspect of implication that makes the execution of a logic program possible. That is why the TIM language is built around some arrow-like symbol.

We suppose the usual definition of *terms* and *atomic formulas* of logic programming. Intuitively, *TIM Horn Clauses* are formulas built with the above connectors, such that dropping the context we may get a classical Horn clauses. Now for each logic programming language we suppose a particular set of context operators. This set depends on the logic programming language we want to implement, e.g. in epistemic logic it is {*knows, comp*} and in temporal logic it is {*always, sometimes*}. Formally we define by mutual recursion:

Definition 2. 1 - contexts

$\mathtt{m}(\mathtt{t_1}, ..., \mathtt{t_n})$ is a context if $\mathtt{m}$ is a context operator $\mathtt{n} \geq 0$, and for $1 \leq \mathtt{i} \leq \mathtt{n}$ every $\mathtt{t_i}$ is either a term or a definite clause.

Definition 2. 2 - goal clauses

$?P$ is a goal clause if P is an atomic formula

$?(G \wedge F)$ is a goal clause if $?G$, $?F$ are goal clauses

$?MOD : F$ is a goal clauses if $?F$ is a goal clause and MOD is a context

Definition 2. 3 - definite clauses

P is a definite clause if P is an atomic formula

$MOD : F$ is a definite clause if F is a definite clause and MOD is a context

$F \leftarrow G$ is a definite clause if F is a definite clause and G is a goal clause

Definition 2. 4 - TIM Horn clause

A TIM Horn clause (or Horn clause for short) is either a goal clause or a definite clause. Note that Horn clauses may contain several implication symbols.

We shall also use the term *Modal Horn clauses* if we are speaking of a modal logic. A set of definite clauses is called a *database*.

In the following sections we shall use the definition of the head of a Horn clause.

Definition 2. 5 - Head of a Horn clause

- H is a head of H.
- H is a head of $F \wedge G$ if H is a head of F.
- H is a head of $F \leftarrow G$ if H is a head of F.
- H is a head of $MOD : F$ if H is a head of F.

3 Writing meta-interpreters

3.1 General Mechanism

Just as in Prolog, to decide whether a given goal follows from the database essentially means to compute step by step new subgoals from given ones. In our

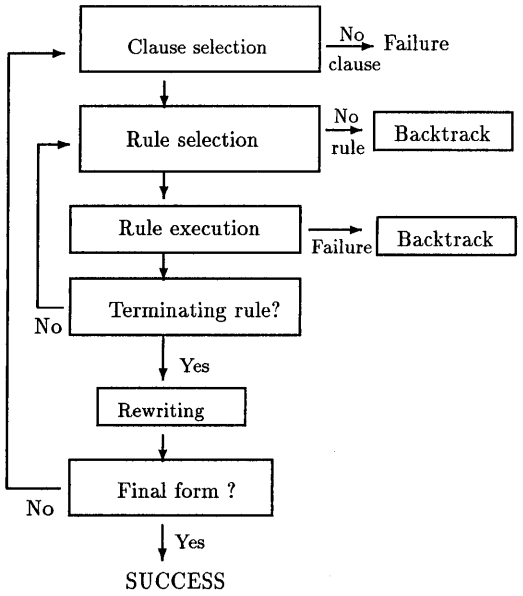


Figure 1: General mechanism of the TIM machine

case, the computation of the new subgoal is specified by the superuser. The general inference mechanism is described in figure 1. There are five steps:

Clause selection: We select a clause to solve the first sub-goal of the question.

Rule selection: We select a rule to be applied to the current clause and the current question.

Rule execution: The execution of the rule "modifies" the current clause and the current question and builds a *resolvent.*

Rewritting of the resolvent: When we reach a termination rule, we rewrite the resolvent into a new question.

End of resolution : A resolution is completed when we reach a *final form* : the goal clause *true.*

This system is doubly non determinist, because we have both a clause selection (as in standard Prolog) and a rule selection.

We are going in the next sections to explain how this mechanism can be implemented. In subsection 3.2, we will discuss rule selection and execution, in subsection 3.4 rewriting and in subsection 3.3 clause selection. In section 6, we will come back to rule selection to show how efficient mechanism can be used to improve resolution speed.

3.2 Selecting and Executing Inference Rules

An inference rule is of the form : $A, ?B \vdash ?C$ where A is a definite clause and B, C are goal clauses. It can be read: If the current goal clause unifies with B and the selected database clause unifies with A then a new goal can be inferred that is unified with C. In the style of Gentzen's sequent calculus, inference rules can be defined recursively as follows:

$$\frac{A, ?B \vdash ?C}{A', ?B' \vdash ?C'}$$

where A,A' are definite clauses and B, C, B', C' are goal clauses. As usual in metaprogramming, objects of the object language are represented by variables of the metalanguage[1].

Essentially, what can be tested here is any condition on the form of A,A', B, C, B', C', or on the existence of a database clause of a certain form. E.g. we can let an inference rule depend on the (non-)existence of some clause in some particular module of the database.

In the recursive definition the following conditions must be met[2]:

- $var(A') \subset var(A)$
- A' is a head of A or A is a head of A'
- C' is a variable
- C' is a head of C

A special category of inference rules are *reflexive rules*:

$$\frac{\text{true}, ?B \vdash ?C}{A', ?B' \vdash ?C'}$$

These rules use the special fact true. The conditions that these rules must meet are:

- A' is either:
  - a variable[3], or
  - any definite clause constructed from the variables in B and C and constants.
- C' is a variable
- C' is a head of C

Partial termination rules are written:

$$A, ?B \vdash ?C \text{ if Condition}$$

They end the recursivity in resolution.

These are some examples : the *Prolog rule for goal conjunctions*:

$$\frac{A, ?B \wedge C \vdash ?D \wedge C}{A, ?B \vdash ?D}$$

[1]To be correct, the real form of inference rule is a little different : a procedural condition expressed with elementary functions of the abstract machine (see section 5) can be added. This enables a more precise control over execution.

[2]It is these conditions on the form of the inference rules that warrant the efficiency of the implementation.

[3]This variable will be unified with a new fact taken in the clause base

the *Prolog rule for implications in database clauses*:

$$\frac{A \leftarrow B, ?C \vdash ?B \wedge D}{A, ?C \vdash ?D}$$

the *Prolog partial termination rule* is:

$$p, ?p \vdash ?true$$

Note that here we make use of unification. These three rules are exactly what is needed to implement Prolog.

To summarize, the execution of an inference rule modifies the current fact and the current question and constructs a resolvent. *The resolvent has the same structure than the question or any other fact.* Partial resolution is achieved when we reach a *partial termination* rule.

How rules are selected is defined by the user. We will see in the section 6 how this is exactly done. For the moment, we say that rules are taken in the order they appear in the rule base.

3.3 Rewriting the Resolvent into a New Question

As soon as we have reached a *partial termination* rule, we *rewrite* the resolvent to create the new question to solve. Rewriting is useful not only in order to simplify goals, but also in order to eliminate the *true* predicate from the new goal clause.

Rewrite rules are of the form:

$$G1 \rightsquigarrow G2$$

and allow to replace a term that is matched by G1 in the resolvent with some substitution σ by the term $(G2)\sigma$ in the new question.

For example, the *Prolog rewrite rule* is:

$$true \wedge A \rightsquigarrow A$$

In epistemic logic, the rule :

$$knows(a) : knows(a) : A \rightsquigarrow knows(a) : A$$

is a useful simplification.

3.4 Selecting Database Clauses

The user can define the way clauses are selected in the base. But this selection "by hand" must be chosen among a given set (that currently implements only two methods: classical Prolog selection and least used clause selection).

Using the abstract machine, it is possible to build another selection mechanism (for example indexing selection on the first operator) but it has not been implemented yet and it is not described in this paper.

4 Examples : Modules

In this section we are going to show how to specify modules with dynamic import. Here, any module name, such as m, $m1$, $m(2)$, etc... is considered to be a context.

Module logic
$\frac{C,?M:G\vdash ?M:NG}{C,?G\vdash ?NG}$
$\frac{M:C,?M:G\vdash ?M:NG}{C,?G\vdash ?NG}$
$true \wedge G \rightsquigarrow true$
$M:true \rightsquigarrow true$

Table 1: Rules for Module logics

The goal $m1 : m2 : G$ succeeds if G can be proved using clauses from the modules $m1$ and $m2$. The inference rules are that for Prolog, plus two supplementary rules to handle module operators (table 1).

The first rule represents the case where a module M is used to compute a new goal, and the second where another module name eventually occurring in G is used.

Others types of modules such as modules with static import or with context extension [MP88], can be specified by just adding as new inference rule. In [BHLM91], we have shown how temporal logics, hypothetical reasoning and logics of knowledge and belief can be specified elegantly in our framework.

5 The abstract machine

The goal of the TARSKI abstract machine is to bridge the gap between the description of inference rules in logical form as shown above, and the real implementation of the rule in an efficient programming language.

Compared to the WAM, the TARSKI abstract machine deals with different objects, and has a quite different goal, but on the whole, principles are identical; we will also define our machine in terms of data, stacks, registers and instructions set. We do not have enough room here to describe completely the machine. So, we shall not speak of the "classical" parts of resolution that are identical: i.e unification or backtracking. Let's say that the machine relies on classical structure sharing for unification, and on depth first search and backtracking.

Before going further, we must tell about the Great Lie. TARSKI does not use classical logic operators $\wedge$ or $\leftarrow$. For consistency and simplicity sake, all operators either modal, temporal, classical, are represented in our formalism in the same way *and are treated by the machine in the same way also.* Let's see that on an example: The logical clause written in Prolog:

$$A \leftarrow B \wedge C$$

will be written in TARSKI:

$$\wedge(C) : \wedge(B) : A$$

Here B is the *argument* of $\wedge$ and A is qualified by $\wedge(B)$. All operators have arguments, and qualify an

object. For example, the S4 modal logic[4] clause:

$$\Box(X) : (\Box(a) : p \leftarrow \Diamond(a) : p)$$

will be written:

$$\Box(X) : \wedge(\Diamond(a) : p) : \Box(a) : p$$

and $\Diamond(a) : p$ is the argument of $\wedge$ that qualifies $\Box(a) : p$.

This could look like the polish reverse notation, but it is not exactly the same. In the polish reverse notation Kpq (that is $p \wedge q$) gives the same role to p and q. In $\wedge(p) : q$, p and q have really different parts to play: p is an operand of $\wedge$ and q is the object qualified by $\wedge(p)$. This destroys the symmetry of $\wedge$, but should be considered as an advantage here. In all classical Prolog, solving $p \wedge q$ is different from solving $q \wedge p$: the operator is not symmetric at all.

This formalism was not adopted lightly. The first versions did not use it, and gave a special place to the classical operators: we had a lot of problems to describe correctly the inference mechanism. Adopting this structure greatly enhanced the simplicity and the efficiency of the system.

5.1 Data structures

First of all, boolean objects (true, false) with classical operations associated (not, or, and) are implemented along with integer and floats, with their standard operations.

All data are organized in stacks. There are currently nine basic data types, and nine corresponding stacks.

The objects stack: holds all the objects on which the machine operates. An object can be either: an operator[5], a predicate[6], a variable, an integer, a float, a cons[7], *alfree*[8]. Elements of this stack will be called *ObjectElement*[9].

The operands stack: Objects do not hold their operands. Each object that has arguments holds the number of its operands and a pointer to an element of this stack that holds pointers to all the operands[10]. Elements of this stack are called *OperandElement.*

The clauses stack: Each element of this stack is composed of:

- a pointer in the object stack to the beginning of the clause
- a pointer to the head predicate[11]
- the number of free variables in the clause.

Elements of this stack are called *ClauseElement.*

The environments stack: Each element is a pair composed of a pointer to an object and a pointer in the environment stack in that the object has to be evaluated (classical structure sharing implementation). Elements of this stack are called *EnvironmentElement.*

The Trail stack: Pointers to the environment list for resetting to *free* some variables when backtracking (classical structure sharing implementation). Elements of this stack are called *TrailElement.*

The backtrack stack: Each element holds all information necessary to backtracking (values of top of stacks). Elements of this stack are called *BactrackElement.*

The question stack: Each element is a pair composed of a pointer of an object and a pointer to the environment where this object must be evaluated. The question stack holds goals to be solved. Elements of this stack are called *QuestionElement.*

The resolvent stack: stack for the resolvent elements. The resolvent is built with the current question and the current selected fact. When reaching a partial termination rule, the resolvent is re-written using rewriting rules on the top of the question and becomes the new question. Elements of this stacks are called *resolventElement.*

The predicates stack: Holds predicate structures.

There are also nine other types : pointers[12] to object in each stack, respectively *ObjectPointer, OperandPointer, ClausePointer, EnvironmentPointer, TrailPointer, BacktrackPointer, resolventPointer, QuestionPointer.*

At last, there is the *rules array.* This array describe how resolution rules behave in the system. We will come back to this later.

5.2 Registers

The registers described here are what we call *global registers* or *main registers* (see figure 2). There exists

[4]From now on, we will only use the S4 modal logic. A classical introduction is [HC72]. We will use the following notations : $\Box$ is *knows*, $\Diamond$ is *compatible.* Modal operators have arguments that must be constants. The new operator $\Diamond_I$ must be added to the original language as shown in ([CH88]).

[5]An operator is an object that has objects as arguments and qualify an other object.

[6]A predicate is an object that has arguments but do not qualify any other object.

[7]The classical LISP *cons*

[8]*alfree* is a special object quite similar in its behaviour to a variable that would always be free (*alfree* is the abbreviation of *always free*).

[9]Strings are currently not implemented.

[10]The operand stack is probably a technical mistake and will probably be suppressed in future versions of the machine

[11]Useful when using classical Prolog clauses selection to increase speed.

[12]We usually use the term *pointer* that is not exactly appropriate. Our *pointers* should be thought as abstract data types, that can be implemented as real pointers, or as indexes of an array, or anything similar.

Register	Description
Qcurr	Pointer to the current object in the question
FCurr	Pointer to the current object in the clause
FEnv	Pointer to the environment of the current clause
CClause	Pointer to the current clause
CRule	Index of the current rule used
TrTop	Pointer to the top of Trail Stack
ObTop	Pointer to the top of Object Stack
BTTop	Pointer to the top of Backtrack stack
Qtop	Pointer to the top of question stack
RTop	Pointer to the top of resolvent stack
EnvTop	Pointer to the top of environment stack

Figure 2: Abstract machine registers

Push(x : object) return pointer
Read(i : pointer) return object
Pull return object
Modify(x : object; i : pointer)
SetTop(i : pointer)
Position return pointer

Figure 3: Operations available on each stack

also general purpose registers that can be temporarily used for calculations. We will note them $R0, R1, \ldots$ in the following pages.

At time t, the machine is completely defined by the values of its stacks and its registers.

5.3 Instructions set

We describe here the instruction set of the abstract machine. We can not, because of lack of space, describe it extensively, but the next few lines give an intensive definitions of all instructions.

For each type of object, there are twice as many functions as there are components in the object, one for getting the value of the component and one for setting this value.

Moreover, for *each* of the nine stacks there are 6 basic operations implemented (see figure 3).

+(p:pointer; i:integer):pointer Increments pointer p by i

-(p:pointer; i:integer):pointer Decrements pointer p by i

-(p1,p2 : pointer):integer Returns the number of elements between $p1$ and $p2$.

There are also some classical functions: **Assignment**, **Equality test**, **Conditional constructions**.

This ends the description of atomic functions. We will need in the following lines the classical macro-instruction **unify**, that unifies (Struct1, Env1) with (Struct2, Env2)[13].

Let's see on an example how the abstract machine code is used to implement rules[14] :

$$\frac{\Box(X) : A, ?\Box(X) : B \vdash ?\Box(X) : C}{\Box(X) : A, ?B \vdash ?C}$$

is translated into:

```
R0:=Read(Qcurr)
if not
 unify(Fcurr,Fenv,GetNumStruct(R0),GetNumEnv(R0)
 then return false
 else Pushresolvent(R0) endif
Qcurr := Qcurr+1
return true
```

6 Rule selection with parallelism

In section 3.4, we said that resolution rules were chosen in the rules base in order of appearance. We are going to show here that this mechanism can be greatly enhanced by indexing the rules base and using parallel execution of rules.

6.1 Indexation of rules

The rules necessary to implement S4 are shown on top of table 2.

Remember that due to the uniform notation of the abstract machine the clause $\wedge(A) : B$ of the second rule is in fact the implication $B \leftarrow A$. We can see that, for a given fact and a given question, we have to try a lot of different rules. This creates a second non-determinism that greatly slows down the implementation of the language.

But trying all rules is usually not useful, because for a given fact and a given question, only a few rules will match the shape of the fact and the shape of the question. For example, if the fact is $\Box(X) : A$ and the question $\Diamond_I(X, I) : B$ only rules 9 and 11 can be used.

So, for a given logic, we can develop extensively all possible cases. For S4, this gives table 2. This way, given a fact and a question, the array gives directly the rules that can be applied *and there is often only one rule that can be applied.* This transforms the double non-determinism in an almost simple non-determinism much closer to Prolog complexity. So, in a large number of cases, it is not necessary to backtrack on rule selection.

[13] **unify** is of course written with atomic instructions.

[14] Other examples can be found in [Alled]: full implementation of S4 logic, among others (Fuzzy logic, module logic).

Type	*Number*	*Form*
Rule	1	$p, ?p \vdash ?true$
Rule	2	$\frac{\wedge(A):B, ?C \vdash ?\wedge(A):D}{B, ?C \vdash ?D}$
Rule	3	$\frac{B, ?\wedge(A):C \vdash ?\wedge(A):D}{B, ?C \vdash ?D}$
Rule	4	$\frac{A, ?\Diamond(X):B \vdash ?C}{A, ?B \vdash ?C}$
Rule	5	$\frac{\Diamond_I(X,I):A, ?\Diamond(X):B \vdash ?\Diamond_I(X,I):C}{A, ?\Diamond(X):B \vdash ?C}$
Rule	6	$\frac{\Diamond_I(X,I):A, ?\Diamond_I(X,I):B \vdash ?\Diamond_I(X,I):C}{A, ?B \vdash ?C}$
Rule	7	$\frac{\Box(X):A, ?\Box(X):B \vdash ?\Box(X):C}{\Box(X):A, ?B \vdash ?C}$
Rule	8	$\frac{\Box(X):A, ?\Diamond(X):B \vdash ?\Diamond(X):C}{\Box(X):A, ?B \vdash ?C}$
Rule	9	$\frac{\Box(X):A, ?\Diamond_I(X,I):B \vdash ?\Diamond_I(X,I):C}{\Box(X):A, ?B \vdash ?C}$
Rule	10	$\frac{\Box(X):A, ?\Diamond(X):B \vdash ?\Diamond(X):C}{A, ?\Diamond(X):B \vdash ?C}$
Rule	11	$\frac{\Box(X):A, ?B \vdash ?C}{A, ?B \vdash ?C}$
Fact	*Question*	*Usable rules*
Pred	Pred	$p, ?p \vdash ?true$
Pred	$\wedge$	$\frac{A, ?\wedge(X):B \vdash ?\wedge(X):C}{A, ?B \vdash ?C}$
Pred	$\Diamond$	$\frac{A, ?\Diamond(X):B \vdash ?C}{A, ?B \vdash ?C}$
$\wedge$	Pred	$\frac{\wedge(X):A, ?B \vdash ?\wedge(X):C}{A, ?B \vdash ?C}$
$\wedge$	$\wedge$	$\frac{\wedge(X):A, ?\wedge(Y):B \vdash ?\wedge(Y):C}{\wedge(X):A, ?B \vdash ?C}$
$\wedge$	$\Diamond$	$\frac{\wedge(X):A, ?\Diamond(Y):B \vdash ?\wedge(X):C}{A, ?\Diamond(Y):B \vdash ?C}$
$\wedge$	$\Box$	$\frac{\wedge(X):A, ?\Box(Y):B \vdash ?\wedge(X):C}{A, ?\Box(Y):B \vdash ?C}$
$\wedge$	$\Diamond_I$	$\frac{\wedge(X):A, ?\Diamond_I(Y,I):B \vdash ?\wedge(X):C}{A, ?\Diamond_I(Y,I):B \vdash ?C}$
$\Box$	Pred	$\frac{\Box(X):A, ?B \vdash ?C}{A, ?B \vdash ?C}$
$\Box$	$\wedge$	$\frac{\Box(Y):A, ?\wedge(X):B \vdash ?\wedge(X):C}{\Box(Y):A, ?B \vdash ?C}$
$\Box$	$\Diamond$	$\frac{\Box(X):A, ?\Diamond(Y):B \vdash ?C}{A, ?\Diamond(Y):B \vdash ?C}$
		$\frac{\Box(Y):A, ?\Diamond(X):B \vdash ?C}{\Box(Y):A, ?B \vdash ?C}$
		$\frac{\Box(X):A, ?\Diamond(X):B \vdash ?\Diamond(X):C}{A, ?\Diamond(X):B \vdash ?C}$
		$\frac{\Box(X):A, ?\Diamond(X):B \vdash ?\Diamond(X):C}{\Box(X):A, ?B \vdash ?C}$
$\Box$	$\Diamond_I$	$\frac{\Box(X):A, ?\Diamond_I(Y,I):B \vdash ?C}{A, ?\Diamond_I(Y,I):B \vdash ?C}$
		$\frac{\Box(X):A, ?\Diamond_I(X,I):B \vdash ?\Diamond_I(X,I):C}{\Box(X):A, ?B \vdash ?C}$
$\Diamond_I$	$\wedge$	$\frac{\Diamond_I(Y):A, ?\wedge(X):B \vdash ?\wedge(X):C}{\Diamond_I(Y):A, ?B \vdash ?C}$
$\Diamond_I$	$\Diamond$	$\frac{\Diamond_I(Y):A, ?\Diamond(X):B \vdash ?C}{\Diamond_I(Y):A, ?B \vdash ?C}$
		$\frac{\Diamond_I(X,I):A, ?\Diamond(X):B \vdash ?\Diamond_I(X,I):C}{A, ?\Diamond(X):B \vdash ?C}$
$\Diamond_I$	$\Diamond_I$	$\frac{\Diamond_I(X,I):A, ?\Diamond_I(X,I):B \vdash ?\Diamond_I(X,I):C}{A, ?B \vdash ?C}$

Table 2: S4 logic rules and their exhaustive development

	Fact	Question	Rules
R1	$\Box$	$\Diamond$	$\frac{\Box(X):A, ?\Diamond(X):B \vdash ?C}{A, ?\Diamond(X):B \vdash ?C}$
R2			$\frac{\Box(X):A, ?\Diamond(X):B \vdash ?C}{\Box(X):A, ?B \vdash ?C}$
R3			$\frac{\Box(X):A, ?\Diamond(X):B \vdash ?\Diamond(X):C}{A, ?\Diamond(X):B \vdash ?C}$
R4			$\frac{\Box(X):A, ?\Diamond(X):B \vdash ?\Diamond(X):C}{\Box(X):A, ?B \vdash ?C}$

Table 3: Rules $\Box$ against $\Diamond$

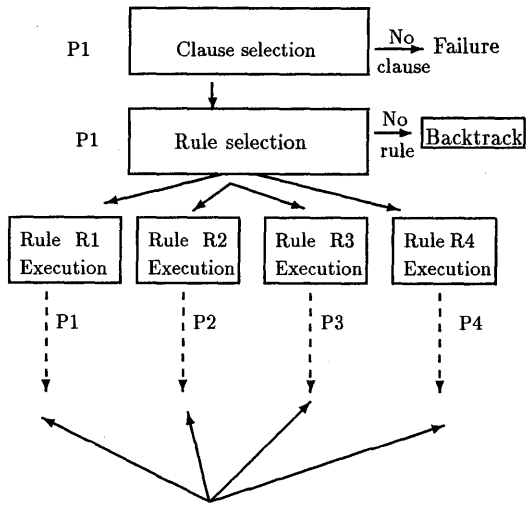


Figure 4: Parallel execution of S4 rules

6.2 Parallel rule execution

The abstract machine was designed to enable an easy implementation of parallelism. Sometimes, for a given definite fact and a given goal clause, more than one rule is possible : we can use a different processor for each rule. For example, in the S4 logic, if the fact is $\Box(X) : A$ and the question is $\Diamond(X) : B$, four rules can be used (table 3). With four processors, each one can continue the resolution with a different rule. Figure 4 shows how the inference system running originally on processor P1. With four processors P1, P2, P3, P4 available, it is possible to solve, in parallel, S4 rules described in table 3.

The information transferred from one processor (P1) to its children (P2, P3, P4) are *the abstract machine data stacks* and *the abstract machine registers.* Some stacks are never transferred (the backtrack stack, the trail stack) because the child does not need to backtrack over the current resolution point. This parallelism induces no side effects : as soon as one processor has received data, it will not have to communicate

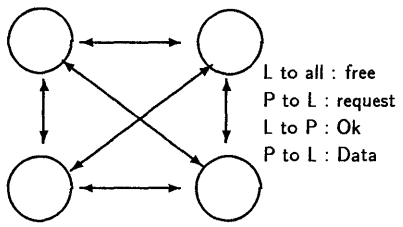


Figure 5: Fully interconnected network

with its parent any more until it has finished its own resolution. Moreover, there is no overhead in processing time because parallelism is explicit in the language itself : overhead comes only from communication between processes.

Four models (Master/slaves network, fully interconnected networks, ring networks, top-down networks) are under development; we just mention them and we will not discuss them in detail[15].

Fully interconnected network: Every processor can distribute work to any other processor that is free. A very simple protocol is used to prevent two processors to send at the same time data to the same processor (figure 5). This protocol will solve problems as represented in figure 4.

Master/slaves network: The master process distributes work to all other processes, which, in turn, can not distribute any work. This protocol will also solve problems as represented in figure 4.

Ring network: Here each processor can send work to the next one, and the last processor can send work to the first.

Top-Down network : In the Top-Down Network, each processor can only send information to the following one but the last processor can't send information to the first one. In ring networks and top-down networks, resolution is not exactly as represented in figure 4.

7 Implementing Parallelism

7.1 The "classical" machine

The new abstract machine specifications was the result that began with the first implementation of MOLOG, in C, in 1988.

Coding the new machine took less than two months. Of course, two years spent in coding other abstract machines (that proved to be unsatisfactory) helped a lot. From the beginning, the stress was on getting a program as close as possible to the specifications of the abstract machine. That was the reason why the ADA language has been chosen: the specifications of the abstract machine *are* exactly the specifications of the main package of the implementation. Moreover, compared to other implementations previously written in C, coding and debugging was a lot easier and faster. We wanted also to be able to easily implement parallelism. So, for example, stacks are implemented with arrays and there is not a single real pointer in the system, only indexes. It has an interesting well known side effect: we never run out of stack space, because if a stack becomes full, we just have to copy it to a new larger stack. All indexes are still valid. The mechanism is invisible to the programmer and the user and very useful with some very recursive non-classical problems.

This was done at the loss of performance. Accessing any object in a stack requires two function calls and three tests plus the classical indirection. The TARSKI machine runs about fifteen times slower than C-Prolog[16] on PROLOG problems. This could easily be enhanced by recoding the machine with efficiency in mind.

Coding a logic is very easy as soon as it follows the general framework given in section 3.2. The S4 logic was implemented in *one* day. and tested with the classical "wise men" puzzle. The puzzle is solved in three minutes on a HP-720 workstation *with the full amount of knowledge* (more than twenty clauses). With only the five clauses necessary to solve the problem, the solution is found in less than a second, hundred times faster than the MOLOG interpreter.

7.2 The parallel machine

The parallel machine was developped with an ETHERNET network as medium for data transfer. The parallel system is made of many TARSKI machines running on different workstations, linked by INTERNET sockets[17]. The only configuration tested was a top-down network. Results are shown in table 4. It would be too long to discuss them here in detail. Full explanations can be found in [Alled].

We can briefly say that, over three processors, the network is clearly too slow and becomes the bottleneck of the system. A large part of time is lost in communicating with other processors. There are different solutions that could be used to enhance performances:

- We can use parallelism only for branches that are

[15] On all practical implementations issues, details can be found in [Alled].

[16] It is however faster than some classical PROLOG written in compiled Common Lisp

[17] It was quite easy to do, because all necessary packages for communication and parallelism had been developped previously for other projects. Reusability of software is a major advantage of ADA.

# of Procs	P1	P2	P3	P4
1	319+1			
2	166+10	145+6		
3	129+24	142+50	77+17	
4	129+26	140+46	46+31	22+9

Table 4: CPU+system time used

close to the root of the tree. This will decrease the number of sent packets.

- We can try a master/slave network. The master processor will be almost devoted to sending packets but slaves would not spare time on this.
- We can improve the amount of sent data; some stacks can only grow, and are never modified under a certain depth. We could only send new data, and not the whole stack.
- We could try to use a different medium. An ethernet network is a very slow device for parallelism, and, moreover, our network is usually crowded with packets coming from other stations or other X-terminals. It would be very interesting to implement the machine on a multi-processor computer with shared memory segments, or on a transputers network. We were not able to do it yet because we lack access to such a machine. We are very eager to try such an approach. If we are able to find a machine with many processors, the inference machine could be almost as fast as a standard PROLOG even when solving non-classical logic problems, because the double non-determinism would be almost reduced to classical PROLOG non-determinism.

8 Conclusion

We think the implementation of any logic given by inference rules of the form defined in the earlier sections can be done in a very short amount of time (one or two days at most). The development of an automatic translator from the logical shape of the rules to the abstract machine specifications suggests itself and is a subject of current work.

Now, it is hoped that fast, general and efficient implementations of such logics could bring a new area of development for expert systems. In particular, in the C.E.N.A.[18] a large expert system (3,000 rules) using fuzzy and temporal logics has been developped in Prolog ([AL91]). This expert systems could be an excellent test for TARSKI.

[18]The CENA is an institution responsible for studies of new systems for Air Traffic Control in France

9 Acknowledgements

We wish to thank Luis Fariñas Del Cerro for valuable discussions.

References

[ABFdC+86] R. Arthaud, P. Bieber, L. Fariñas del Cerro, J. Henry, and A. Herzig. Automated modal reasoning. In *Proc. of the Int. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Paris, july 1986.

[AG88] J. M. Alliot and J. Garmendia. *Une Implementation en "C" de MOLOG.* Rapport D.E.A, Université Paul Sabatier, Toulouse, France, 1988.

[AL91] Jean-Marc Alliot and Marcel Leroux. En route air traffic organizer, un système expert pour le contrôle du trafic aérien. In *Proceedings of the International Conference on Expert systems and their applications*, Avignon, May 1991.

[Alled] Jean-Marc Alliot. Tarski: une machine parallèle pour l'implémentation d'extensions de prolog. Master's thesis, Université Paul Sabatier, To be published.

[Bau89] Marianne Baudinet. *Logic Programming Semantics: Techniques and Applications.* PhD thesis, Stanford University, feb 1989.

[BFdCH88] P. Bieber, L. Fariñas del Cerro, and A. Herzig. MOLOG – a modal PROLOG. In E. Lusk and R. Overbeek, editors, *Proc. of the 9th Int. Conf. on Automated Deduction*, LNCS 310, pages 487–499, Argonne – USA, may 1988. Springer Verlag.

[BHLM91] P. Balbiani, A. Herzig, and M. Lima-Marques. TIM: The Toulouse Inference Machine for non–classical logic programming. In M.M. Richter and H. Boley, editors, *Processing Declarative Knowledge*, number 567 in Lecture Notes in Artificial Intelligence, pages 365–382. Springe–Verlag, 1991.

[BK82] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[Bri87] M. Bricard. Une machine abstraite pour compiler MOLOG. Rapport D.E.A., Université Paul Sabatier – LSI, 1987.

[CH88] Luis Fariñas Del Cerro and Andreas Herzig. Linear modal deductions. In E. Lusk and R. Overbeek, editors, *Proc. of the 9th Int. Conf. on Automated Deduction Computer Systems.* Springer-Verlag, 1988.

[Esp87a] Esprit Project p973 "ALPES". *MOLOG Technical Report*, may 1987. Esprit Technical Report.

[Esp87b] Esprit Project p973 "ALPES". *MOLOG User Manual*, may 1987. Esprit Technical Report.

[FdC86] L. Fariñas del Cerro. MOLOG: A system that extends PROLOG with modal logic. *New Generation Computing*, 4:35–50, 1986.

[FKTMO86] M. Fujita, S. Kono, H. Tanaka, and T. Moto-Oka. Tokio: Logic programming language based on temporal logic and its compilation to prolog. In *Third Int. Conf. on Logic Programming*, pages 695–709, jul 1986.

[GL82] M. Gallaire and C. Lasserre. Meta-level control for logic programs. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 173–188. Academic Press, 1982.

[GR84] D. Gabbay and U. Reyle. N-prolog: An extension of prolog with hypothetical implications. *Jounal of Logic Programming*, 1:319–355, 1984.

[HC72] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logics.* Methuen & Co. Ltd, USA, 2 edition, 1972.

[MP88] Luis Monteiro and Antonio Porto. Modules for logic programming based on context extension. In *Int. Conf. on Logic Programming*, 1988.

[Sak87] Y Sakakibara. Programming in modal logic: An extension of PROLOG based on modal logic. In *Int. Conf. on Logic Programming*, 1987.

[Sak89] Takashi Sakuragawa. Temporal PROLOG. In *RIMS Conf. on software science and engineering*, 1989.

[SS86] L Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, USA, 1986.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Constraint Solving in Andorra-I

Steve Gregory and Rong Yang

Department of Computer Science
University of Bristol
Bristol BS8 1TR, England

steve/rong@cs.bris.ac.uk

Abstract

The subject of this paper is the integration of two active areas of research: a parallel implementation of a constraint logic programming language. Specifically, we report on some experiments with the and/or-parallel logic programming system Andorra-I extended with support for finite domain constraint solving.

We describe how the language supported by Andorra-I can be extended with finite domain constraints, and show that the computational model underlying Andorra-I is well suited to execute such programs. For example, most constraints are automatically executed eagerly so as to reduce the search space; moreover, they are executed concurrently, using dependent and-parallelism.

We have compared the performance of some constrained search programs on Andorra-I with that of conventional generate-and-test programs. The results show that the use of constraints not only reduces the sequential execution time, but also significantly increases the and-parallel speedup.

1 Introduction

Much of the success of Prolog has been due to its suitability for applications involving search: the language provides a relational notation which is very convenient for expressing non-deterministic problems and it can be implemented with impressive efficiency. However, the search strategy built into Prolog is a rather naive one, which tends to perform an unnecessary amount of search for problems that are stated in a simple manner. To solve realistic search problems in Prolog, it is often necessary to perform additional forward computation in order to reduce the search space to a manageable size. However, since this extra computation must be programmed in Prolog itself, it may be an expensive overhead which partly offsets the speed benefits of the reduced search. Moreover, the resulting program is more opaque and difficult to write than a natural solution in Prolog.

To improve on the search strategy of Prolog while retaining its advantages is the motivation for the development of constraint logic programming (CLP) systems. Most of the CLP languages that have been proposed are based on Prolog, extended with the ability to solve constraints in one or more domains. CLP languages use knowledge specific to their domain to execute certain goals ("constraints") earlier than would be possible in Prolog, thus potentially reducing the search space. Provided that the constraint solving mechanism is implemented efficiently and that the language is simple to use, the search time can be reduced at little cost in either forward computation time or increased program complexity. One type of CLP language, which has proved particularly useful for combinatorial search problems, is that based on finite domains; this is described in a little more detail in Section 2.

There have been many projects in recent years to develop parallel implementations of Prolog. Most of these systems incorporate either or-parallelism, independent and-parallelism, or both. In contrast, the Andorra-I system is an implementation of Prolog that exploits or-parallelism together with dependent and-parallelism, which is the sole form of parallelism exploited in most implementations of concurrent logic programming languages such as Parlog and GHC. Andorra-I has proved effective in obtaining speedups in programs that have potential or-parallelism and those with potential and-parallelism, while in some programs both forms of parallelism can be exploited. Andorra-I, and the Basic Andorra model on which it is based, are described briefly in Section 3.

The subject of this paper is the integration of the above strands of research: a parallel implementation of a constraint logic programming language. Specifically, we report on our experiences with extending the Prolog-like language supported by Andorra-I to support finite domain constraint solving. There are two main reasons why this is of interest:

1. **Language.** To investigate how easily the required language extensions can be supported by the Basic Andorra model.

2. **Performance.** To ensure that the finite domain extensions can be implemented efficiently in Andorra-I and that the efficiency is retained in parallel execution.

Although a prototype or-parallel implementation of the Chip language has been developed [Van Hentenryck 1989b], we are not aware of any previous investigation of and-parallelism with finite domain constraints. By adding these extensions to Andorra-I we can experiment with both forms of parallelism and compare them.

It is particularly interesting to compare the performance of constrained search programs on the Basic Andorra model with that of conventional generate-and-test programs (apart from the expected reduction in overall execution time). The constraint solving represents additional forward computation, so — provided that the constraints can be effectively solved in parallel — we would expect and-parallelism to be increased. At the same time, since the search space is reduced, there may be less scope for or-parallelism. The performance results obtained with Andorra-I confirm these expectations.

The next two sections describe the background to the paper. Section 4 discusses the implementation of finite domain constraints on the Basic Andorra model. It describes in detail the language extensions that we have implemented and the structure of programs that use them. Section 5 presents some results of running constrained search problems on Andorra-I. Section 6 concludes the paper.

2 Finite domain constraints

The idea of adding finite domain constraints to logic programming originated with the work of Van Hentenryck and his colleagues, and was first implemented in the language Chip [Van Hentenryck and Dincbas 1986; Dincbas *et al.* 1988; Van Hentenryck 1989a]. Chip extends Prolog in several ways to handle constraints; the principal extensions relevant to finite domains are outlined below.

2.1 Domain variables

Some variables in a program may be designated *domain variables*, ranging over any specified finite domain. Domain variables appear to the programmer like normal logical variables but are treated differently by unification and by constraints.

2.2 Constraints on finite domains

Goals for certain *constraint* relations behave in a special way when they have domain variables as arguments. For example, if `X` is a domain variable, the goal `X ≤ 5` can be executed by removing from the domain of `X` all items greater than 5. This in turn may reduce the search space that the program explores. A user-defined predicate may be made a constraint by using a `'forward'` or `'lookahead'` declaration, while some primitives (e.g., inequality) have such declarations implicitly. (Unification can have a similar effect: unifying two domain variables reduces the domain of both to the intersection of their original domains, while unifying a domain variable and a constant may fail.)

2.3 Coroutining

Constraints should be executed as early as possible in order to reduce the search space. For example, `X ≤ Y` could be executed as soon as either `X` or `Y` has a value and the other is a domain variable. In general, a coroutining mechanism ensures that control switches to a constraint goal as soon as it can be executed. The simplest such control rule is *forward checking*, used for `forward`-declared constraints, whereby a constraint is executed as soon as its arguments contain at most one domain variable and are otherwise ground. The constraint goal is then effectively executed for each member of its argument's domain and values that cause failure are removed from the domain.

The *lookahead* rule, often used for inequality relations such as '≤', can even execute constraints whose arguments contain more than one domain variable; we shall not consider this further in this paper.

3 The Basic Andorra model

The Basic Andorra model is a computational model for logic programs which exploits both or-parallelism and dependent (stream) and-parallelism. The model works by alternating between two phases:

1. **Determinate phase.** Determinate goals are executed in preference to non-determinate goals. While determinate goals exist they are executed in parallel, giving dependent and-parallelism. (A goal is considered *determinate* if the system can detect that it can match at most one clause.) This phase ends when no determinate goals are available or when some goal fails.

2. **Non-determinate phase.** When no determinate goals remain, one goal — namely, the leftmost one that is not `det_only` (see below) — is selected and a choicepoint created for it. Or-parallelism can be obtained by exploring choicepoints in parallel.

The model and its prototype implementation, Andorra-I, are described in [Santos Costa *et al.* 1991].

Andorra-I supports the Prolog language augmented with a few features specific to the model. For example, `det_only` declarations allow the programmer to specify that goals for some predicate

can only be executed in the determinate phase; if such a goal remains in the non-determinate phase it cannot be used to create a choicepoint, even if it is the leftmost goal. Conversely, `non_det_only` declarations can be used to prevent goals from executing in the determinate phase even if they are determinate.

Performance results for Andorra-I show that the system obtains good speedups from both or-parallelism and and-parallelism. The best and-parallel speedups are obtained for programs that are completely determinate (and therefore have no or-parallelism to exploit). The best or-parallel speedups come from search programs, especially when searching for all solutions.

Unfortunately, very little and-parallel speedup has typically been observed in running standard Prolog search programs on Andorra-I. One reason for this is the sequential bottleneck inherent in the Basic Andorra model: the periods (both during the non-determinate phases and while backtracking) when no and-parallel execution is performed.

This suggests that the key to obtaining greater and-parallel speedup is to increase the "granularity" of the and-parallelism. That is, it is important to minimize the number of choicepoints created and the number of goal failures, relative to the total number of inferences. One way to achieve this in search programs is by the use of constraint satisfaction techniques.

4 Implementing finite domains in Andorra-I

In order to experiment with finite domain constraint solving on Andorra-I, we have defined and implemented finite domains and a few simple primitives to operate on them. Our system defines a new data type, a *domain*, which exists alongside numbers, structures, etc. Domains can only be used as arguments to the domain primitives and have no meaning elsewhere in a program; for example, they cannot be printed. A domain is created with a set of possible values that it may take; eventually it may become instantiated to one of those values, at which time we call it an *instantiated domain*. In contrast with the Chip concept of domain *variables*, a domain instantiated to t is *not* identical to t. We write a domain as a set $\{t_1,\ldots,t_n\}$, where $t_1,\ldots,t_n$ are its current possible values; $\{t\}$ represents an instantiated domain.

Our domains are easier to implement than domain variables because there is no need to change many basic operations of the system such as unification, suspension on variables, etc. At the same time, the efficiency of implementation should be comparable with that of domain variables, while our primitives are still quite convenient to use.

We describe our primitives first and then outline their use and implementation.

4.1 Finite domain primitives

Domains can be created by the primitives `make_domain` and `make_domains`. The latter is potentially more efficient when creating many domains ranging over the same values since the table of values can be shared.

All of the other primitives operate on existing domains; they can only be executed when their first argument is instantiated and will fail if this is not a domain. `domain_var` performs the mapping between a domain and its ultimate value, while `domain_remove` allows the removal of values from a domain. Either of these may cause the domain to be instantiated: the first in a positive way, the second by removing all but one of the values. `domain_guess` is the only non-determinate primitive. The last two, `domain_size` and `domain_values`, may yield different results depending on when they are called and should therefore be used with care.

`make_domain(D,Set)`

Can be executed when `Set` is instantiated to a non-empty list of distinct atomic terms, $[t_1,\ldots,t_n]$. `D`, which should be an unbound variable, is bound to a new domain, $\{t_1,\ldots,t_n\}$.

`make_domains(Ds,Set)`

Can be executed when `Set` is instantiated to a non-empty list of distinct atomic terms, $[t_1,\ldots,t_n]$, and `Ds` is a list of variables. Each variable in `Ds` is bound to a new domain, $\{t_1,\ldots,t_n\}$.

`domain_var(D,Var)`

Unifies `Var` with the *value variable* (a normal logical variable) of domain `D`. Subsequently, if `D` becomes an instantiated domain $\{t\}$, t is unified with `Var`. Alternatively, if `Var` becomes instantiated to t, if t is currently in the domain `D`, `D` becomes an instantiated domain $\{t\}$, otherwise failure occurs.

`domain_remove(D,Value)`

Can be executed when `Value` is ground. If `Value` is not currently in the domain `D`, there is no effect. If `D` is the instantiated domain `{Value}` the primitive fails. Otherwise `Value` is removed from the domain; if only one value, t, remains in the domain `D` becomes instantiated to $\{t\}$.

`domain_guess(D)`

Instantiates `D` non-determinately to one of its possible values. If `D` is the domain $\{t_1,\ldots,t_n\}$, `D` is instantiated successively to $\{t_1\}$, ..., $\{t_n\}$.

Note that `domain_guess(D)` is non-determinate (unless `D` is already instantiated) and can therefore be executed only if there are no determinate goals to execute.

domain_size(D,Size)

`Size` is unified with a positive integer which indicates the number of values currently in domain `D`.

domain_values(D,Values)

`Values` is unified with a list of the values currently in domain `D`.

4.2 Finite domain programming

Like Chip, our aim is to provide the programmer with a language as close as possible to Prolog but with the extensions necessary for constraint programming. However, the "Prolog" language supported by the Basic Andorra model differs in behaviour from that of regular Prolog, and this affects how the language is used. In this section we outline how our primitives can be employed in the context of Prolog on Andorra-I to solve constraint problems.

Program 1 is our solution to the familiar N-queens problem. This program is almost identical to the Chip one on p123 of [Van Hentenryck 1989a], except that the result of the goal `four_queens(Qs)` is a list of domains (which can be converted to a numeric value by `domain_var`). However, it executes differently. The execution order in Chip is the same as in Prolog, repeatedly executing a `domain_guess` goal for one domain followed by a `noattack` goal to remove inconsistent values from the other domains. On Andorra-I the program executes all of the `queens` and `noattack` goals first, since they are determinate, and sets up all '≠' constraints before `domain_guess` is called to non-determinately generate domain values.

```
four_queens(Qs) :-
   Qs = [Q1,Q2,Q3,Q4],
   make_domains(Qs, [1,2,3,4]),
   queens(Qs).

queens([]).
queens([Q|Qs]) :-
   domain_guess(Q),
   noattack(Q, Qs, 1),
   queens(Qs).

noattack(_, [], _).
noattack(Q1, [Q2|Qs], N) :-
   Q1 ≠ Q2,
   Q1 ≠ Q2 - N,
   Q1 ≠ Q2 + N,
   N1 is N + 1,
   noattack(Q1, Qs, N1).
```

Program 1: N-queens

At the end of the first determinate phase, the resolvent contains only the following goals, for `domain_guess` and the inequality predicate '≠', where each of `Q1`, `Q2`, `Q3`, and `Q4` is an uninstantiated domain:

```
domain_guess(Q1),
   Q1 ≠ Q2, Q1 ≠ Q2 - 1, Q1 ≠ Q2 + 1,
   Q1 ≠ Q3, Q1 ≠ Q3 - 2, Q1 ≠ Q3 + 2,
   Q1 ≠ Q4, Q1 ≠ Q4 - 3, Q1 ≠ Q4 + 3,
domain_guess(Q2),
   Q2 ≠ Q3, Q2 ≠ Q3 - 1, Q2 ≠ Q3 + 1,
   Q2 ≠ Q4, Q2 ≠ Q4 - 2, Q2 ≠ Q4 + 2,
domain_guess(Q3),
   Q3 ≠ Q4, Q3 ≠ Q4 - 1, Q3 ≠ Q4 + 1,
domain_guess(Q4).
```

The only goals that can be executed in the non-determinate phase are for `domain_guess`, since the '≠' goals are treated as `det_only` (see Section 3). Selecting the leftmost goal, `domain_guess(Q1)`, `Q1` is instantiated non-determinately to the domain {1} and a new determinate phase begins, in which all nine '≠' goals containing `Q1` can be executed in parallel.

This example illustrates a difference between our language and Chip, which follows from the Basic Andorra model: that the order of goals in a clause is irrelevant. Constraints and generators can appear in any order, but the constraints will always be set up before any non-determinate bindings are made. This is important, since it results in a smaller search space. In order to get the same effect (called "generalized forward checking") in Chip, the structure of the program has to be changed. However, we do have to make sure that constraints can be executed determinately, so that they execute first, whereas constraints need not be determinate in Chip.

The inequality predicate '≠' used above is an example of a constraint that is to be executed by forward checking. Such predicates can be programmed using the primitives of Section 4.1. As an example, Program 2 defines a constraint `plusorminus(X,Y,C)`, which means `X=Y-C` or `X=Y+C`. This can be executed in a forward checking way when either of domains `X` and `Y` is instantiated and the third argument is ground; it then leaves only (at most) the two values `Y-C` and `Y+C` (resp. `X-C` and `X+C`) in the domain of `X` (resp. `Y`).

In Program 2 we use Pandora syntax [Bahgat and Gregory 1989]. The `plusorminus` procedure is a "don't-care procedure" in the style of Parlog: the first clause removes the appropriate values from the domain of `Y` if domain `X` is instantiated, while the second does the converse. This procedure uses the `data` primitive to wait for the domain to be instantiated and the operator ':' to commit to the appropriate clause. A sequential conjunction operator '&' is used in the `pm` procedure, so that the values currently in domain `Y` are found (by a call to `domain_values`) only after the other arguments are instantiated. It then `filters` these values to find which ones must be removed from the domain, and removes them by calling `domain_remove`.

In addition to primitive constraints such as inequality, Chip allows user-defined constraints. These are conventional Prolog procedures augmented with a `'forward'` declaration indicating which

arguments should be ground and which should be domain variables. For example, `plusorminus` is defined [Van Hentenryck 1989a: p134] as follows:

```
forward plusorminus(d,d,g).
plusorminus(X,Y,C) :- X is Y - C.
plusorminus(X,Y,C) :- X is Y + C.
```

The problem with allowing user-defined constraints in Andorra-I is that the procedures may in general be non-determinate and, in any case, a search is required through the elements of a domain. One way to handle such constraints is by transforming the procedure to a determinate, forward checking, equivalent, as we did with `plusorminus` in Program 2. Another way would be to use a "determinate `bagof`" primitive which is currently being implemented in Andorra-I. This is similar to the `bagof` of Prolog but it executes as part of the determinate phase as a new subcomputation, even if it has to create internal choicepoints.

```
mode plusorminus(?, ?, ?).
plusorminus(X, Y, C) <-
   domain_var(X, Xv), data(Xv) :
   pm(Xv, Y, C).
plusorminus(X, Y, C) <-
   domain_var(Y, Yv), data(Yv) :
   pm(Yv, X, C).

mode pm(?, ?, ?).
pm(Xv, Y, C) <-
   Yv1 is Xv - C,
   Yv2 is Xv + C &
   domain_values(Y, Yvs),
   filter(Yvs, Yv1, Yv2, Remove),
   remove_all(Y, Remove).

mode filter(?, ?, ?, ^).
filter([], _, _, []).
filter([V1|Vs], V1, V2, R) <-
   filter(Vs, V1, V2, R).
filter([V2|Vs], V1, V2, R) <-
   filter(Vs, V1, V2, R).
filter([V|Vs], V1, V2, [V|Vs1]) <-
   V \== V1, V \== V2,
   filter(Vs, V1, V2, Vs1).

mode remove_all(?, ?).
remove_all(_, []).
remove_all(D, [V|Vs]) <-
   domain_remove(D, V),
   remove_all(D, Vs).
```

Program 2: Pandora program for the `plusorminus` constraint

The `deleteff` predicate of Chip, which is used to implement the first-fail heuristic, can easily be programmed using another of our primitives, `domain_size`. `deleteff(Best,Ds,Rest)` finds `Best` as the domain in list `Ds` that has the smallest current size; `Rest` contains the remaining elements of `Ds`. Program 3 is a program for N-queens which implements the first-fail heuristic (the `noattack` procedure is the same as in Program 1), and illustrates the general structure of such programs. Note that the "guessing" and "checking" components (the `guess_queens` and `check_queens` procedures) must be separated, though their order is unimportant.

```
four_queens(Qs) :-
   Qs = [Q1,Q2,Q3,Q4],
   make_domains(Qs, [1,2,3,4]),
   guess_queens(Qs),
   check_queens(Qs).

guess_queens([]).
guess_queens([Q|Qs]) :-
   deleteff(Best, [Q|Qs], Rest),
   domain_guess(Best),
   guess_queens(Rest).

check_queens([]).
check_queens([Q|Qs]) :-
   noattack(Q, Qs, 1),
   check_queens(Qs).
```

Program 3: Changes to N-queens to implement first-fail heuristic

The main issue in using `deleteff` in an Andorra-I program is to ensure that it is called at the right time, i.e., immediately before a choicepoint is created. By default, Andorra-I would execute all the `deleteff` goals immediately, since they are determinate. This would just choose the domains to guess in a fixed order. The easiest way to avoid this problem is to declare `deleteff` to be `non_det_only` (see Section 3).

During the first determinate phase, the `check_queens` goal executes to completion, spawning the same inequality ('≠') goals as in Program 1, while `guess_queens([Q1,Q2,Q3,Q4])` reduces to the following:

```
deleteff(Best, [Q1,Q2,Q3,Q4], Rest),
domain_guess(Best),
guess_queens(Rest)
```

Now the leftmost goal, `deleteff`, runs and finds the smallest domain from `[Q1,Q2,Q3,Q4]`. In the next non-determinate phase `domain_guess` is called for the chosen domain, allowing some of the constraints to execute; when no more constraints can be executed, the next `deleteff` goal can execute, and so on.

4.3 Implementation

There are several ways to represent domains and to implement the predicates listed in Section 4.1. The predicates could be implemented by logic programs, provided we design a suitable representation of domains. Two of them, `domain_var` and `domain_remove`, modify the state of a domain but, happily, domains have the property that their size monotonically decreases. This enables us to represent

each domain by a tuple of logical variables, one for each possible domain value; the variable is bound to 0 when the value is removed, or 1 when the domain is instantiated to that value.

Given such a representation, the properties of a domain (e.g., it must not be empty, it cannot be instantiated to a value that has been removed, and so on) must be preserved. One way to do this is for each operation to check the state of the domain before modifying it. This works well in a sequential logic programming system, but is extremely complex to implement correctly in an and-parallel context because of contention by several operations modifying the same domain in parallel. A better method in the presence of and-parallelism is to spawn a process network to maintain the properties of a domain at the time it is created. This technique was described in [Bahgat and Gregory 1989].

Both of the above techniques were used to prototype our domain operations. However, to get more meaningful performance results, we wished to implement them as efficiently as possible, so a lower-level implementation was developed. A domain is represented by a structure containing the following fields:

1. A term (initially an unbound variable) representing the ultimate value of the domain. This term can be accessed by the `domain_var` primitive.

2. A boolean array with one bit for each potential member of the domain.

3. The number of elements currently in the domain. This field is accessed by the `domain_size` primitive.

4. The position of the last element guessed non-determinately.

5. A reference to a table mapping between domain values and positions in the domain.

The key implementation issues concern how to update the domains. Conditional modifications to domains (fields 2, 3, 4) need to be trailed. Fortunately, this can be achieved using the "updatable variables" which are already implemented in Andorra-I and used for many other purposes.

Each domain may be concurrently accessed by many constraints. To implement the required mutual exclusion, the value variable of a domain (field 1) is locked while the domain is modified, using the normal variable locking mechanism of Andorra-I. Each constraint locks only one domain at a time, so there is no danger of deadlock. Starvation is avoided because a domain is locked only when values are to be removed, and the size of domains is finite.

Both the updatable variables and variable locking features of Andorra-I are described in [Santos Costa *et al.* 1991].

5 Performance results

In this section we present some results obtained on the Andorra-I system running on a Sequent Symmetry. Each of the tables gives the results of running a particular program on different problem sizes. The respective columns show:

BT the number of backtrackings,

Time the execution time (in seconds) on one processor,

And-// the and-parallel speedup when run on 10 processors,

Or-// the or-parallel speedup when run on 10 processors.

(The speedup figures are simply the ratio of execution time on one processor to that on 10 processors.)

Table 1 shows the results of a standard Prolog program for N-queens. The structure of this program is similar to that of Program 1, but it makes no use of forward checking: it simply places a queen on each row non-deterministically and tests each time that the resulting configuration is safe with respect to previously-placed queens. The top part of the table gives results of a search for all solutions, while the bottom part shows a search for the first solution.

	N	BT	Time	And-//	Or-//
All	4	18	0.22	1.05	1.57
solns	6	208	2.92	1.11	4.23
	8	3544	54.32	1.17	8.83
	10	75190	1250.41	1.21	9.82
First	4	7	0.11	1.00	0.92
soln	6	46	0.65	1.12	2.41
	8	223	3.27	1.16	3.85
	10	276	3.71	1.16	1.98
	12	873	11.94	1.19	1.51
	16	42865	653.78	1.26	1.10

Table 1: Standard backtracking program for N-queens

Table 1 confirms that the search space and execution time increase dramatically as the problem size increases. It also shows that the or-parallel speedup for the first solution is very variable. This is usual, since an or-parallel search for one solution explores a different part of the search tree than a sequential search, so the backtrack count will differ

from that shown in the **BT** column and indeed will vary between runs. (We give the best or-parallel speedup obtained from several runs.) The consistent results are that a large or-parallel speedup is seen when searching for all solutions, while there is a very small and-parallel speedup in all cases. Both of these increase as the problem size increases. In every case, the or-parallel speedup observed is better than the and-parallel one.

Table 2 gives the same results for the forward checking program (Program 1). As expected, the search space is much reduced. The fact that the total execution time is also much smaller indicates that our implementation of finite domains is efficient enough that the cost of constraint solving pays off. The and-parallel speedup for this program is substantially larger than for the standard backtracking program (though it is still rather small), while the or-parallel speedup is generally less. In contrast to Table 1, for the first-solution search, the and-parallel speedup always exceeds the or-parallel one.

	N	BT	Time	And-//	Or-//
All	4	7	0.09	1.50	1.00
solns	6	41	0.64	1.78	2.91
	8	417	8.34	1.86	6.95
	10	6667	142.54	1.99	9.37
First	4	2	0.05	1.67	1.00
soln	6	8	0.19	1.73	1.46
	8	24	0.52	2.08	2.00
	10	24	0.61	2.18	1.49
	12	54	1.42	2.22	1.34
	16	1833	53.56	2.44	2.12

Table 2: Forward checking program for N-queens

We carried out similar experiments for the graph colouring problem: to colour a graph so that neighbouring nodes have distinct colours, and so that the number of colours used (the chromatic number) is minimized. The programs for this problem perform a depth-first branch-and-bound search by first finding an approximate solution with chromatic number C, then restarting the search with the added constraint that no node can be given a colour greater than C–1; this is repeated until no better solution is found.

Two programs for this problem were tested. The standard backtracking program colours nodes in descending order of degree; each time a node is coloured, each possible colour (from 1 to the current upper bound, C–1) is compared against the colour of each coloured neighbour. The forward checking program uses a domain of size C–1 for the colour of each node; when a node is coloured, the chosen colour is removed from its neighbours' domains. The latter program uses the first-fail heuristic to decide the order in which to colour nodes; when more than one node has the smallest domain, the one with the greatest degree is chosen.

Tables 3 and 4 give the results of our two graph colouring programs run on several randomly generated, constant density, graphs. **N** is the number of nodes and **D** is the density (the probability that any two nodes are connected). **CN** is the chromatic number of the graph. In the top half of each table we keep the size constant and vary the density; below we keep the density constant and vary the size.

N	D	CN	BT	Time	And-//
30	0.1	3	92	8.04	4.62
30	0.3	5	1768	29.20	2.47
30	0.5	7	2891	49.53	2.50
30	0.7	10	5567	81.47	2.23
30	0.9	16	3610	87.72	3.10
10	0.5	4	26	1.26	3.71
20	0.5	6	372	12.86	3.75
30	0.5	7	2891	49.53	2.50
40	0.5	8	256888	1557.59	1.09

Table 3: Standard backtracking program for graph colouring

N	D	CN	BT	Time	And-//
30	0.1	3	1	10.68	3.63
30	0.3	5	2	23.09	5.30
30	0.5	7	3	35.72	6.01
30	0.7	10	5	58.98	6.16
30	0.9	16	1	60.55	6.89
10	0.5	4	1	1.67	3.41
20	0.5	6	1	11.56	5.28
30	0.5	7	3	35.72	6.01
40	0.5	8	9	97.96	6.35

Table 4: Forward checking program for graph colouring

The results show that the use of forward checking dramatically reduces the search space, and also reduces the sequential execution time, especially for larger graphs. Moreover, the and-parallel speedup is much greater for the forward checking program.

6 Conclusions

We have described some extensions to Andorra-I that allow us to experiment with finite domain constraint logic programming in a parallel context. These extensions were implemented with very little effort, thanks to the existing features of the Andorra-I system, such as its coroutining mechanism, updatable variables, variable locking, etc. We have also shown

how easily constraint programs can be written in the Prolog variant supported by Andorra-I. For example, provided that constraints are determinate — a very common case — they are automatically executed "actively", in preference to non-determinate guessing.

Our experiments have confirmed that programs which use constraints are much faster than similar generate-and-test programs, demonstrating that our implementation of forward checking has no substantial overhead.

The results of parallel execution are particularly interesting. Constraint programs exhibit greater and-parallelism than generate-and-test programs, because the extra computation involved in forward checking can be parallelized by solving constraints in parallel. Evidence of this is the difference between Tables 1 and 2, and between Tables 3 and 4. For example, on one processor, forward checking solves the 16-queens problem 12 times faster than standard backtracking, and colours the 40-node graph 16 times faster. On 10 processors, the speed improvement due to forward checking increases to 24 times and 92 times, respectively.

Or-parallelism is usually measured for all-solutions search, mainly because this gives more consistent results than a search for one solution since the whole search tree is explored. The or-parallel speedup for a first-solution search is very variable and depends heavily upon the nature of the or-parallel scheduler built into the system. However, in many combinatorial search problems it is impractical to search for all (or many) solutions, so it is arguably more realistic to measure performance for first-solution search. Our results always give a much smaller or-parallel speedup for the first solution than for all solutions.

For the generate-and-test program of Table 1, the or-parallel speedup does exceed the and-parallel one, which is negligible. However, for the forward checking program of Table 2, the opposite is true. Although the and-parallel speedup in Table 2 is not large, it is enough to tip the balance in favour of exploiting and-parallelism, given a choice.

Finally, we should mention that all of our results concerning and-parallelism are specific to the Basic Andorra model. This is because Andorra-I is the only serious Prolog implementation that features dependent and-parallelism. (It seems unlikely that a system with independent and-parallelism could give similar results, since forward checking involves the solution of constraints that are mutually dependent.) As we noted in Section 3, the Basic Andorra model has a sequential bottleneck with respect to and-parallelism, which is ameliorated by the use of constraint solving. It would be interesting to see whether our results extend to other computational models combining dependent and-parallelism and search. An example of such a model, not yet implemented, is the Extended Andorra model [Warren 1990], which can execute even non-determinate dependent goals in parallel and therefore should not have such a bottleneck.

Acknowledgements

This work would not have been possible without the work of colleagues who implemented various components of the Andorra-I system, in particular Tony Beaumont, Inês Dutra, and Vítor Santos Costa.

We are grateful to Reem Bahgat and the referees for helpful comments on a draft of this paper.

Rong Yang is supported by ESPRIT contract 2471.

References

[Bahgat and Gregory 1989] R. Bahgat and S. Gregory. Pandora: non-deterministic parallel logic programming. In *Proc. 6th Intl. Conf. on Logic Programming* (Lisbon, June). MIT Press, 1989, pp. 471-486.

[Dincbas *et al.* 1988] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a cutting-stock problem in constraint logic programming. In *Proc. 5th Intl. Conf. on Logic Programming* (Seattle, August). MIT Press, 1988, pp. 42-58.

[Santos Costa *et al.* 1991] V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I engine: a parallel implementation of the Basic Andorra model. In *Proc. 8th Intl. Conf. on Logic Programming* (Paris, June). MIT Press, 1991.

[Van Hentenryck 1989a] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[Van Hentenryck 1989b] P. Van Hentenryck. Parallel constraint satisfaction in logic programming. In *Proc. 6th Intl. Conf. on Logic Programming* (Lisbon, June). MIT Press, 1989, pp. 164-180.

[Van Hentenryck and Dincbas 1986] P. Van Hentenryck and M. Dincbas. Domains in logic programming. In *Proc. AAAI-86* (Philadelphia, August 1986).

[Warren 1990] D.H.D. Warren. The Extended Andorra model with implicit control. *ICLP90 Workshop on Parallel Logic Programming* (Eilat, Israel, June 1990).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Parallel Execution of Functional Logic Language with Lazy Evaluation

Jong H. Nang, D. W. Shin, S. R. Maeng, and Jung W. Cho
Department of Computer Science & Center for Artificial Intelligence Research
Korea Advanced Institute of Science and Technology
373-1 Kusong-Dong, Yusung-Ku, Taejon 305-701, Korea
e-mail : jhnang@adam.kaist.ac.kr

Abstract

In this paper, we propose a parallel computational model, called PR^3 (Parallel Resolution and Reduction with RAP), and its abstract machine for the parallel execution of Lazy Aflog program. Lazy Aflog together with its abstract machine FWAM-II was proposed as a cost-effective functional logic language. Since the parallel reduction of the arguments of a function can be regarded as a parallel evaluation of independent subgoals, only Independent-And Parallelism is exploited in PR^3 in order to simplify the execution control. PR^3 is an extension of DeGroot's RAP, and it is proposed as a simple and coherent parallelizing method that can be applied both of the function and logic. A parallel abstract machine for PR^3 based on the RAP-WAM is also developed, which is an extension of FWAM-II equipped with the run-time structures and the primitive instructions to spawn the parallel executions and gather the results. Simulation results show that both of the parallel resolution and parallel lazy reduction can be provided efficiently in the PR^3 and abstract machine.

1 Introduction

During the last couple of decades, there has been growing interest in functional languages and logic languages as potential alternatives to conventional languages, because of their declarative semantics and no side-effect. They have been widely used as system programming as well as application programming languages. Functional languages are characterized by reduction rules which make them procedural, while logic languages have the declarative flavour owing to their logical backgrounds. However, there exist software components which include both procedural and declarative part. Since defining them in one paradigm, procedurally or declaratively, would be unnatural and leads to inefficiency [Bellia and Levi 1986], there have emerged a lot of research efforts on the combination of two languages.

Lazy Aflog [Nang *et al.* 1991] is an E-Unification (Equality-Unification) based functional logic language, in which an E-Unification, called *E-Unification with lazy evaluation*, is developed to combine the lazy reduction of functional language and two-way argument passing of logic languages. Thanks to this E-Unification, the noticeable functional language features such as infinite data structures and higher-order function can be expressed naturally, while the expressiveness of the logic language such as non-determinism and unification is also maintained in the single framework. FWAM-II [Nang *et al.* 1991] is an abstract machine for Lazy Aflog, in which instructions and run-time structures to support the suspension and reactivation of functional closure are incorporated into WAM. We already demonstrated in [Nang *et al.* 1991] that this pair would be a good compromise between the expressiveness and efficiency of the combination.

Although FWAM-II is designed to maximize the performance on the conventional von-Neumann computers, it has the speed limitation because of its sequential nature. A natural way to improve the performance is to extend Lazy Aflog and FWAM-II pair in parallel, while keeping the performance optimizations and storage efficiency of sequential system. However, parallelizing Lazy Aflog computation is not a trivial problem, because we should deal with two different styles of parallelisms, one for logic part and the other for functional part. The simplest way in parallelization is to adopt already developed parallelizing schemes for each part, for example, Conery model [Conery 1983] for logic part and parallel graph reduction model such as GRIP [Peyton Jones *et al.* 1987] for functional part. It, however, requires a complex control mechanism to switch between the parallel execution of logic and functional part. Hence, instead of having two different schemes, it is highly desirable to develop a coherent one that could be applied to both logic and functional part.

Since the main parallelism in the functional part of Lazy Aflog program is the parallel reduction of arguments and it can be viewed as an Independent-AND Parallelism in the view point of logic language, the parallelisms in both parts can be exploited easily if there is a parallelizing method for Independent-AND Parallelism. The RAP Model [DeGroot 1984] is such a parallelizing method to spawn the parallel executions when there are independent subgoals in a clause. In this paper, we propose a parallel execution model for Lazy Aflog, called PR^3 (Parallel Resolution and Reduction with RAP), which is an extension of RAP. In PR^3, only independent subgoals in a clause and all the arguments of a strict function are resolved and reduced in parallel. Although this approach overlooks some available parallelisms in a Lazy Aflog program such as OR-Parallelism in logic part, it helps to avoid the complex run-time support.

In addition, this paper proposes an abstract machine for PR^3, called PFWAM-II (Parallel FWAM-II). It is an extension of FWAM-II equipped with the run-time structures and primitive instructions to spawn the parallel execution and gather the results. These run-time structures and instructions are inherited from the RAP-WAM [Hermenegildo 1986] with some modifications for the parallel lazy reduction of functional terms. Simulation

results to show the p-processor speed-up ratio over single processor are also presented to show the efficiency of PR^3 and PFWAM-II.

This paper is structured as follows. Section 2 briefly recalls our previous works on the Lazy Aflog and FWAM-II. A parallel computational model based on RAP for Lazy Aflog is presented in Section 3, while a parallel extension of FWAM-II for the parallel model is followed in Section 4. The simulation results that show the performance of the parallel extensions and a comparison with the related works are presented in Section 5. Finally, a summary of paper is presented in Section 6.

2 Lazy Aflog and FWAM-II

Lazy Aflog [Nang *et al.* 1991] is a successor of Aflog [Shin *et. al* 1987, Shin *et al.* 1988], to which the capability to process infinite data structures and higher-order function are added. FWAM-II is also a successor of the abstract machine for Aflog [Shin *et al.* 1992] with the primitives to suspend and reactive the functional closure at the machine instruction level. Lazy Aflog and FWAM-II pair was proposed as an effective mechanism to incorporate the functional features into logic. Now, let us explain Lazy Aflog and FWAM-II in more detail.

A Lazy Aflog program consists of a set of Prolog clauses and a set of function definitions (or rewrite rules) written in a constructor based functional language. The functional symbols in Lazy Aflog programs are classified into two disjoint sets : a set of *constructors* and a set of *defined functions*. A symbol f is a defined function if it appears at the left hand side of a rewrite rule, otherwise it is treated as a constructor symbol. In a Lazy Aflog program, a function application occurs as an argument of Prolog subgoal, which is reduced to its WHNF (Weak Head Normal Form) [Peyton Jones 1986] when the subgoal is resolved. This is the way in Lazy Aflog to incorporate functional programming into logic programming. Lazy Aflog imposes a restriction that all the arguments of a function should be ground before the function is reduced. Even though it prevents Lazy Aflog from having the powerful inferencing mechanism such as narrowing, it greatly contributes to the efficiency of the underlying E-Unification algorithm, because it assures that the E-unifier of two terms is unitary.

Let us explain the programming style and operational semantics of Lazy Aflog. Example 1 is the famous *Sieve of Eratosthenes* program which generates the list of all the prime numbers infinitely using lazy evaluation technique.

Example 1 *Sieve-of-Eratosthenes*

```
C1 : test(X) :- truncate(X, sieve(from(2))).
C2 : truncate(0,L).
C3 : truncate(X,[H | T]) :-
        print_era(X,H), Y is X - 1, truncate(Y,T).
C4 : print_era(X,H) :- write(X), tab(2), write(H), nl.

F1 : from(N) ==> [N | from(N+1)].
F2 : sieve([P | L]) ==> [P | sieve(filterp(P,L))].
F3 : filterp(P, [X | L]) ==> ((X%P) == 0 | filterp(P,L))
F4 :                         [X | filterp(P,L)].
```

In Example 1, a query ":- test(100)." generates 100 consecutive prime numbers as its result. In the course of the refutation of the query, the unification of *truncate(100, sieve(from(2))* in C_1 and *truncate(X, [H | T])* in C_3 is tried as follows :

```
      call E-Unify(sieve(from(2)), [H | T])
→       call E-Unify(from(2), [P | L])   /* by F2 */
→       exit E-Unify([2 | from(2+1)],
                     [2 | from((2+1))])  /* by F1 */
→     call E-Unify(sieve([2 | from(2+1)], [H | T])
→     exit E-Unify([2 | sieve(filterp(2, from(2+1)))],
            [2 | sieve(filterp(2,from(2+1)))])  /* by F2 */
```

In this E-Unification† process, the reduction of a functional term is initiated when a head pattern of a clause or rewrite rule is a non-variable term and the corresponding argument of the caller is a functional term. Note that the functional term is not completely reduced to its normal form, but to WHNF, which makes it possible to handle the infinite data structures. The complete description of the E-Unification algorithm, called *E-Unification with Lazy Evaluation*, is presented in [Nang *et al.* 1991].

FWAM-II, an abstract machine for Lazy Aflog, is an extension of WAM augmented with the manipulation of functional closure. It is characterized by that :

- it adds the reduction mechanism to the WAM architecture, and
- it employs an environment-based reduction rather than graph reduction.

Since WAM uses an environment for the variables in the body of a clause, the conventional environment-based reduction scheme is more suitable to WAM than the graph reduction is in the combination. Therefore, FWAM-II behaves similarly to the WAM in the execution of a clause, whereas it works similarly to an environment-based reduction machine in the reduction of functional term. This WAM-based approach has been also adopted in other abstract machines for the functional logic language, such as K-WAM [Bosco *et al.* 1989] for K-LEAF and a WAM model [Nadathur and Jayaraman 1989] for λ-Prolog. The E-unification of Lazy Aflog is realized in FWAM-II via the reducibility checking in the unification instructions, which immediately calls the reduction process if the passed argument is a functional term and corresponding pattern is not a non-variable term. To implement the suspension and reactivation of functional closure, a run-time structure (called, *Reduction Stack*) is added to WAM structure. Figure 1 shows a compiled FWAM-II code for the *filterp* function in Example 1, where *mode* and *eq* are predefined strict functions.

Upon the benchmark testing [Nang *et al.* 1991], the reduction mechanism of FWAM-II is relatively less efficient than WAM executing pure Prolog programs because of its overhead to construct and reference the functional closure, but it can support lazy evaluation in logic in the abstract machine level. Consequently, it is argued that FWAM-II can support not only all the features of logic language but also the essential features of functional language with the performance comparable to WAM.

3 A Parallel Computational Model for Lazy Aflog

Although FWAM-II would be an efficient sequential abstract machine for Lazy Aflog, it has the speed limitation because of its sequential nature. A natural way to overcome this obstacle is to extend it in parallel. This

F_1 : filterp(P, [X | L]) => ((X%P) == 0 | filterp(P,L))
F_2 : [X | filterp(P,L)].

F_1:	allocate	3
	% Pattern Matching	
	fget_value	'P', X1
	fget_list	X2
	match_value	'X'
	match_value	'L'
	%Guard Checking	
	try_me_else_L	F_2
	put_value	X, X1
	put_value	P, X2
	call_P_Arity_N	*mode*/2, 2
	put_integer	0, X2
	call_P_Arity_N	*eq*/2, 2
	%Committing	
	commit	
	%Construct WHNF	
	write_function	'filterp/2', X1
	write_value	'P'
	write_value	'L'
	rewrite_value	X1
	% Returning	
	return	
F_2:	*trust_me_else_fail*	
	write_function	'filterp/2', X1
	write_value	'P'
	write_value	'L'
	write_list	X2
	write_value	'X'
	write_value	X1
	rewrite_value	X2
	return	

Figure 1 A Compilation Example

section addresses our point of view that adopts the RAP as our starting point, and presents a parallel computational model for Lazy Aflog.

3.1 Parallelisms in Lazy Aflog Programs

Lazy Aflog has various kinds of parallelisms inherited from both function and logic, such as AND-Parallelism, OR-Parallelism, and Argument-Parallelism. Among these parallelisms, we adopt the Independent AND-Parallelism as the primary parallelism owing that :

- Ideally, all parallelisms in the Lazy Aflog program can be exploited in the parallel extension. However, it may require a complex control mechanism that may degrade the performance gains obtained through the parallel execution.
- Since the Argument-Parallelism in the functional language part can be viewed as a kind of Independent-AND Parallelism in the logic language part, we can exploit parallelisms in both of the functional and the logic parts in a simple and coherent manner if there is a parallelizing method for it.
- There have emerged an efficient and powerful computational model and an abstract machine for Independent-AND Parallelism of logic programs. DeGroot's RAP Model and RAP-WAM [Hermenegildo 1986] are such a computational model and an abstract machine, respectively.

3.2 A Parallel Computational Model : PR^3

A parallel computational model for Lazy Aflog, called PR^3 [Nang 1992], is a parallel model which can support both of the parallel resolution and parallel lazy reduction simultaneously. The basic principle to spawn a parallel task is as follows ;

Rule 1) the subgoals in a clause are executed in parallel when their arguments are *independent* or *ground*

Rule 2) the arguments of a functional term are reduced in parallel when their WHNFs are demanded and the function is a strict one

Rule 3) the alternative clauses and rewrite rules are tried sequentially using the top-down strategy

---

The algorithm of *independent* and *ground* are same as the ones defined in [DeGroot 1984]. This principle can be expressed with an intermediate code, called CGE^+ (*Conditional Graph Expression*$^+$), which is an extension of DeGroot's CGE [DeGroot 1984]. It is used to express the necessary conditions to spawn the subgoals or function reductions in parallel. The body of a clause and right-hand side of a rewrite rule are expressed by the CGE^+, which is informally defined as follows;

---

1) G : a simple goal (or subgoal) whose argument can be a functional term.
2) $(SEQ\ E_1 \cdots E_n)$: execute expressions E_1 through E_n sequentially
3) $(PAR\ E_1 \cdots E_n)$: execute expressions E_1 through E_n in parallel
4) $(GPAR\ (V_1 \cdots V_k)\ E_1 \cdots E_n)$: if all the variables V_1 through V_k are ground, then execute expressions E_1 through E_n in parallel ; otherwise, execute them sequentially
5) $(IPAR\ (V_1 \cdots V_k)\ E_1 \cdots E_n)$: if all the variables V_1 through V_k are mutually independent, then execute expression E_1 through E_n in parallel; otherwise, execute them sequentially
6) $(IF\ B\ E_1\ E_2)$: if the expression B is evaluated to true, execute expression E_1; otherwise, execute expression E_2
7) $F(SEQ\ F_1 \cdots F_n)$: if F is a construct symbol or non-strict function symbol, then construct WHNF $F(F_1 \cdots F_n)$ sequentially ; otherwise (*i.e.* F is a strict function symbol) evaluate expressions F_1 through F_n sequentially and eventually evaluate $F(F_1' \cdots F_n')$
8) $F(PAR\ F_1 \cdots F_n)$: if F is a construct symbol or non-strict function symbol, then construct WHNF $F(F_1 \cdots F_n)$ sequentially ; otherwise (*i.e.* F is a strict function symbol) evaluate expressions F_1 through F_n in parallel and eventually evaluate $F(F_1' \cdots F_n')$

The expressions 1) through 6) are the same as the DeGroot's CGE for the clause (actually they are improved CGE defined in [Hermenegildo 1986]), while expressions 7) and 8) are new expressions for rewrite

rules. Note that there are no conditions to check the groundness of function arguments in the expressions 7) and 8), since they are automatically checked by pattern matching semantics of the rewrite rules. That is, the arguments of a rewrite rule are always ground, hence, they can be always evaluated in parallel. In expression 8), the arguments are reduced in parallel only if the function is strict, which results in its WHNF. Otherwise, it is rewritten to the term in the right-hand side, which is returned as the result. In this case, as it is not WHNF, it induces another reduction process. The reason to adopt this reduction strategy rather than directly call the non-strict function, is in order to keep the storage optimization based on tail-recursion.

Example 2 is a CGE^+ for a Lazy Aflog program. It can be automatically generated from the Lazy Aflog program by the parallelizing compiler, or programmed directly by the programmer.

Example 2 A CGE^+ for the Lazy Aflog program

C_1 : test(X,Y) :- (**IPAR** (X,Y) p(X,Z) q(Y,W)), r(f(Z), g(W)).
C_2 : test(X,Y).
C_3 : p(a,1). C_5 : q(c,3).
C_4 : p(b,2). C_6 : q(d,4).
C_7 : r(2,5).
C_8 : r(4,72).
F_1 : f(X) ==> (X == 0) | 0,
+(**PAR** fib(X) fib(2*X)).
F_2 : g(Y) ==> *(**PAR** factorial(Y), fib(Y)).

In Example 2, as the subgoals $p(X,Z)$ and $q(Y,W)$ would generate the values of Z and W that are taken into the terms $f(Z)$ and $g(W)$, the goal $r(f(Z),g(W))$ should be executed after the evaluation of them. Figure 2 is the snapshots of the parallel execution of the CGE^+ in Example 2 when "Q_1 :- test(b,d)" is given. In Figure 2, the rectangle, circle and rounded-rectangle represent OR node, AND node and reduction node, respectively. The number attached to each node represents the order of execution, while the filled nodes represent the activated nodes at that time. Note that, since the *Unification Parallelism* is not exploited in PR^3, the functional terms $f(1)$ and $g(3)$ in the step (c) are reduced sequentially, although they can be evaluated in parallel if the *innermost-like* reduction strategy is used. The backward execution of the PR^3 is the same as the one presented in [Hermenegildo 1986] because there are no backtracking in the reduction phases after a functional term is eventually reduced to WHNF. For example, in the step (d), the subgoal $q(Y,W)$ which generate the arguments W would search alternative solutions for Y and W when a fail is occurred, rather than to generate another WHNF for $g(3)$ or $f(1)$.

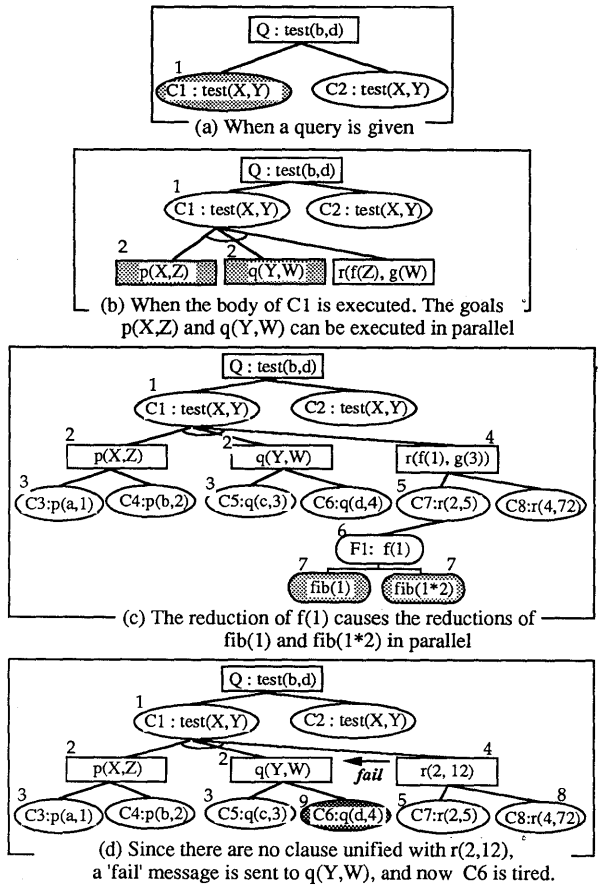


Figure 2 The Parallel Execution Snapshots of the Lazy Aflog Program in Example 2

4 A Parallel Extension of FWAM-II for PR³

The desirable characteristics of parallel abstract machine is to support the parallel execution while retaining the performance optimizations offered by the current sequential systems. To achieve this goal, a parallel abstract machine for PR^3, called PFWAM-II (Parallel FWAM-II), is designed as an extension of the sequential abstract machine FWAM-II. It is equipped with the run-time structures and instruction set to fork and join the parallel executions. We adopted the run-time structures and instructions of the RAP-WAM for the extension of FWAM-II because it is also an extension of WAM for AND-Parallel execution of Prolog and has a general primitive to fork and join the parallel tasks. Figure 3 shows the relationships between WAM, FWAM-II, RAP-WAM, and PFWAM-II.

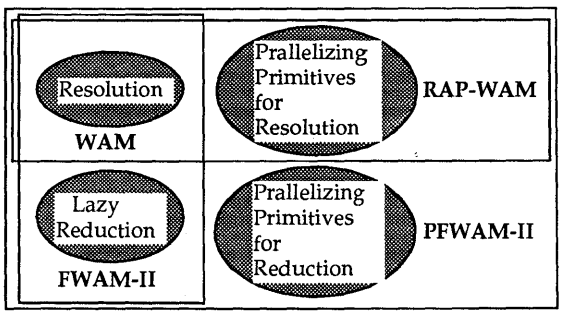


Figure 3 The Relationships Between WAM, FWAM-II, RAP-WAM and PFWAM-II

4.1 Run-Time Structures for Parallel Execution

The run-time structure of PFWAM-II is an extension of FWAM-II for parallel executions as shown in Figure 4. It consists of three parts ; First, the Heap, Trail, Environment, and Choice Point are structures for the execution of the logic part, and inherited from WAM ; Secondly, RS (Reduction Stack) is the structure only for the function reduction, and inherited from FWAM-II ; Finally, GS (Goal Stack), ParCall Frame, Local Goal Marker, Input Goal Marker, and Wait Marker are run-time structures for the parallel executions of subgoal or function reduction, that are inherited from RAP-WAM with slight modifications.

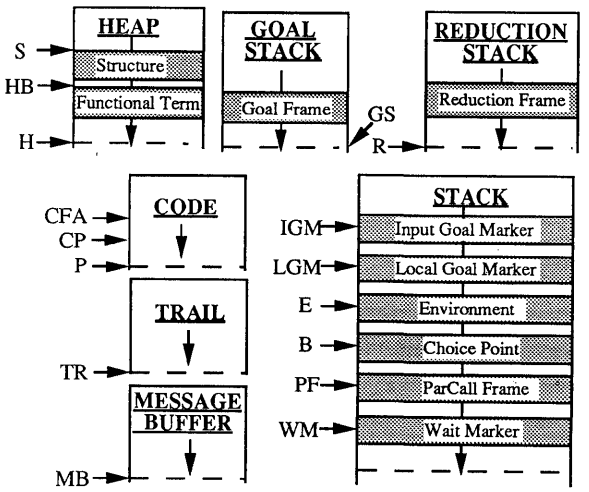


Figure 4 Data Areas and Registers for One PFWAM-II

In fact the run-time structures of the parallel execution is almost the same as that of RAP-WAM except that a parallel task in PFWAM-II can be a reduction of a functional term as well as the evaluation of subgoal, whereas in RAP-WAM, only the evaluation of a subgoal can be a parallel task. The run-time structure for parallel execution are the *Goal Frame, ParCall Frame, Input Goal Marker, Local Goal Marker,* and *Wait Marker.* Let us explain them focusing on the extensions which allow them to be also used for function reduction.

- *The Goal Frame* :
  The subgoals or the functional terms which are ready to be executed in parallel are pushed onto the Goal Stack. Each entry in the GS is also called a *Goal Frame* as in RAP-WAM. A *Goal Frame* contains all the necessary information for the remote execution of tasks. There are two kinds of *Goal Frame* in PFWAM-II; one is for a subgoal, and the other is for a function reduction. They are distinguished by the special tag in the *Goal Frame.* When a *Goal Frame* is the one for the subgoal, the structure of *Goal Frame* is the same as in RAP-WAM ; otherwise (*i.e.* it is one for the function reduction), it contains the extra pointer to the functional term to be reduced. In both cases, they are stolen from *Goal Stack* by a remote processor, and executed remotely in the same way.
- *The ParCall Frame* :
  It is used to keep track of the parallel tasks during forward and backward executions of PR^3. The entries and meanings of the ParCall Frame that is created for each parallel task are the same as in RAP-WAM. If a *ParCall Frame* is the one for the parallel function reductions, it immediately disappears from the *Local Stack* when the parallel reductions are completed because there is no backtracking in the reduction process. It is different from the case of parallel subgoal calls, in which it remains in the *Local Stack* in order to select the appropriate actions during backtracking.
- The entries and meanings of the *Input Goal Marker, Local Goal Marker,* and *Wait Marker* are the same as in RAP-WAM. However, they also immediately disappears when the task is a function reduction and it is reduced to WHNF.

The general execution scenario of PFWAM-II is as follows. As soon as a processor steals a task from another processor's *Goal Stack,* it creates an *Input Goal Marker* on its top of *Local Stack,* and checks whether it is a subgoal or a function reduction. If it is a subgoal, the processor starts working on the stolen subgoal by loading its argument registers from the parameter register fields in the *Goal Frame* and fetching instructions starting at the location (procedure address) received. If the stolen task is a function reduction, the processor loads the arguments and finds the starting address of the corresponding rewrite rule by referencing the functional term stored in the *Heap* of the parent processor. It was recorded on the *Goal Frame* by the parent processor. At any case, the local stacks of the processor will then grow (and shrink) as indicated by the semantics of FWAM-II.

When a parallel call is reached, a ParCall frame is created on the top of the *Local Stack* and tasks are pushed on to the *Goal Stack.* If there are no idle processors in the system at that time, the processor itself gets the goal from its *Goal Stack* again, makes a *Local Goal Marker,* and executes the task locally. If the parallel call is one for the subgoals, an *Wait Marker* is created on the top of the *Local Stack* as soon as all subgoals succeed. It is used for the backward execution of PFWAM-II. However, if the parallel call is for the function reduction, the *ParCall Frame, Local Goal Marker,* or *Input Goal Marker,* created on the local Stack can be removed since there is no backtracking in the reduction process. After the parallel call is finished, the execution can continue normally beyond the parallel call.

4.2 Instruction Set

The instruction set of PFWAM-II consists of the FWAM-II instructions and the new instructions implementing RAP as shown in Table-1. Since the FWAM-II instructions were explained in [Nang *et al.* 1991] and the instructions to fork and join the parallel call when tasks are subgoals are almost the same as the RAP-WAM, we only explain the instructions to control the parallel reduction. To fork and join the parallel executions are actually the same as the RAP-WAM when the parallel call is a determinate one. However, some attentions are required since the tasks to be forked can be functional terms.

- *push_reduce Vn, Slot_Num*
  It makes a new goal frame on the *Goal Stack* with the *Slot_Num* for the functional term pointed by Vn.

<Table-1> The PFWAM-II Instruction Set

The PFWAM-II Instruction Set					
WAM Instructions					
Procedure Control		**Indexing**		**Clause Control**	
try	L	switch_on_term	Ai, v,c,l,s	call	P/arity
retry	L	switch_on_constant	n, ff	execute	
trust	L	switch_on_structure	n, ff	proceed	
try_me_else	L			allocate	
retry_me_else	L			deallocate	
trust_me_else	fail				
Get		**Put**		**Unify**	
get_variable	Vi, Ai	put_variable	Vi, Ai	unify_variable	Vi
get_value	Vi, Ai	put_value	Vi, Ai	unify_value	Vi
		put_unsafe_value	Yi, Ai	unify_unsafe_value	Yi
get_constant	C, Ai	put_constant	C, Ai	unify_constant	C
get_list	Ai	put_list	Ai	unify_list	
get_structure	S, Ai	put_structure	S, Ai	unify_structure	S
get_nil	Ai	put_nil	Ai	unify_nil	
				unify_void	
Reduction Instructions					
Fget		**Matching**		**Writing**	
fget_value	Vi, Ai	match_value	Vi	write_value	Ai
fget_constant	C, Ai	match_constant	C	write_constant	C
fget_list	Ai	match_structure	S	write_structure	S
fget_structure	S, Ai	match_list		write_list	Ai
fget_nil	Ai			write_function	F, Ai
				write_structure_value	S, Ai
Reduction Control		**Rewriting**		**Reducing**	
commit		rewrite_value	Vi	reduce_value	Ai
return					
Parallel Abstract Machine Specific Instructions					
RAP-WAM Instructions				**Parallel Reduction Specifics**	
check_me_else_label	Label	push_call	Pid,Arity,Slot#	push_reduce	Vn, Slot_#
check_ground	Vn	check_ready	Slot_#, Label	deallocate_pcall	
allocate_pcall	#_of_slot, M	check_independent	Vn, Vm		
pop_pending_goal		waiting_on_siblings			

- *deallocate_pcall*
  It is used to join the parallel reductions. It waits until the number of goals to wait on in current *ParCall Frame* is 0; then, removes the current *ParCall Frame* from the local Stack.

Figure 5 shows the simplified PFWAM-II codes for F_2 of the CGE^+ in Example 2, in which since '+' and '*' are strict functions, their arguments are reduced directly rather than constructing the functional closure.

F_2 : g(Y) ==> *(**PAR** factorial(Y), fib(Y)).

```
F2:   allocate
      % Pattern Matching
      fget_value                Y1, X1
      % Spawn Parallel Reduction for factorial(Y)
      allocate_pcall            2, 2
      put_value                 Y1, X1
      write_function            factorial/1, Y2
      write_value               X1
      push_reduce               Y2, 2
      % Spawn Parallel Reduction for fib(Y)
      put_value                 Y1, X1
      write_function            fib/1, Y3
      write_value               X1
      push_reduce               Y3, 1
      % Gather the Results
      pop_pending_goal
      deallocate_pcall
      % Construct WHNF
      put_value                 Y2, X1
      put_value                 Y3, X2
      call_P_Arity_N            */2, 2, 1
      rewrite_value             X1
      % Returning
      return
```

Figure 5 An Compilation Example for CGE^+ in Example 2

5 Analysis

5.1 Performance Evaluation

In order to estimate the performance of our parallel extension, a simulator for PFWAM-II is developed. In this simulation, we assumed that there is a common shared memory for the run-time structures of each processor which are interconnected by a network. Each processor can access the run-time structures of other processors without additional overheads. The performance of PFWAM-II is estimated by counting the number of memory and register references, where the time for referencing data stored in the shared memory (whether it is local or not) is assumed 3 times longer than the time for register referencing, and the times for other operations such as arithmetic are ignored for the sake of simplicity.

We use three benchmark programs : the first one is *Fibonacci10* that is to compute the 10th fibonacci number, the second is *Check50* [Hermenegildo 1986] in which there are 10 parallel tasks each of which calls itself 50 times, and the third is *Symbolic Derivation* [Hermenegildo 1986] which is to find the derivative with respect to a variable. There are 176 parallel tasks in the *Fibonacci10*,

10 parallel tasks in the *Check50*, and 152 parallel tasks in the *Symbolic Derivation*. These benchmarks are programmed in both of logic and functional programming. In the simulation of function reduction, the effect of different reduction strategies is also measured. The simulated reduction strategies are *Innermost Reduction* in which the innermost functional terms are reduced first before the outer is tried, *Semi-Lazy* in which only the strict functions are reduced in the innermost fashion, and *Lazy Reduction* in which all functions are reduced in the outermost fashion.

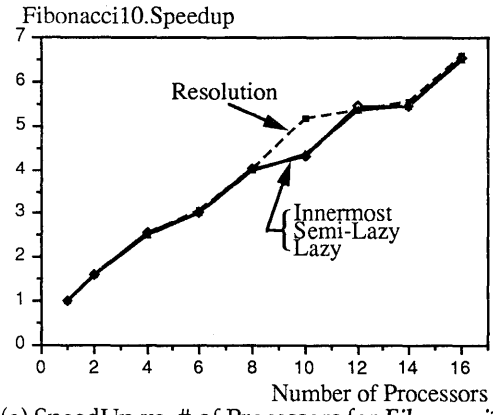
Upon the simulation results, the parallelizing overhead, which is defined as the extra execution time for parallel code running on the single processor, is measured as about 30~60 % when the grain size is relatively small (for example, *Fibonacci10* and *Symbolic Derivation*), whereas about less than 1 % when the grain size of parallel task is large enough to ignore the overhead (for example, *Check50*). Figure 6 graphically shows the speedup of the execution time of all benchmark programs as a function of the number of processors. In this figure, since *Check50* has only 10 parallel tasks, the speedup doest not increase when the number of processors is larger than 10. The speedup of other benchmark programs are not linear because they have too fine-grained parallelism. The most important fact which can be identified from Figure 5 is that, *whether they are programmed in the logic or functional style, and whether the reduction strategy is innermost or outermost, the speedup behaviour is almost same.* The speedup ratio is not dependent on the execution mechanisms, but the availability and grain size of parallelism in the benchmark programs. In other words, PFWAM-II can support both of the parallel resolution and parallel reduction with the almost same efficiency.

Figure 7 shows the Working, Waiting, and Idle times for *Symbolic Derivation* as a function of the number of processors. It is from identified from Figure 7 that the processor utilization ratio is reduced proportional to the number of processors, and the parallel reduction mechanism permits higher utilization ratio than parallel resolution because there is no restriction to steal a task from other processors when the task is a function reduction (*i.e.*, there is no "garbage slot problem [Hermenegildo 1986]" when executing the function reduction).

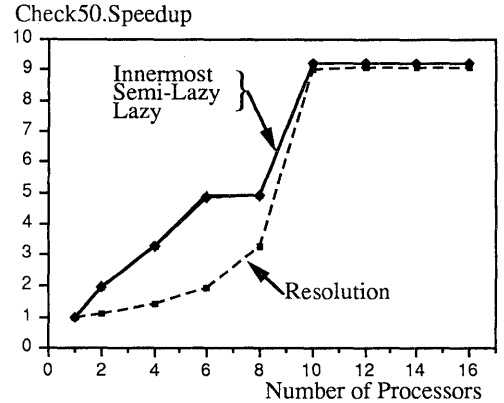
5.2 Comparison with Related Work

One of the most related works is the CSELT's work centering around K-LEAF. K-LEAF [Levi and Bosco 1987] is a functional logic language based on the transformation. A rewrite rule in K-LEAF program is transformed into Prolog clause with an extra argument for the return value, and the nested function is flattened with *produced variable* for the outermost search strategy. K-WAM is an abstract machine to support outermost-SLD resolution which is the inference rule of K-LEAF. Accordingly, there is no real reduction mechanism in K-LEAF and K-WAM.

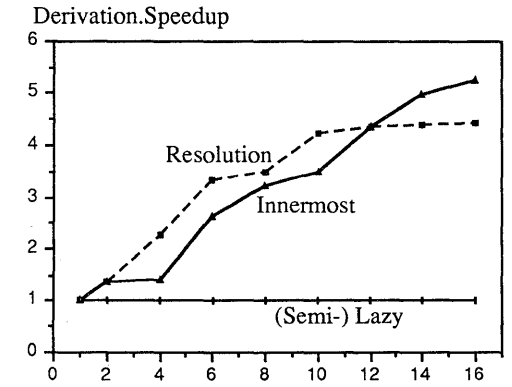
A parallel extension of K-WAM on a distributed memory multiprocessor is also developed [Bosco *et al.* 1990]. In this work, K-WAM is extended to control the OR-parallel execution of K-LEAF programs, and AND-parallelism is restricted to be *one-solution*. The major difference between the parallel extension of K-WAM and PFWAM-II is that the former is designed for exploiting only OR-parallelism in the flattened K-LEAF programs, while the later is designed for exploiting only AND-parallelism of Lazy Aflog programs.



(a) SpeedUp vs. # of Processors for ***Fibonacci10***



(b) SpeedUp vs. # of Processors for ***Check50***

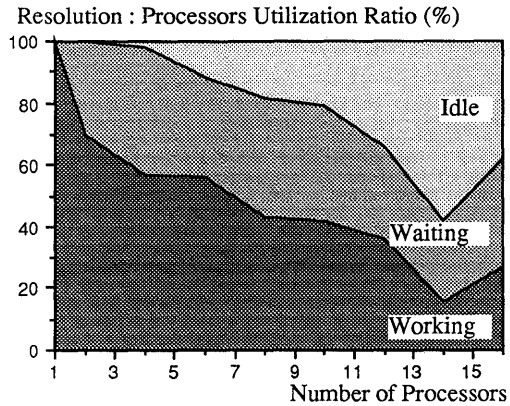


(c) SpeedUp vs. # of Processors for ***Symbolic Derivation***

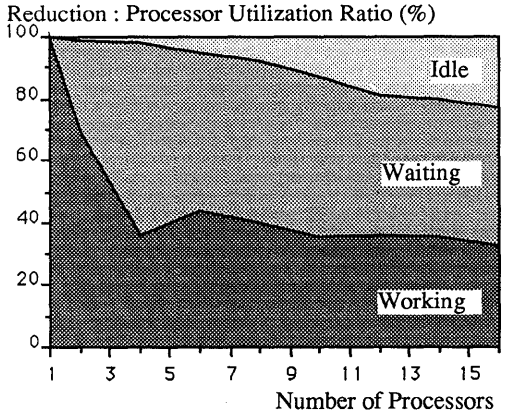
Figure 6 SpeedUp vs. Number of Processors for Benchmark Programs

6 Summary

This paper presents a pair of a parallel computational model and its abstract machine for a functional logic language, called Lazy Aflog, which was proposed as a cost-effective mechanism to incorporate functional language features into logic language. The proposed



(a) Case of Logic Programming
(Parallel Resolution)



(b) Case of Functional Programming
(Parallel Reduction)

Figure 7 Working, Waiting Idle Times for *Symbolic Derivation*

computational model underlies DeGroot's RAP model because the Restricted-AND Parallelism could be easily exploited in both of the function and logic. However, some extensions are required since there is a parallel function reduction in the functional part of Lazy Aflog programs. Since RAP-WAM includes the general structures to fork and join the parallel tasks, the parallel function reductions can be also supported efficiently with slight extension. A parallel abstract machine based on the RAP-WAM and extension of FWAM-II, called PFWAM-II, is also proposed as an implementation method on a multiprocessor. Several simulation results show that PFWAM-II can support not only the parallel resolution, but also parallel reduction with the almost same efficiency.

References

[Bellia and Levi 1986] M. Bellia and G. Levi, The Relation between Logic and Functional Languages : A Survey, *Journal of Logic Programming*, Vol. 3, No. 3, 1986, pp. 217-236.

[Bosco *et al.* 1989] P. G. Bosco, C. Cecchi, C. Moiso, An Extension of WAM for K-LEAF : a WAM-based compilation of conditional narrowing, *Proc. of 6th Int'l Conference on Logic Programming*, MIT Press, 1989, pp. 318-333.

[Bosco *et al.* 1990] P. G. Bosco, C. Cecchi, C. Moiso, M. Porta, and G. Sofi, Logic and Functional Programming on Distributed Memory Architectures, *Proc. of 7th Int'l Conference on Logic Programming*, 1990, pp. 325-339.

[Conery 1983] J. S. Conery, *The AND/OR Process Model for Parallel Execution of Logic Programs*, PhD These, Univ. of California, Irvine, 1983.

[DeGroot 1984] D. DeGroot, Restricted AND-Parallelism, *Proc. of FGCS84, ICOT, 1984, pp. 471-478.*

[Hermenegildo 1986] M. V. Hermenegildo, *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, TR-86-20, Dept. of Computer Science, Univ. of Texas at Austin, 1986.

[Levi and Bosco 1987] G. Levi and P. G. Bosco, A Complete Semantic Characterization of K-LEAF, A Logic Language with Partial Functions, *Proc. 1987 Symposium on Logic Programming*, IEEE Computer Society Press, 1987, pp. 318-327.

[Nadathur and Jayaraman 1989] G. Nadathur, B. Jayaraman, Towards a WAM Model for λProlog, *Logic Programming : Proceedings of the North American Conference*, MIT Press, 1989, pp. 1180-1198.

[Nang *et al.* 1991] Jong H. Nang, D. W. Shin, S. R. Maeng, and J. W. Cho, An Effective Incorporation of Function into Logic, *Information and Software Technology*, Vol 33, No. 8, Butterworth Heimann Ltd., U.K, 1991.

[Nang 1992] Jong H. Nang, *A Study on the Execution Mechanisms of a Functional Logic Language with Lazy Evaluation*, PhD Dissertation, KAIST, 1992.

[Peyton Jones 1986] S. L. Peyton Jones, *The implementation of functional programming languages*, Prentice-Hall, 1986.

[Peyton Jones *et al.* 1987] S. L. Peyton Jones, C. Clack, J. Salkid, and M. Haridi, GRIP - A High Performance Architecture for Parallel Graph Reduction, *Proc. of Functional Programming Language and Computer Architecture*, Springer-Verlag LNCS 274, 1987, pp.98-112.

[Shin *et. al* 1987] D. W. Shin, J. H. Nang, S. Han, and S. R. Maeng, A Functional Logic Language Based on Canonical Unification, *Proc. 1987 Symposium on Logic Programming*, IEEE Computer Society Press, 1987, pp. 328-334.

[Shin *et al.* 1988] D. W. Shin, J. H. Nang, S. R. Maeng, and J. W. Cho, The Semantics of a Functional Logic Language with Input Mode, *Proc. of the International Conference On the Fifth Generation Computer Systems 1988*, 1988.

[Shin *et al.* 1992] D. W. Shin, J. H. Nang, S. R. Maeng, and J. W. Cho, A Functional Extension of Logic Programming, *New Generation Computing*, 1992 (accepted for publication).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Self-organizing Task Scheduling for Parallel Execution of Logic Programs

Zheng Lin*

Department of Computer Science
University of Maryland
College Park, Maryland 20742
zlin@cs.umd.edu

Abstract

A new scheduling scheme is proposed which directs processors to share the search space according to universal task distribution rules obeyed by all processors involved. Load balancing is achieved by altering the shape of a search tree to remove the so-called structural imbalance, and following a statistically even distribution rule. A condition for task distribution is derived which minimizes the average parallel runtime. We present data showing the effectiveness of the proposed scheme. Simulation results from benchmark programs that can be found in literature demonstrate that the method is able to efficiently treat programs that render mostly fine-grained parallel tasks under a typical existing scheduler. The peak speed-up factors with the proposed technique exceed by a substantial margin that achieved by Aurora Parallel Prolog on the same set of benchmarks.

Key Words: Efficiency, Logic programming, Load balancing, Parallel execution, Scheduling, Speed-up.

1 Introduction

Load balancing is the key to obtaining maximum utilization of a multiprocessor system. Parallel execution of a logic program creates many tasks that need to be assigned to processors at run time. Detecting available tasks at run time and migrating tasks among processors is expensive. This is particularly acute for systems in which communication overhead is high due either to architectural reasons, or to a large number of processors being used, because a traditional task scheduler relies heavily on shared resources, shared memory or interconnection network, to perform its functions. As the scale of a multiprocessor system grows, and the speed of implementing resolution in local processor improves[1], task scheduling becomes increasingly frequent. However, the speed of the scheduler cannot be expected to increase proportionally if the scheduler continues to operate on resources shared by all processors. This motivates us to search for schemes that are less reliant on resources subject to competition by all processors in a multiprocessor system.

In this paper we discuss a scheduling scheme called *self-organizing scheduling* which directs processors to share the search space, the search tree defined implicitly by a program, according to task distribution rules followed by all processors. We discuss methods, including program restructuring and a new interpretation of so-called choice predicates, that help to alter the shape of the search tree so as to facilitate maintaining load balance with a probabilistic task distribution rule. We derive a condition for task distribution that minimizes the parallel runtime. Experimental data are presented showing the effectiveness of the methods. Empirically, many programs that were frequently used as Or-parallelism benchmarks in the literature can be restructured to effectively exploit the advantage provided by the proposed scheduling method. For problems with fine-grained parallelism (e.g. a tightly written 8-queens, zebra, turtles programs, running on 30 or more processors) whose speed-up factors reach peaks at less than 30 processors on a typical Or-parallel Prolog system, we found that the peak speed-up factors can be doubled or tripled using the self-organizing scheduling method *even without* resorting to communication.

The paper is organized as follows: section 2 provides background on parallel execution of logic programs; section 3 discusses the proposed methods; section 4 presents the experimental results, and comparison with existing systems; section 5 discusses advantage and limitation of the proposed method, and possible solutions; section 6 describes related work; section 7 concludes our work.

2 Background

We consider a logic program to be a set of Horn clauses written as,

$$H : -B_1, B_2, ..., B_n$$

*This work is supported by AFOSR grant AFOSR-91-0350 and NSF grand IRI-89-16059.

[1] Speed of sequential Prolog implementation has been improved drastically over the past several years. New developments have been reported [VR90] which could lead to improvement in speed in the order of several times that of the current best Prolog implementations.

where H, the head of the clause, is a positive literal and the $B_i's$, the body of the clause, are conjunction of either positive of negated literals (possibly empty). The intuitive interpretation of the above rule is *if all B_i's are solved then H is considered solved.*

A query is written as : $-Q$, where Q is a conjunction of literals. Evaluation of Q starts with clause : $-Q$, using resolution [Lloyd84] to derive an empty clause should one exist. There may be multiple selection of rules at each resolution step. All solutions can be found by exhausting every possible alternative in the program. The resolution process can be visualized as the construction of a search tree (backtracking tree, proof tree) for the given query. Given a program and a query, the tree is implicitly defined.

We define a *partition* of the tree as a part of the tree that consists of a set of nodes reachable from the root of the tree. Two partition are *disjoint* if there is no common leaf node in the partitions. We note that a *partition* always contains a path from the root.

2.1 Or-Parallel Execution of a Logic Program

Or-parallel execution of a logic program can be viewed as having multiple processors (resolution engine, workers) simultaneously exploring different parts of a search tree defined implicitly by the program. Execution starts with the original goal sent to one of the workers. The goal is expanded by resolving one of its atoms (the leftmost one in the case of Prolog) with clauses which have matching heads. If more then one potential subgoal is generated, and if there are idle workers, the extra subgoals are made available to the idle workers. Any unsolved subgoal that remains is solved upon backtracking. The procedure repeats until all workers finish their tasks. In this paper, we are concerned only with the situation in which the tree is finite and all solutions need to be found. In other words, the entire search tree is explored.

Task scheduling consists of searching for available tasks (or processors) and transferring a task. Transferring a task from one processor to another means migrating the state (variable bindings, control information, etc.) of one processor corresponding to the task to another processor. Different execution models handle task migration differently [Ali90, But88, Mud91, Kale85, Lusk, Clock88, Giul90], with the objective of balancing load distribution with as little communication as possible. A common characteristic of existing methods is that processors cope with the dynamically changing search space by interchanging messages to detect where a task is available and migrate to the task. While this approach has an obvious advantage of automatically adapting to the shape of the search tree, the overhead of scheduling can be unnecessarily high especially for fine-grained tasks. This will become clear when performance data is presented from a typical Or-parallel system later in this paper.

With increasingly fast implementation of sequential resolution engines, and even larger scale multiprocessor systems available, the issue of scheduling has added a new element of how to keep up with the speed of the resolution engine which operates primarily on local and private resources. The computing power of fast local resolution engines can be utilized fully *only when the scheduler is able to allocate tasks for them efficiently.*

We investigate a method that divides the search space and coordinates the search by following universal rules rather than via communication among processors. We describe the method and present performance results in following sections.

3 Self-organizing Scheduling

The idea of the method is to allow each processor to decide, locally, a partition (defined in above section) in the search tree to explore, according to universal rules agreed on by the whole system. It works as follows: every processor obtains a copy of the original goal (the root of the search tree), and performs a depth-first search on the tree. At each node, a processor expands *all* children of the node and claims those belonging to it according to rules agreed on by every other processor in the system, then processes them *independently.* The decision of which path(s) to pursue is made locally by each processor. No dialogue among processors is necessary until the first processor completes the task it claims.

An ideal situation would be that each processor obtains a partition of equal size. However, this is unlikely unless the granularity of a task is predictable. We propose a program restructuring method which alters the shape of the search tree so as to facilitate a probabilist task distribution rule, which will be discussed later.

Or-parallel branches in the search tree are created by the selected literal (for expansion) unifying the heads of multiple rules. Imbalances of the tree are results of either 1) terminated branches (cut-offs) or 2) syntactic characteristic of the program which results in an imbalanced search tree, as will be referred to as *structural imbalance* in the rest of the paper.

An important class of programs written in a logic programming language are the generate-and-test programs, where the generating phrase produces candidates, stored in a structure, and the testing phrase retrieves and tests a candidate from the structure. Generating and testing can be interwaiving. While we do not know yet how to speculate on cut-offs, structural imbalances can be cured by changing the way the candidates are retrieved.

To illustrate the idea, we examine a Prolog predicate, the *member* predicate. This predicate, and its variation, can be found in many normal style generate-and-test programs as a mean to create alternative choices. The predicate is usually defined as,

```
member(X,[X|Y]).
member(X,[H|Y]) :- member(X, Y).
```

Given a list as the second argument, *member* returns an element from the list in the first argument of the predicate. All element can be retrieved eventually by exhausting, recursively, all alternatives.

Predicates which represent multiple choices in the resolution are refered to as *choice* predicates, as oppose to *determinate* predicates which has only one valid choice. The *member* predicate defined above is a choice predicate when called with an instantiated second argument and uninstantiated first argument. Notice that whether or not a predicate is a choice predicate is contingent on not only the way it is written but also the argument pattern it is called with. A recursive choice predicate and a recursive determinate predicate is not distinguishable syntactically in Prolog. We assume choice predicate are explicitly identified with annotation suplied by users. This assumption is consistent with practice in many existing parallel Prolog systems[But88, Ali90], which require explicitly distinction between predicates to be evaluated sequentially or in parallel.

At run time the normal style *member* predicate defined above produces a search tree "biased" to the right: the left child of a node in the tree corresponds to the first rule and the right subtree of a node corresponds to the second recursive rule of the definition. When this predicate is embeded in a program, a left branch so generated represents **one** element of the given list to be processed, and a right branch represents **the rest** elements to be processed. *The difference cannot be observed by the resolution engines being at the parent node of the branches.* Furthermore, the degree of bias is magnified if the predicate is called from inside a loop.

3.1 Flattening Choice Predicates

Program Restructuring: Retrieving members of a given structure can be written in a non-recursive form. For instance, the *member* predicate can be defined as,

```
member(X,[X|Y]).
member(X,[_,X|Y]).
member(X,[_,_,X|Y]).
member(X,[_,_,_,X|Y]).
member(X,[_,_,_,_,X|Y]).
member(X,[_,_,_,_,_,X|Y]).
member(X,[_,_,_,_,_,_,X|Y]).
```

if the number of elements the predicate will be called with is known at compile time. Otherwise a recursive rule has to be added to ensure the correctness of the definition,

```
member(X,[_,_,_,_,_,_,_|Y]) :- member(X,Y).
```

We consider this approach a partial solution to the problem because it is not sufficient for all programs in general. It is a useful program pre-processing technique until a new interpreter is built that takes care of general cases as suggested below.

Flattening Choice Predicates at Run Time: We propose that the evaluation of a choice predicate be separated from normal resolution so that choices can be represented in the search tree in a flatten form regardless how the choice predicate is written. A choice predicate is compiled into a special structure distinguishable from the rest of the code and is evaluated at run time separately. We identify such a structure as a *choice graph.* The choice graph is intended to help expand all possible alternatives at run time. Ideally it should also provide mechanism to recognize "bogus" choices, i.e. choices that quickly lead to failure. For this purpose a *guard* can be incorporated to validate an alternative. A predicate will then be defined as

$$Head : -Guard : Body$$

At run time, the *Guard* is evaluated before a branch is actually expanded in the search tree. A choice graph is constructed at compile-time as follows:

- the choice predicate forms a node called the root;
- the right-hand-side of each alternative definition is a child node of the root. There is a directed arc from the root to every child. A child node has two part, the *guard* and the *body.* The body can contain an arc, in position of the recursive call to the choice predicate, leading to the root of the graph, representing recursion. We limit our discussion to direct recursion in this paper.

At run time, a choice predicate is evaluated according to its choice graph. Choices generated by the evaluation become *immediate children to the node at which the choice predicate is calleld.* The tree so generated will be as if the choice predicates in the program were flattened syntactically, achieving the same effect of removing structural imbalances in the search tree while keeping the original program intact. We note that to create a choice branch in the search tree, it is sufficient to evaluate only the guard and predicates positioned to the left of the recursive call (the recursive arc in the choice graph).

3.2 Task Distribution Rules

Effectiveness of the self-organizing scheduling approach lies in whether a balanced load distribution can be obtained. By removing structural imbalance of a program, cut-offs are the only factor that remains causing imbalanced load distribution. Cut-offs exhibit high degree of

Figure 1: Sample Probability Density Functions

Strategy	L_{alloc}	N_{alloc}	C_{rd}
eager-splitting	$log_d\ n$	$< d$	0
lazy-splitting	$log_2\ n$	$(\frac{d}{2})^{log_2\ n}$	$\simeq (\frac{1}{1-2/d}) * (\frac{d}{2})^{log_2\ n}$

Table 1: Comparison of the Two Splitting Strategies. See Text for Further Explanation.

uncertainty, or randomness. Here we investigate task distribution rules that minimizes *average* parallel runtime in theory. In the next section, we study the effectiveness of these rules on benchmark programs.

Until now, we have been using the term task without formally defining it. A task is a sequence of consecutive resolution steps including backtracking performed by a processor. A task is a basic unit of work to be assigned to one or more processor(s). It can be represented by a node or several nodes in the search tree. Task is created dynamically at run time.

We assume that runtime is proportional to the number of nodes traversed. Runtime of a parallel execution is the longest runtime of all processors. In the following discussion, runtime *is* measured by number of nodes traversed, as to simplify the description.

We prove, in the Appendix, the following result:

Theorem: Let N be the number of processors, let m ($\frac{N}{m}$ is an integer) be the number of tasks whose sizes are *statistically identical* and exhibits the following property:

1. the probability density function is non-increasing, or
2. the probability density function is symmetric with respect to a positive central point.

then the average parallel runtime is minimized iff identical number of processors are assigned to each of the tasks.

The conditions in the theorem are satisfied by distribution of shapes illustrated in 1, including, but not limited to, *uniform*, *exponential*, and *normal* distributions.

Statistical identicality of tasks can be guaranteed by enforcing fairness in creating a task, that is, a particular node from a pool of available nodes has equal chance to be included in any task.

Problem remains as to how many tasks are to be created under any particular node. We could create as many tasks as the number of processors being present the node, evenly dividing them among processors, or create only one task, assigning it to all processors. In the former case the search space is divided among processors in the fast possible way. In the latter case the search space is not divided at the current node of the search tree. Redundant computation is incurred, but the ability to adapt to the shape of the search tree can be improved as will be explained later.

Here, we focused on the following task distribution rules, both satisfying the statistical identicality condition:

1. the eager-splitting strategy: at each choice point where m processors are present, assume there is n valid choices. m tasks are created and assigned evenly to m processors. If $n \geq m$, each task contains $\frac{n}{m}$ choices, the left-over choices are randomly included in some of the tasks. If $n < m$, each choice constitute $\frac{m}{n}$ tasks, the rest tasks are formed by randomly picking one choice for each.
2. the lazy-splitting strategy: at each choice point, two tasks are created and assigned to each of the half of the processors. In case of choices being not evenly dividable, left-overs are treated in a way similar to that in the eager-splitting rule.

With the eager-splitting strategy, the search tree is divided among processors in the fastest possible way. The lazy-splitting strategy is the opposite, trading computational overhead for better adaptability.

Assuming that there are $n = 2^k$ processors, and the search tree is balanced and is of degree d (i.e. every node has d branches). Under these conditions, the two task distribution rules are compared in terms of parameters described below:

- allocation level, L_{alloc}: the depth (from the root, level 0) in the search tree where an individual processor commits itself to one or more nodes exclusively.
- number of nodes allocated, N_{alloc}: the number of nodes a processor commits to at the allocation level.
- redundant computation C_{rd}: redundant node expansion compared to the eager-splitting rule, which is considered 0.

Table 1 summarizes results comparing the two splitting strategies. We expect that the eager-splitting strategy minimizes redundant computation, but it is not very adaptive to the shape of the search tree, in the sense that some processors may quickly be out of work due to encountering cut-offs in the tree. This strategy is suitable for a shallow search tree. On the other hand, the lazy-splitting strategy introduces redundant computation but it commits a processor to much more nodes in the search tree compared to the eager-splitting strategy. It is expected to be more adaptive because there are more "alternative" tasks for a processor. For a deep (i.e. the height of the tree is much greater that *log n*) and bushy search tree, the lazy-splitting strategy is expected to perform better since it is more adaptive to the shape of the tree and the redundant computation is relative insignificant in such a case.

Program	Size (res. steps)	Description
9-queens	225926	placing 9 queens such that they cannot attack each other
n-square	77217	testing if all but one of the elements of a square grid can be removed using tic-tac-toe like jumps
patten	50520	testing if certain pattern of a list can be otained
8-queens	47483	placing 8 queens such that they cannot attack each other
tree	22676	traversing a tree generated by pruning branches in a quad-tree randomly with probability set equal to 0.5 The height of the tree is 16
turtles	19678	fitting 9 sqaure pieces into a 3 by 3 board so that certain constraints on matching edges are satistied
zebra	17478	solving the puzzle of who owns the zebra

Table 2: Benchmark Programs

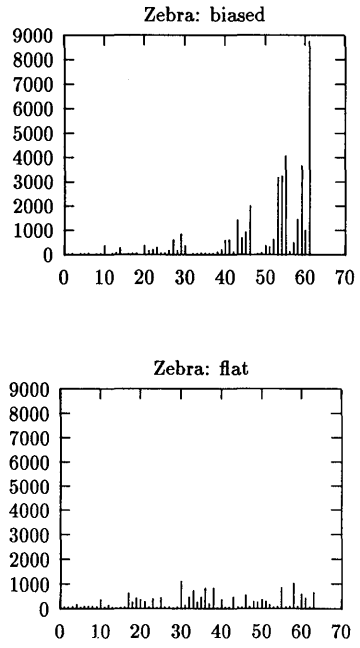


Figure 2: Load Distribution from Running the Zebra Programs on 64 Processors

4 Performance Study

Performance of the self-organize scheduling approach is studied on a set of benchmark programs listed in Table 2. The size of a program is the size of the search constructed during execution of the program. It is the number of resolution steps (logic inferences) during the execution, excluding evaluating Prolog built-in predicates. These programs are pre-processed with the program restructuring method described in the previous section. We note that there is no significant performance changes due to the restructuring in any of the benchmarks running with Sicstus Prolog 0.6.

4.1 Load Distribution

First, we are interested in how effectively the task distribution rules can balance the load, with structural imbalance in a program removed. We defined *balance factor* as,

$$B = \frac{\frac{1}{n}\sum_i^n T_i}{Max(T_i)}$$

where T_i is the total number of nodes in the search tree allocated to processor i, n is the total number of processors. A better balanced load distribution will be reflected in a larger B value. The load balance factor is similar to the efficiency factor e used in other literature [Kumar87], defined as

$$e = \frac{1}{n}\frac{T}{Max(T_i)}$$

where T is the total number of nodes in the tree. $B = e$ if $\sum_i^n T_i = T$, which in many cases is untrue due to that the load on each processor (measured by the number of nodes it possesses) cannot always be $\frac{T}{n}$, because the search tree may not have sufficiently many branches at any particular moment to keep every processor busy. The notion B tries to reflect a realistic load distribution that is possible under a particular load balancing strategy.

The first set of results shows how the balance factor is improved by eliminating the structural imbalance in a program. These results are obtained by extracting the search tree of a program and then exploring the tree with the self-organizing scheduling rule in a simulation with a uniprocessor machine. The eager-splitting rule is used unless specified otherwise.

Figure 2 shows the difference of load (in term of tree nodes) distribution on 64 processors between two versions of a *zebra* program, one with a regular choice predicate and the other with a flattened choice predicate. Load balancing in vastly improved due to program restructuring. It is generally true that flattening the choice predicate results in a better balanced load distribution, though the improvement varies depending on different programs.

We summarize the result by presenting the curves of balance factor for several other benchmarks, shown in Figure 3. The eager-splitting rule is used in this experiment. As can be seen, the balance factor is significantly improved for all but the *n-square* and *tree* programs, which have a deep and bushy search tree that cannot be sufficiently taken care of by the eager-splitting rule. The *n-square* program, and the *tree* program were run

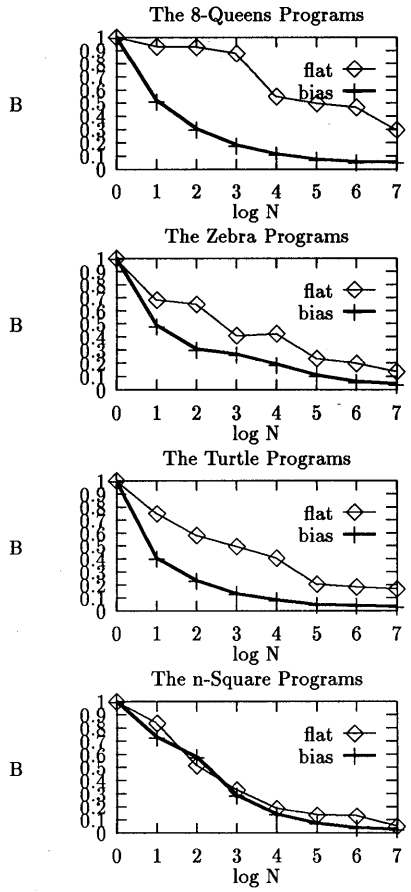


Figure 3: Comparison of Balance Factors (B) between Programs with Flattened Choice Predicates (labeled 'flat' in the figure) and with Normal Style Choice Predicates (labeled 'bias' in the figure)

with the lazy-splitting rule. Results are given Fig. 4. The balance factor is substantially improved (i.e > 100% with 128 processors) since the lazy-splitting is better in coping with irregular shaped tree. However, the overhead of redundant computation makes the lazy-splitting rule unsuitable for a shallow search tree such as that of the *8-queens*, the *zebra*, or the *turtles* program. The height of the search trees for these programs is not sufficiently larger than $log(128)$, the level at which each of the 128 processors commits to its own tasks.

4.2 Speed-up Factors

Speed-up factor is defined as sequential runtime divided by the parallel run time. It is a generally accepted indication of how well a parallel system is able to improve the runtime of a program. Next, we present data showing speed-up factors of the proposed approach on the selected benchmark programs.

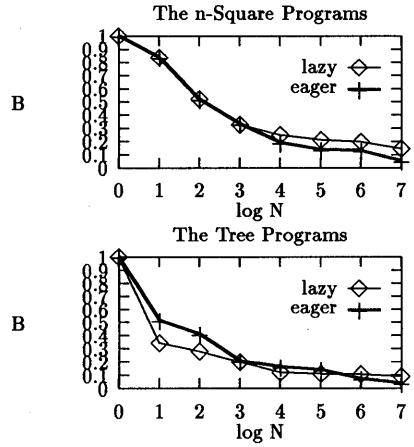


Figure 4: Comparison of the Eager-splitting Rule (labeled 'eager' in the figure) and the Lazy-splitting Rule (labeled 'lazy' in the figure), With Flattened Choice Predicates in Both Programs

Table 3 lists speed-up factors from a simulation study running on a uniprocessors. In this simulation, the run time is measured by the number of resolutions performed in the execution (number of nodes traversed in the proof tree).

Proc. / Prog.	4	8	16	32	64	128
	Eager-splitting					
8-queens	3.9	7.5	9.4	17.0	31.9	**40.1**
9-queens	2.9	4.5	8.6	16.7	22.7	**42.2**
zebra	3.2	4.0	8.3	9.1	15.3	**20.6**
turtles	3.0	5.2	8.6	8.6	15.3	**27.6**
pattern	2.8	5.5	6.2	12.5	21.7	**21.7**
n-square	2.6	2.8	3.2	3.7	3.7	**6.7**
tree	2.4	2.4	3.7	6.5	6.8	**6.8**
	Lazy-splitting					
n-square	2.2	2.8	4.3	7.2	13.0	**17.7**
tree	1.6	2.3	2.7	4.9	9.0	**14.2**

Table 3: Speed-up from Simulation Study. Speed-up is defined as sequential runtime divided by parallel runtime.

Table 4 lists speed-up factors from a parallel emulation study running on a BBN Butterfly TC2000 with 32 processors. The run time is measured by the physical clock. We assume that each resolution step takes constant time. Cost of a real resolution step varies in general. However, here we are merely interested in the total time of a task which consists of a large number of resolution steps. The total time (the sum of the time by all resolution steps) can be considered as the *average cost of each resolution step* times the *number of resolutions*. In other words, the difference of time spent on each resolution step is immaterial. For a given program, the constant can be regarded as the *average cost* of each resolution step.

In order to observe the real overhead of task allocation, which is the time to compute the partition of tasks, the resolution speed must be realistic. In the emulation, resolution engine speed is set equal to that of Aurora Parallel Prolog[2], a well known parallel Prolog implementation, running on one Butterfly processor. Both the eager and the lazy scheduling strategies are implemented in the emulator. The eager-splitting rule was used for the programs *n-queens*, *zebra patten* and *turtles*. The lazy-splitting rule was used for the programs *n-square* and *tree*. From the emulation study, we are able to verified that the sequential simulation, which measures run time by the number of resolution steps performed, accurately reflects the speed-up result by the parallel emulation, which measures run time by the real clock, for up to 32 processor. The overhead of calculating the task distribution, the only overhead not considered in the simulation, is nearly invisible in the emulation, given that the speed-up factors are almost identical to that from the sequential simulation. Notice that there is no communication involved here.

Program	1	4 proc	8 proc	16 proc	32 proc
	Eager-splitting				
8-queens	1	4.0	7.6	9.3	16.5
9-queens	1	3.0	4.6	8.4	16.4
zebra	1	3.2	4.0	8.0	8.9
turtles	1	3.1	5.2	8.2	8.2
patten	1	2.8	5.5	6.1	12.1
	Lazy-splitting				
n-square	1	2.2	2.8	4.2	6.9
tree	1	1.6	2.3	2.7	4.6

Table 4: Speed-up from Emulation Study.

4.3 Performance Comparison with Aurora Parallel Prolog

The same set of benchmarks were run with Aurora Parallel Prolog on the Butterfly machine. Runtime and speed-up factors (the best out of 10 runs) are listed in table 5.

The Peak Speed-up Factors: The speed-up curves for all benchmark programs either have reached the peak (bold face numbers) or at least level off with Aurora Parallel Prolog on 32 processors, as shown in Table 5. Using the self-organizing scheduling approach, simulation results (Table 3) on up to 128 processors showed that:

- the peak speed-up factors for the *8-queens*, *zebra* and *turtles* programs (with fine grain parallelism) exceed, by a margin of at least 200%, experimental results on Aurora;
- the peak speed-up factors for the *9-queens* program is twice as that on Aurora;
- the peak speed-up factors for the *n-square* program (with a very bushy search tree) is about 30% faster that that on Aurora.

[2] Aurora 0.6/Foxtrot, patch #8, with the Manchester Scheduler.

Program	1 proc	16 proc	24 proc	32 proc
8-queens	1,620	141/11.5	**122/13.3**	123/13.2
9-queens	7,500	533/14.1	367/20.4	**350/21.4**
zebra	2,600	490/5.3	**500/5.2**	525/4.9
turtles	4,300	**550/7.8**	580/7.4	569/7.5
pattern	1,084	**130/8.3**	160/6.8	240/4.5
n-square	2,230	190/11.7	**170/13.1**	178/12.6

Table 5: Runtime (ms.) / Speed-up factors with Aurora Parallel Prolog

Speed-up Comparison: Given the number of processors, the speed-ups achieved by self-organizing scheduling appears to be comparable to that of Aurora, but somewhat lower when the number of processors is small (e.g < 16). Note that these results are obtained without communication. The same speed-up result is expected to hold regardless of the speed at which resolution engine is running. Therefore, absolute speed comparison will favor the self-organizing scheduling scheme.

5 Discussion

In the above experiment we studied the behavior of the proposed technique without communication among processors. We demonstrated that the scheme is able to effectively deal with problems which render mostly fine-grained parallel tasks under a traditional scheduler. The loss of processor utilization due to the unevenness in load distribution can be more than covered by the benefit of reduced scheduling overhead. The advantage of the proposed technique is its non-communicating nature, as frees it from possible constraints such as communication bandwidth among processors that could otherwise limit the ability of a scheduler to function effectively. The limitation, however, is its unable to re-use processors that complete tasks they allocated before the termination of the (parallel) execution. We have shown, in the above simulation study, that this would not necessarily compromise performance of programs specially those that generate mostly fine-grained tasks at run time under a traditional scheduler. But the worse case scenario could happen despite the effort to obtain a better balanced load distribution by removing structural imbalance of the search tree and using a statistically even distribution rule. Below, we discuss options to deal with the problem.

One possible solution to the problem is to resort to dynamic task redistribution as existing schedulers do. As we know, the overhead of dynamic task redistribution is relatively small for medium to large-grained tasks, and it provides us with the adaptiveness necessary to deal

with some extraordinary shape search space. On the other hand, the self-organizing scheduling approach introduces low overhead and thus ensures that when it does not help improve performance it is not expect degrade it either. When the two methods are careful integrated, it can be a combination that takes advantage of what the two methods are best at. The issue is when and how dynamic task redistribution should be invoked to achieve the best result. Preliminary research has been conducted in this direction and we will present results in a separate paper. Another option that alleviates the problem is to have idle processors collected by a higher level scheduler (e.g. the operating system) and assigned to other queries. The idea is to use dynamic scheduling only at the level of user queries which usually offer larger granule. In a multi-user environment, this approach can yield a high system throughput given sufficient queries. Global load balancing is involved here. It appears an interesting subject for future investigation.

Static program analysis that provides probability of cut-offs according to given query patterns will be very helpful to guide task distribution. More research is yet to be done before this becomes a feasible alternative to the currently used statistical distribution rule.

Finally, we note that an interesting feature of the self-organizing scheduling approach is that it establishes linkage between processor mapping and the syntax of a program. This feature provides user a mean to influence the mapping of processors to tasks, as would be particularly helpful for applications in which tasks are clearly defined and dynamic task redistribution is known to be not beneficial (there are many such applications). Again, dynamic task redistribution can be used to guard against abuse of this feature.

6 Conclusion and Future Work

A task scheduling technique, self-organizing scheduling, is proposed in this paper. The method directs processors to share the search space, a search tree defined implicitly by the program, according to universal rules followed by every processor in the system. Load balance is achieved by altering the shape of the search tree to remove the so-called structural imbalance (see section 3), and imposing a statistically even task distribution rule to deal with the randomness in cut-offs in the tree. Resolution engines only share the program and the original query. A condition for task distribution that minimizes the average parallel runtime is given and proved. An advantage of the method is that it allows all processors to operate independently on private resources both for resolution and task allocation, while being able to maintain a fairly balanced load distribution among processors. The effectiveness of the self-organizing scheduling scheme is independent of the speed of the resolution engine, and architectural characteristics of the multiprocessor.

We presented data showing the effectiveness of the proposed methods on programs that belong to the generate-and-test category. By removing structural imbalances in a program, it was found that a reasonably balanced load distribution can be obtained by following a statistically even distribution rule. We discussed two distinct task distribution rules, the eager-splitting rule and lazy-splitting rule and examed their effectiveness. We showed that the peak speed-up factors with self-organizing scheduling for a set of benchmark programs exceeds, by a substantial margin, results achieved on the same programs by Aurora Parallel Prolog, a well-known parallel Prolog implementation. Given a fixed number of processors, the speed-up factors by the self-organizing scheduling scheme are competitive. By experimenting with the two near-extreme case task distribution rules we also demonstrated that adaptability can be gained on the cost of redundant computation within this framework.

We believe that the condition for task distribution derived in the paper can be useful for other scheduling schemes. Also, the idea of removing structural imbalances in a program will help with a tree-based scheduler that employs the top-most dispatching strategy [But88, Cald88].

We are currently investigating incorporating traditional task redistribution techniques in order to handle large but highly uneven shaped search trees. Preliminary results indicate that allowing limited communication among processors one can substantially improve the efficiency of the execution. Global load balancing, aimed at maximizing throughput of a system that supports multiple user and multiple queries, is an interesting topic for future research.

References

[Ali90] Ali, K. and Karlsson, R., "The Muse Or-Parallel Prolog Model and its Performance", *Proceeding of the North American Conference of Logic Programming, 1990*, MIT press, 1990.

[Ali91] Ali, K. and Karlsson, R., "Scheduling Or-Parallelism in Muse", *Proceeding of the 8th International Conference on Logic Programming*, MIT Press, 1991.

[But88] Butler, R., Disz, T., Lusk, E., Overbeek, R., and Stevens, R., "Scheduling OR-Parallelism: an Argonne perspective", *Logic Programming, Proceedings of the Fifth International Conference and Symposium on Logic Programming*, MIT press, 1988.

[Cald88] Calderwood, A., Szeredi P., "Scheduling Or-parallelism in Aurora - the Manchester Scheduler", *Proceedings of the Sixth International Conference on*

Logic Programming, pages 419-435, MIT Press, Jun. 1989.

[Clock88] Clocksin, W. F. and Alshawi, H., "A Method for Efficiently Executing Hown Clause Programs Using Multiple Processors", *New Generation Computing*, 5, 1988 p 361-376 OHMSHA, Ltd. and Springer Veriag.

[Giul90] Giuliano, M., Kohli, M., Minker, J., Durand, I., "Prism: A Testbed for Parallel Control", *Parallel Algorithms for Machine Intelligence*, edited by Kanal, L., and Kumar, V., to appear.

[Kale85] Kale, L. V., "Parallel Architectures for Problem Solving", Technical report No. UIUCDCS-R-85-1237, Department of Computer Science, University of Illinois at Urbana-Champaign.

[Kumar87] Kumar V. and Nageshwara Rao V. "Parallel Depth First Search. Part II. Analysis" *International Journal of Parallel Programming*, Vol. 16, No.6, 1987.

[Lloyd84] Lloyd, J. W. "Foundations of Logic Programming", Springer-Verlag, 1984.

[Lusk] Lusk, E., Warren H. D., Haridi, S., *et al.* "The Aurora Or-Parallel Prolog System", Argonne internal technical report.

[Mud91] Mudambi, S., "Performance of Aurora on NUMA Machines", *Proceeding of the 8th International Conference on Logic Programming*, MIT Press, 1991.

[VR90] Van Roy, P. L., "Can Logic Programming Execute as Fast as Imperative Programming", Univ. of California, Berkeley Technical Report UCB/CSD 90/600, Dec., 1990.

Appendix

We prove the following theorem:

Theorem: Let N be the number of processors, let m ($\frac{N}{m}$ is an integer) be the number of tasks whose sizes are *statistically identical* and exhibits the following property:

1. the probability density function is non-increasing, or
2. the probability density function is symmetric with respect to a positive central point.

then the average parallel runtime is minimized iff identical number of processors are assigned to each of the tasks.

Before the proof, we describe some basic terminology and notations to be used.

Capital letters X, Y, Z are used for random variables. The probability density function for X is $f_X(x)$, the cumulative probability distribution function for X is $F_X(x)$, we have $F_X(x) = \int_{-\infty}^{x} f_X(t)dt$ by definition. Or in other words, $f_X(x) = F'_X(x)$. In addition, $f_X(x) \geq 0$ and $0 \leq F_X(x) \leq 1$. $F_X(x)$ is non-decreasing since $f_X(x) \geq 0$.

Runtime of a parallel execution is the longest runtime of all processors. Runtime is measured by the size of a task, in our case, the number of nodes to be traversed in a search tree.

N is the number of processor available. $T_1, T_2, ..., T_m$ are random variables denoting the size of m tasks which are statistically identical, that is, with an identical probability distribution function $f(x)$ and $F(x)$. Let $k_1, k_2, ..., k_m$ be the number of processors assigned to $T_1, ..., T_m$, respectively. $k_1 + k_2 + ... + k_m = N$.

We illustrate the proof with a special case when $m = 2$.

Proof:

Let Z be a random variable denoting the runtime by assigning k_1 to task T_1 and k_2 to task k_2. We assume that T_1 is processed in time $\frac{T_1}{k_1}$ and T_2 is processed in time $\frac{T_2}{k_2}$.

$$Z = max(\frac{T_1}{k_1}, \frac{T_2}{k_2})$$

The cumulative distribution function for Z is $F_z(x)$,

$$\begin{aligned} F_z(x) &= \textit{probability that } Z \leq x \\ &= \textit{probability that } (\frac{T_1}{k_1} \leq x) \textbf{ AND } (\frac{T_2}{k_2} \leq x) \\ &= \textit{probability that } (T_1 \leq k_1 x) \textbf{ AND } (T_2 \leq k_2 x) \\ &= F(k_1 x)F(k_2 x) \end{aligned}$$

Average runtime is the mean of Z,

$$\overline{Z} = \int_{-\infty}^{\infty} (1 - F_z(x))dx$$

We need to show that $\overline{Z}$ is minimized when $k_1 = k_2$, given that $k_1 + k_2 = N$, a constant.

For fixed k_1, k_2, define function $G(x) = \frac{F(k_1 x) + F(k_2 x)}{2}$. We have

$$\int_{-\infty}^{\infty} (1 - F_z(x))dx \geq \int_{-\infty}^{\infty} (1 - G^2(x))dx$$

since $F(k_1 x)F(k_2 x) \leq G^2(x)$, given that $F(x)$ is non-negative. Equality holds when $k_1 = k_2$.

Case I: the probability density function f(x) is non-increasing.

It can be shown that the curve of $F(x)$ is either of an arch shape, or a straight line, as illustrated in figure 5. The curve of $G(x)$ lies below (or on) that of $F(x)$ because the curve of $G(x)$ is composed from center points in lines whose two ends are on curve $F(x)$. $G(x) - F(x) \leq 0$, hence $G^2(x) - F^2(x) = (G(x) - F(x))(G(x) + F(x)) \leq 0$ Therefore,

$$\int_{-\infty}^{\infty} (1 - G^2(x))dx \geq \int_{-\infty}^{\infty} (1 - F^2(x))dx$$

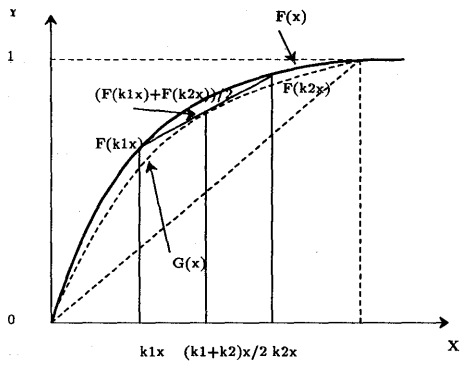


Figure 5: An Arch Shape Distribution

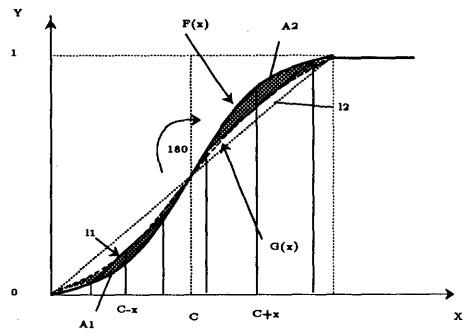


Figure 6: An S Shape Distribution

Equation holds when $k_1 = k_2$.
Thus, we have

$$\int_{-\infty}^{\infty}(1-F_z(x))dx \geq \int_{-\infty}^{\infty}(1-G^2(x))dx \geq \int_{-\infty}^{\infty}(1-F^2(x))dx$$

and equality holds when $k_1 = k_2$. Thus the mean of Z is minimized when $k_1 = k_2$.

Case II: the probability density function f(x) is symmetric with respect to a positive center point, denoted by C.

The curve of $F(x)$ is of the shape an S tilted to the right, as illustrated in figure 6. The curve of $G(x)$ is another S shape curve "contained" in that of $F(x)$. We want to show that

$$\int_{-\infty}^{\infty}(1-G^2(x))dx \geq \int_{-\infty}^{\infty}(1-F^2(x))dx$$

or,

$$\int_{-\infty}^{\infty}(F^2(x)-G^2(x))dx \geq 0$$

This is equivalent to showing

$$\int_{-\infty}^{\infty}(F(x)-G(x))dx \geq 0$$

since

$$\int_{-\infty}^{\infty}(F(x)+G(x))dx > 0$$

Notice that we can no longer have $(F(x) - G(x)) \geq 0$ for all x. However, the integral of (F(x) - G(x)) can still be non-negative if we can proof the shaded areas A_2 is larger or equal to A_1 in figure 6. It suffices to show that for any $(C - x)$ and $(C + x)$ on the X axe, $F(C + x) - G(C + x) \geq G(C - x) - F(C - x)$, and equality holds when $k1 = k2$.

Observe that (C-x,G(C-x)) is the center point of a line, l_1, whose end points are on the curve of $F(x)$. (C+x,G(C+x)) is the center point of another line, l_2, whose end points are on the curve of $F(x)$. Now, rotate the lower part of the S shaped curve of $F(x)$ 180°. The two part of the S matches each other and it can be shown that l_1, after the rotation, completely lies above or on l_2. Thus,

$$F(C+x) - G(C+x) \geq G(C-x) - F(C-x)$$

Equality holds when $k1 = k2$. Proof done for $m = 2$. □

The same idea can be used to prove the general case. A formal proof of the general case will not be presented here, but we note that a property of polygon that is crucial to the proof is that the center of a convex polygon resides *inside* the polygon.

Acknowledgement: I wish to thank Professor Jack Minker for his guidance on this work. Thanks to Dr. Mark Guiliano for his comments on an early draft of this paper. Also, I would like to express my appreciation to Argonne National Laboratory for providing parallel computing facilities.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Asymptotic Load Balance of Distributed Hash Tables

Nobuyuki Ichiyoshi and Kouichi Kimura
Institute for New Generation Computer Technology
1 - 4 - 28 Mita, Minato-ku, Tokyo 108, Japan
{ichiyoshi,kokimura}@icot.or.jp

Abstract

The distributed hash table is a parallelization of the hash table obtained by dividing the table into subtables of equal size and allocating them to the processors. It can handle a number of search/insert operations simultaneously, increasing the throughput by up to p times that of the sequential version, where p is the number of processors. However, in the average case, the peak throughput is not attained due to load imbalance.

It is clear that the table size m must grow at least linearly in p to balance the load. In this paper, we study the rate of growth of m relative to p necessary to maintain the load balance on the average (or to make it approach the perfect load balance). It turns out that linear growth is not enough, but that moderate growth—namely $\omega(p \log^2 p)$—is sufficient. The probabilistic model we used is fairly general and can be applied to other load balancing problems.

We also discuss communication overheads, and find that, in the case of mesh multicomputers, unless the network channel bandwidth grows sufficiently as p grows, the network will eventually become a performance bottleneck for distributed hash tables.

1 Introduction

Parallel computation achieves speedup over sequential computation by sharing the computational load among processors. The load balance between processors is central in determining the parallel runtime (though other factors also affect performance). Unlike uniform computational tasks in which almost perfect load balance is achieved by allocating data uniformly to the processors, non-uniform computational tasks such as search problems pose non-trivial load balancing problems.

In most non-uniform tasks, worst-case computational complexity is far larger than average-case complexity; and the worst case is usually a very rare case. Thus, the study of average case performance is important, and it has been conducted for sorting and searching [Knuth 1973], optimization problems [Coffman and Lueker 1991], and many others [Vitter and Flajolet 1990]. However, there seems to have been little work on average-case performance analysis in regard to parallel algorithms, especially on highly-parallel computers, a notable exception being [Kruskal and Weiss 1985].

In this paper, we study the average-case load balance of distributed hash tables on highly parallel computers. A distributed hash table is a parallelization of a hash table, in which the table is divided into subtables of equal size to be allocated to the processors. It can handle a number of search/insert operations simultaneously, increasing the throughput up to p times that of the sequential version, where p is the number of processors.

However, in average cases, the peak throughput is not attained due to load imbalance. Intuitively, the more buckets allocated to each processor, the better the average load balance becomes. It is clear that under a constant load factor $\alpha = n/m$ (n is the number of elements in the table, m is the table size), m must grow at least linearly in p to balance the load. We shall investigate the necessary/sufficient rate of growth of m relative to p so that the load balance factor—the average processor load divided by the maximum processor load—approaches 1 as $p \to \infty$. It turns out that linear growth is not enough, but that moderate growth—namely, $\omega(p \log^2 p)$—is sufficient. This means that the distributed hash table is a data structure that can exploit the massive computational power of highly parallel computers, with problems of a reasonable size.

We also briefly discuss communication overheads on multicomputers, and find that, in the case of mesh multicomputers, unless the network channel bandwidth grows sufficiently as p grows, the network will eventually become a performance bottleneck for distributed hash tables.

The rest of the paper is organized as follows. Section 2 describes the distributed hash table and defines the problem we shall analyze. The terminology of average-case scalability analysis is introduced in Section 3. The analysis of load balance is presented in Section 4. The full proofs of the propositions appear in [Kimura and Ichiyoshi 1991]. The communication overheads are considered in Section 5. The last section summarizes the paper.

2 Distributed Hash Tables

2.1 Distributed Hash Tables

The distributed hash table is a parallelization of the hash table. A hash table of size $m = pq$ is divided into subtables of equal size q and the subtables are allocated to p processors. The two most simple bucket allocations are:

The block allocation

The k-th bucket ($k \geq 1$) belongs to the ($\lfloor (k-1)/q \rfloor + 1$)-th subtable,[1] and

The modular allocation

The k-th bucket ($k \geq 1$) belongs to the ($((k-1) \bmod p) + 1$)-th processor.

At the beginning of a hash operation (search or insert) for an element x, the hash function is computed for x to generate a number h ($1 \leq h \leq m$), and the element (or the key) is dispatched to the processor which contains the h-th bucket. The rest of the operation is processed at the target processor.

For better performance, it is desirable to maximize the locality. Thus, when the indirect chaining scheme is employed for hash collision, the entire hash chain for a given bucket should be contained in the same processor which contains the bucket. With open addressing, linear probing has the best locality (under the allocation scheme (1)) but its performance degrades quickly as the load factor increases. Other open addressing schemes have better sequential performance characteristics [Knuth 1973], but have less locality. For this reason and also for simplicity of analysis, we choose the indirect chaining scheme. The bucket allocation scheme does not influence the load balance analysis in this case.

The absence of a single entry point that can become a bottleneck makes the distributed hash table a suitable data structure for highly parallel processing. The peak throughput increases linearly with the number of processors. The problem is: When does the "real" performance approach the "peak" performance? When elements are evenly distributed over the processors, linear growth in the number of data elements is sufficient for linear growth in performance. On the other hand, in the worst case, all elements in the hash table might belong to a single subtable so that performance does not increase at all. We are not interested in these two extremes, but in average performance, just as we are more interested in the average complexity of hash operations in sequential hash tables rather than worst-case complexity.

[1] When p does not divide m, taking $q = \lceil m/p \rceil$ works but it may lead to a sub-optimal load balance (e.g., consider the case $m = p + 1$). A better load balance can be realized by a mapping function which is a little more complicated than simple division.

2.2 Problem Definition

There can be a number of uses of hash tables depending on the application. Here we examine the following particular use of the hash table.

Concurrent Data Generation, Search and Insertion

Initially, there is an "old" distributed hash table containing "old elements" and an empty "new" distributed hash table. The old and new tables are of the same size $m = pq$ (p is the number of processors and q is the number of buckets assigned to each processor) and use the same hash function. Also, some "seeds" of new elements are distributed randomly across the processors.

(1) Concurrent Data Generation

Each processor generates "new elements" from the allocated seeds. It is assumed that the time it takes each processor to generate new elements is proportional to the number of generated elements.

(2) Concurrent Data Dispatch

Each processor computes the hash values of the new elements and dispatches the elements to the target processors accordingly.

(3) Concurrent Search

Each processor does a search in the old table for each of the new elements it has received.

(4) Concurrent Insert

Each processor inserts those new elements that are not found in the old table into the new table. No interprocessor communication arises, because the old and new hash tables use the same hash function.

The above usage may seem a little artificial, but the probabilistic model and the analysis for it should be easily applicable to other usages. In the analysis of load balance, the data dispatch step is ignored (equivalently, instantaneous communication is assumed). This is discussed in Section 5.

3 Scalability Analysis

Average Speedup and Efficiency We denote the sequential runtime by $T(1)$ and the parallel runtime using p processors by $T(p)$. The *speedup* is defined by $S(p) = T(1)/T(p)$, and the *efficiency* by $E(p) = S(p)/p$. Efficiency is the ratio between the "real" performance (obtained for a particular problem instance) and the "peak" performance of the parallel computer. In the absence of speculative computation, the efficiency is less than or equal to 1.

Since we intend to engage ourselves in an average-case analysis, we need to define the "average speedup" and the "average efficiency".

Definition 1 We define the *average collective speedup* $\sigma(p)$ by $E(T(1))/E(T(p))$ ($E(X)$ denotes the expectation of X) and the *average collective efficiency* $\eta(p)$ by $\sigma(p)/p$.

The reason why we analyze the above defined average collective speedup, and not the expected speedup in the literal sense—$E(T(1)/T(p))$—is that: (1) it is much simpler to analyze $E(T(1))/E(T(p))$ than analyze $E(T(1)/T(p))$, and (2) in cases where any average speedup figure is meaningful our definition is a better indicator of overall speedup. Suppose we run a number of instances $I_1, I_2, \ldots$ from some problem class, then the *collective speedup* defined by $\sum_i T(1, I_i)/\sum_i T(p, I_i)$ ($T(1, I_i)$ and $T(p, I_i)$ are sequential and parallel runtimes for problem instance I_i) and represent overall speedup. This is more meaningful than any one of arithmetical mean, geometric mean, or harmonic mean that may be calculated from the individual speedups $T(1, T_i)/T(p, T_i)$.

Scalability Analysis and Isoefficiency We would like to study the behavior of $\eta(p)$ as p becomes very large. In general, for a fixed amount of total computation W, $\eta(p)$ decreases as p increases, because there is only finite parallelism in a fixed problem. On the other hand, in many parallel programs, for a fixed p, $\eta(p)$ increases as W grows. Kumar and Rao [1987] introduced the notion of *isoefficiency*: if W needs to grow according to $f(p)$ to maintain an efficiency E, then $f(p)$ is defined to be the *isoefficiency function* for efficiency E. A rapid rate of growth in the isoefficiency function indicates that near-peak performance of a large-scale parallel computer can be attained only when very—sometimes unrealistically—large problems are run. Such a parallel algorithm and/or data structure is not suitable for utilizing a large-scale parallel computer. (We will refer to the isoefficiency by this original definition by *exact isoefficiency.*)

Since it is sometimes impossible to maintain an exact E because of the discrete nature of the problem, the following weaker definitions of isoefficiency may be more suitable or easier to handle.

Asymptotic Isoefficiency f is an *asymptotic isoefficiency* function for E if

$$\lim_{p\to\infty} \eta(p) = E \quad \text{under } W = f(p).$$

Asymptotic Super-Isoefficiency f is an *asymptotic super-isoefficiency* function for E if

$$\liminf_{p\to\infty} \eta(p) \geq E \quad \text{under } W = f(p).$$

f is an *asymptotic super-isoefficiency* function if it is an asymptotic super-isoefficiency function for some $E > 0$, i.e., the efficiency is bounded away from 0 as $p \to \infty$.

An exact isoefficiency function for E is an asymptotic isoefficiency function for E; and an asymptotic isoefficiency function for E is an asymptotic super-isoefficiency function for E.

In the analysis of load balance, we study the balance of essential computation. Essential computation is the total computation performed by processors excluding the parallelization overheads. The amount of essential computation is equal to $pT(p)$ minus the total overhead time spent on things such as message handling and idle time. In the absence of speculative computation, we can identify the amount of essential computation with the sequential runtime.[2] The terminology for load balance analysis is defined like that for speedup/efficiency analysis, except that "essential computation" replaces "runtime": the *total essential computation* corresponds to sequential runtime; *maximum processor load* corresponds to parallel runtime; and *load balance factor*[3] corresponds to efficiency. We use the same terminology for isoefficiency functions. In the following analysis, we study asymptotic isoefficiency for 1 and asymptotic super-isoefficiency. (Since we are not dealing with exact isoefficiency, we drop the adjective "asymptotic" for brevity.)

4 Analysis of Load Balance

4.1 Assumptions

For the sake of probabilistic analysis, we consider a model in which the following values are treated as random variables (RVs): the number of old and new elements belonging to the j-th bucket on the i-th processor ($1 \leq i \leq p$, $1 \leq j \leq q$) denoted by A_{ij} and B_{ij} respectively, and the number of new elements generated at the i-th processor denoted by G_i.

First, we make some assumptions on the distributions of these random variables. The two alternative models of hash tables are the Bernoulli model in which the number of elements n inserted in m buckets is fixed ($\alpha = n/m$) and the probability that an element has a given hash value is uniformly $1/m$, and the Poisson model in which the occupancy of each bucket is an independent Poisson random variable with parameter α [Vitter and Flajolet 1990]. We choose the Poisson model, because it is simpler to analyze directly, and because, with regard to the distributions of maximum

[2] If we ignore various sequential overheads such as cache miss, process switching, and paging.

[3] Not to be confused with the load factor of hash tables.

bucket occupancy in which we are interested, those under the Bernoulli model approach those under the Poisson model as $m \to \infty$ [Kolchin *et al.* 1978].

For a similar reason, we assume that G_i $(1 \le i \le p)$ are independent identically distributed (i.i.d.) random variables having a Poisson distribution with some parameter γ. It follows that the total number of new elements has a Poisson distribution with parameter $p\gamma$, and by the assumption on the hash function, B_{ij}'s are i.i.d. random variables having a Poisson distribution with parameter $\beta = p\gamma/m = \gamma/q$. We assume that load factors α and β of the old and new hash tables are constant (do not change with p, q).

To summarize, A_{ij} and B_{ij} are i.i.d. random variables having a Poisson distribution with parameters α and β, and G_i are i.i.d. random variables with a Poisson distribution with parameter $q\beta$. Note that G_i's and B_k's are not independent because $\sum_i G_i = \sum_{ij} B_{ij}$.

4.2 Essential Computation and Load Balance Factor

Since each data generation is assumed to take the same time, the essential computation of the data generation step is:

$$W_{gen} = \sum_{1 \le i \le p} G_i.$$

(ignoring the constant factor).

As for the search step, some searches are successful (the new element is found in the old table) and others are unsuccessful. For simplicity of analysis, we choose a pessimistic estimate of the essential computation and assume that all searches are unsuccessful. We also assume that an unsuccessful search involves comparison of the new elements against all the old elements in the bucket. Thus, the number of comparisons made by an unsuccessful search in the bucket with A_{ij} elements is $A_{ij}+1$ (the number of elements plus one for the hash table slot containing the pointer to the collision chain). Therefore, the essential computation of the search step is:

$$W_{search} = \sum_{1 \le i \le p} \sum_{1 \le j \le q} (A_{ij}+1)B_{ij}.$$

(again ignoring the constant factor).

We make a similar assumption for the insert step: every insert is done after an unsuccessful search in the new table. Thus, the essential computation of the search step for bucket j on processor i is:

$$\sum_{0 \le l \le B_{ij}-1} (l+1) = B_{ij}(B_{ij}+1)/2,$$

and the total essential computation for the search step is

$$W_{insert} = \sum_{1 \le i \le p} \sum_{1 \le j \le q} B_{ij}(B_{ij}+1)/2.$$

Thus, the total essential computation is:

$$W(1) = \sum_{1 \le i \le p} (W_i' + W_i'' + W_i'''),$$

where

$$W_i' = G_i, \quad W_i'' = \textstyle\sum_{1 \le j \le q}(A_{ij}+1)B_{ij}, \quad \text{and}$$
$$W_i''' = \textstyle\sum_{1 \le j \le q} B_{ij}(B_{ij}+1)/2.$$

The maximum processor load is

$$W(p) = \max_{1 \le i \le p}(W_i' + W_i'' + W_i''')$$

The average load balance factor $\eta(p)$ is $E(W(1))/pE(W(p))$. We would like to know what rate of growth of q is necessary/sufficient so that $\eta(p) \to 1$ as $p \to \infty$.

Since

$$\begin{aligned} & E\left(\sum_{1 \le i \le p} (W_i' + W_i'' + W_i''')\right) \\ &= E\left(\sum_{1 \le i \le p} W_i'\right) + E\left(\sum_{1 \le i \le p} W_i''\right) + E\left(\sum_{1 \le i \le p} W_i'''\right) \\ &= p(E(W_1') + E(W_1'') + E(W_1''')), \end{aligned}$$

and

$$\begin{aligned} & E\left(\max_{1 \le i \le p}(W_i' + W_i'' + W_i''')\right) \\ & \quad \le E\left(\max_{1 \le i \le p} W_i'\right) + E\left(\max_{1 \le i \le p} W_i''\right) + E\left(\max_{1 \le i \le p} W_i'''\right), \end{aligned}$$

we have

$$\eta(p) \ge \frac{E(W_1') + E(W_1'') + E(W_1''')}{E\left(\max_{1 \le i \le p} W_i'\right) + E\left(\max_{1 \le i \le p} W_i''\right) + E\left(\max_{1 \le i \le p} W_i'''\right)}.$$

Thus, if

$$\begin{aligned} E\left(\max_{1 \le i \le p} W_i'\right) &\sim E(W_1'), \\ E\left(\max_{1 \le i \le p} W_i''\right) &\sim E(W_1''), \text{ and} \\ E\left(\max_{1 \le i \le p} W_i'''\right) &\sim E(W_1''') \\ & \qquad (\text{as } p \to \infty), \end{aligned}$$

then $\eta(p) \to 1$. The above are also necessary conditions, because all three summands are significant as $p \to \infty$.

The random variable G_i, having a Poisson distribution with parameter $q\beta$, has the same distribution as the sum of q i.i.d. random variables H_{ij} $(1 \le j \le q)$ with a Poisson distribution with parameter β. Thus, we are led to the study of the average maximum of p sums of q i.i.d. random variables W_{ij} $(1 \le i \le p,\ 1 \le j \le q)$ with a distribution that does not change with p and q. In our distributed hash table example, we are interested in the cases in which each W_{ij} is either a Poisson variable, the product of two Poisson variables, or a polynomial of a Poisson variable.

4.3 Average Maximum of Sum of i.i.d. Random Variables

We give sketches of the proofs or cite the results. The details are presented in [Kimura and Ichiyoshi 1991].

4.3.1 Poisson Variable

The asymptotic distribution of the *maximum bucket occupancy* has been analyzed by Kolchin *et al.* [1978]. The following is the result as cited in [Vitter and Flajolet 1990].

Theorem 1 (Kolchin et al.) *If X_i $(1 \le i \le p)$ are i.i.d. random variables having a Poisson distribution with parameter μ, the expected maximum bucket occupancy is*

$$\overline{M_\mu} = E\left(\max_{1\le i\le p} X_i\right) \sim \begin{cases} \tilde{b} & \text{if } \mu = o(\log p); \\ \mu & \text{if } \mu = \omega(\log p), \end{cases}$$

where $\tilde{b}$ is an integer greater than μ such that

$$\frac{e^{-\mu}\mu^{\tilde{b}+1}}{(\tilde{b}+1)!} \le \frac{1}{p} < \frac{e^{-\mu}\mu^{\tilde{b}}}{\tilde{b}!}.$$

When $\mu = \Theta(1)$, $\tilde{b} \sim \log p / \log\log p$.

The proof is based on the observation that, as p becomes large, $P\{M_\mu > b\}$ as a function of b approaches the step function having value 1 for b smaller than $\tilde{b}$ and 0 for b larger than $\tilde{b}$, and the expectation of M_μ is equal to its summation from $b = 0$ to $b = \infty$.

We extend Kolchin's theorem to the product of Poisson variables and polynomials of a Poisson variable.

4.3.2 Product of Two Poisson Variables

We introduce a partial order on the class $\mathcal{M}$ of non-negative random variables with a finite mean.

Definition 2 For $X, Y \in \mathcal{M}$, we define $X \prec Y$ iff $E(\max\{X, c\}) \le E(\max\{Y, c\})$ for all $c \ge 0$.

There are a number of natural properties concerning this partial order. For example, if $X \prec Y$ and Z is independent of X, Y, then $X + Z \prec Y + Z$, $XZ \prec YZ$, $\max\{X, Z\} \prec \max\{Y, Z\}$, etc. Note $X \prec Y_1$ and $X \prec Y_2$ do not imply $2X \prec Y_1 + Y_2$. The utility of $\prec$ in analyzing the expected maximum is illustrated by the following lemma.

Lemma 1 *Let X_i $(1 \le i \le p)$ and Y_i $(1 \le i \le p)$ be i.i.d. random variables distributed as X and Y. If $X \prec Y$, then*

$$E\left(\max_{1\le i\le p} X_i\right) \le E\left(\max_{1\le i\le p} Y_i\right).$$

SKETCH OF PROOF:

$$\max\{X_1, X_2, \ldots, X_p\} \prec \max\{Y_1, X_2, \ldots, X_p\} \prec \cdots \prec \max\{Y_1, \ldots, Y_p\} \quad \square$$

For the convex sum of i.i.d. variables, we have the following lemma.

Lemma 2 *Let X_i $(1 \le i \le p)$ be i.i.d. random variables distributed as X. For all $a_i \ge 0$ $(1 \le i \le p)$ such that $a_1 + \cdots + a_p = 1$, $a_1X_1 + \cdots + a_pX_p \prec X$.*

SKETCH OF PROOF: Let $a_1, a_2 \ge 0$ and $a_1 + a_2 = 1$. For arbitrary $c \ge 0$, $\max\{a_1X_1 + a_2X_2, c\} + \max\{a_1X_2 + a_2X_1, c\} \le \max\{X_1, c\} + \max\{X_2, c\}$. The expectation of the left hand side is equal to $2E(\max\{a_1X_1 + a_2X_2, c\})$, and that of the right hand side is

$$\begin{aligned} & E(\max\{X_1, c\} + \max\{X_2, c\}) \\ &= E(\max\{X_1, c\}) + E(\max\{X_2, c\}) \\ &= 2E(\max\{X, c\}). \end{aligned}$$

Thus, $a_1X_1 + a_2X_2 \prec X$. The case for $p > 2$ can be reduced to $p - 1$ using the above. $\square$

Finally, the following lemma gives an upper bound on the sum of the product of two sets of i.i.d. random variables.

Lemma 3 *Let X_i $(1 \le i \le rs)$ and Y_i $(1 \le i \le rs)$ be i.i.d. random variables. We have*

$$X_1Y_1 + \cdots + X_{rs}Y_{rs} \prec (X_1 + \cdots + X_r)(Y_1 + \cdots + Y_s).$$

SKETCH OF PROOF: We can prove $X_1Y_1 + \cdots X_tY_t + Z \prec X_1(Y_1 + \cdots + Y_t) + Z$ (Z independent of X_is, Y_is) by conditioning Z and using Lemma 2. By repeatedly "collecting" the $X_{ij}Y_{ij}$'s and replacing them with the bracketed terms, we have the desired result. $\square$

Theorem 2 *Let X_i $(1 \le i \le q)$ and Y_i $(1 \le i \le q)$ be i.i.d. having a Poisson distribution with parameter α and β. If $q = \omega(\log^2 p)$, then*

$$E\left(\max_{1\le i\le p} \sum_{1\le j\le q} X_{ij}Y_{ij}\right) \sim q\alpha\beta = E\left(\sum_{1\le j\le q} X_{1j}Y_{1j}\right)$$

(as $p \to \infty$).

SKETCH OF PROOF: Let $q = r^2$,

$$\begin{aligned} & E\left(\max_{1\le i\le p}(X_{i1}Y_{i1} + \cdots + X_{iq}Y_{iq})\right) \\ &\le E\left(\max_{1\le i\le p}(X_{i1} + \cdots + X_{ir})(Y_{i1} + \cdots + Y_{ir})\right) \\ &\le E\left(\max_{1\le i\le p}(X_{i1} + \cdots)\right) E\left(\max_{1\le i\le p}(Y_{i1} + \cdots)\right) \end{aligned}$$

by the Lemma 2 and 3. The sum of r i.i.d. Poisson variables with parameter α is distributed as a Poisson variable with parameter $r\alpha$. Thus, if $r = \omega(\log p)$, then

$$E\left(\max_{1\le i\le p}(X_{i1} + \cdots + X_{ir})\right) \sim r\alpha = E(X_{11} + \cdots + X_{1r})$$

by Kolchin's theorem. This is similar for the sum of Y_{ij}. This is what we needed. $\square$

4.3.3 Polynomial of Poisson Variable

The treatment of upper bounds on the expected maximum of the sums of a polynomial of i.i.d. random variables is more involved. We only list the result.

Theorem 3 *Let X_i $(1 \le i \le q)$ be i.i.d. having a Poisson distribution with parameter α, and $c(X)$ be a polynomial of degree $d > 0$ with non-negative coefficients. If $q = \omega(\log^d p)$, then*

$$E\left(\max_{1\le i\le p}\sum_{1\le j\le q} c(X_{ij})\right) \sim qc^*(\alpha) = E\left(\sum_{1\le j\le q} c(X_{ij})\right)$$

(as $p \to \infty$), where $c(X) = a_d^* X^{(d)} + \cdots + a_1^* X^{(1)} + a_0^*$, and $c^*(X) = a_d^* X^d + \cdots + a_1^* X^1 + a_0^*$ ($X^{(k)} = X(X-1)\cdots(X-k+1)$ is the falling power of X).

As for corresponding lower bounds on the necessary growth rate of q, we only know at present that if $q = o((\log p/\log\log p)^2)$, the ratio between the expected maximum and the mean tends to ∞ as $p \to \infty$.

4.4 The Isoefficiency for Load Balance

Now, let us suppose $q = \omega(\log^2 p)$. Then,

$$E\left(\max_{1\le i\le p} G_i\right) \sim E(G_1) \quad (p \to \infty)$$

is immediate from Kolchin's theorem. Also,

$$\begin{aligned} & E\left(\max_{1\le i\le p}\sum_{1\le j\le q}(A_{ij}+1)B_{ij}\right) \\ \le\ & E\left(\max_{1\le i\le p}\sum_{1\le j\le q} A_{ij}B_{ij}\right) + E\left(\max_{1\le i\le p}\sum_{1\le j\le q} B_{ij}\right) \\ \sim\ & E\left(\sum_{1\le j\le q} A_{1j}B_{1j}\right) + E\left(\sum_{1\le j\le q} B_{1j}\right) \end{aligned}$$

by Kolchin's theorem and the proposition for the product of two Poisson variables. Finally, since $X(X+1)/2$ is a polynomial of degree 2,

$$E\left(\max_{1\le i\le p}\sum_{1\le j\le q}\frac{B_{ij}(B_{ij}+1)}{2}\right) \sim E\left(\sum_{1\le j\le q}\frac{B_{1j}(B_{1j}+1)}{2}\right).$$

if $q = \omega(\log^2 p)$.

We have shown that if $q = \omega(\log^2 p)$, the average collective load balance factor $\eta(p) \to 1$ as $p \to \infty$. Therefore, $W = \Theta(pq) = \omega(p\log^2 p)$ is a sufficient condition for isoefficiency for 1.

4.5 Simulation

A simple simulation program was run to test the applicability of the asymptotic analysis for p up to 4096. Fig. 1 shows the results for $\alpha = \beta = 4$, $p = 4$, 16, 64, 256, 1024, and 4096 and $q = 1$, $\lg p$, $\lg^2 p$, and $\lg^3 p$ ($\lg$ denotes the logarithm with base 2). The experimental load balance factors (on the vertical axis) are plotted against the number of processors (on the horizontal axis). The experimental load balance factor $\overline{\eta_{p,q}}$ for p, q are calculated by

$$\overline{\eta_{p,q}} = \frac{E\left(\sum_{1\le j\le q} W_{1j}\right)}{\overline{\max_{1\le i\le p}\sum_{1\le j\le q} W_{ij}}},$$

where W_{ij} is one of $X_{ij}, X_{ij}Y_{ij}$ and $X_{ij}^{(2)}$ ((a), (b) and (c), respectively in the figure), and the average $\overline{\max_{1\le i\le p}\sum_{1\le j\le q} W_{ij}}$ is calculated from the result of 50 simulation runs.

X_{ij} and Y_{ij} are generated according to the Bernoulli model (*i.e.*, a table $X[1..pq]$ is prepared, and $n = pq\alpha$ random numbers x's with $x \ge 0$ were generated, each x going to the $((x \bmod p)+1)$-th table entry, etc.). The coefficient of variation (the ratio of standard deviation to average) of $\max_{1\le i\le p}\sum_{1\le j\le q} W_{ij}$ is larger for $X^{(2)}$ and XY than for X, and it decreases as p becomes larger or q becomes larger. Table 1 gives the coefficients of variation for $p = 64$ and 4096.

By and large, the results seem to confirm the asymptotic analysis. For the product and the second falling power, $\Theta(\log^2 p)$ appears to be a sufficient rate of growth of q for η to converge to 1. Even logarithmic growth $(q = \lg p)$ does not lead to very poor load balance factors at least up to $p = 4096$ (approx. 0.5 for XY and approx. 0.4 for $X^{(2)}$).

5 Communication Overheads

We briefly discuss the communication overheads when distributed hash tables are implemented on multicomputers. A multicomputer (also referred to as a distributed-memory computer and a message-passing parallel computer) consists of p identical processors connected by some interconnection network. On such computers, the time it takes to transfer a message of length L (in words) from a processor to another which is D hops away in the absence of network contention[4] is $t_s + t_h D + t_w L$, where t_s is the constant start-up time, t_h is the per-hop time, and t_w is the per-word communication time. We choose the mesh architecture for consideration (two-dimensional square meshes in particular) since many of the recent "second generation" multicomputers have such topologies. Examples include J-Machine, Intel Paragon, and parallel inference machines Multi-PSI and PIM/m.

We note that the average traveling distance of a random message (a message from a randomly chosen processor i to another randomly chosen processor i', allowing $i = i'$) is $\frac{2}{3}(\sqrt{p} - \frac{1}{\sqrt{p}}) \sim \frac{2}{3}\sqrt{p}$ on the meshes. It is roughly

[4] Communication latency in the absence of network contention is called *zero-load latency*.

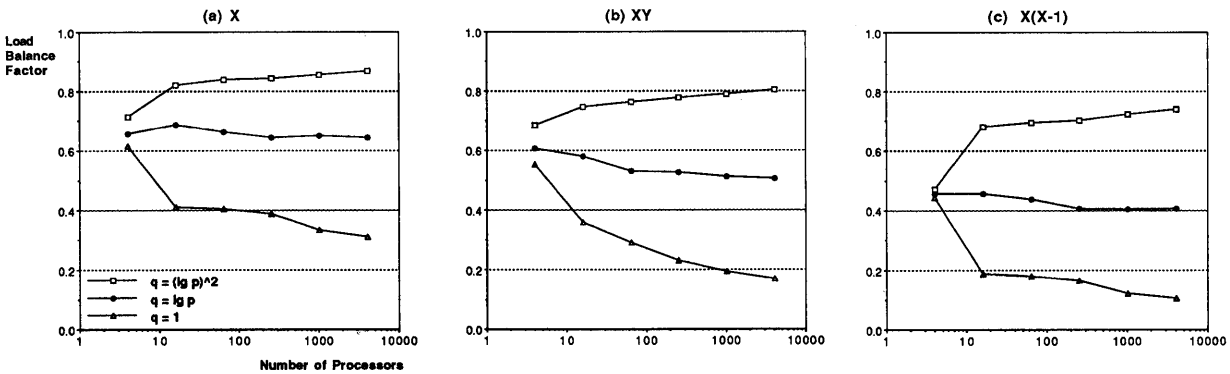


Figure 1: Experimental Load Balance Factors ($\alpha = \beta = 4$)

Table 1: Coefficients of Variation of Maximum Load ($\alpha = \beta = 4$)

	$p = 64$			$p = 4096$		
	$q = 1$	$q = 6$	$q = 36$	$q = 1$	$q = 12$	$q = 144$
X	11.0%	6.3%	2.6%	7.1%	3.5%	1.0%
XY	17.8%	12.2%	5.0%	12.3%	5.3%	2.1%
$X^{(2)}$	24.8%	13.1%	6.0%	15.4%	6.8%	2.6%

1/3 of the diameter of the network, which is $2(\sqrt{p} - 1)$. We can easily see that $W = \Omega(p^{3/2})$ is a necessary and sufficient condition for super-isoefficiency due to zero-load latency, which is a situation worse than that due to load imbalance.

In real networks, the impact of message collisions must be taken into account. Instead of estimating the time required for data dispatch using a precise model of contention, we compare the amount of traffic generated by random communication and the capacity of the network. The *traffic* of a message is defined by the product of its traveling distance and its length. It indicates how much network resource (measured by channel × network cycle time) the message consumes. The *capacity* of a network is defined by the sum of the bandwidth of all network channels (channels that connect routers). It indicates the peak throughput of the network. The basic fact is that the time required for completely delivering a set of messages is at least M/C, where M is the total message traffic and C is the capacity of the network.

The average traffic generated by $\sum_{1 \le i \le p} G_i$ random messages is $\sim \frac{2}{3}p^{3/2}q\beta L$ (L is the constant message length). The network capacity is $2\sqrt{p}(\sqrt{p} - 1)/t_w \sim 2p/t_w$. Thus, the average data dispatch time is at least $\sim \frac{1}{3}\sqrt{p}q\beta L t_w = \Theta(\sqrt{p}q) = \omega(T(1)/p)$. This means that meshes with constant channel bandwidth cannot sustain the traffic generated by random communication, forcing the efficiency to approach zero as $p \to \infty$. The network channel bandwidth must grow at least in proportion to $\sqrt{p}$, to maintain the communication latency under heavy random communication within a constant factor of the zero-load latency.

A similar analysis for the hypercube architecture shows that $W = \Omega(p \log p)$ is a necessary and sufficient condition for super-isoefficiency due to zero-load latency, and is less than that due to load imbalance, and that the network capacity has the same growth rate as that of the random traffic.

The degradation of performance due to network contention in the mesh architecture has been pointed out by several authors. Gupta and Kumar [1990] have done scalability analysis for a parallel FFT algorithm, and Singh *et al.* [1990] for parallel quicksort algorithms. In both of these types of algorithms, the communication patterns are nonlocal as in our distributed hash table example, and the growth in the problem size makes local computation per message increase very slowly. This means that isoefficiency function must grow very rapidly (nearly exponential). In our case, since local computation per message does not increase with problem size, it is impossible to maintain efficiency as p gets larger.

Our analysis does *not* suggest that hypercubes are superior to mesh networks for building very large-scale multicomputers. On the contrary, Dally [1990] showed that if we fix the wire bisection of the network, low dimensional cubes (k-ary n-cubes with n small) provide larger throughput than high dimensional cubes (k-ary n-cubes with n large). We believe that future very large-scale

multicomputers should provide network bandwidths that can meet the traffic generated by nonlocal communication, if they are to support a wide variety of parallel algorithms, not restricted to ones with high communication locality. Dally [1991] proposes a design of such network architectures.

6 Conclusions

An asymptotic analysis of the load balance of distributed hash tables was conducted, and it was found that, with a constant load factor, $m = \omega(p \log^2 p)$ is a sufficient rate of growth of table size m to balance the load as the number of processors p grows. Communication overheads on multicomputers was also briefly discussed. In the case of mesh multicomputers, unless the network channel bandwidth grows sufficiently as p grows, the network will eventually become a performance bottleneck.

Because of the rather high overheads in encoding and decoding message packets on the part of the processing node, small- to medium-scale multicomputers may not generate enough message traffic to make contention—or, even communication latency—a performance bottleneck [Nakajima and Ichiyoshi 1990, Chittor and Enbody 1990]. But, the bottleneck is bound to show itself in very large-scale multicomputers.

The probabilistic analysis in this paper is fairly general and can be applied to similar load balance problems, such as parallel A^* search with distributed OPEN lists [Kumar *et al.* 1988, Huang and Davis 1988, Manzini 1990]. Kruskal and Weiss [1985] studied parallel runtimes when independent subtasks are allocated on processors, with an (rather restrictive) assumption that the distribution of subtask running times is one with increasing failure rate (IFR). Their analysis was also asymptotic as the number of subtasks and processors becomes large. This paper differs from their study mainly in that (1) the IFR assumption does not hold for the distribution of hash operation costs, and (2) asymptotic (super-)isoefficiency is investigated.

References

[Chittor and Enbody 1990] S. Chittor and R. Enbody. Performance evaluation of mesh-connected wormhole-routed networks for interprocessor communication in multicomputers. In *Proceedings of Supercomputing Conference*, 1990, pp. 647–656.

[Coffman and Lueker 1991] E. G. Coffman, Jr. and G. S. Lueker. *Probabilistic Analysis of Packing and Partitioning Algorithm.* John Wiley & Sons, 1991.

[Dally 1990] W. Dally. Network and processor architecture for message-driven computers. In R. Suaya and G. Birtwistle, editors, *VLSI and Parallel Computation*, chapter 3. Morgan Kauffman Publishers, 1990.

[Dally 1991] W. J. Dally. Express cubes. *IEEE Transactions on Computers*, Vol. 40, No. 9 (1991), pp. 1016–1023.

[Gupta and Kumar 1990] A. Gupta and V. Kumar. On the scalability of FFT on parallel computers. In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*, 1990.

[Huang and Davis 1988] S. Huang and L. S. Davis. Parallel iterative A* search: An admissible distributed heuristic search algorithm. In *Proceedings of IJCAI-89*, 1989, pp. 23–29.

[Kimura and Ichiyoshi 1991] K. Kimura and N. Ichiyoshi. Scalability analysis of static load balancing under unpredictable subproblem sizes. ICOT Technical Report, ICOT, 1991. To appear.

[Knuth 1973] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, 1973.

[Kolchin *et al.* 1978] V. F. Kolchin, B. A. Sevast'yanov, and V. P. Chistyakov. *Random Allocations.* V. H. Winston & Sons, 1978.

[Kruskal and Weiss 1985] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 10 (1985), pp. 1001–1016.

[Kumar *et al.* 1988] V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state space graphs: A summary of results. In *Proceedings of AAAI-88*, 1988, pp. 122–127.

[Kumar and Rao 1987] V. Kumar and V. N. Rao. Parallel depth-first search, part II. *International Journal of Parallel Programming*, Vol. 16, No. 6 (1987), pp. 501–519.

[Manzini 1990] G. Manzini. Probabilistic performance analysis of heuristic search using parallel hash tables. 1990 (Written while the author was at MIT).

[Nakajima and Ichiyoshi 1990] K. Nakajima and N. Ichiyoshi. Evaluation of inter-processor communication in the KL1 implementation on the Multi-PSI. In *Proceedings of ICPP'90*, 1990, pp. 613–614.

[Singh *et al.* 1990] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Scalability of parallel sorting on mesh multicomputers. Technical Report TR-90-45, Institute of Technology, University of Minnesota, 1990.

[Vitter and Flajolet 1990] J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, chapter 9. The MIT Press/Elsevier, 1990.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Constructing and Collapsing a Reflective Tower in Reflective Guarded Horn Clauses

Jiro Tanaka† and Fumio Matono‡

†FUJITSU LABORATORIES, IIAS,
1-17-25 Shinkamata, Ota-ku, Tokyo 144, JAPAN
Email: jiro@iias.flab.fujitsu.co.jp

‡FUJITSU SOCIAL SCIENCE LABORATORY,
Parale-Mitsui-Build., 8 Higashida-cho, Kawasaki 210, JAPAN

Abstract

The meta-level representation of *Guarded Horn Clauses* (GHC) is considered and a GHC meta-computation system is constructed by enhancing the simple GHC meta-program. Then the *Reflective Guarded Horn Clauses* (RGHC) system is described, where a *reflective tower* can be constructed and collapsed in a dynamic manner, using *reflective predicates*. The implementation of RGHC is shown. Finally a simple execution example is also shown. This paper assumes a basic knowledge of parallel logic languages.

1. Introduction

If we look for an ideal programming language, it must be simple and, at the same time, powerful language. Looking back the history of programming language, we note that the development of the programming language is generated by the repeated trials which look for such languages within a limitation of the available hardware.

Recently, it seems that the mechanism, called *meta* or *reflection*, is attracting wide spread attention in programming language community. Though the concept of *computational reflection* goes back to [Weyhrauch 80, Smith 84], this concept is becoming popular especially in the object-oriented language community [Maes 88].

In this paper, we assume the parallel logic programming language *GHC* [Ueda 85, Tanaka 86] as our underlying language. The reasons for picking this language are in its structural simplicity, semantical clearness and applicability to the system programming.

We have already proposed *reflection* mechanism and shown several application examples [Tanaka 88, Tanaka 90, Tanaka 91]. However, *reflection* has been introduced in an *ad hoc* manner. It lacks the generality seen in 3-Lisp [Smith 84]. We would like to propose *Reflective Guarded Horn Clauses* (RGHC), which has the expressive power comparable to 3-Lisp, in this paper.

The organization of this paper is as follows. In Section 2, we try to describe the meta-computation system of GHC. After considering meta-presentation of the object-level system, we describe GHC meta-computation system by enhancing a simple 4-line GHC meta-program. The language features of RGHC and several reflective programming examples are described in Section 3. RGHC implementation is described in Section 4. An actual program execution example is shown in Section 4. Related works and conclusion are described in Section 6.

2. Meta-computation system in GHC

A meta-system can be defined as a computational system whose problem domain is another computational system. The latter computational system is called the *object-system*. The program of *meta-system* is called *meta-program*.

2.1. A simple GHC meta-program

In Prolog world, a simple 4-line program is well-known as *Prolog in Prolog* or *vanilla* interpreter [Bowen 83]. The GHC version of this program can be described as follows:

```
exec(true):-true|true.
exec((P,Q)):-true|exec(P),exec(Q).
exec(P):-user_defined(P)|
    reduce(P,Body),exec(Body).
exec(P):-system(P)|sys_exe(P).
```

Using this meta-program, we can execute a goal as an argument of "exec." This program tries to execute a given goal in an interpretive manner. We can see two levels, i.e., the *meta-level*, where the top level execution is performed, and the *object-level*, where goal execution is simulated inside the meta-program.

The meaning of this meta-interpreter is as follows: If the given goal is "true," the execution of the goal succeeds. If it is a sequence, it is decomposed and executed separately. In the case of a user-defined goal, the predicate "reduce" finds the clause which satisfies the guard and the goal is decomposed to the body goals of that clause. If it is a system-defined goal, it is executed directly.

Though this 4-line program is very simple, it certainly works as *GHC in GHC*. However, this *GHC in GHC* is insufficient as a real meta-program because of the following reasons.

- There is no distinction between the variables at the meta-level and those at the object-level. The object-level variables cannot be manipulated at the meta-level.
- The predicate "reduce(P,Q)" finds *potentially unifiable clauses* for the given argument "P." The object-level program must also be defined as a program. Therefore, the object-level program cannot be manipulated without using *assert* or *retract*.
- This program only simulate the top level execution of the program. The more detailed executing information such as *current continuation*, *environment* or *execution result* is not *explicit* in this interpreter.

2.2. Meta-level representation of the object-level system

We would like to have a real meta-computation system which does not have the disadvantages described in the previous section. As a first step, we consider the meta-level representation of the object-system.

2.2.1. Constants, function symbols and predicate symbols

We assume that constants, function symbols and predicate symbols are expressed by the same symbols. The other possibility is using *quote* to distinguish the level. In this approach, '3 (quote three) corresponds to the 3 at the object-level. 3-Lisp and Gödel [Lloyd 88b] adopt this approach.

However, we do not adopt this approach. Though *quote* is usually used to separate the *data* from the *program*, there exists a clear separation between predicates and functors in logic programming languages. Though the implementation of *quote* is not difficult, our claim is that there is little merit in using *quote* in logic programming languages.

2.2.2. Variables and variable bindings

As explained previously, we cannot manipulate object-level variables well if it is expressed as variables. To manipulate object-level variables, we need information about the representation of variables, i.e., we need to know where and how the given variable is realized.

Therefore, we use a *special ground term* to express an object-level variable. This *special ground term* has a one-to-one correspondence to the object-level term and is distinguished from the ordinary *ground term*.

An object-level variables are expressed as "@number" at the meta-level. A unique number is assigned for each variable. Though we are afraid that this representation of variables is not abstract enough, compared to the approach using *quote*, we have chosen it for implementation simplicity. Similarly, we also assume that the object-level variable is expressed as "@!number" at the meta-meta-level; "@!!number" at the meta-meta-meta-level, and so on.

The variable bindings at the object-level can *conceptually* be represented as a list of address-value pairs at the meta-level. The followings are the examples of such pairs.

(@1, undf)	the value of @1 is undefined
(@2, a)	the value of @2 is the constant "a"
(@3, @2)	the value of @3 is the reference pointer to @2
(@4,f(@1,@2))	the value of @4 is the structure whose function symbol is "f," the first argument is the reference pointer to @1, and the second argument is the reference pointer to @2

We can regard these pair as expressing the memory cells of the object-level. Similar to the ordinary Prolog implementation, reference pointers are generated when two variables are unified. Therefore, we need to *derefer* the pointers when the value of a variable is needed.

2.2.3. Terms and object-level programs

Keeping the consistency with the notations explained before, we denote object-level terms by corresponding meta-level *special ground terms*, where every variable is replaced by its meta-level notation.

For example, the object-level term "p(a,[H|T], f(T,b))" is expressed as "p(a,[@1|@2],f(@2,b))" at the meta-level.

It is also expressed as "p(a,[@!1|@!2],f(@!2,b))" at the meta-meta-level.

On the other hand, the program of object-level are expressed as a *ground term* at the meta-level, where all variables are replaced by "var(number)" notation. Note that "var(number)" is just a *ground term* and not the *special ground term*.

For example, the following "append/3" program

```
append([A|B],C,D):-true|
```

```
    D=[A|E], append(B,C,E).
append([],A,B):-true|A=B.
```

is expressed as

```
((append,3),
[(append([var(1)|var(2)],var(3),var(4))
      :-true|var(4)=[var(1)|var(5)],
      append(var(2),var(3),var(5))),
 (append([],var(1),var(2))
      :-true|var(1)=var(2))])
```

at the meta-level. Note that, this representation also works at the meta-meta-level, since "var(number)" is just a *ground term.*

2.3. An enhanced meta-program

The simple GHC meta-program in Section 2.1 can be enhanced to fit to the requirements of the real meta-program using the meta-level representation in Section 2.2. The enhancement can be done by making *explicit* what is *implicit* in the simple GHC meta-program.

- There was no distinction between the variable at the meta-level and the one at the object-level. We express object-level variables as *special ground terms* at the meta-level.
- We manipulate object-level program as a *ground term* at meta-level. "exec" keeps it program as its argument.
- "exec" also keeps the *goal queue* and the *environment* in its arguments for expressing *continuation* and *variable bindings.*

The top level description of GHC meta-system can be written as follows:

```
m_ghc(Goal,Db,Out) :- true|
    transfer(Goal,GRep,Env),
    exec([GRep],Env,Db,NEnv,Res),
    make-result(Res,GRep,NEnv,Out).
```

For given object-level goal "Goal" and given object-level program "Db," "m_ghc" outputs the computation result to "Out." "transfer" changes given goal "Goal" to object-level representation "GRep." In "GRep," every variable in "Goal" has been replaced to "@number" form. The third argument contains the environment of this goal representation.

For example, if we input "exam([H|T],T)" to "Goal," "transfer(exam([H|T],T),GRep,Env)" will be executed. The computation result becomes

```
GRep = exam([@1|@2],@2)
Env = (2,[(@1,undf),(@2,undf)]).
```

Note that *environment* is expressed as a pair of a number and a list. The first element of the pair shows how many variables are allocated in the environment. In this case, two numbers have already been allocated. The second element of the pair shows the variable bindings.

The enhanced "exec" executes this *goal representation* and the computation result "Out" will be generated by "make_result" predicate.

The enhanced "exec" has five arguments. These five arguments, in turn, denote the *goal queue*, the *environment*, the *program*, the new *environment* and the *execution result.* The enhanced "exec" can be programmed as follows:

```
exec([],Env,Db,NEnv,R)
    :- true|
        (NEnv,R)=(Env,success).
exec([true|Rest],Env,Db,NEnv,R)
    :- true|
        exec(Rest,Env,Db,NEnv,R).
exec([false|Rest],Env,Db,NEnv,R)
    :- true|
        (NEnv,R)=(Env,failure).
exec([GRep|Rest],Env,Db,NEnv,R)
    :- user_defined(GRep,Db)|
        reduce(GRep,Rest,Env,Db,
            NGRep,Env1),
        exec(NGRep,Env1,Db,NEnv,R).
exec([GRep|Rest],Env,Db,NEnv,R)
    :- system(GRep)|
        sys_exe(GRep,Rest,Env,NGRep,Env1),
        exec(NGRep,Env1,Db,NEnv,R).
```

Though we omit the detailed explanation, the meaning of this program is self-explanatory. We easily notice that this is the extension of the simple GHC meta-program in Section 2.1. Note that the use of *list* for expressing *goal queue* imposes us inefficiency and some sequentiality. The *difference list* is used in the actual implementation. Also note that the use of *shared-variable* and *short-circuit* techniques [Hirsch 86, Safra 86] might be effective in the parallel implementation.

3. Reflective Guarded Horn Clauses

Reflection is the capability to feel or modify the current state of the system dynamically. The form of *reflection* we are interested in is the *computational reflection* proposed by [Smith 84]. A reflective system can be defined as a computational system which takes itself as its problem domain.

In 3-Lisp, an infinite tower of meta is conceptually assumed. A program is not executed directly. Instead, it is assumed to be executed on the bottom of the infinite tower of meta. A meta-system executes the object-system in an interpretive manner. Similarly, the meta-meta-system executes the meta-system in an interpretive manner. Conceptually, the infinite tower of meta all

moves in a synchronous manner. A reflective function can be defined as a mechanism which shift the control one level up.

If a computational system has such reflective capability, it becomes possible to catch the current state while executing the program and takes the appropriate action according to the obtained information.

3.1. Two approaches in implementing reflection

How should such reflective capability be implemented? Apparently, implementing an infinite tower of meta directly is not possible.

There exist two approaches to realizing such reflective system. One is utilizing a pre-exist layer of meta-systems. We can modify the meta-program and add the means of communication between the levels, namely, we prepare a set of built-in predicates which can catch or replace the current state of the object system. If we adopt this approach, it becomes possible to catch or modify the *internal state* of the executing program by using those built-in predicates.

In [Tanaka 88, Tanaka 90, Tanaka 91], we proposed several reflective built-in predicates, such as "get_q," "put_q," "get_env" and "put_env." "get_q" gets the current goal queue of "exec." "put_q" resets the current goal queue to the given argument. Similarly, "get_env" and "put_env" operate on the variable binding environment. Though this approach has merit in that the implementation is relatively straightforward, we should note that this approach is not an accurate implementation of *reflection*. It is because the *internal state* is always changing, even while processing the obtained information at the object-level.

The other way is to create meta-system dynamically when needed. If a *reflective* predicate is called from the object-system, the meta-system is dynamically created and the control transfers to the meta-level in order to perform the necessary computation. The *reflective* function may also be called while executing the meta-system. In this case, the system creates the meta-meta-system and the control transfers to that system. Similarly, it is possible to consider the meta-meta-meta-system, the meta-meta-meta-meta-system, and so on. When the meta-level computation terminates, the control automatically returns to the one-level-lower computation system.

We adopted the second approach in implementing *Reflective Guarded Horn Clauses* (RGHC). RGHC is the *reflective extension* of GHC and can be defined as a superset of GHC. Language features of RGHC are shown in the following sections.

3.2. Reflective predicate

Reflective predicates are user-defined predicates which invoke *reflection* when called. *Reflective predicates* can

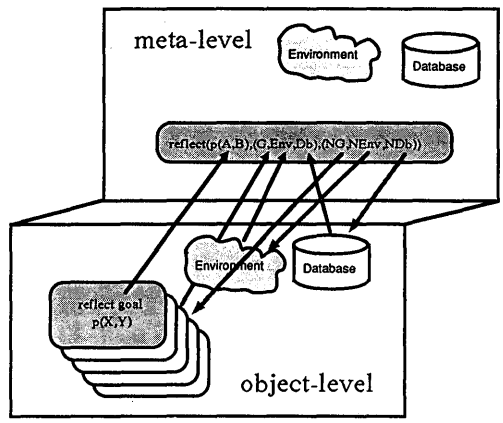


Figure 1: Execution of the reflective predicate

be defined quite easily. It can used wherever we want, in the user program or in the initial query. Similar to 3-Lisp, it is possible to *access* or *modify* the internal state of the computation system by this predicates.

For example, reflective predicate for goal "p(A,B)" can be defined as follows:

```
reflect(p(X,Y),(G,Env,Db),(NG,NEnv,NDb))
    :- guard | body.
```

Note that "p(A,B)" is changed to "p(X,Y)" and two extra arguments, i.e., "(G,Env,Db)" and "(NG,NEnv,NDb)" are added. When the goal "p(A,B)" is called at the object-level, we automatically shift one level up and this goal is executed at the meta-level. (See Figure 1.) At this level, "p(A,B)" is transformed to "p(X,Y)," where "X" and "Y" are the meta-level representation of the arguments.

The computation state of the object-level is also represented as "(G,Env,Db)," where "G" represents the *remaining goals* which should be executed at the object-level, "Env" represents the *environments* and "Db" represents the object-level program. Note that they are the representations of the state and we can freely access and manipulate these arguments. "(NG,NEnv,NDb)" denotes the new computation state of the object-level to which the system should return when the meta-level execution finishes. We assume that (NG,NEnv,NDb) are instantiated while executing meta-level goals. When the execution of this reflective goal is finished, we automatically shift one level down and "(NG,NEnv,NDb)" becomes to the new object-level state.

For example, a reflective predicate "var(X,R)," which checks whether the given argument "X" is unbound or not, can be defined as follows:

```
reflect(var(X,R),(G,Env,Db),(NG,NEnv,NDb))
```

```
    :- unbound(X,Env) |
        add_env((R,unbound),Env,NEnv),
        (NG,NDb)=(G,Db).

reflect(var(X,R),(G,Env,Db),(NG,NEnv,NDb))
    :- bound(X,Env) |
        add_env((R,bound),Env,NEnv),
        (NG,NDb)=(G,Db).
```

Since an object-level variable is handled as a *special ground term* and its value is contained in the *environment*, we examine the representation of *environment* to check whether the variable is bound or not and the result is added to the environment list as a value of "R."

The "current_load(N)" predicate, which obtains the number of goals in the *goal queue* of the object-system, can be defined as follows:

```
reflect(current_load(N),(G,Env,Db),
    (NG,NEnv,NDb)):- true |
        length(G,X),
        add_env((N,X),Env,NEnv),
        (NG,NDb)=(G,Db).
```

We shift up to the meta-level and computes the length "X" of "G." This value "X" is contained in the environment list as a value of "N."

The "add_clause(CL)" predicate, which adds a given clause definition to the program of the object-system can be defined as follows:

```
reflect(add_clause(CL),(G,Env,Db),
    (NG,NEnv,NDb)) :- true |
        add_db(CL,Db,NDb),
        (NG,NEnv)=(G,Env).
```

The next example is the "interpretive" predicate which execute a given goal "p" in an interpretive manner.

```
reflect(interpretive(P),(G,Env,Db),
    (NG,NEnv,NDb)) :- true |
        exec([P],Env,Db,NEnv,_),
        (NG,NDb)=(G,Db).
```

Note that this interpretive execution can be executed in parallel with other execution. Therefore, it is possible to execute the specific goals in an interpretive manner and execute others directly. One possibility is modifying this "exec" to keep the debugging information. In such case, this predicate can be used as a "debugger." This kind of modification can be performed in a quite straightforward manner.

3.3. Shift-down and shift-up

It has been explained that, when a reflective predicate is called, the system is automatically shifted one-level up. When the execution of the reflective procedure finishes, the system is automatically shifted one-level down. In that sense, shift-up and shift-down are automatically carried out by using *reflective predicates* and we do not need to specify them explicitly.

However, we sometimes need to obtain the information about the representation, not the information itself. This typically happens when we want to *implement* a reflective system. For such purposes, we prepare two built-in predicates, i.e., "shift_down" and "shift_up."

"shift_down(Exp,Down_Exp)" transforms the given representation "Exp" to the one-level lower representation "Down_Exp." "shift_up(Exp,Up_Exp)" transforms the given representation "Exp" to the one-level higher representation "Up_Exp." We should note that "shift_down" and "shift_up" just converts the representations. Therefore, they do not need to use environment for the conversion. For example, if we *shift-down* "p(a,[@1|@2],f(@2,b))" we obtain "p(a,[@!1|@!2],f(@!2,b))."

Though the use of "shift_up" and "shift_down" is not recommended for the casual user, we can use these predicates and obtain the information about the representation if we want. For example, "get_q" predicate which obtains the content of *execution goals* as its *representation* can be defined as follows:

```
reflect(get_q(Q),(G,Env,Db),(NG,NEnv,NDb))
    :- true |
        shift_down(G,Down_G),
        add_env((Q,Down_G),Env,NEnv),
        (NG,NDb)=(G,Db).
```

We need to shift down the *execution goals* because we want to get the content of *execution goals* as its *representation.*

On the other hand, "put_q" predicate, which replaces the contents of *goal queue* to the given expression "Q," can be defined as follows:

```
reflect(put_q(Q),(G,Env,Db),(NG,NEnv,NDb))
    :- true |
        shift_up(Q,NG),
        (NEnv,NDb)=(Env,Db).
```

Note that we cannot get the expected result, if we forget to *shift-up* "Q."

3.4. Meta-level databases

It is explained that reflective predicates are executed at the meta-level. The remaining question is how to build a meta-level computation system *dynamically* when the reflective predicate is executed.

Please see Figure 1 again. In general, a computation system of GHC consists of three elements, i.e., *goal queue*, *environment* and *database*. We have already explained how the meta-level *goal queue* is created, i.e., it

only consists of the reflect goal. The meta-level *environment* only contains the binding information of this goal.

How about *database*? It must be different from the object-level database. If all of guard and body goals of the reflective predicate consist of system-defined goals, no problems occur. If it includes user-defined goals, they must be defined in the *database* at the meta-level.

How can we create meta-level database different from object-level database? We assume that *initially* only object-level database exists.

"meta" and "global" predicates are prepared for such purpose. For example, if we would like to define a clause

```
G :- H | B.
```

at the the meta-level, we define it as

```
meta(G):- H | B.
```

at the object-level. When the meta-level is *dynamically* created by executing the reflective predicate, all *meta* definitions are searched from the object-level definition. Top level predicate "meta" is removed from those definitions and they are copied to the meta-level database. Similarly the meta-meta-level predicates can be defined as

```
meta(meta(G)):- H | B.
```

It is also assumed that *reflective* definitions are all copied to the meta-level, since they can be used recursively. The *global* predicate

```
global(G):- H | B.
```

is also prepared to define user-defined predicates which are common to all levels.

4. RGHC implementation

In implementing RGHC, there exists several possibilities. The most efficient implementation is re-designing the abstract machine code, which corresponds to Warren code, for RGHC. In this case, the abstract machine code must have the capability to handle system's *internal state* as *data*, or, conversely, to convert the given *data* into its *internal state*.

The other possibility is realizing RGHC system as an interpreter on top of an ordinary GHC system. Though we cannot expect too much for the execution efficiency in this case, this method has a merit that the implementation is relatively simple. We actually implemented RGHC using this method.

4.1. Description of RGHC

The top level description of RGHC can be expressed as follows:

```
r_ghc(Goal,Db,Out)
    :- true |
        transfer(Goal,GRep,Env),
        exec([GRep],Env,Db,NEnv,Res),
        make_result(Res,GRep,NEnv,Out).
```

Note that this code is exactly the same as that of "m_ghc" in Section 2.3. This means that we realize a *reflective system* as a object-level system in the meta-computation system.

However, "exec" must be enhanced to realize *reflection*. This can simply be performed by adding one program clause to the "exec" program in Section 2.3, as shown below.

```
exec([GRep|Rest],Env,Id,Db,NEnv,R)
    :- reflective(GRep,Db)|
    create_meta_db(Db,Meta_Db),
    shift_down((GRep,Rest,Env,Db),
        (D_GRep,D_Rest,D_Env,D_Db)),
    exec([reflect(D_GRep,(D_Rest,D_Env,D_Db),
        (@1,@2,@3))],
        (3,[(@1,undf),(@2,undf),(@3,undf)])
        Meta_Db,New_Meta_Env,_),
    deref_variable((@1,@2,@3),New_Meta_Env,
        (D_Rest2,D_Env2,D_Db2)),
    shift_up((D_Rest2,D_Env2,D_Db2),
        (N_Rest,N_Env,N_Db)),
    exec(N_Rest,N_Env,Id,N_Db,NEnv,R).
```

This program definition clause takes care of the creation of the reflective tower. "create_meta_db" creates the meta-database from the object-system database. "(GRep,Rest,Env,Db)" is shifted down and the representation "(D_GRep,D_Rest,D_Env,D_Db)" is generated.

Then "exec" at line 6 starts the meta-level computation using these arguments. This "exec" essentially responsible for the creation of the meta-level computation system. The *trick* of the program is in using the same "exec" at the meta-level computation. Note that variables at the meta-level computation are assigned differently from the object-level computation.

Therefore, in our implementation, the meta-level computation is executed at the same speed as its object-level computation.

When the meta-level execution finishes, "@1,@2,@3" must be instantiated. We derefer these variables, shift up this information and get "(N_Rest,N_Env,N_Db)" which denotes the new object-level information. Then we return to the object-level execution using this information.

Figure 2 shows how the reflective tower is constructed by calling reflective predicates and how it is collapsed by finishing up their execution.

4.2. RGHC implementation on PSI-II

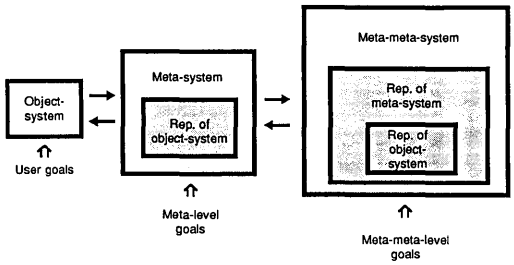


Figure 2: Constructing and collapsing a reflective tower

@1	undf
@2	a
@3	@2
@4	f(@1,@2)
@5	undf
@6	undf
...	...

Figure 3: Vector representation of variable bindings

The prototype implementation of RGHC has already been finished up using PSI-II [Nakashima 87] sequential Prolog machine. We used KL1 [ICOT 89] and ESP [Chikayama 84] as our implementation programming languages. KL1 is the extension of GHC, running on PSI-II hardware. Various extensions has been made to GHC, considering the actual program development on PSI-II. KL1 is used to describe the core part of the program. On the other hand, user interface and i/o part are written in ESP, the object-oriented dialect of Prolog. The total size of the program is 1530 lines, where KL1 part consists of 985 lines.

Though the RGHC system can *conceptually* be described as shown in Section 4.1, this implementation is very expensive since *list* is used for expressing *environment*. It sequentially searches the element and it leads to the inefficiency when the length of the *list* becomes long.

Therefore, the *vector* data type is used instead of *list* for *internal* implementation. KL1 provides us with *vector* data type, where the *index search* is possible. Figure 3 shows the *vector representation* which corresponds to the *variable bindings* shown in Section 2.2.2. This *vector* implementation is initiated by [Koshimura 90] and it has turned out to be very efficient.

In KL1, a vector can be created by executing "`new_vector(Vector,N)`" goal, where "`N`" is the input for specifying the vector size and "`Vector`" is the output keeping the reference pointer to the vector. The content of i-th element can be examined by "`vector_element(Vector,I,Element)`."

"`set_vector_element(Vector,I,OldElem,NewElem, NewVector)`" is used for setting value "`NewElem`" to the i-th element of vector "`Vector`." We should note that the old value of i-th element is given as "`OldElem`" and the new reference pointer to the modified vector is given as "`NewVector`." It can be seen that this "`set_vector_element`" predicate is defined in a quite declarative manner. However, at the language implementation level, the KL1 system is managing the reference count for the vector and destructive assignment is performed when no other goals are pointing the vector.

Our policy for RGHC implementation is as follows: A *vector* is used instead of *list* for *internal* implementation. However, we still continue to use *list* structures for the *external* representation. Therefore, the *reflective* programming examples shown in the previous sections are still effective. On the other hand, *exec* must be modified slightly to handle *vectors*, though we omit the implementation details because of the space limitation.

Note that the use of the internal database, such as seen in DEC-10 Prolog [Bowen 83], may also be effective for the more efficient implementation. If we use the internal database, fast program look-up becomes possible using the key. Though we have not used the internal database in representing programs, these kinds of considerations become more important, especially when the size of the object-level program becomes larger.

5. Program execution example

The snapshots of a program execution example are shown in Figures 4 and 5. When the RGHC system is started, "`I/O WINDOW Level 1`" is automatically opened, where "`level 1`" means the *object level*. The initial query can be typed in from this window. In this example, we have typed in "`<- test(Q,A,B).`" The definition of "`test`" predicate is shown in the "`pmacs_window`," located to the right side of Figure 4, for the reference.

As seen in this program, "`get_q(Q)`" is defined as a reflective goal. When this "`get_q(Q)`" goal is executed, the meta-level computation system is *dynamically* created and "`I/O WINDOW Level 2`" is opened.

Figure 4 shows the instant when the meta-level computation has just finished up. "`reflect(get_q(...)...)`" in "`I/O WINDOW Level 2`" shows the reflective goal to be executed at the meta-level. This window shows that the execution of this goal has been finished successfully by 42 steps. The bindings of variables allocated at the meta-level are also shown. Variables at the meta-level are shown by `@(1),@(2),@(3)` and the object-level variables are shown by `@!5,@!7`. These representations are slightly different from those explained in Section 2.2.2, since it includes `()` at the meta-level. The differences mainly come from the regulations of our GHC system and are not essential.

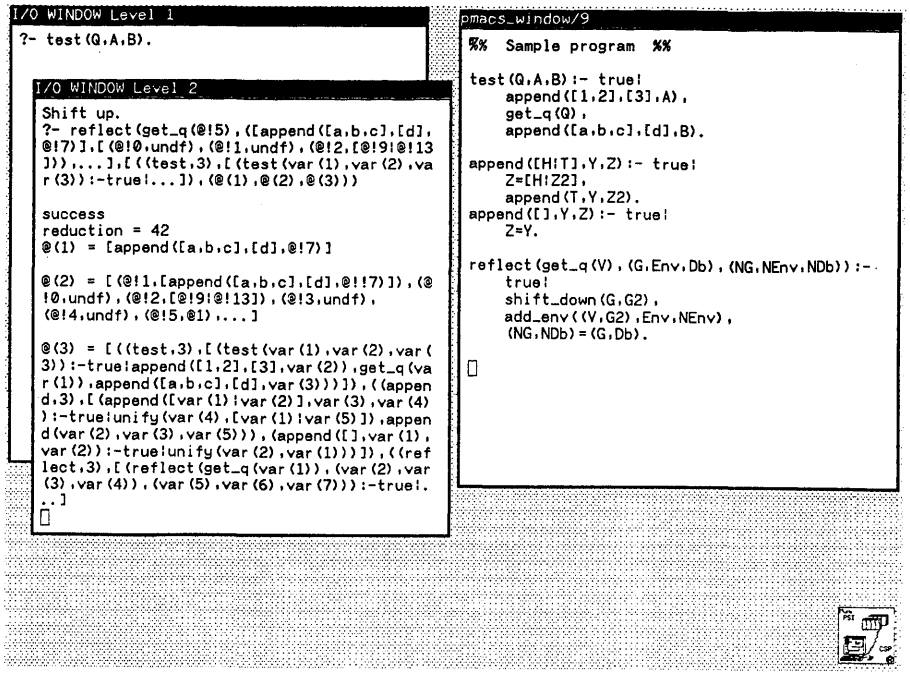


Figure 4: Execution example of RGHC (1)

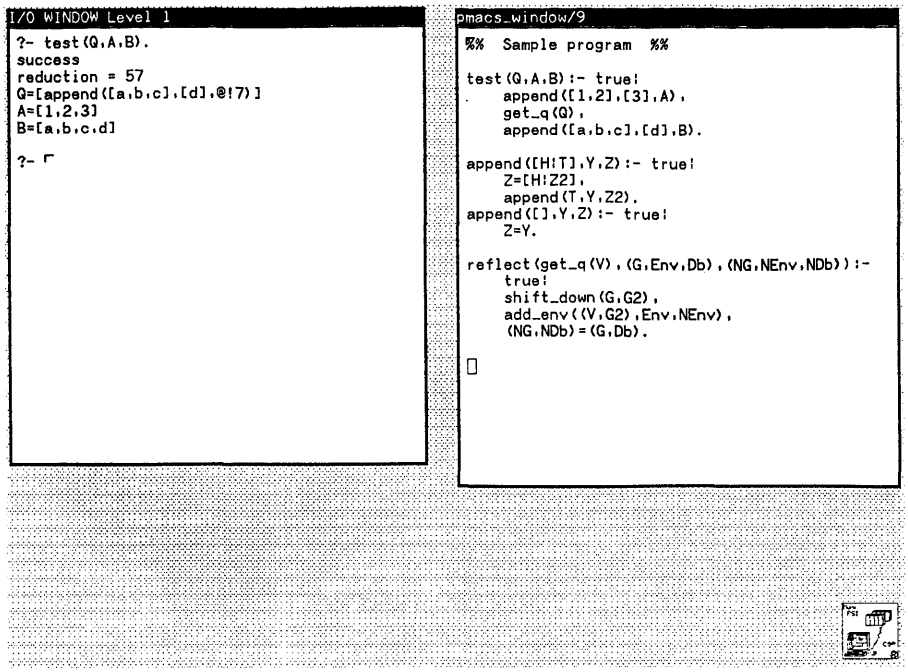


Figure 5: Execution example of RGHC (2)

When the meta-level computation terminates, "`I/O WINDOW Level 2`" also disappears. Figure 5 shows the instance when the whole computation terminates. The final computation result is shown in "`I/O WINDOW Level 1`." It shows that the execution has been terminated successfully by 57 steps and the variable bindings at that time are also shown.

6. Related works and conclusion

In logic programming field, [Bowen 82] provides us the starting point for *meta-programming* research. There exists related research, such as [Porto 84, Eshghi 86]. Also some related research was carried out in parallel logic programming field [Shapiro 84, Clark 84, Hirsch 86, Safra 86].

Recently, two workshops on *meta-programming* in logic programming, i.e., Meta 88 and Meta 90, have been held [Lloyd 88a, Bruynooghe 90]. It seems that the attention has been paid to the theoretical foundation of meta-programming. Several considerations have been done for the meta-level *representation* of the object-system [Lloyd 88b, Lim 90]. Our representation, described in Section 2, is very close to Lim's approach.

Though we described the GHC meta-computation system first in this paper, it seems that our object-system representation is quite straightforward one and that most people agree with our representation scheme.

However, regarding to *reflection* in logic programming, there exist few research works so far. The exceptions are [Costantini 89, Sugano 90, Lamma 91]. They all worked for *reflection* in Prolog.

In [Costantini 89, Lamma 91] the interests mainly lie in controlling the program execution dynamically by redefining the *solve* predicate at the meta-level. In spite of his claim on *computational reflection*, their systems have only a very limited expressive power,

On the other hand, [Sugano 90] assumes the similar kind of *reflective predicate definition* as proposed in this paper. However, his interest is mainly in semantics. Not much consideration has done for the implementation, the execution efficiency and the application.

The features of our RGHC system can be summarized as follows:

1. Reflection mechanism by *reflective predicate definition*. The user can freely define *reflective predicates* which invoke *reflection* when called. We can handle *current continuation*, *environment* and *program* at the meta-level. This mechanism is the GHC version of *reflective function* in 3-Lisp.

2. Dynamic constructing and collapsing of a reflective tower. In our system, a new level is generated when a reflective predicate is called. When finished, that level is collapsed and the system automatically returns to its original level. By calling *reflective predicate* recursively, the arbitrary level of *meta* can be created dynamically.

3. Creation of the arbitrary layers of databases. We can define the meta-level database, which is different from the object-level by "`meta`" predicate. It is also possible to define the arbitrary layers of databases by using "`reflect`," "`meta`" and "`global`" predicates.

It seems that the unique feature of RGHC is the implementation simplicity of *reflection*. As shown in Section 4, the trick is in using the same "`exec`" at the meta-level computation. Therefore, the meta-level computation can be executed at the same speed as its object-level computation.

This elegance of the implementation mainly comes from the simplicity of the language. This seems to be its most critical difference from the implementation of *reflection* in Lisp or the one in object-oriented languages. Though we did not mentioned the semantics of RGHC, we should note that [Sugano 90] worked out for defining the semantics of his R-Prolog* by the extended Herbrant base with i/o pair.

Our final goal exists in building a sophisticated distributed operating system on top of the distributed inference machine such as PIM [Uchida 88]. Some trials for describing such systems can be seen in [Tanaka 88, Tanaka 90, Tanaka 91].

7. Acknowledgments

This research has been carried out as a part of the Fifth Generation Computer Project of Japan. The authors would like to express thanks to Yukiko Ohta, Simon Martin, Hiroyasu Sugano and Youji Kohda their useful discussions.

References

[Bowen 82] K. Bowen and R. Kowalski: Amalgamating Language and Metalanguage in Logic Programming, *Logic Programming*, pp.153-172, Academic Press, London, 1982.

[Bowen 83] D.L. Bowen, L. Byrd, F.C.N. Pereira, L.M. Pereira and D.H.D. Warren: DECsystem-10 Prolog User's Manual, University of Edinburgh, August 1983.

[Bruynooghe 90] M. Bruynooghe eds.: *Proceedings of the Second Workshop on Meta-Programming in Logic*, Leuven, Belgium, April, 1990.

[Chikayama 84] T. Chikayama: Unique Features of ESP, Proceedings of the International Conference on Fifth Generation Computer Systems 1984, pp.292-298, ICOT, 1984.

[Clark 84] K. Clark and S. Gregory: Notes on Systems Programming in Parlog, Proceedings of the International Conference on Fifth Generation Computer Systems 1984, pp.299-306, ICOT, 1984

[Costantini 89] S. Costantini and G.A. Lanzarone: A Metalogic Programming Language, *Logic Programming: Proceedings of the Sixth International Conference*, pp.218-233, The MIT Press.

[Eshghi 86] K. Eshghi: it Meta-Language in Logic Programming, Ph.D. Thesis, Department of Computing, Imperial College, July 1986.

[Hirsch 86] M. Hirsch, W. Silverman and E. Shapiro: Layers of Protection and Control in the Logix System, Weizmann Institute of Science Technical Report CS86-19, 1986.

[ICOT 89] ICOT: PIMOS Manual, Version 1, ICOT, July 1989 (*in Japanese*).

[Koshimura 90] M. Koshimura, H. Fujita and R. Hasegawa: Meta-programming in KL1, SIGSYM Meeting, No.57-2, IPSJ, November 1990 (*in Japanese*).

[Lamma 91] E. Lamma, P. Mello and A. Natali: Reflection Mechanisms for Combining Prolog Databases, *Software Practice and Experience*, Vol.21, No.6, pp.602-624, June 1991.

[Lim 90] P. Lim and P.J. Stucky: Meta Programming as Constraint Programming, *Logic Programming, Proceedings of the 1990 North American Conference*, The MIT Press pp.416-430, 1990.

[Lloyd 88a] J. W. Lloyd eds.: *Proceedings of the Workshop on Meta Programming in Logic Programming*, University of Bristol, June 1988.

[Lloyd 88b] J. W. Lloyd: Directions for Meta-Programming, Proceedings of of the International Conference on Fifth Generation Computer Systems 1988, pp.609-617, ICOT, November 1988.

[Maes 88] P. Maes and D. Nardi eds: *Meta-Level Architectures and Reflection*, North-Holland, 1988.

[Nakashima 87] H. Nakashima and K. Nakajima: Hardware Architecture of the Sequential Inference Machine: PSI-II, Proceedings of 1987 Symposium on Logic Programming, San Francisco, pp.104-113, 1987.

[Porto 84] A. Porto: Two-level Prolog, Proceedings of of the International Conference on Fifth Generation Computer Systems 1984, pp.356-360, ICOT, November 1984.

[Safra 86] S. Safra and E. Shapiro: Meta Interpreters for Real, Proceedings of IFIP 86, pp.271-278, 1986.

[Shapiro 84] E. Shapiro: Systems programming in Concurrent Prolog, Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, pp.93-105, ACM, January 1984.

[Smith 84] B.C. Smith: Reflection and Semantics in Lisp, Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, pp.23-35, ACM, January 1984.

[Sugano 90] H. Sugano: Meta and Reflective Computation in Logic Programs and its Semantics, Proceedings of the Second Workshop on Meta-Programming in Logic, Leuven, Belgium, pp.19-34, April, 1990.

[Tanaka 86] J. Tanaka, K. Ueda, T. Miyazaki A. Takeuchi, Y. Matsumoto and K. Furukawa: Guarded Horn Clauses and Experiences with Parallel Logic Programming, Proceedings of FJCC '86, ACM, Dallas, Texas, pp.948-954, November 1986.

[Tanaka 88] J. Tanaka: Meta-interpreters and Reflective Operations in GHC, Proceedings of of the International Conference on Fifth Generation Computer Systems 1988, pp.774-783, ICOT, November 1988.

[Tanaka 90] J. Tanaka, Y. Ohta and F. Matono: Overview of Experimental Reflective Programming System ExReps, *Fujitsu Scientific & Technical Journal*, Vol.26, No.1, pp.86-97, April 1990.

[Tanaka 91] J. Tanaka: An Experimental Reflective Programming System Written in GHC, *Journal of Information Processing*, Vol.14, No.1, pp.74-84, 1991.

[Ueda 85] K. Ueda: Guarded Horn Clauses, ICOT Technical Report, TR-103, 1985.

[Uchida 88] S. Uchida, K. Taki, K. Nakajima, A. Goto and T. Chikayama: Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project, Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pp.16-36, ICOT, November 1988.

[Weyhrauch 80] R. Weyhrauch: Prolegomena to a Theory of Mechanized Formal Reasoning, *Artificial Intelligence*, 13, pp.133-170, 1980.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

CHARM: Concurrency and Hiding in an Abstract Rewriting Machine*

Andrea Corradini **Ugo Montanari** **Francesca Rossi**

University of Pisa
Computer Science Department
Corso Italia 40, 56100 Pisa, Italy
{andrea,ugo,rossi}@dipisa.di.unipi.it

Abstract

CHARM (for Concurrency and Hiding in an Abstract Rewriting Machine) is an abstract machine which allows to naturally model the behaviour of distributed systems consisting of a collection of processes sharing variables. *CHARM* is equipped with a clean operational semantics based on term rewriting over a suitable algebra, and it exhibits a sophisticated treatment of concurrency and modularity, which is obtained through the partition of each state into a global and a local part. To show the expressivness and generality of this abstract machine, two relevant computational formalisms, graph grammars and concurrent constraint programming, are mapped onto the *CHARM* framework.

1 Introduction

Various formalisms have been proposed in the last decades for describing and specifying concurrent programming and distributed systems. Among them we recall Petri nets [Reisig 1985], CCS [Milner 1989], CSP [Hoare 1985], the Chemical Abstract Machine [Berry and Boudol 1990], Graph Grammars [Ehrig 1979], and Concurrent Constraint Programming [Saraswat 1989]. However, the high number of such formalisms shows the need for a unifying framework, which should be able to capture the essence of concurrent computations. Such a framework should be general enough, so that most of the formalisms already proposed could be embedded in it, but it should be also expressive enough, so to be able to prove interesting properties about it. We have found that a reasonable balance of generality and expressiveness can be enjoyed by a formalism able to express in a simple way both concurrency and modularity. In fact, such notions are fundamental in order to describe how concurrent systems interact and synchronize, evolve, compose, or embed in other systems.

In this paper we propose an abstract machine, called *CHARM* (for Concurrency and Hiding in an Abstract Rewriting Machine), which is intended to satisfy the above need for a unifying framework for concurrent programming. Such a machine exhibits a sophisticated treatment of the above cited issues of concurrency and modularity, which subsumes and surpasses the corresponding treatment of many other formalisms.

States of a *CHARM* are collections of processes interacting through shared variables. The issue of modularity is addressed basically by partitioning the states into a global (i.e., visible) and a local (i.e., hidden) part. In fact, the global items of a system are those which allow the interaction with other systems, and thus are used to compose them in a non-trivial way. Transitions of the machine are rewrite rules described by pairs of systems with an identical global part, which expresses the part being preserved by the application of the rule (i.e., the part that the rule, being local to the rewritten state, cannot change). The presence of a global and a local part allows also a degree of concurrency higher than the one provided, for example, by the Chemical Abstract Machine [Berry and Boudol 1990] or by Petri nets [Reisig 1985]. In fact, two transitions may be applied in parallel not only when the subsystems they affect are disjoint, but also when their intersection is preserved by both of them.

The technique used for the formal definition of the *CHARM* follows the algebraic approach introduced in [Meseguer and Montanari 1990] for Petri nets, and further developed for structured transition sys-

*Research partially supported by the GRAGRA Basic Research Esprit Working Group n.3299 and by Alenia S.p.A.

tems in [Corradini 1990,Corradini et al. 1990] and for concurrent rewriting systems in [Meseguer 1990]. This approach is characterized by the fact that states and transitions of a system have the same algebraic structure, which can also be consistently extended to computations. This algebraic construction equips a system with a calculus of computations, which provides a rich modular proof system.

To show the expressiveness and generality of the *CHARM* computational model, we describe how the classical algebraic approach to graph grammars [Ehrig 1979], which has been widely used for algebraic system specification, can be implemented in our framework. Also, the *CHARM* provides a very natural interpretation of concurrent constraint programming [Saraswat and Rinard 1990] [Saraswat et al. 1991], since the sharing of variables and the possibility of "asking" (i.e., testing while preserving) a constraint are the two main notions in such a paradigm. Notice that the ability of expressing concurrent constraint programming in the *CHARM* framework is very significant, since such a paradigm is already very general and subsumes many widely used programming paradigms like logic programming [Lloyd 1987], constraint logic programming [Jaffar and Lassez 1987], and concurrent logic programming [Shapiro 1989].

Although this issue is not addressed in this paper, we are confident that also process description languages, like CCS [Milner 1989] and CSP [Hoare 1985], can be modelled within the *CHARM* framework. This hope is supported by the fact that we use an algebraic approach, where some basic operators of such languages (i.e., parallel composition, hiding, and relabelling) are already present, while other mechanisms (like synchronization and non-deterministic choice) may be coded via suitable techniques, as we will hint in Section 2.

We first give an informal description of the *CHARM* in Section 2, and then we present the formal theory underlying our approach by presenting, in Section 3, an algebra for the states of the machine and also for the rewriting rules. We then address the relationship between the *CHARM* and graph grammars and concurrent constraint programming in Sections 4 and Section 5 respectively.

2 An informal description of the abstract machine

In this section we informally give the main ideas underlying the design of the abstract machine we propose (called *CHARM* in the rest of the paper), and we also enlighten some of its advantages w.r.t. other transition systems and/or machines which have already been proposed in the literature for describing concurrent systems.

Each state of a *CHARM* is a *(distributed) system*, i.e., a collection of processes and a set of (possibly shared) variables, where each process is connected to a subset of the variables. This notion of state is very general. In fact, we do not assume any requirement on the structure of processes and variables, which thus may be interpreted in various way. For example, processes may also be though of as predicates or constraints or relations, and variables may represent communication channels or shared data structures. It is important to notice that many of the approaches proposed to represent the evolution of concurrent systems [Berry and Boudol 1990] [Reisig 1985] cannot model directly the sharing of variables, since a state is simply a multiset of processes.

Each state is partitioned into a local part and a global part, and thus will be informally indicated in the rest of this section by the pair $S = (G, L)$, where G stands for the global part and L for the local part. In terms of distributed systems, we may think of the local (resp., global) part as the hidden (resp., visible) set of variables and processes. Intuitively, the local items are those whose identity is known only to the system under consideration, while the global ones are the interface of the system with the rest of the world and thus may be known by other systems as well. For example, such an interface may contain common data structures, as well as processes implementing services of global utility.

States can be built from smaller states. For example, the parallel composition of two states (given later by the operator "|") is defined as the state whose global part is the set union of their global parts, and whose local part is the *disjoint* union of their local parts. This reflects the fact that, as we said above, the identity of the items in the global parts of the two states are known by both of them, so that items with the same name should be identified. In the state resulting from the composition, it is possible to force some items, which were global in both the composing sub-states, to become local. This can be done by using a suitable *hiding* operator, which will be denoted by "\". Parallel composition and hiding, together with a *renaming* operator (denoted by "[Φ]") define an algebra (introduced in the next section) whose terms are the states of a *CHARM*.

The dynamic behaviour of a *CHARM* is given by a collection of rewrite rules. Every rewrite rule $R : S \to S'$ maps its left-hand side $S = (G, L)$ to its right-hand side $S' = (G, L')$, both having the same

global part G, which is also called the global part of R. The graphical representation of a rewrite rule can be seen in Figure 1. The idea is that L can be cancelled and L' can be generated provided that G is present. Thus our notion of rewriting is context-dependent, the global part of a rule playing the role of the context. It is worth stressing that the global part G is not affected by the application of R, but it is simply tested for existence. Although this goes beyond the scope of this paper, this fact should allow us to define a satisfactory truly concurrent semantics for *CHARM*'s, since it minimizes the causal dependencies among rewrite rules.

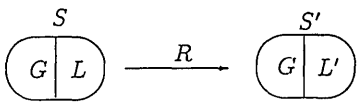


Figure 1: A rewrite rule.

Intuitively, the global part G of R contains those items (processes and variables) which are needed for the transformation of the state to take place, but which are not changed by the rewrite rule. For example, we may want to do some operation only if some data structure contains some given information. In this case, the data structure is considered to be global and thus it is not affected by the rewrite rule. It is important at this point to notice that, unlike our approach, many transition systems or abstract machines proposed in the literature (like Petri nets [Reisig 1985] and the Chemical Abstract Machine [Berry and Boudol 1990]) cannot distinguish between the situation where some item is preserved by a rewrite rule, and the one where the same item is cancelled and then generated again. For example, the rule "a rewrites to b only if c is present" must be represented in those formalisms as $\{a, c\} \rightarrow \{b, c\}$, which also represents the rule "a and c rewrite to b and c".

On the other hand, some other formalisms explicitly consider the issue of context-dependent rewriting, and allow one to formally indicate which items should be present for the application of a rule, but are not affected by it. For example, in the algebraic approach to graph grammars [Ehrig 1979], the role of the context is played by the so-called "gluing graph" of a graph production, while in concurrent constraint programming [Saraswat 1989] [Saraswat and Rinard 1990] [Saraswat et al. 1991], the items the presence of which must be tested are explicitly mentioned by the use of the "ask" primitive. The relationship between the *CHARM* and these two formalisms will be explored deeply in later sections of this paper.

A rewrite rule R from $S = (G, L)$ to $S' = (G, L')$ can be thought of as modelling the evolution of a (small) subsystem, represented by its left hand side S. To apply this rule to a given state $Q = (G_Q, L_Q)$, one first has to find an occurrence of S in Q, i.e., a subsystem of Q "isomorphic" to S (as we shall see later, this requirement will be relaxed in the formal definitions). Following the usual intuition about structured systems, it is evident that all the items which are local for a part of a system are local for the whole system as well, while items which are global for a subsystem can be either global or local for any enclosing system. In the application of a rule like the one above, this observation is formalized by requiring that the occurrence of L in Q is contained in its local part L_Q.

The application of R to a system Q yields a new system $Q' = (G_{Q'}, L_{Q'})$, where $G_{Q'} = G_Q$, i.e., the global part remains unchanged, and $L_{Q'} = (L_Q \setminus L) \cup L'$. In words, the local part of the new state coincides with the local part of the old one, except that the occurrence of L has been replaced by an occurrence of L'. Thus, the part of the state Q which is preserved by the application of R is partitioned in two parts: the occurrence of G, which is necessary to apply R, and the rest, which does not take part in the rewriting. The graphical representation of the application of R to Q can be seen in Figure 2.

The fact that the application of a rewrite rule preserves the global part of the state can be justified by interpreting the items contained in the global part as an interface for a possible composition with other states. Thus, such an interface cannot be modified by any rewrite rule, since a rewrite rule is local to the rewritten state. Notice that, as a consequence of the above considerations, a closed system, i.e., a system which is not supposed to be composed further, is represented by a state with no global part.

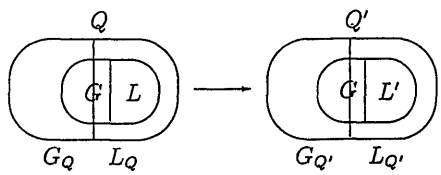


Figure 2: The application of a rewrite rule.

The above construction describes the application of a single rewrite rule of a *CHARM* to a state. However, this mechanism is intrinsically concurrent, in the sense that many rewrite rules may be applied in

parallel to a state, provided that their occurrences do not interfere. In particular, if the occurrences of the rules are pairwise disjoint, we have a degree of parallelism which is supported also by many other models of concurrent computation, like Petri nets [Reisig 1985], the Chemical Abstract Machine [Berry and Boudol 1990], and the concurrent rewriting of [Meseguer 1990]. However, our approach provides a finer perception of the causal dependencies among rewrite rules, because rules whose occurrences in a state are not disjoint but intersect only on their global parts can be considered not to depend on each other, and thus can be applied concurrently. This fact reflects the intuition, since such rules interact only on items which are preserved by all of them. This corresponds to what is called "parallel independence" of production applications in the algebraic theory of graph grammars [Ehrig 1979], which can in fact be faithfully implemented within the *CHARM* framework, as we will see in Section 4.

From a technical point of view, the application of one or more rules of a *CHARM* is modelled by extending the algebra of states to the rules (for similar approaches in the case of Petri nets or structured transition systems see [Meseguer 1990], [Corradini et al. 1990] and [Meseguer and Montanari 1990]). As we shall see in Section 3, this is possible because each rule has an associated global part, just like states. The resulting algebra, called the *algebra of transitions*, contains, as elements, all the rewrite rules of the abstract machine, an identity rule $S : S \to S$ for each state S, and it is closed w.r.t. parallel composition, hiding, and substitution operations. The left and the right-hand sides of a transition (i.e., of an element of the algebra of transitions) are easily obtained by structural induction from its syntax. For example, if $R : S \to Q$ and $R' : S' \to Q'$ are two rewrite rules, then $R \mid R' : S \mid S' \to Q \mid Q'$ is a new *parallel* transition. Like rewrite rules, transitions preserve the global part of the state they are applied to.

In this paper we will assume that the algebra of transitions is freely generated by the set of rewrite rules defining a *CHARM*, and by the identity rules for all states. As a consequence of this fact, if a transition can be applied to a state S, then it can be applied to any state containing S as well. Informally, this can be considered as a meta-rule governing the behaviour of a *CHARM*, and directly corresponds to the so-called "membrane law" of the Chemical Abstract Machine [Berry and Boudol 1990].

Although the choice of a free algebra of transitions is satisfactory for the formalisms treated in this paper, more general kinds of algebras would be needed in order to deal with other formalisms, like for example process description languages [Milner 1989] [Hoare 1985]. In fact, some features of those languages (e.g., the parallel composition of agents with synchronization in the presence of restriction, and the description of atomic sequences of actions, useful to provide a low-level implementation of the non-deterministic choice operator "+" [Gorrieri et al. 1990] [Gorrieri and Montanari 1990]) cannot be modelled adequately within a free algebra of transitions. Nevertheless, as shown in [Ferrari 1990] and [Gorrieri and Montanari 1990] respectively, both those aspects can be faithfully modelled in an algebraic framework by labelling transitions with observations which include an error label and by specifying suitable algebraic theories of computations where the atomic sequences are basic operators. Thus we are confident that, although this goes beyond the scope of this paper, these topics could be fruitfully addressed in the algebraic framework introduced here, by slightly generalizing the construction of the algebra of transitions of *CHARM*'s presented in the next section.

A computation of a *CHARM* is a sequence of transitions, starting from a given initial state. Since each transition preserves the global part of its left-hand side state, the final state of a computation has the same global part as the initial state. Thus every computation is naturally associated with a global part as well. As for transitions, this will allow us to define an algebra of computations, having the same operations of the algebra of states, plus a sequential composition operation denoted by "; ". The elements of the algebra of computations are subject to the same axioms as for states, plus some axioms stating that all the operations distribute over sequential composition. Thus we have a rich language of computations, where some computations can be proved to be equivalent by using the axioms.

The interesting fact is that the algebra of computations allows one to relate the global evolution of a closed system to the local behaviour of its subsystems. For example, suppose to consider the closed system $P = (S \mid S')\backslash x$, where the two subsystems S and S' cooperate through the common global variable x which is hidden by the use of the $\backslash x$ operator. Furthermore, consider the computations $\rho : S \Rightarrow Q$ and $\rho' : S' \Rightarrow Q'$ for S and S' respectively. Then, by using the algebra of computations it is possible to construct the computation $\sigma = (\rho \mid \rho')\backslash x$ which models the evolution of the closed system P, i.e., $\sigma : P \Rightarrow P'$, where $P' = (Q \mid Q')\backslash x$.

The algebra of computations provides also some

basic mechanisms which should allow to model process synchronization. In fact, consider for example the two computations $\sigma = (\rho \mid \rho')\backslash x$ and $\sigma' = (\rho\backslash x) \mid (\rho'\backslash x)$. Now, $\sigma \neq \sigma'$, since ρ and ρ' can synchronize through the common variable x in σ, but not in σ'.

Another relevant advantage of the definition of the algebra of computations of a *CHARM* consists of the possibility of providing a truly concurrent semantics in a natural way. In fact, computations differing only in the order in which independent rewrite rules are applied fall within the same equivalence class. For example, considering again the computation σ' introduced above, we have that $\sigma' = (\rho\backslash x) \mid (\rho'\backslash x) = ((\rho\backslash x) \mid S')\ ;\ (Q \mid (\rho'\backslash x))$, where S' and Q stand for the identity computations on such states. This means that, since $\rho\backslash x$ and $\rho'\backslash x$ are independent, they can be performed either in parallel or sequentially, and the two resulting computations are equivalent.

With each equivalence class of computations it is possible to associate a partial ordering, recording the causal dependencies among the rewrite rules used in the computations. For the two formalisms we shall consider in this paper (i.e., graph grammars and concurrent constraint programming) the truly concurrent semantics obtained via their translation to a *CHARM* is significant. In fact, it is possible to show that some of the classical results about concurrency and parallelism in graph grammars directly derive from the axioms of our algebra. Also, the truly concurrent semantics proposed in [Montanari and Rossi 1991] for the concurrent constraint programming framework coincides with the one induced by its compilation into a *CHARM*. However, the true concurrency aspects go beyond the scope of this paper.

3 Formal Definitions

In this section we present the formal description of a *CHARM*, following the outline of the informal presentation given in the previous section. After introducing the algebra of states, a *CHARM* will be defined as a collection of rewrite rules over this algebra which preserve the global part of a term. Next we will introduce the algebra of transitions and the algebra of computations of a *CHARM*, respectively.

The states of a *CHARM* are going to be represented by the terms of an algebra S, which is parametric w.r.t. a fixed pair of disjoint infinite collections $(\mathcal{P}, \mathcal{V})$, called *process instances* and *variables* respectively. The terms of this algebra are subject to the axioms presented below in Definition 3.

Definition 1 *Let $\mathcal{P}$ be a set of* process instances *(ranged over by p, q, ...), and $\mathcal{V}$ be a set of* variables *(ranged over by v, z, ...). Each $x \in (\mathcal{P} \cup \mathcal{V})$ is called an* item. *The algebra of states $\mathcal{S}$ is the algebra having as elements the equivalence classes of terms generated by the following syntax, modulo the least equivalence relation induced by the axioms listed in Definition 3:*

$$S ::= 0 \mid v \mid p(v_1, \ldots, v_n) \mid S \mid S \mid S[\Phi] \mid S\backslash x$$

where v, v_1, ..., $v_n \in \mathcal{V}$; $p \in \mathcal{P}$; " $\mid$ " is called parallel composition*; Φ is a (finite domain) substitution, i.e., a function $\Phi : (\mathcal{P} \cup \mathcal{V}) \to (\mathcal{P} \cup \mathcal{V})$ such that $\Phi(\mathcal{V}) \subseteq \mathcal{V}$, $\Phi(\mathcal{P}) \subseteq \mathcal{P}$, and such that the set of items for which $x \neq \Phi(x)$ is finite; and x is an item ($\backslash x$ is called a* hiding *operator). Term of the form 0, v, or $p(v_1, \ldots, v_n)$ are called* atoms.■

Intuitively, 0 is the empty system, v is the system containing only one variable, and $p(v_1, \ldots, v_n)$ represents a system with one process which has access to n variables. The term $S_1 \mid S_2$ represents the composition of system S_1 and system S_2, $S[\Phi]$ is the system obtained from state S by renaming its items, and $S\backslash x$ is the system which coincides with system S except that item x is local.

Definition 2 *Given a term $S \in \mathcal{S}$, its set of* free items *$\mathcal{F}(S)$ is inductively defined as $\mathcal{F}(0) = \emptyset$; $\mathcal{F}(v) = \{v\}$; $\mathcal{F}(p(v_1, \ldots, v_n)) = \{p, v_1, \ldots, v_n\}$; $\mathcal{F}(S_1 \mid S_2) = \mathcal{F}(S_1) \cup \mathcal{F}(S_2)$; $\mathcal{F}(S[\Phi]) = \Phi(\mathcal{F}(S)) = \{\Phi(x) \mid x \in \mathcal{F}(S)\}$; and $\mathcal{F}(S\backslash x) = \mathcal{F}(S) \setminus \{x\}$ if $x \in \mathcal{F}(S)$ and $\mathcal{F}(S\backslash x) = \mathcal{F}(S)$ otherwise. A term S is* closed *iff $\mathcal{F}(S) = \emptyset$. A term is* concrete *if it does not contain any hiding operator. A term is* open *if no variable appearing in the term is restricted. Formally, all atoms are open; $S_1 \mid S_2$ is open if both S_1 and S_2 are open; $S[\Phi]$ is open if S is open; and $S\backslash x$ is open if S is open and $x \notin \mathcal{F}(S)$. Clearly, all concrete terms are open.*■

The free items of a term S are the process instances and the variables of the global part of the system represented by S. Thus a closed term represents a system with no global part, while an open term corresponds to a system where everything is global. The above interpretation of the operators of the algebra of states is supported by the following axioms, which determine when two terms are equivalent, i.e., represent the same system.

Definition 3 *The terms of algebra $\mathcal{S}$ introduced in Definition 1 are subject to the following conditional axioms.*

ACI: $(S_1 \mid S_2) \mid S_3 = S_1 \mid (S_2 \mid S_3)$; $S_1 \mid S_2 = S_2 \mid S_1$; $S \mid 0 = S$

ABS: $p(v_1,\ldots,v_n) \mid v_i = p(v_1,\ldots,v_n)$, for $1 \le i \le n$;
$S \mid S = S$, if S is open
COMP: $S[\Phi][\Psi] = S[\Psi \circ \Phi]$
EXC: $S\backslash x\backslash y = S\backslash y\backslash x$
EL: $S\backslash x = S$, if x is not free in S
MAP: $p(v_1,\ldots,v_n)[\Phi] = \Phi(p)(\Phi(v_1),\ldots,\Phi(v_n))$;
$v[\Phi] = \Phi(v)$; $0[\Phi] = 0$
DIS: $(S_1 \mid S_2)[\Phi] = S_1[\Phi] \mid S_2[\Phi]$
FAC: $S_1\backslash x \mid S_2 = (S_1 \mid S_2)\backslash x$, if x is not free in S_2
SWAP: $(S\backslash x)[\Phi] = S[\Phi]\backslash\Phi(x)$, if $\nexists y \in \mathcal{F}(S\backslash x)$ such that $\Phi(y) = \Phi(x)$
α-CONV: $S[\Phi] = S$, if Φ is bijective and $\Phi(x) = x\ \forall x \in \mathcal{F}(S)$

Two terms S and S' are equivalent *(written $S \approx S'$) if they are in the least congruence relation (w.r.t. all the operators of the algebra) induced by the above axioms.*■

In words, axioms ACI and ABS state that the parallel composition of systems behaves like disjoint union on the local parts of systems, and like set union on the global ones. Axioms $COMP$ and MAP deal with substitution composition and application respectively, while axiom DIS states that substitutions distribute over parallel composition. Axioms EXC and EL state that the items of a system can be made local only once and in any order. Finally, axioms FAC and $SWAP$ describe in an obvious way the interplay between the hiding operator and the operators for parallel composition and substitution, while axiom α-CONV formalizes the intuition that the names of the hidden items are not meaningful.

As anticipated in Section 2, the rewrite rules which define a *CHARM* must preserve the global part of the states they can be applied to. Therefore we define a function $\mathcal{GP}$ which extracts from each term a concrete subterm, corresponding to its global part. Actually, $\mathcal{GP}$ is a partial function on $\mathcal{S}$, because some term may denote a system whose global part is not a legal system. This happens when a variable is made local, but some process using it is considered as global. Terms on which $\mathcal{GP}$ is defined are called well-formed. The function $\mathcal{GP}$ is defined by exploiting the existence of a canonical form of terms.

Proposition 4 *Every term S of the algebra of states $\mathcal{S}$ has an equivalent canonical form $S_1 \mid S_2\backslash x_1 \ldots \backslash x_n$, where S_1 and S_2 are parallel compositions of atoms. If $S_2 = S_{21} \mid \ldots \mid S_{2k}$, then $\forall i = 1,\ldots,k$, either $S_{2i} = 0$ or $\mathcal{F}(S_{2i}) \cap \{x_1,\ldots,x_n\} \neq \emptyset$. Moreover, for each atom of the form $q(v_1,\ldots,v_m) \in S_2$, $\forall i = 1,\ldots,m$, either $v_i = x_j$ for some $j = 1,\ldots,n$, or v_i occurs in S_1. If $S_1 \mid S_2\backslash x_1 \ldots \backslash x_n$ and $S_1' \mid S_2'\backslash y_1 \ldots \backslash y_k$ are two canonical forms for term S, then $S_1 \approx S_1'$.* ■

Definition 5 *Let S be a term and $S_1 \mid S_2\backslash x_1 \ldots \backslash x_n$ be one of its canonical forms. Term S is* well-formed *if and only if for each atom of the form $q(v_1,\ldots,v_m) \in S_2$, $q = x_i$ for some i. Then, the global part of a well-formed term S is defined as $\mathcal{GP}(S) = S_1$.*■

Definition 6 *A* rewrite rule *R over $\mathcal{S}$ is a pair of well-formed terms of $\mathcal{S}$, $R = (S, S')$ (also written $R : S \to S'$) such that $\mathcal{GP}(S) \approx \mathcal{GP}(S')$. A* CHARM *$\mathcal{M}$ (over $\mathcal{S}$) is a collection of rewrite rules over $\mathcal{S}$, i.e., $\mathcal{M} = \{R_i : S_i \to S_i'\}_{i \in I}$.*■

Definition 7 *Let $\mathcal{M} = \{R_i : S_i \to S_i'\}_{i \in I}$ be a* CHARM *over $\mathcal{S}$. Then the* algebra of transitions *of $\mathcal{M}$, $\mathcal{T}(\mathcal{M})$, is generated by the following inference rules, which also give the left and the right-hand side of each transition.*

$$\frac{i \in I}{R_i : S_i \to S_i'} \qquad \frac{S \in \mathcal{S}}{S : S \to S}$$

$$\frac{T : S \to Q,\ T' : S' \to Q'}{T \mid T' : S \mid S' \to Q \mid Q'} \qquad \frac{T : S \to Q}{T[\Phi] : S[\Phi] \to Q[\Phi]}$$

$$\frac{T : S \to Q}{T\backslash x : S\backslash x \to Q\backslash x}$$

The free items of a transition are the free items of its left (or right) hand side. A transition is open iff it has the form $S : S \to S$, with S open. The terms of $\mathcal{T}(\mathcal{M})$ are subject to the same axioms as in Definition 3.■

Definition 8 *Let $\mathcal{M} = \{R_i : S_i \to S_i'\}_{i \in I}$ be a* CHARM *over $\mathcal{S}$, and $\mathcal{T}(\mathcal{M})$ be its algebra of transitions. Then the* algebra of computations *of $\mathcal{M}$, $\mathcal{C}(\mathcal{M})$, is generated by the following inference rules, where $\rho : S \Rightarrow S'$ means that computation ρ starts from state S and ends in state S':*

$$\frac{T : S \to Q \in \mathcal{T}(\mathcal{M})}{T : S \Rightarrow Q} \qquad \frac{\rho : S \Rightarrow S',\ \rho' : S' \Rightarrow S''}{\rho;\rho' : S \Rightarrow S''}$$

$$\frac{\rho : S \Rightarrow Q,\ \rho' : S' \Rightarrow Q'}{\rho \mid \rho' : S \mid S' \Rightarrow Q \mid Q'} \qquad \frac{\rho : S \Rightarrow Q}{\rho[\Phi] : S[\Phi] \Rightarrow Q[\Phi]}$$

$$\frac{\rho : S \Rightarrow Q}{\rho\backslash x : S\backslash x \Rightarrow Q\backslash x}$$

The free items of a computation are the free items of its starting (or ending) state. The terms of $\mathcal{C}(\mathcal{M})$ are subject to the same axioms as in Definition 3, plus the following functoriality axioms, *valid whenever both sides are defined, stating that the operations of the algebra distribute over sequential composition.*

$(\rho \mid \rho');(\sigma \mid \sigma') = (\rho;\sigma) \mid (\rho';\sigma')$
$(\rho;\sigma)[\Phi] = \rho[\Phi];\sigma[\Phi]$
$(\rho;\sigma)\backslash x = \rho\backslash x;\sigma\backslash x$ ■

4 Modelling graph grammars

The "theory of graph grammars" studies a variety of formalisms which extend the theory of formal languages in order to deal with structures more general than strings, like graphs and maps. A graph grammar allows one to describe finitely a (possibly infinite) collection of graphs, i.e., those graphs which can be obtained from an initial graph through repeated application of graph productions. In this section we shortly show how to translate a graph grammar into a *CHARM* which faithfully implements its behaviour. Because of space limitations, the discussion will be very informal: a more formal presentation of this translation can be found in [Corradini and Montanari 1991].

Following the so-called *algebraic approach* to graph grammars [Ehrig 1979], a graph production $p = (L \stackrel{l}{\hookleftarrow} K \stackrel{r}{\hookrightarrow} R)$ is a pair of graph monomorphisms having as common source a graph K, the *gluing graph*, indicating which edges and nodes have to be preserved by the application of the production. Throughout this section, for graph we mean *unlabelled, directed hypergraph*, i.e., a triple $G = (N, E, c)$, where N is a set of *nodes*, E is a set of *edges*, and $c : E \to N^*$ is the *connection function* (thus each edge can be connected to a list of nodes). Production p can be applied to a graph G yielding H (written $G \Rightarrow_p H$) if there is an *occurrence* (i.e., a graph morphism) $g : L \to G$, and H is obtained as the result of the *double pushout* construction of Figure 3.

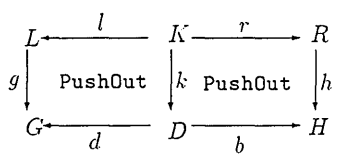


Figure 3: Graph rewriting via double pushout construction.

This construction may be interpreted as follows. In order to delete the occurrence of L in G, we construct the "pushout complement" of g and l, i.e., we have to find a graph D (with morphism $k : K \to D$ and $d : D \to G$) such that the resulting square is a "pushout". Intuitively, graph G in Figure 3 is the pushout object of morphisms l and k if it is obtained from the disjoint union of L and D by identifying the images of K in L and in D. Next, we have to embed the right-hand side R in D via a second pushout, which produces graph H. In this case we say that there is a *direct derivation* form G to H via p.

A *graph rewriting system* is a set $\mathcal{R}$ of graph productions. A *derivation* from G to H over $\mathcal{R}$ (shortly $G \Rightarrow^*_{\mathcal{R}} H$), is a finite sequence of direct derivations of the form $G \Rightarrow_{p_1} G_1 \Rightarrow_{p_2} \ldots \Rightarrow_{p_n} G_n = H$, where $p_1, \ldots, p_n$ are in $\mathcal{R}$.

To define the *CHARM* which implements a given graph rewriting system, we have to define the sets of process instances and of variables (see Definition 1). Quite obviously, we can regard a graph as a distributed system where the edges are processes, and the nodes are variables. Thus we consider a *CHARM* over the pair of sets $(\mathcal{E}, \mathcal{N})$ which are two collections including all edges and all nodes, respectively. The precise relationship between the algebra of states of such a *CHARM* and the graphs introduced above has been explored in [Corradini and Montanari 1991]. It has been shown there that concrete terms of such an algebra (i.e., terms without hiding operators, see Definition 2), faithfully model finite graphs, i.e., if *FGraph* is the collection of all finite graphs and $\mathcal{CS}$ is the sub-algebra of concrete terms of $\mathcal{S}$, there are injective functions $Gr : \mathcal{CS} \to FGraph$ and $Tm : FGraph \to \mathcal{CS}$ such that $Gr(Tm(G)) \cong G$ for each graph G. Furthermore, well-formed terms (see Definition 5) model in a similar way "partially abstract graphs", i.e., suitable equivalence classes of graph monomorphisms, where the target graph is defined up to isomorphism. For our goals, it is sufficient to introduce the function WfT which associates a well-formed term with each graph monomorphism.

Definition 9 *Let $G = (N, E, c)$ be a graph, with $N = \{n_i\}_{i \le m}$, $E = \{e_j\}_{j \le r}$, and $c(e_i) = n_{i1} \cdot \ldots \cdot n_{ik_i}$ for all $1 \le i \le r$. Then the concrete term representing G is defined as*

$Tm(G) = n_1 \mid \ldots \mid n_m \mid e_1(n_{11}, \ldots, n_{1k_1}) \mid \ldots \mid e_r(n_{r1}, \ldots, n_{rk_r}) \mid 0$.

Let $h : G \hookrightarrow H$ be a graph monomorphism. Then the well-formed term representing h is defined as

$$WfT(h) = (Tm(H) \backslash x_1 \backslash \ldots \backslash x_n)[h^{-1}]$$

where $\{x_1, \ldots, x_n\}$ is the set of items of H which are not in the image of G through h, and h^{-1} improperly denotes the substitution such that $h^{-1}(y) = x$ if $h(x) = y$, and $h^{-1}(y) = y$ otherwise (which is well defined because h is injective).■

From the last definition it can be checked that the global part of the well-formed term representing a monomorphism $h : G \hookrightarrow H$ is equivalent to the concrete term representing G, i.e., $Tm(G)$. Using this observation, and since a graph production is a pair of graph monomorphisms with common domain,

it is easy to associate a *CHARM* rewrite rule (in the sense of Definition 6) with each graph production.

Definition 10 *Let $\mathcal{R}$ be a graph rewriting system. For each graph production $p = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ in $\mathcal{R}$, its associated rewrite rule $\mathcal{M}(p)$ is defined as $\mathcal{M}(p) : WfT(l) \rightarrow WfT(r)$. The* CHARM *implementing $\mathcal{R}$ is defined as $\mathcal{M}(\mathcal{R}) = \{\mathcal{M}(p_i) \mid p_i \in \mathcal{R}\}$.*■

In order to correctly relate the operational behaviours of a graph rewriting system $\mathcal{R}$ and of its associated *CHARM* $\mathcal{M}(\mathcal{R})$, we have to take care of the translation of the starting graph of a derivation into a term. In fact, if G is such a graph, it would not be sound to take as starting state of $\mathcal{M}(\mathcal{R})$ the concrete term $Tm(G)$. Indeed, we must observe that the graph derivations informally introduced above are defined up to isomorphism, i.e., if $G \Rightarrow_p H$, then $G' \Rightarrow_p H'$ for each $G' \cong G$ and $H' \cong H$. This is due to the fact that the pushout objects of Figure 3 are defined up to isomorphism. As a consequence, graph derivations actually define a relation among equivalence classes of graphs, rather than among graphs. Such equivalence classes are faithfully represented by closed terms of the algebra of states: using Definition 9, the class of all graphs isomorphic to G is represented as $WfT(0_G)$, where 0_G is the unique (mono)morphism from the empty graph to G.

The next theorem states that the translation of a graph rewriting system into a *CHARM* is sound and complete. This result is not trivial, and is based on the fact that every transition of the algebra $\mathcal{T}(\mathcal{M}(\mathcal{R}))$ (see Definition 7) represents a pair of graph monomorphisms with common source, which are the bottom line of a double pushout construction like the one depicted in Figure 3. We refer to [Corradini and Montanari 1991] for the formal proofs.

Theorem 11 *Let $\mathcal{R}$ be a graph rewriting system and $\mathcal{M}(\mathcal{R})$ be the associated* CHARM. Soundness: *If G is a graph and $\rho : WfT(0_G) \Rightarrow Q$ is a term of the algebra of computations of $\mathcal{M}(\mathcal{R})$, i.e., of $\mathcal{C}(\mathcal{M}(\mathcal{R}))$ (see Definition 8), then there is a derivation $G \Rightarrow^*_{\mathcal{R}} H$ such that $WfT(0_H) \approx Q$.* Completeness: *If $G \Rightarrow^*_{\mathcal{R}} H$, then there is a computation ρ in the algebra of computations of $\mathcal{M}(\mathcal{R})$, such that $\rho : WfT(0_G) \Rightarrow WfT(0_H)$.*■

5 Modelling concurrent constraint programming

The concurrent constraint (*cc*) programming paradigm [Saraswat 1989] is a very elegant framework which captures and generalizes most of the concepts of logic programming [Lloyd 1987], concurrent logic programming [Shapiro 1989], and constraint logic programming [Jaffar and Lassez 1987]. The basic idea is that a program is a collection of concurrent agents which share a set of variables, over which they may pose ("tell") or check ("ask") constraints. Agents are defined by clauses as the parallel composition ("||"), or the existential quantification ("∃"), or the nondeterministic choice ("+"), of other agents. A computation refines the initial constraint on the shared variables (i.e., the store) through a monotonic addition of information until a stable configuration (if any) is obtained, which is the final constraint returned as the result of such a computation.

The *cc* paradigm is parametric w.r.t. the kind of constraints that are handled. Any choice of the constraint system (i.e., kind of constraints and solution algorithm) gives a specific *cc* language. For example, by choosing the Herbrand constraint system we get concurrent logic programming, and by further eliminating concurrency we get logic programming. The constraint system is very simply modelled by a *partial information system* [Saraswat et al. 1991], i.e. a pair $< D, \vdash >$, where D is the set of the primitive constraints and $\vdash \subseteq \wp(D) \times D$ is the *entailment relation* which states which tokens are entailed by which sets of other tokens, and which must be reflexive and transitive. Then, a constraint is a set of primitive constraints, closed under entailment.

In this section we will informally show how any *cc* program can be modelled by a *CHARM*. The idea is to consider each state as the current collection of constraints (on the shared variables) and of active agents (together with the variables they involve), and then to represent each computation step as the application of a rewrite rule. More precisely, both agents and primitive constraints are going to be modelled as process instances, while the shared variables are the variables of the abstract machine.

Basic computation steps are an ask operation, a tell operation, the decomposition of an agent into other agents, but also the generation of new constraints by the entailment relation. In the following, each agent or constraint always comes together with the variables it involves, even though we sometimes will not say it explicitly.

In a state Q, the agent $A = tell(c) \rightarrow A1$ adds constraint c to Q and then transforms itself into agent $A1$. This can be faithfully modelled by a rewrite rule R from $S = (G, L)$ to $S' = (G, L')$ where L contains agent A, L' contains agent $A1$ and constraint c, and G contains the variables involved in A (since these are the only items connecting A to the rest of the

state Q). This rule may be seen in Figure 4. Note that the fact that c is present only in the local part L' of S' does not mean that c is visible only locally. In fact, the mechanism of rule application allows to treat a local item as a global one (see Figure 2).

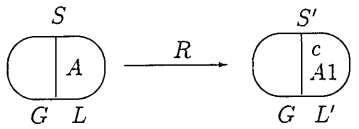


Figure 4: The *CHARM* rewrite rule for the agent $A = tell(c) \rightarrow A1$.

In a state Q, the agent $A = ask(c) \rightarrow A1$ transforms itself into $A1$ if c is in Q and suspends otherwise. The corresponding rewrite rule is R from $S = (G, L)$ to $S' = (G, L')$, where L contains agent A, L' contains agent $A1$, and G contains c. In fact, constraints, once generated, are never cancelled, since the accumulation of constraints is monotonic. Since the rewrite rule cannot be applied if there is no occurrence of the lhs in Q, the ask suspension is given for free. This rule may be seen in Figure 5.

Parallel and nondeterministic composition, as well as existential quantification of agents, are straightforwardly modelled by corresponding rewrite rules. Note that, in an "atomic" interpretation, *tell* and *ask* operations fail if c is inconsistent with the constraints in Q. Our rewrite rules model instead the "eventual" interpretation [Saraswat 1989], where inconsistency is discovered sooner or later, but possibly not immediately. Thus immediate failure is not directly modelled. However, since the difference between the two interpretations basically depends on the way the nondeterministic choice is implemented, the specification of suitable algebraic theories, as suggested in Section 2, could be of help for the implementation of the atomic interpretation of the *cc* framework.

Each pair $\langle C, t\rangle \in \vdash$ may be modelled by a state change as well. In fact, in a state Q, $\langle C, t\rangle$ can be interpreted as a tell of t whenever C is in Q, and can thus be represented by a rewrite rule R from $S = (G, L)$ to $S' = (G, L')$, where L is empty, G contains C, and L' contains t. Note that L is empty, since nothing has to be cancelled, and all items involved are either tested for presence and thus preserved (C) or generated (t). This rule may be seen in Figure 6.

Figure 5: The *CHARM* rewrite rule for the agent $A = ask(c) \rightarrow A1$.

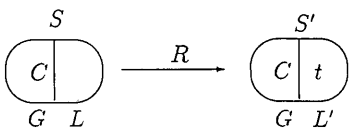


Figure 6: The *CHARM* rewrite rule for the pair $\langle C, t\rangle$ of the entailment relation $\vdash$.

In summary, (the eventual interpretation of) a *cc* program, together with the underlying constraint system, is modelled in a sound and complete way by a *CHARM* with as many rewrite rules as agents (and subagents) and pairs of the entailment relation (note that, while the number of agents is always finite, in general there may be an infinite number of pairs in the entailment relation). It is important to stress the naturality of the *CHARM* as an abstract machine for *cc* programming. In fact, the global part of the rules exactly corresponds to the idea that constraints are never cancelled, and thus, once generated locally (by one of the subsystems), are global forever. This description of *cc* programming within the *CHARM* framework follows a similar one, given in [Montanari and Rossi 1991], where the classical "double-pushout" approach to graph rewriting was used to model *cc* programs and to provide them with a truly concurrent semantics. Thus, the results of this section are not surprising, given the results in [Montanari and Rossi 1991] and those of the previous section, which show how to model graph grammars through a *CHARM*.

6 Future Work

As pointed out in Section 2, one of the subjects which seem most interesting to investigate is the possibility to provide the *CHARM* with a true-concurrency semantics. Another one is instead the implementation of process description languages onto the *CHARM*. As briefly discussed in sections 2 and 3, both these issues seem to be fruitfully addressable within the algebraic framework we have depicted in this paper.

In [Laneve and Montanari 1991] it has been shown that concurrent constraint programming may encode the lazy and the call-by-value λ-calculus.

This encoding exploits a technique similar to the one used by Milner to encode λ-calculus in π-calculus [Milner et al. 1989], since the mobility of processes (which is one of the main features of π-calculus) can be simulated in *cc* programming via a clever use of the shared logical variables. This result, combined with our implementation of *cc* programming in the *CHARM*, described in Section 5, suggests that also higher order aspects of functional languages may be expressed within the *CHARM*.

References

[Berry and Boudol 1990] G. Berry and G. Boudol. The Chemical Abstract Machine. In *Proc. POPL90*, ACM, 1990.

[Corradini 1990] A. Corradini. *An Algebraic Semantics for Transition Systems and Logic Programming.* Ph.D. Thesis TD-8/90. Dipartimento di Informatica, Università di Pisa. Italy. March 1990.

[Corradini et al. 1990] A. Corradini, G. Ferrari, and U. Montanari. Transition Systems with Algebraic Structure as Models of Computations. In *Semantics of Systems of Concurrent Processes*, Guessarian I. ed, Springer-Verlag, LNCS 468, 1990.

[Corradini and Montanari 1991] A. Corradini and U. Montanari. An Algebra of Graphs and Graph Rewriting. In *Proc. 4th Conference on Category Theory and Computer Science*. Springer-Verlag, LNCS, 1991.

[De Boer and Palamidessi 1991] F.S. De Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In *Proc. CAAP*, 1991.

[Ehrig 1979] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In *Proc. International Workshop on Graph Grammars*, Springer-Verlag, LNCS 73, 1979.

[Ferrari 1990] G. Ferrari. *Unifying Models of Concurrency.* Ph.D. Thesis, Computer Science Department, University of Pisa, Italy, 1990.

[Gorrieri et al. 1990] R. Gorrieri, S. Marchetti, and U. Montanari. A^2CCS: Atomic Actions for CCS. In *TCS 72*, vol. 2-3, 1990.

[Gorrieri and Montanari 1990] R. Gorrieri and U. Montanari. A Simple Calculus Of Nets. In *Proc. CONCUR90*, Springer-Verlag, LNCS 458, 1990.

[Hoare 1985] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Jaffar and Lassez 1987] J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *Proc. POPL*, ACM, 1987.

[Laneve and Montanari 1991] C. Laneve and U. Montanari. Mobility in the cc paradigm. Submitted for publication, 1991.

[Lloyd 1987] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.

[Meseguer 1990] J. Meseguer. Rewriting as a Unified Model of Concurrency. In *Proc. CONCUR90*, Springer-Verlag, LNCS 458, 1990.

[Meseguer and Montanari 1990] J. Meseguer and U. Montanari. Petri Nets are Monoids. *Information and Computation*, vol.88, n.2, 1990.

[Milner 1989] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Milner et al. 1989] R. Milner, J.G. Parrow, and D.J. Walker. *A calculus of mobile processes.* LFCS Reports ECS-LFCS-89-85/86, University of Edinburgh, 1989.

[Montanari and Rossi 1991] U. Montanari and F. Rossi. True Concurrency in Concurrent Constraint Programming. In *Proc. ILPS91*, MIT Press, 1991.

[Reisig 1985] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science, Springer Verlag, 1985.

[Shapiro 1989] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, vol.21, n.3, 1989.

[Saraswat 1989] V.A. Saraswat. *Concurrent Constraint Programming Languages.* Ph.D. Thesis, Carnegie-Mellon University, 1989. Also 1989 ACM Dissertation Award, MIT Press.

[Saraswat and Rinard 1990] V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proc. POPL*, ACM, 1990.

[Saraswat et al. 1991] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proc. POPL*, ACM, 1991.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Less Abstract Semantics for Abstract Interpretation of FGHC Programs

Kenji Horiuchi

Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN
horiuchi@icot.or.jp

Abstract

In this paper we present a denotational semantics for Flat GHC. In the semantics, the reactive behavior of a goal is represented by a sequence of substitutions, which are annotated with + or − depending on whether the bindings are given from, or posted to the environment of the goal. Our objective in investigating the semantics is to develop a framework for abstract interpretation. So, the semantics is *less abstract* enough to allow an analysis of various properties closely related to program sources. We also demonstrate moded type inference of FGHC programs using abstract interpretation based on the semantics.

1 Introduction

Various work on the semantics for concurrent logic languages has been investigated by many researchers [Gerth *et al.* 1988][Murakami 1988][Gaifman *et al.* 1989] [Gabbrielli and Levi 1990][de Boer and Palamidessi 1990]. One of their main purposes is to identify one program with another syntactically different program, or distinguish between syntactically similar programs. And, since some researchers are interested in properties like *fully abstractness*, they may want to hide *internal communications* from the semantics or want to abstract even observable behaviors much further.

Since our main objective is to analyze a program unlike the above researchers, we want to have a fixpoint semantics suitable to the collecting semantics, on which our framework of abstract interpretation is based. But once we try to introduce one of their semantics to a framework of abstract interpretation, the semantics may be too abstract to obtain some of the properties we require.

In this paper we present a denotational fixpoint semantics for Flat GHC. In the semantics the reactive behavior of a goal is represented by a sequence of substitutions which are annotated with + or − depending on whether the bindings are given from, or posted to the environment of the goal. The semantics presented here is *less abstract* enough to allow an analysis of various properties closely related to program sources, e.g., on occurrences of symbols in programs or internal communications. We also demonstrate moded type inference of FGHC programs using abstract interpretation.

We briefly explain the concurrent logic programming language Flat GHC and its operational semantics in Section 3 after we introduce the preliminary notions in the next section. Next we present the fixpoint approach to the semantics of Flat GHC in Section 4, and then in Section 5 we show the relationship between the fixpoint semantics and the operational semantics. After reviewing a general framework for abstract interpretation, we show examples of analyzing FGHC programs.

2 Preliminaries

In this section, we introduce the following basic notions used in this paper, many of which are defined as usual [Lloyd 1987][Palamidessi 1990].

Definition 2.1 (Functor, Term, Atom, Predicate and Expression)
Let *Var* be a non-empty set of *variables*, *Func* be a set of *functors*, *Term* be a set of all *terms* defined on *Var* and on *Func*, *Pred* be a set of all *predicates* and *Atom* be a set of all *atoms* defined on *Term* and *Pred*.

An *expression* is a term, an atom, a tuple of expressions or a (multi)set of expressions, and we denote a set of all expressions by *Exp*. We also denote the set of all variables appearing in an expression E by $var(E)$.

Definition 2.2 (Substitution)
A *substitution* θ is a mapping from *Var* to *Term* such that the *domain* of θ is finite, where the domain of θ, denoted by $dom(\theta)$, is defined by $\{V \in Var \mid \theta(V) \not\equiv V\}$. The substitution θ is also represented by a set of *assignments* such that $\{V \leftarrow t \mid V \in dom(\theta) \wedge \theta(V) \equiv t\}$. The identity mapping on *Var*, called an *identity substitution*, is denoted by $\emptyset$. The *range* of θ, denoted by $ran(\theta)$, is a set of all variables appearing in terms at the right hand side of each assignment of θ, i.e., $\bigcup_{V \in dom(\theta)} var(\theta(V))$. $var(\theta)$ also denotes the set of variables $dom(\theta) \cup ran(\theta)$

When E is an expression, $E\theta$ (or $(E)\theta$) denotes an expression obtained by replacing each variable V in E with $\theta(V)$. The *composition* of two substitution θ and σ, denoted by $\theta\sigma$, is defined as usual [Lassez *et al.* 1987][Palamidessi 1990]. A substitution θ is assumed to be always *idempotent* [Lassez *et al.* 1987]. (i.e.,

$dom(\theta) \cap ran(\theta) = \emptyset$, where $\emptyset$ denotes an empty set.) And the result of composing substitutions is also assumed to be idempotent. The set of all idempotent substitutions is denoted by *Subst* and the set of all renamings is denoted by *Ren*. A *restriction* of θ *onto* $var(E)$ is denoted by $\theta_{|E}$.

Definition 2.3 (Equivalence Class and Partial Ordering)
A pre-ordering $\preceq$ on *Subst*, called an *instantiation ordering*, is defined as follows: $\theta_1 \preceq \theta_2$ iff $\exists\sigma(\theta_1\sigma = \theta_2)$, where $\theta_1, \theta_2, \sigma \in Subst$. The equivalence relation w.r.t. an instantiation ordering $\preceq$, denoted by $\sim$, is defined as follows: $\theta_1 \sim \theta_2$ iff $\exists\eta(\theta_1\eta \equiv \theta_2)$, where $\eta \in Ren$.

And substitutions θ_1 and θ_2 are said to *be in an equivalence class* when $\theta_1 \sim \theta_2$. A set of the equivalence classes of *Subst* is denoted by $Subst_{/\sim}$. A partial ordering on $Subst_{/\sim}$, also denoted by $\preceq$, is naturally induced from a pre-ordering $\preceq$ on *Exp*. We denote the equivalence class of a substitution θ by $\theta_\sim$, or simply by θ. Given $\top$ as the greatest element on $\preceq$ of $Subst_{/\sim}$, $Subst_{/\sim}$ can be naturally extended to $Subst^{\top}_{/\sim}$. Then $(Subst^{\top}_{/\sim}, \preceq)$ forms a complete lattice.

Definition 2.4 (Most General Unifier)
A *most general unifier (mgu)* θ of expressions E_1, E_2, denoted by $mgu(E_1, E_2)$, iff $E_1\theta \equiv E_2\theta$ and $E_1\theta' \equiv E_2\theta' \supset \theta \preceq \theta'$ for all θ'. Let U be a set of equations $\{s_1{=}t_1, \ldots, s_n{=}t_n\}$. Then $mgu([s_1, \ldots, s_n], [t_1, \ldots, t_n])$ is also denoted by $mgu(U)$. A substitution θ can also be represented by a set of equations, denoted by $Eq(\theta)$, such that $Eq(\theta) = \{X{=}t \mid (X \leftarrow t) \in \theta\}$.

Definition 2.5 (Directed)
Let θ_1, θ_2 be substitutions. Then θ_1 and θ_2 are said to *be directed*, denoted by $\theta_1 \bowtie \theta_2$, iff $var(\theta_1) \cap var(\theta_2) = dom(\theta_1) \cap dom(\theta_2)$.

Example 2.1 Consider two substitutions $\theta_1 = \{X \leftarrow U, Y \leftarrow U\}$, $\theta_2 = \{X \leftarrow Y\}$ in an equivalence class and a substitution $\sigma = \{X \leftarrow f(V), Y \leftarrow f(a)\}$. Then θ_1 and σ are directed, but θ_2 and σ are not directed.

As $(Subst^{\top}_{/\sim}, \preceq)$ forms a complete lattice, every subset of $Subst^{\top}_{/\sim}$ has the *lub (least upper bound)* and the *glb (greatest lower bound)* w.r.t. $\preceq$. Several algorithms for computing the *lub* and the *glb* have already been presented [Lassez *et al.* 1987][Palamidessi 1990]. In [Palamidessi 1990], two operations: $Subst_{/\sim} \times Subst_{/\sim} \rightarrow Subst_{/\sim}$ are provided, which are called a *parallel composition*$\uparrow$ and a *parallel factorization*$\downarrow$. This has shown $\theta_1 \uparrow \theta_2 = lub(\theta_1, \theta_2)$ and $\theta_1 \downarrow \theta_2 = glb(\theta_1, \theta_2)$.

We now review the two operations in [Palamidessi 1990] briefly. Let θ_1, θ_2 be (equivalence classes) of idempotent substitutions. $\theta_1 \uparrow \theta_2$ is defined $mgu(Eq(\theta_1) \cup Eq(\theta_2))$. And $\theta_1 \downarrow \theta_2$ is defined by using the *factorization algorithm* which repeatedly replaces the different symbol at the same position in the bindings by a variable and finally generates (an equivalence class of) a substitution η as the $glb(\theta_1, \theta_2)$ with two addenda σ_1, σ_2. Then, the following property is also shown between these substitutions: $\eta\sigma_1 = \theta_1$ and $\eta\sigma_2 = \theta_2$, where σ_1, σ_2 are called *side substitutions*. Here we call σ_1 (or σ_2) a *most general difference (mgd) of* θ_2 (or θ_1) *from* θ_1 (or θ_2), and denote it by $mgd(\theta_1, \theta_2)$ (or $mgd(\theta_2, \theta_1)$.)

Definition 2.6 (Compatibility and Complement)
θ_1 and θ_2 are said to *be compatible*, denoted by $\theta_1 \approx \theta_2$, iff $lub(\theta_1, \theta_2) \neq \top$. And they are said to *be incompatible*, denoted by $\theta_1 \not\approx \theta_2$, iff $lub(\theta_1, \theta_2) = \top$. A *complement* of a substitution $\theta_{/\sim} \in Subst_{/\sim}$, denoted by $\overline{\theta}_\sim$ or $\overline{\theta}$, is a set of all (equivalence classes of) substitutions incompatible with θ, which is defined by $\{\theta'_\sim \in Subst_{/\sim} \mid \theta_\sim \not\approx \theta'_\sim\}$

Example 2.2 Consider substitutions $\theta_1 = \{X \leftarrow f(Z), Y \leftarrow f(Z)\}$ and $\theta_2 = \{X \leftarrow f(a), Y \leftarrow f(b)\}$. Since the parallel composition $\theta_1 \uparrow \theta_2$ is $\top$, they are *incompatible*. And the parallel factorization $\theta_1 \downarrow \theta_2$ is the substitution $\{X \leftarrow f(U), Y \leftarrow f(V)\}$, and the most general difference $mgd(\theta_1, \theta_2)$ is $\{U \leftarrow V\}$ and $mgd(\theta_2, \theta_1)$ is $\{U \leftarrow a, V \leftarrow b\}$.

3 Flat Guarded Horn Clauses

Now, we briefly recall a concurrent programming language *Flat Guarded Horn Clauses (FGHC)*, and then define the operational semantics of Flat GHC in terms of a transition system [Ueda 1990b].

3.1 Syntax of FGHC

An *FGHC program* is a set of *flat guarded clauses*. A flat guarded clause (simply, *clause*) is of the form:

$$p(t_1, \ldots, t_k) \text{ :- } G_1, \ldots, G_m \mid B_1, \ldots, B_n. \qquad (k, m, n \geq 0),$$

where p is a k-ary predicate symbol, $t_1, \ldots, t_k$ are terms, and $G_1, \ldots, G_m$, $B_1, \ldots, B_n$ are atoms. The atom $p(t_1, \ldots, t_k)$ is called a *head*, the head and $G_1, \ldots, G_m$ are called a *guard* and $B_1, \ldots, B_n$ is called a *body*. One binary predicate "=" for unifying two terms is predefined by the language, a goal of which is called a *unification goal*. Each guard goal G_i must be a unification goal.

3.2 Operational Semantics of Flat GHC

In [Ueda 1990b], Ueda has defined the operational semantics of FGHC in the style of Plotkin. Here we present it by following his definition.

Definition 3.1 (Transition System of FGHC)
A *transition system of* an FGHC program P is defined by using a *configuration* and a *transition relation*. A configuration is a pair of the form $\langle B, E \rangle$ where B is a multiset of goals and E is a binding environment of B. A binding environment E is a multiset of equations C with a set of variables V such that $var(B) \cup var(C) \subseteq V$, denoted by $C : V$.

A transition relation under P, denoted by $Trans_P$, is the smallest set of binary relations on configurations, denoted by $\cdot \rightarrow \cdot$, such that:

$$\frac{\langle B_1,\, C_1 : V_1\rangle \rightarrow \langle B_1',\, C_1' : V_1'\rangle}{\langle B_1 \cup B_2,\, C_1 : V_1\rangle \rightarrow \langle B_1' \cup B_2,\, C_1' : V_1'\rangle} \quad (1)$$

$$\frac{\langle \{A=H\} \cup G,\, C : V \cup var((H,G))\rangle \xrightarrow{*} \langle \emptyset,\, C \cup C_g : V'\rangle}{\langle \{A\},\, C : V\rangle \rightarrow \langle B,\, C \cup C_g : V' \cup var(B)\rangle}$$
$$\text{if } \exists c \in P\, \exists \eta \in Ren((H :\!- G \mid B) \equiv c\eta \wedge V \cap var(c\eta) \equiv \emptyset)$$
$$\text{and } \models \forall.(C \supset \exists(var(C_g) \setminus var(A)).C_g) \quad (2)$$

$$\frac{}{\langle s=t,\, C : V\rangle \rightarrow \langle \emptyset,\, C \cup \{s=t\} : V\rangle} \quad (3)$$

When $c_1 \rightarrow c_2 \in Trans_P$, $c_1 \rightarrow c_2$ is said to *be in the transition system of* P or c_1 is said to *be reduced to* c_2 *under a program* P. Then, a computation of a program P with an initial goal B is represented by a (possibly infinite) sequence of transitions in $Trans_P$; $c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow c_i \rightarrow \ldots$ such that c_1 is $\langle B, \emptyset : var(B)\rangle$. Each configuration $c_i (i \geq 1)$ is called a *possible configuration* from B.

We may use alternative denotations $\xrightarrow{\cup}$, $\xrightarrow{c}$ and $\xrightarrow{s=t}$ corresponding to transition rules (1),(2) and (3) respectively if it is necessary to identify them. The reflexive and transitive closure of $\rightarrow$, by applying $\xrightarrow{c}$ (or $\xrightarrow{s=t}$) once only, is denoted by $\stackrel{c}{\Rightarrow}$ (or $\stackrel{s=t}{\Rightarrow}$, respectively,) or by $\Rightarrow$ simply.

4 Fixpoint Approach to the Semantics

In this section, we show that a computation of a multiset of goals G is modeled as interleaving computations of each goal in G and the model can be computed as the fixpoint of the semantic function defined here.

4.1 Atom Reaction

We are interested in reactive behaviors between a given initial goal and a (possibly altered) environment which may be implemented by other goals, rather than the fixed behavior and the final result induced from the initial goal and the initial environment. In such a case the environment (i.e., the other goals) may also be (monotonically) altered by reacting against the initial goal and/or its subgoals during the computation of the initial goal.

Here possible reactive behaviors of a initial goal corresponding to various environments are denotationally modeled by using sequences of substitutions.

Definition 4.1 (Unit Reaction)
A *unit reaction* is a substitution with an annotation '+' or '−', denoted by θ^+ or θ^- where θ is a substitution. θ^+ is called an *input unit reaction* and θ^- is called an *output unit reaction*. We may denote a unit reaction without an annotation when we do not need to distinguish *input* or *output* from each other. A substitution θ obtained from a unit reaction $\delta \equiv \theta^a$ by removing an annotation a is denoted by $|\delta|$. A set of all input unit reactions $\{\theta^+ \mid \theta \in Subst\}$ is denoted by $Ureact^+$, a set of all output unit reactions $\{\theta^- \mid \theta \in Subst\}$ is denoted by $Ureact^-$, and a set of all unit reaction $Ureact^+ \cup Ureact^-$ is denoted by $Ureact$.

Next we introduce special symbols, called *termination symbols*, which represent special states in reactive behaviors.

Definition 4.2 (Terminal Symbol)
A *termination symbol* is $\bot_{\mathrm{suc}}, \bot_{\mathrm{rf}}, \bot_{\mathrm{uf}}$ or $\bot_{\mathrm{dl}}$, (or simply by $\bot$), which represent *finite success*, *reduction failure*, *unification failure* and *deadlock* respectively. Then $Ureact^\bot$ denotes $Ureact \cup \{\bot_{\mathrm{suc}}, \bot_{\mathrm{rf}}, \bot_{\mathrm{uf}}, \bot_{\mathrm{dl}}\}$. $|\bot_{\mathrm{suc}}| = |\bot_{\mathrm{rf}}| = |\bot_{\mathrm{dl}}| = \emptyset$, and $|\bot_{\mathrm{uf}}| = \top$.

Now we define various operations on unit reactions by extending operations on the substitutions defined above.

Definition 4.3 (Operations on Unit Reactions)
Let σ be a substitution, δ be a unit reaction and a be an annotation of δ. Then *domain* and *range* of unit reaction are defined by $dom(\delta) \equiv dom(|\delta|)$ and $ran(\delta) \equiv ran(|\delta|)$. *mgu* and *mgd* of a substitution σ and a unit reaction δ are defined by $mgu(\sigma, \delta) \equiv mgu(\delta, \sigma) \equiv mgu(|\delta|, \sigma)^a$, $mgd(\delta, \sigma) \equiv mgd(|\delta|, \sigma)^a$ and $mgd(\sigma, \delta) \equiv mgd(\sigma, |\delta|)$. $\sigma \bowtie \delta$ and $\delta \bowtie \sigma$ iff $\sigma \bowtie |\delta|$ or $|\delta| \bowtie \sigma$.

For a unit reaction δ_i and a sequence of unit reactions Δ, δ_i is said to *be in* Δ iff $\exists \delta_i (1 \leq i \leq n)(\Delta = \delta_1 \delta_2 \ldots \delta_n)$, and denoted by $\delta_i \in \Delta$. An empty sequence of unit reactions is denoted by $\Box$.

Definition 4.4 (Reaction Sequence)
A *reaction sequence* is an empty sequence $\Box$, a sequence of one unit reaction δ, or a sequence of more than two unit reactions Δ such that $\forall \delta_i, \delta_j \in \Delta\, (1 \leq i < j \leq n \wedge dom(\delta_i) \cap dom(\delta_j) \equiv \emptyset \wedge dom(\delta_i) \cap ran(\delta_j) \equiv \emptyset)$. A set of all reaction sequences is denoted by $Rseq$.

A *domain* of $\Delta \in Rseq$ is a set of variables such that $\{V \mid \exists \delta \in \Delta(V \in dom(\delta)) \wedge \forall \delta' \in \Delta(V \notin ran(\delta'))\}$. $var(\Delta)$ also denotes the set of variables $\bigcup_{\delta \in \Delta} var(\delta)$. A substitution σ and a reaction sequence Δ are said to *be directed*, denoted by $\sigma \bowtie \Delta$, iff $var(\sigma) \cap var(\Delta) = dom(\sigma) \cap dom(\Delta)$. Reaction sequences Δ_1 and Δ_2 are said to *be directed*, also denoted by $\Delta_1 \bowtie \Delta_2$, iff $var(\Delta_1) \cap var(\Delta_2) = dom(\Delta_1) \cap dom(\Delta_2)$.

When $\Delta = \delta_1 \ldots \delta_n \in Rseq$ and $\delta \in Ureact$, a concatenation of Δ and δ is denoted by $\Delta \cdot \delta$ or $\delta \cdot \Delta$, defined by $\Delta \cdot \delta = \delta_1 \ldots \delta_n \delta$ or $\delta \cdot \Delta = \delta \delta_1 \ldots \delta_n$. A sequence of unit reaction $\Delta \cdot \delta$ such that $\Delta \in Rseq$ and $\delta \in Ureact^\bot$ is also a reaction sequence. A set of all such reaction sequences is denoted by $Rseq^\bot$.

Definition 4.7 (Atom Reaction)
An *atom reaction* is a pair of an atom $A \in Atom$ and a reaction sequence $\Delta \in Rseq^\bot$, denoted by (A, Δ), such that $dom(\Delta) \subseteq var(A)$. Here a set of all atom

reactions is denoted by *Areact*, i.e., $Areact = \{(A, \Delta) \mid A \in Atom \wedge \Delta \in Rseq^{\perp}\}$.

A substitution θ and an atom reaction (A, Δ) are also said to *be directed*, denoted by $\theta \bowtie (A, \Delta)$, iff $\theta \bowtie \Delta$. Atom reactions (A_1, Δ_1) and (A_2, Δ_2) are said to *be directed*, denoted by $(A_1, \Delta_1) \bowtie (A_2, \Delta_2)$ iff $\Delta_1 \bowtie \Delta_2$.

An equivalence class of E, i.e., $E_\sim$, may be represented by E, as mentioned in Section 2. When we say "E_1 and E_2 such that $E_1 \bowtie E_2$" where E_1, E_2 are substitutions, reaction sequences or atom reactions, we mean that each E_1 or E_2 is restricted to a subset of $E_{1\sim}$ or $E_{2\sim}$ such that $E_1 \in E_{1\sim}$, $E_2 \in E_{2\sim}$ and $E_1 \bowtie E_2$.

Let $(A, \delta_1 \ldots \delta_n), (A, \delta'_1 \ldots \delta'_n)$ be atom reactions that are directed. Then $(A, \delta_1 \ldots \delta_n)$ is said to be *more general than* $(A, \delta'_1 \ldots \delta'_n)$ when the following condition hold:

(1) if $\delta_n \in \perp$ or $\delta'_n \in \perp$, then $\delta_n = \delta'_n$, and
(2) for all i $(1 \le i \le n)$,
  (a) $\delta_i \in Ureact^+$ iff $\delta'_i \in Ureact^+$,
  (b) $\delta_i \in Ureact^-$ iff $\delta'_i \in Ureact^-$,
  (c) if $\delta_i, \delta'_i \in Ureact^+$, then $\Pi\delta_i \preceq \Pi\delta'_i$, and
  (d) if $\delta_i, \delta'_i \in Ureact^-$, then $\Pi\delta'_i \sim lub(\Pi\delta'_{i-1}, \Pi\delta_i)$,

where $\Pi\delta_i$ is a composition of substitutions $|\delta_1| \ldots |\delta_i|$.

Here we want to explain intuitively what is the notion that (A, Δ) is more general than (A, Δ'). (A, Δ') represents a reactive behavior such that a goal A gets more instantiated bindings from, and posts not more instantiated bindings to the environment of the goal A, than the reactive behavior represented by (A, Δ).

Definition 4.6 (Atom on a Program)
Given a FGHC program P, an *atom on a program* P is an atom A such that the predicate symbol of A appears in P (not necessarily at head parts.) A set of all atoms on P is denoted by $Atom_P$, and a set of all atom reactions (A, Δ) such that $A \in Atom_P$ and $\Delta \in Rseq$ is denoted by $Areact_P$.

Now we define the relation between atom reactions and operational behaviors more formally.

Definition 4.7 (Correct Atom Reaction)
When a program P and an atom $G_0 \in Atom_P$ are given, an atom reaction $(G_0, \delta_1 \ldots \delta_n)$ is called a *correct atom reaction w.r.t.* a program P, when the following conditions hold, where $B_0 = \{G_0\}$, $C_0 = \emptyset$ and $V_0 = var(G_0)$, and, for all $i (1 \le i \le n)$, let C_{δ_i} be a set of equations such that $mgu(C_{\delta_i}) \sim |\delta_i|$.

(1) If $n = 0$, i.e., $\delta_1\delta_2 \ldots \delta_n = \Box$, then $(G_0, \delta_1 \ldots \delta_n)$ is always correct,
(2) if $\delta_i \in Ureact^+$, there exists a transition $\langle B_{i-1}, C_{i-1} \cup C_{\delta_i} : V_{i-1}\rangle \stackrel{\varsigma}{\Rightarrow} \langle B_i, C_i : V_{i-1} \cup var(c\eta)\rangle$ such that $var(C_{\delta_i}) \cap V_i \subseteq var(G_0\Theta)$ and $B_i = (B_{i-1} \setminus \{H\}) \cup B$, where $(H \text{ :- } G \mid B) \equiv c\eta$, $\eta \in Ren$ and $\Theta = mgu(C_{i-1})$,
(3) if $\delta_i \in Ureact^-$, there exists a transition $\langle B_{i-1}, C_{i-1} : V_{i-1}\rangle \stackrel{t=s}{\Rightarrow} \langle B_i, C_i : V_{i-1} \cup var(s=t)\rangle$ such that $C_i = C_{i-1} \cup C_{\delta_i}$ and $B_i = B_{i-1} \setminus \{t=s\}$, or
(4) if $\delta_n \in \{\perp\}$, at least one of the conditions (1)–(3) holds for all i $(1 \le i \le n-1)$, and
  (a) if δ_n is $\perp_{\mathrm{suc}}$, then B_{n-1} is $\emptyset$,
  (b) if δ_n is $\perp_{\mathrm{rf}}$, then there exists $A \in B_{n-1}$ such that $mgu(C_{n-1}) \not\approx mgu(\{A=H\} \cup G)$ for all clauses such that $(H \text{ :- } G \mid B) \equiv c\eta, \eta \in Ren$ and $c \in P$,
  (c) if δ_n is $\perp_{\mathrm{uf}}$, then there exists $(t=s) \in B_{n-1}$ such that $mgu(C_{n-1}) \not\approx mgu(t, s))$, or
  (d) if δ_n is $\perp_{\mathrm{dl}}$, then there exists the same transition as in case (2) unless the condition $var(C_{\delta_{n-1}}) \cap V_i \subseteq var(G_0\Theta)$ exists, where $\Theta = mgu(C_{i-1})$.

In the following we define the most important atom reaction in correct atom reactions.

Definition 4.8 (Most General Correct Atom Reaction)
Let (A, Δ) be a correct atom reaction w.r.t. a program P. Then, (A, Δ) is called a *most general correct atom reaction w.r.t.* a program P, denoted by $(A, \Delta) \rightsquigarrow_P$, when (A, Δ) is more general than any other correct atom reactions (A, Δ') w.r.t. P.

Example 4.1 Suppose that $(A, \delta_1\delta_2\delta_3) \rightsquigarrow_P$ such that $\delta_1, \delta_2 \in Ureact^+$ and $\delta_3 \in Ureact^-$. Intuitively we can explain the notion of a *correct atom reaction* by considering a chain of the following transitions:

$$\langle G, \emptyset\rangle$$
$$\langle G, C_{\delta_1}\rangle \stackrel{\varsigma}{\Rightarrow} \langle B_1, C_1\rangle$$
$$\langle B_1, C_1 \cup C_{\delta_2}\rangle \stackrel{\varsigma}{\Rightarrow} \langle B_2, C_2\rangle$$
$$\langle B_2, C_2\rangle \stackrel{s=t}{\Rightarrow} \langle B_3, C_2 \cup C_{\delta_3}\rangle$$

where $G = \{A\}$ and $mgu(C_{\delta_i}) \sim |\delta_i|$ $(1 \le i \le 3)$.

Example 4.2 Let P be a program
$\{p(A,B,C) \text{:-} A=f(D), C=g(a,E) \mid B=f(a)\}$,
Δ_1 be $\{X \leftarrow f(U), Z \leftarrow g(a,V)\}^+\{Y \leftarrow f(a)\}^-$,
Δ_2 be $\{X \leftarrow f(U), Z \leftarrow g(a,V), Y \leftarrow f(a)\}^+$, and
Δ_3 be $\{X \leftarrow f(U)\}^+\{Z \leftarrow g(a,V), Y \leftarrow f(a)\}^-$.
Then the following two atom reactions $(p(X,Y,Z), \Delta_1)$ and $(p(X,Y,Z), \Delta_2)$ are correct atom reaction w.r.t. P. $(p(X,Y,Z), \Delta_1)$ is a most general correct reaction, i.e., $(p(X,Y,Z), \Delta_1) \rightsquigarrow_P$. But $(p(X,Y,Z), \Delta_3)$ is *not correct* because the configuration $\langle\{p(X,Y,Z)\}, \{X = f(U)\}\rangle$ can not be reduced to any configuration under the program P.

4.2 Fixpoint Semantics

In this section, we present the semantic function after defining some operations on reaction sequences and the semantic domain. Next we show the least fixpoint of the function gives the semantics of the programs in the same way as used in an ordinary fixpoint semantics theory.

Firstly, we define an application of a substitution to an atom reaction when they are directed. Let $\theta \in Subst_{/\sim}$ and $\Delta = \delta_1\delta_2 \ldots \delta_n \in Rseq^{\perp}$ such that $\theta \bowtie \Delta$. Then *an application of* a substitution θ *to* a reaction sequence Δ, denoted by $\Delta\theta$, is a sequence of unit reactions $\delta'_1\delta'_2 \ldots \delta'_n$ such that $\delta'_i = mgd(\sigma_i, \sigma_{i-1})$ for all i $(1 \le i \le n)$, where $\sigma_0 = \theta$ and $\sigma_i = lub(\sigma_{i-1}, \delta_i)$.

Example 4.3 Let θ be a substitution
$\{X \leftarrow M, Y \leftarrow M, Z \leftarrow g(N,b)\}$,
and $\delta_1\delta_2$ be the reactive sequence Δ_1 same as in Example 4.2. Then, θ and Δ_1 are directed because
$var(\theta) \cap var(\Delta_1) = dom(\theta) \cap dom(\Delta_1) = \{X, Y, Z\}$.
Let σ_1 be $lub(\theta, \delta_1)$, i.e.,
$\{ X \leftarrow f(U), Y \leftarrow f(U), Z \leftarrow g(a,b), V \leftarrow b,$
$M \leftarrow f(U), N \leftarrow a \}$,
and let σ_2 be $lub(\sigma_1, \delta_2)$, i.e.,
$\{ X \leftarrow f(a), Y \leftarrow f(a), Z \leftarrow g(a,b), V \leftarrow b, U \leftarrow a$
$M \leftarrow f(a), N \leftarrow a \}$,
Therefore, the application of θ to Δ_1 is
$\{M \leftarrow f(U), N \leftarrow a\}^+\{U \leftarrow a\}^-(= \delta_1'\delta_2')$, where $\delta_1' = mgd(\sigma_1, \theta)$ and $\delta_2' = mgd(\sigma_2, \sigma_1)$.

If $\delta_i \in Ureact^+$ and $\delta_i \not\approx \sigma_{i-1}$ for some i, then such an application is not defined, that is, we can ignore the result and remove it from our system. Because, although such a reduction can *not* be done by *the* clause (corresponding to the input unit reaction δ_i), it may be done by *another alternative* clause. That is, it is not necessary that a *reduction failure* is immediately induced by this application. On the other hand, in the case that $\delta_i \in Ureact^-$ and $\delta_i \not\approx \sigma_{i-1}$ for some i, $\Delta\theta$ is $\delta_1'\delta_2' \ldots \delta_{i-1}\bot_{\rm uf}$. This is because such an application induces a *unification failure* immediately.

Definition 4.9 (Application to Atom Reaction)
Let (A, Δ) be an atom reaction. Then *an application of* a substitution θ *to* an atom reaction (A, Δ) is $(A\theta, \Delta\theta)$, which is also an atom reaction.

Example 4.4 Let P be the same program and $(p(X,Y,Z), \Delta_1)$ be the same atom reaction as in Example 4.2, and let θ be the same substitution as in Example 4.3. Then, the application of θ to $(p(X,Y,Z), \Delta_1)$, i.e., $(p(X,Y,Z)\theta, \Delta_1\theta)$, is
$(p(M,M,g(N,b)), \{M \leftarrow f(U), N \leftarrow a\}^+\{U \leftarrow a\}^-)$.
Now the application of θ to it, $(p(X,Y,Z)\theta, \Delta_1\theta)$, intuitively represents a reactive behavior of a goal $p(M,M,g(N,b))$ under the program P. In fact, the atom reactions $(p(X,Y,Z), \Delta_1)$ and $(p(X,Y,Z)\theta, \Delta_1\theta)$ is both correct w.r.t. P.

Next we define possible interleavings of reaction sequences.

Definition 4.10 (Interleaving)
Possible interleavings of a set of reaction sequences $\{\Delta_1, \ldots, \Delta_n\}$ *on* a set of variables V, denoted by $int(\Delta_1, \ldots, \Delta_n)_{|V}$, is a set of all reaction sequences $\delta \cdot \Delta$ defined inductively as follows, where Δ_i is $\delta_i \cdot \Delta_i'$ for each i $(1 \le i \le n)$ such that Δ_i is not $\Box$:

(1) if $\delta_i \in Ureact^+$ and $V \subset dom(\delta_i)$ for all i $(1 \le i \le n)$, then $\Delta = \Box$ and $\delta = \bot_{\rm dl}$, or
(2) otherwise, for some i $(1 \le i \le n)$,
  (a) if $\Delta_i = \Box$,
    then $\delta \cdot \Delta = int(\Delta_1, \ldots, \Delta_{i-1}, \Delta_{i+1}, \ldots, \Delta_n)_{|V}$,
  (b) if $\delta_i \in \{\bot_{\rm rf}, \bot_{\rm uf}, \bot_{\rm dl}\}$, then $\Delta = \Box$ and $\delta = \delta_i$,
  (c) if $\Delta_j = \Delta_j'' \cdot \bot_{\rm suc}$ for all j $(1 \le j \le n)$, then
    $\Delta = int(\Delta_1''\delta_i, \ldots, \Delta_i', \ldots, \Delta_n''\delta_i)_{|var(V|\delta_i|)} \cdot \bot_{\rm suc}$
    and $\delta = \delta_i$, or
  (d) otherwise,
    $\Delta = int(\Delta_1\delta_i, \ldots, \Delta_i', \ldots, \Delta_n\delta_i)_{|var((V)|\delta_i|)}$ and $\delta = \delta_i$.

Definition 4.11 (Semantic Function)
Given a program P, we denote a power set of $Areact_P$ by Den_P, and let it be a *domain of* the following semantic function. Given a program P and a goal G_0, we define a semantic function $T_{P,G_0} : Den_P \rightarrow Den_P$ as follows:

$$
\begin{array}{l}
T_{P,G_0}(I) \ = \\
\{(G_0, \Box)\} \cup \\
\{(s{=}t, \theta^-) \mid (s{=}t, \Box) \in I \wedge \theta = mgu(s,t)\} \cup \\
\{(B_i\theta_g, \Box) \mid \exists(A, \Box) \in I \wedge \exists(H \text{ :- } G \mid B) \in P\,(B_i \in B)\} \cup \\
\{(A, \theta_g^+\Delta) \mid (A, \Box) \in I \wedge \\
\qquad \exists(H \text{ :- } G \mid B) \in P\, \forall B_i, B_j \in B \\
\qquad \exists(B_i\theta_g, \Delta_i), (B_j\theta_g, \Delta_j) \in I \\
\qquad\quad ((B_i\theta_g, \Delta_i) \bowtie (B_j\theta_g, \Delta_j) \wedge \\
\qquad\quad \Delta \in int(\Delta_1, \Delta_2, \ldots, \Delta_n)_{|var(A)})\} \cup \\
\{(A, \theta^+\bot_{\rm rf}) \mid (A, \Box) \in I \wedge \forall(H \text{ :- } G \mid B) \in P\,(\theta \in \overline{\theta}_g)\}
\end{array}
$$

where $\theta_g = mgu(\{A{=}H\} \cup G)$ and $\overline{\theta}_g = \{\theta \mid \theta \not\approx \theta_g\}$

The set Den_P forms a complete lattice under the ordering of set inclusion $\subseteq$ with a bottom element $\emptyset$ and a top element $Areact_P$.

The $T_{P,G_0}(I)$ is recursively defined by using I as the union of four sets of atom reactions each of which represents the following situation:

(1) when a unification goal $s{=}t$ is called, the binding $mgu(s,t)$ is posted to the environment,
(2) when a goal A exists, each goal $B_i\theta_g$ is generated as a sub-goal of A and may invoke the new process,
(3) and a goal A affects the environment as θ_g^+ followed by a sequence of reactive behaviors represented by Δ which is obtained from interleaving reactive behaviors generated by all sub-goals of A, that is, A may perform the computation represented by Δ after A gets the binding θ_g, or
(4) when A meets with the binding θ incompatible with all bindings to solve the guard $\{A{=}H\} \cup G$ for all clauses, A will suspend. This situation is called a *reduction failure.*

Lemma 4.1 *Let P be an FGHC program and G be a goal. The function $T_{P,G}$ is continuous, i.e., $T_{P,G}(lub(X)) = lub(T_{P,G}(X))$ for any directed subset X of Den_P*

Proof: It is proved in a similar way to the proof of continuity of the semantic function of a standard logic program. (See pp.37-38 [Lloyd 1987].) ∎

From Lemma 4.1, $T_{P,G}$ has the least fixpoint, $lfp(T_{P,G})$, and $lfp(T_{P,G}) = glb\{X | T_{P,G}(X) \subseteq X\}$. Furthermore, $lfp(T_{P,G}) = T_{P,G}\uparrow\omega$.

Definition 4.12 (Topdown Semantics)
Let P be a FGHC program and G be a goal. Then $lfp(T_{P,G})$ is called a *topdown semantics* of P with G, and denoted by $[\![P]\!]_G$.

5 Relation between Operational Semantics and Fixpoint Semantics

In this section, we show that the topdown semantics defined in Section 4 is closely related to the operational semantics of FGHC introduced in Section 3.

Theorem 5.1 (Soundness) *Let P be an FGHC program and G_0 be a goal. If $(G_0, \Delta) \rightsquigarrow_P$, then $(G_0, \Delta) \in [\![P]\!]_{G_0}$.*

Proof (A Sketch of the Proof):
Let k be a length of Δ, denoted by $|\Delta|$. The proof is by induction on the length k.

If $k = 0$, then the theorem is trivial since $(G, \Box)$ is always correct.

Otherwise, i.e., $k > 0$, suppose $\Delta = \Delta' \cdot \delta$ such that $|\Delta'| = \geq 0$.

If G_0 is a unification goal, then the theorem is trivial.

Otherwise, G_0 is a non-unification goal. Now, since $(G_0, \Delta) \rightsquigarrow_P$ hold, $(G_0, \Delta') \rightsquigarrow_P$ holds. By the induction hypothesis, since $|\Delta'| < k$ and $(G_0, \Delta') \rightsquigarrow_P$, $(G_0, \Delta') \in [\![P]\!]_{G_0}$.

Hence, from the definition of T_{P,G_0}, $\exists \theta_g^+ (\Delta' = \theta_g^+ \cdot \Delta'')$ such that $(H \text{ :- } G \mid B) \in P$ and $\theta_g = mgu(\{A = H\} \cup G)$ and $\forall B_i \in B \exists (B_i\theta_g, \Delta_i) \in [\![P]\!]_{G_0}$ and $\Delta'' \in int(\Delta_1, \ldots, \Delta_n)_{|var(G_0)}$. Now we have $(G_0, \theta_g^+ \cdot \Delta'') \rightsquigarrow_P$ and $\Delta'' \in int(\Delta_1, \ldots, \Delta_n)_{|var(G_0)}$. Then we can get Δ_i such that $(B_i\theta_g, \Delta_i) \rightsquigarrow_P$ by selecting a unit reaction from the only i-th argument (i.e., Δ_i) in the definition of int.

Suppose that the last transition of $(G_0, \Delta' \cdot \delta) \rightsquigarrow_P$ is a transition on a sub-goal of $B_i\theta_g$. Then $(B_i\theta_g, \Delta_i\delta) \rightsquigarrow_P$.

Since $k > |\Delta'| > |\Delta''| \geq |\Delta_i|$, $k > |\Delta'| \geq |\Delta_i\delta|$.

By induction hypothesis again, since $(B_i\theta_g, \Delta_i\delta) \rightsquigarrow_P$, $(B_i\theta_g, \Delta_i\delta) \in [\![P]\!]_{G_0}$.

Therefore, from the definition of T_{P,G_0}, since $\Delta'\delta = int(\Delta_1, \ldots, \Delta_i, \ldots, \Delta_n)$, $(G_0, \Delta'\delta) \in [\![P]\!]_{G_0}$. ∎

In Theorem 5.1 we show that any most general correct atom reaction (G_0, Δ) w.r.t. a program P is in the topdown semantics $[\![P]\!]_{G_0}$. In general it is necessary to prove the only-if part of the theorem (usually called *Completeness Theorem*), and we think this is possible by introducing a kind of downward closure of (A, Δ) by using the *'more general than' relation* in Section 4.1, as *subsumption relation* in [Falaschi *et al.* 1990]. This, however, is beyond the scope of this paper. Because Theorem 5.1 is sufficient to guarantee the correctness of the framework of abstract interpretation based on the topdown semantics since we want to use this semantics as a collecting semantics.

6 General Framework for Abstract Interpretation

In this section we briefly review a general framework of abstract interpretation for programs whose semantics can be defined from a fixpoint approach, and some conditions to guarantee that the abstract interpretation is 'safe' for the semantics.

When a standard semantics is given by the least fixpoint of some semantic function, an abstract semantics is given by another semantic function obtained by directly abstracting the concrete semantic function such that the *safe* relation exists between their two semantics.

6.1 Concrete Fixpoint Semantics

Suppose that the meaning of a program P is given by the least fixpoint of a *(concrete) semantic function* T_P, denoted by $lfp(T_P)$, where $T_P : Den \rightarrow Den$ is a continuous function and Den is a powerset of D, called a *concrete domain*, such that each element of D expresses a concrete computation state of the program. For example, in an ordinary logic program, is an Herbrand Base. And Den forms a complete lattice with regard to the set inclusion ordering $\subseteq$ on Den. Then, the least fixpoint of T_P exists and we can get it by $lfp(T_P) = T_P\uparrow\omega$.

Definition 6.1 (Concrete Semantics)
$[\![P]\!] = lfp(T_P)$ is called the *least fixpoint semantics* of a program P. Especially, we call it the *concrete semantics* of a program P since the semantics is obtained from the concrete semantic function T_P

6.2 Abstract Fixpoint Semantics

We define an abstract fixpoint semantics by abstracting the concrete domain and the concrete semantic function introduced in 6.1.

Definition 6.2 (Abstract Domain)
Given a concrete domain D, an *abstract domain* $\underline{D}$ is a finite set of denotations satisfying the following conditions:

(1) every element of $\underline{D}$ represents a subset of D,
(2) $\underline{D}$ forms a complete lattice with respect to an order relation $\sqsubseteq$ defined on $\underline{D}$, and
(3) there exist two monotonic mappings, that is, *abstraction* $\alpha : D \rightarrow \underline{D}$ and *concretization* $\gamma : \underline{D} \rightarrow D$ defined as follows: $\forall \underline{d} \in \underline{D} \, (\underline{d} = \alpha(\gamma(\underline{d}))) \wedge \forall d \in D \, (d \subseteq \gamma(\alpha(d)))$

In order to define the abstract semantics of a program P, we should define (or design) a monotonic and continuous mapping of a program P; $\underline{T_P} : \underline{Den} \rightarrow \underline{Den}$, called the *abstract semantic function*, as well as the abstract domain $\underline{D}$, corresponding to the concrete domain D and the concrete semantic function T_P of P. Then we have to define the abstract versions of various operations, e.g., a composition or an application of substitutions, used in the definition of T_P.

Definition 6.3 (Abstract Semantics)
Then the least fixpoint semantics $[\![P]\!] = lfp(\underline{T}_P)$, obtained from the abstract semantic function $\underline{T}_P$, is called the *abstract semantics* of a program P.

Now we claim the termination property with respect to the abstract fixpoint semantics.

Lemma 6.1 *There exists the least fixpoint $lfp(\underline{T}_P)$ of $\underline{T}_P$ such that $lfp(\underline{T}_P) = \underline{T}_P \uparrow k$ for some finite k*

Lastly, we attach the following acceptable relation between the abstract semantics and the concrete semantics:

Definition 6.5 (Safeness Condition)
A *safeness condition for the abstract semantics* is as follows: $[\![P]\!] \subseteq \gamma([\![P]\!])$.

Lemma 6.2 *If $T_P(\gamma(\underline{d})) \subseteq \gamma(\underline{T}_P(\underline{d}))$ for all $\underline{d} \in \underline{D}$, then the abstract semantics is safe, i.e., a safeness condition holds, where $T_P(\gamma(\underline{d})) = \{T_P(d) \mid d \in \gamma(\underline{d})\}$.*

7 Applications for Analysis of FGHC Programs

In this section we show some examples of analyzing FGHC programs by using abstract interpretation based on the topdown semantics in Section 4, which is an instance of the general framework in Section 6

7.1 Moded Type Graph

The abstract domain presented here is so similar to the one based on type graphs in [Bruynooghe and Janssens 1988], that most necessary operations on the abstract domain will be well-defined similarly to [Bruynooghe and Janssens 1988][Janssens and Bruynooghe 1989].

Here we introduce a moded type graph, and show briefly that a reaction sequence and an atom reaction can be abstracted by a moded type graph.

Definition 7.1 (Moded Type Constructor and Generic Types)
A(n n-ary) *moded type constructor* is a(n n-ary) function symbol $f/n \in Func$ with a mode annotation + (or −), denoted by $f^+_{/n}$ (or $f^-_{/n}$) or simply f^+ (or f^-), which represents a(n n-ary) function symbol f appearing in input (or output) unit reactions (respectively). Four *generic (moded) types* are an *any type*, a *variable type*, an *undefined type* and an *empty type*, denoted by $\underline{any}$, $\underline{V}$, _ and $\emptyset$ respectively. An *any type* represents the set of all moded terms, both $\underline{V}$ and _ represent the set of variables, and $\emptyset$ represents the empty set of terms.

Definition 7.2 (Moded Term and Moded Type)
A *moded term* is a term constructed from moded type constructors over a set of variables *Var*. A moded term represents the same term without all mode annotations such that a moded type constructor with + (or −) corresponds to a function symbol appearing in an input (or an output) unit reaction. A *moded type* is a set of moded terms.

Definition 7.3 (Moded Type Graph)
A *moded type graph* is a representation of a moded type, which is a directed graph such that each node is labeled with either a moded type constructor, a generic type, or a special label 'or'.

The relation between a parent node and (possibly no) child nodes in a moded type graph $\mathcal{G}$ is defined as follows:

(1) a node labeled with $f^+_{/n}$ or $f^-_{/n} (n \geq 0)$ has n *ordered* arcs to n nodes, i.e., has n ordered child nodes,
(2) a node labeled with 'or' has n *non-ordered* arcs to n nodes ($n \geq 2$), i.e., has n non-ordered child nodes,
(3) a node labeled with a generic type has no child node,
(4) there exists at least one node, called a *root node*, such that there are paths from the root node to any other nodes in $\mathcal{G}$, and
(5) the number of occurrences of nodes with the same label on each path from the root node of $\mathcal{G}$ is bounded by a constant d, called a *moded type depth.*

Suppose that a node N tries to be newly added as a child node of N_p in a moded type graph $\mathcal{G}$. Then, if the creation of the node N violates the condition (5) in the above definition, that is, if there exist more than d numbers of nodes with the same label as N on the path from the root node to N, then the new node N will not be added to $\mathcal{G}$ as a new child node of N_p but will be shared with the farthest ancestor node of N_p with the same label as N. In such a case, a circular path must be created. (Nodes with the same label aren't shared with each other when their nodes are on different paths from root.) The restriction of (5) is the same as the *depth restriction* in [Janssens and Bruynooghe 1989]. They call a type graph satisfying the depth restriction a *restricted type graph*, and they have presented an algorithm for transforming a non-restricted type graph to restricted one.

A concretization for a moded type graph with a root node N_0, denoted by $\gamma(N_0)$, is defined as follows:

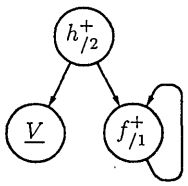
(1) $\gamma(N)$ is *Var* if the label of N is $\underline{V}$ or _,
(2) $\gamma(N)$ is the empty set $\emptyset$ if the label of N is $\emptyset$,
(3) $\gamma(N)$ is $\{f^+(t_1, \ldots, t_n) \mid t_i \in \gamma(N_i) \wedge 0 \leq i \leq n\}$ if the label of N is $f^+_{/n}$ and $N_1, \ldots, N_n$ are child nodes of N,
(4) $\gamma(N)$ is $\{f^-(t_1, \ldots, t_n) \mid t_i \in \gamma(N_i) \wedge 0 \leq i \leq n\}$ if the label of N is $f^-_{/n}$ and $N_1, \ldots, N_n$ are child nodes of N, or
(5) $\gamma(N)$ is $\gamma(N_1) \cup \ldots \cup \gamma(N_n)$ if the label of N is 'or' and $N_1, \ldots, N_n$ are child nodes of N.

A moded type graph represents a set of moded term, i.e., a moded type, defined by γ. A set of all moded types is denoted by $\underline{Term}$.

A moded type graph $\mathcal{G}$ can be also represented by an expression, called a *moded type definition*, like a context free grammar with (possibly no) non-terminal symbols, called *type variables*, and one start symbol, called a *root*

type variable, corresponding to the root node of $\mathcal{G}$, as in [Janssens and Bruynooghe 1989]. A moded type or a mode type graph represented by a moded type definition may be referred to the root type variable.

Example 7.1 The following graph $\mathcal{G}$ is a moded type graph whose root node is labeled with $h^+_{/2}$:



Then the moded type graph may also be denoted by the following moded type definition:

$\tau = h^+(\underline{V}, \tau_1)$,
$\tau_1 = f^+(\tau_1)$,

where τ, τ_1 is type variable and τ is a root type variable.

This moded type definition represents a set of moded terms:

$\gamma(\tau) = \{\, h^+(V_1, f^+(V_2)),\ h^+(V_1, f^+(f^+(V_2))), \ldots \}$

An abstraction α for a moded term satisfying the condition (3) in Definition 6.2 is also well-defined in a similar way to [Janssens and Bruynooghe 1989].

A *moded type substitution* $\underline{\theta}$ is a mapping *Var* to $\underline{Term}$, and is also represented by a set of assignments of variables to moded types. A concretization and an abstraction for a moded type substitution is defined:

$$\gamma(\underline{\theta}) = \{\theta \mid \forall X \in dom(\underline{\theta})\ (t \in \gamma(X\underline{\theta}) \supset (X \leftarrow t) \in \theta)$$
$$\alpha(\theta) = \{X \leftarrow \alpha(X\theta) \mid X \in dom(\theta)\}$$

And an ordering relation $\sqsubseteq$ over moded type substitutions is defined as follows: $\underline{\theta}_1 \sqsubseteq \underline{\theta}_2$ iff $\gamma(V\underline{\theta}_1) \subseteq \gamma(V\underline{\theta}_2)$ for all variables $V \in Var$.

A *moded type reaction sequence* $\underline{\triangle}$ is a sequence of moded type substitutions $\underline{\delta}_1\underline{\delta}_2 \ldots \underline{\delta}_n$ such that

$\forall i, j (1 \leq i < j \leq n)(\underline{\delta}_i \sqsubseteq \underline{\delta}_j \wedge\ dom(\underline{\delta}_i) = dom(\underline{\delta}_j))$,

and $dom(\underline{\triangle})$ is defined $dom(\underline{\delta}_i)$. A concretization for a moded type reaction sequence, denoted by $\gamma(\underline{\delta}_1 \ldots \underline{\delta}_n)$, is defined as follows:

$$\{\delta_1 \ldots \delta_n \mid \delta_1 \ldots \delta_n \in Rseq \wedge \Pi\delta_i \in \gamma(\delta_i)\},$$

where $\Pi\delta_i$ is a composition of substitutions $|\delta_1| \ldots |\delta_i|$.

And an instantiation ordering $\sqsubseteq$ on a moded type reaction sequence is defined: $\underline{\triangle}_1 \sqsubseteq \underline{\triangle}_2$ iff $\gamma(\underline{\triangle}_1) \subseteq \gamma(\underline{\triangle}_2)$.

Definition 7.4 (Moded Type Atom Reaction) A *moded type atom reaction* is a pair of an atom A and moded type reaction sequence $\underline{\triangle}$ such that $dom(\underline{\triangle}) \subseteq var(A)$. $\underline{Areact}$ is a set of all moded type atom reactions.

Example 7.2 Let $\triangle$ be a reaction sequence $\{X \leftarrow f(Y)\}^+\{Y \leftarrow g(Z)\}^-$. Then $\alpha(\{X \leftarrow f(Y)\}^+\{Y \leftarrow g(Z)\}^-)$ is $\{X \leftarrow \tau_1\}\{X \leftarrow \tau_2\}$, where τ_1 and τ_2 is defined by the following type definitions:

$\tau_1 = f^+(\underline{V})$,
$\tau_2 = f^+(g^-(\underline{V}))$.

An application of a moded type substitution $\underline{\delta}$ to a moded type reaction sequence $\underline{\delta}_1 \ldots \underline{\delta}_n$ is a moded type reaction sequence $\delta'_1 \ldots \delta'_n$ such that $\delta'_i = lub(\delta_{i-1}, \delta_i)$ for all i $(0 \leq i \leq n)$ where $\delta_0 = \delta$.

A possible interleaving of moded type reaction sequences $\underline{int}$ can be well-defined by using the definition of possible interleaving on a concrete domain in Section 4.2. And $\underline{Den}$ is a power set of $\underline{Areact}$.

Now we can define the abstract semantic function $\underline{T}_{P,G}$: $\underline{Den} \rightarrow \underline{Den}$ for a program P and a goal G by using abstract operations and denotations defined above.

7.2 An Example of Detecting Multiple Writers

Consider that two goals try to instantiate a shared variable to a (possibly different) symbol(s). In such a case, the goals may cause inconsistent assignments to the same variable, which are called *multiple writers*. Recently, in the family of concurrent logic languages, several languages have been proposed that do not allow multiple writers, and many advantages have been discussed [Saraswat 1990][Ueda 1990a][Kleinman *et al.* 1991][Foster and Taylor 1989]. For examples, *moded FGHC* presented in [Ueda 1990a] has the following advantages: (1) an efficient implementation based on a message-oriented technique, (2) unification failure free, and (3) easy mode analysis. So moded FGHC seems to lead FGHC programmers into a good style of FGHC programming.

Although you can write most programs without using multiple writers, you may want to use them in *a few* cases. *Stop signal* may be one of these examples.

Stop signal is a programming technique such that, when some goal find the answer to a searching problem, the goal broadcasts a *stop signal* to any other goals which are solving the same searching problem (or its sub-problems) and the goal forces any other goals to terminate their process by instantiating a flag symbol to a variable shared by all goals. Several flaggings may occur on different goals at the same time, or some goal may broadcast a flag at any stage if a flag is not received but has been sent from other goals. In such cases, multiple writing problems may occur.

Now we show a method of detecting multiple writers as an application of the moded type inference in the previous section. The following program implements a very simple example of 'stop signal'. A subscript number of each function symbol is used to distinguish occurrences.

```
main(T,F) :- true | generate(T),search(T, F).
search(t1(_,a1,_),F) :- true | F=f1.
search(_,f2) :- true | true.
search(t2(L,b1,R),F) :- true |
                    search(L,F),search(R,F).
generate(T) :- true | T=t3(L,N,R),genNode(N),
                    generate(L),generate(R).
genNode(N) :- true | N=a2.
genNode(N) :- true | N=b2.
```

A goal generate(T) generates a binary tree with each node labeled with a or b, and a goal search(T,F) searches a node labeled with a. Body goals of search/2 share the second argument as a 'stop signal'. Now we try to analyze the moded type of a goal main(T,F) by computing $[\![P]\!]_{\texttt{main(T,F)}}$ on the abstract domain for the moded type. Here each moded type constructor has a subscript number. When we apply $\underline{\theta} = \{\texttt{X} \leftarrow \texttt{a}_1^+\}$ to $\underline{\delta} = \{\texttt{X} \leftarrow \texttt{a}_2^-\}$, we can get a moded type substitution $(\underline{\theta})\underline{\delta} = \{\texttt{X} \leftarrow \texttt{a}_{21}^-\}$. This represents a moded type $\{\texttt{X} \leftarrow \texttt{a}^-\}$ by engaging $\texttt{a}_2$ to $\texttt{a}_1$. When goals try to engage a moded type constructor with $-$ to a moded type constructor with $-$, the goals are multiple writers.

In the above program, we can compute the following moded type atom reaction in $[\![P]\!]_{\texttt{main(T,F)}}$:

(main(T,F), ... $\{\texttt{F} \leftarrow \texttt{f}_{11}^-\}$ ...).

Then we can get information such that

(1) the goal main(T,F) may cause multiple writes, and
(2) the problematic goal is a unification goal writing $\texttt{f}_1$, i.e. in the body of the first clause of search.

8 Discussions

Much research has been presented on the fixpoint approaches to the semantics for concurrent logic languages.

Atom reactions are essentially the same as *reactive clauses* introduced in *reactive behavior semantics* [Gaifman *et al.* 1989]. Since reactive behavior semantics is defined by the self-unfolding of reactive clauses, we cannot always define some reasonable abstraction of the semantics when the semantics is applied to abstract interpretation. That is, the same non-terminating problem may occur as in the example below. While using our semantics, since we define by computing all possible reaction sequences corresponding to atoms in a body at one time by *int*, such a problem does not occur.

Our semantics distinguishes *reduction failure* from *deadlock* as well as *unification failure*, although the operational semantics of FGHC say nothing w.r.t. reduction failure, that is, reduction failure is regarded as *suspension*. Then the case that a goal is reduced by no clause is distinguished from *failure* (*unification failure*), but not distinguished from deadlock. But we introduce reduction failure as a termination symbol. In a practical system of FGHC, reduction failure may be reported as a *system service* to users if the system *fortunately* detects it at run time. It is helpful to users if reduction failure can be detected since such failure causes deadlock. So, we will want to detect the possibility of reduction failure at analysis time too. This is why we must introduce reduction failure to the semantics.

In [de Boer *et.al.* 1989], they have presented a denotational and a fixpoint approach to the semantics for (non-flat) GHC. They have presented the declarative semantics based on a fixpoint approach over the semantic domain similar to our atom reaction. They have mentioned that the fixpoint semantics is sound and complete w.r.t. the operational semantics giving only the results of finite success computations. Whereas, since our approach keeps more information by using the complement of all correct input unit reactions and $\bot_{\mathrm{rf}}$, it can be *correctly* related to the operational semantics including the cases of deadlock and finite failure.

A few works on abstract interpretation for concurrent logic programs have been presented. The approaches of [Codognet *et al.* 1990] and [Codish *et al.* 1991] are based on the operational semantics.

In [Codognet *et al.* 1990], they have presented a meta-algorithm for FCP(:) and an abstracted version of it. They also show the correctness relation of the algorithm to the operational semantics, which is defined by a transition system similar to this paper.

In [Codish *et al.* 1991], they directly abstracted a standard transition system semantics, where a set of configurations is approximated to an abstract configuration. One of the advantages of their approach is that the analysis is simple and easy to prove correct.

These two are essentially the same approaches and it is easy to understand the correspondence to the operational semantics in both approaches.

In the approach of [Codish *et al.* 1991], the termination of abstract interpretation may not be guaranteed for some programs such that a goal may infinitely generate more and more sub-goals. For example, the following program is taken from [Codish *et al.* 1991]. They must abstract the domain (i.e., configuration) too much (called *star abstraction*) in order to solve such a problem. The star abstraction is enough and not too abstract to analyze suspension. But it may not be suitable to call and/or success pattern analysis. These problems may be solved by adopting some abstraction on goals other than the star abstraction [Codish 1992].

```
producer(X) :- true |
      X=f(X1,X2), producer(X1), producer(X2).
consumer(X) :- X=f(X1,X2) |
      consumer(X1), consumer(X2).
```

But our abstract interpretation can analyze call pattern of the program, and return the following moded type atom reaction when the moded type depth is 1:

(producer(X), $\{\texttt{X} \leftarrow \tau_1\}\{\texttt{X} \leftarrow \tau_2\}\{\texttt{X} \leftarrow \tau_3\}$)

$\tau_1 = \texttt{f}^+(-,-)$
$\tau_2 = \texttt{f}^+(\tau_2,-)$
$\tau_3 = \texttt{f}^+(\tau_3,\tau_3)$

Although our *possible interleavings* may be a little difficult to define and understand, these problems can be solved by the abstraction only on the domain, i.e., reaction sequences.

9 Conclusions

We have presented a denotational semantics for FGHC which is less abstract semantics and is suitable as a basis for abstract interpretation. Since the semantics is defined by a fixpoint approach on atom reactions which represent the reactive behaviors of atoms, we can easily develop a program analysis system only to abstract a (possibly infinite) domain to a finite domain. We have also demonstrated moded type inference of FGHC programs.

Acknowledgments

I thank Kazunori Ueda and Michael Codish for valuable comments and suggestions and the referees for their helpful comments.

References

[Bruynooghe and Janssens 1988] Bruynooghe, M. and G. Janssens, "An Instance of Abstract Interpretation Integrating Type and Mode Inferencing", Proc. of the 5th International Conference and Symposium on Logic Programming, R .A. Kowalski and K. A. Bowen (eds.), pp.669–683, 1988.

[Codish and Gallagher 1989] Codish, M., J. Gallagher, "A Semantic Basis for the Abstract Interpretation of Concurrent Logic Programs", Technical Report CS89–26, November, 1989.

[Codish *et al.* 1991] Codish, M., M. Falaschi and K. Marriott, "Suspension Analysis for Concurrent Logic Programs", Proc. of the 8th International Conference on Logic Programming, Furukawa, K. (ed.), pp.331–345, 1991

[Codish 1992] Codish, M., *personal communication*, Feb, 1992

[Codognet *et al.* 1990] Codognet, C., P. Codognet and M. M. Corsini, "Abstract Interpretation for Concurrent Logic Languages", Proc. of the North American Conference on Logic Programming, S. Debray and M. Hermenegildo (eds.), pp.215–232, 1990.

[de Boer *et.al.* 1989] de Boer, F. S., J. N. Kok and C. Palamidessi, "Control Flow versus Logic: a denotational and declarative model for Guarded Horn Clauses", Proc. of Mathematical Foundations of Computer Science, A. Kreczmar and G. Mirkowska (eds.), pp.165–176, LNCS-379, Springer-Verlag, 1989.

[de Boer and Palamidessi 1990] de Boer, F. S., and C. Palamidessi, "Concurrent Logic Programming: Asynchronism and Language Comparison", Proc. of the North American Conference on Logic Programming, S. Debray and M. Hermenegildo (eds.), pp.175–194, 1990.

[Falaschi *et al.* 1989] Falaschi, M., G. Levi, M. Martelli, C. Palamidessi, "A Model-theoretic Reconstruction of the Operational Semantics of Logic Programs", Università di Pisa, Technical Report TR-32/89, 1989.

[Falaschi *et al.* 1990] Falaschi, M., M. Gabbrielli, G. Levi and M. Murakami, "Nested Guarded Horn Clauses", International Journal of Foundations of Computer Science, Vol.1, no. 3, pp.249–263, 1990.

[Foster and Taylor 1989] Foster, I. and S. Taylor, "Strand: A Practical Parallel Programming Tool", Proc. of the North American Conference on Logic Programming, E. L. Lusk and R. A. Overbeek (eds.), pp.497–512, 1989.

[Gabbrielli and Levi 1990] Gabbrielli, M. and G. Levi, "Unfolding and Fixpoint Semantics for Concurrent Constraint Logic Programs", Proc. of the 2nd International Conference on Algebraic and Logic Programs, LNCS, Nancy, France, 1990.

[Gaifman *et al.* 1989] Gaifman, H., M. J. Maher and E. Shapiro, "Reactive Behavior Semantics for Concurrent Constraint Logic Programs", Proc. of the North American Conference on Logic Programming, E. L. Lusk and R. A. Overbeek (eds.), pp.551–569, 1989.

[Gerth *et al.* 1988] Gerth, R., M. Codish, Y. Lichtenstein and E. Shapiro "Fully Abstract Denotational Semantics for Concurrent Prolog", Proc. of 3rd Annual Conference on Logic in Computer Science, IEEE, pp.320–335, 1988.

[Janssens and Bruynooghe 1989] Janssens, G. and M. Bruynooghe, "An Application of Abstract Interpretation: Integrated Type and Mode Inferencing", Report CW86, Katholieke Universiteit Leuven, April, 1989.

[Kleinman *et al.* 1991] Kleinman, A., Y. Moscowitz, A. Pnueli and E. Shapiro, "Communication with Directed Logic Variables", Proc. of the 8th Annual ACM Symposium on Principles of Programming Languages, pp.221–232, 1991.

[Lassez *et al.* 1987] Lassez, J. L., M. J. Maher, and K. Marriott, "Unification Revised", Foundations of Deductive Databases and Logic Programming, Minker, J. (ed.), Morgan Kaufmann, pp. 587–625, 1987.

[Levi 1988] Levi, G., "A New Declarative Semantics of Flat Guarded Horn Clauses", Technical Report, ICOT, 1988.

[Lloyd 1987] Lloyd, J.W., "Foundation of Logic Programming", Second, Extended Edition, Springer-Verlag, 1087.

[Murakami 1988] Murakami, M., "A Declarative Semantics of Parallel Logic Programs with Perpetual Processes", Proc. of the International Conference on FGCS'88, pp.374–388, Tokyo, 1988.

[Palamidessi 1990] Palamidessi, C., "Algebraic Properties of Idempotent Substitutions", Proc. of the 17th ICALP, pp 386–399, 1990.

[Saraswat 1990] Saraswat, V. A., K. Kahn and J. Levy, "Janus: A Step Towards Distributed Constraint Programming", Proc. of the North American Conference on Logic Programming, S. Debray and M. Hermenegildo (eds.), 1990.

[Ueda 1990a] Ueda, K. and M. Morita, "New Implementation Technique for Flat GHC", Proc. of the 7th International Conference on Logic Programming, D. H. D. Warren and P. Szeredi (eds.), pp.3–17, 1990

[Ueda 1990b] Ueda, K., "Designing a Concurrent Programming Language", Proc. of an International Conference organized by the IPSJ to Commemorate the 30th Anniversary: InfoJapan'90, pp.87–94, Tokyo, 1990.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Optimization and Execution of Large Join Queries

Eileen Tien Lin *
Edward Omiecinski

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

Sudhakar Yalamanchili

School of Electrical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

Optimizing large join queries that consist of many joins has been recognized as NP-hard. In this paper, we examine the feasibility of exploiting the inherent parallelism in optimizing large join queries, on a hypercube multiprocessor. This includes not only using the multiprocessor to answer the large join query, but also to optimize it. Two heuristics are provided for generating an initial solution, which is further optimized by an iterative local-improvement method. The entire process of parallel query optimization and execution is simulated on an Intel iPSC/2 hypercube machine.

1 Introduction

A large join query consists of a series of relational database join operations. The order in which these joins are executed has a great impact on the response time. The fundamental problem with optimizing large join queries is searching the large solution space of possible query execution plans.

In [IK84], the optimization of N-relational joins, using the nested-loop join method is proven to be NP-complete. In [KBZ86], a generic cost formula is assumed applicable to the join methods used. They extend a polynomial time optimization algorithm for tree queries[IK84] to the more general case. This algorithm is also improved to an $O(N^2)$ solution where N is the number of relations in the query.

Several researchers have been studying the feasibility of applying general *combinatorial optimization techniques*, such as Simulated Annealing and Iterative Local-Improvement to avoid exhaustive enumeration of all plans. In [SG88], the solution space consists of only *outer linear join processing trees* where at most one intermediate result is active and the inner relation is always a base relation. Furthermore, they assumed that the database resides in main memory. Later in [Swa89], they propose a set of heuristics to be combined with the combinatorial techniques in order to improve the performance. In [IK90], a new Two Phase Optimization algorithm is presented which runs Iterative Local-Improvement for a small period of time and uses the output of this phase as the initial solution for the second phase that runs Simulated Annealing.

In [DKT90] and [SD90], different strategies for processing large join queries in a parallel environment are discussed. In [DKT90], the authors study how to execute a large join query on a shared memory parallel computer. In [SD90], they show how a different representation of a query tree can affect the degree of parallelism within a query and performance. Specifically, they compare *left-deep* and *right-deep* tree representations.

In this paper, we investigate the issue of using the inherent parallelism in a hypercube multiprocessor to optimize large join queries. Both *inter-join* and *intra-join* parallelism are exploited in forming the plan, which implies that a join can be performed on a subcube of any size and more than one join can be performed at a time.

*Currently at IBM Corporation, 555 Bailey Avenue, San Jose, California 95150

2 The Parallel Query Processing Model

Our parallel query processing model is predicated on the following parallel architecture model. We have P = 2^d processors interconnected in a d-dimensional binary hypercube. Each processor, with address $p_{d-1}, p_{d-2}, \ldots, p_i, \ldots, p_1, p_0$, is connected to every other processor whose address is $p_{d-1}, p_{d-2}, \ldots, \overline{p_i}, \ldots, p_1, p_0, \forall i$, where $\overline{p_i}$ is the bit complement of p_i. Communication between non-adjacent processors is realized by routing messages between intermediate nodes. Every processor (*node*) has its own memory and interacts with other processors via message passing. In this paper, we refer to an n-dimensional hypercube of 2^n nodes as an n-cube. A subcube is a subset of processors that forms a smaller hypercube. For the purposes of our study we assume the complete hypercube is available for performing the joins.

Based on this architecture model, the query processing model consists of the following steps.

1. The host preprocesses a query and transforms it into an internal form such as a join graph.
2. The host accesses the global database dictionary for relevant statistics for each relation and each join.
3. The host selects a query optimization strategy for each node. The query and the selected strategy are sent to all nodes in the system.
4. Each node follows its specified strategy to generate an initial plan and to optimize it to make the best parallel query execution plan. This plan is then reported to the host.
5. The host selects the best plan from all of the nodes, schedules the query, and sends the plan to all participating nodes. Each node then executes the plan.

A distinct feature of our research is to exploit the inherent parallelism of the optimization step, instead of relying on only the host to generate a good plan.

3 Assumptions

1. Each relation is *horizontally partitioned* over a subcube within the system. Relations may be allocated to different subcubes of different sizes. Tuples are assumed to be *uniformly* distributed across nodes within a subcube.
2. The queries considered only involve natural joins, i.e. equi-joins. For simplicity, we consider only two-way joins that use the Cube Hybrid-Hash join algorithm[OL89].
3. The system is assumed to be dedicated to this application. Every node is available for both optimization and computation of the joins.
4. A join can be performed on any subcube of any size. More than one join can be simultaneously performed in disjoint subcubes.
5. The values of attributes are distributed *uniformly and independently* of each other. This implies that the size of $R \bowtie S \bowtie T$ can be estimated by multiplying the cardinalities of the three relations and the two join selectivity factors.

4 Definitions

Parallelism in the execution of joins requires the allocation/deallocation of subcubes to relations and join computations. In our approach we use the *binary buddy* system to manage subcubes for relations and join operations. In the binary buddy system the hypercube is recursively partitioned into subcubes. The subcubes can be represented by a binary tree as follows. Associate with each node a status bit that is 1 (0) if the processor is available (busy). The leaf nodes represent the status bits of the nodes. The status bit associated with any interior node is 0 if any of the leaf nodes in the corresponding subtree is 0, and 1 otherwise. The root is at level 0 and the nodes at level i are associated with subcubes of dimension $n - i$. When a request for 2^k processors arrives, nodes at level $n-k$ in the tree are searched to find the first available one. If found, it is allocated and the status bits of all the parent nodes are adjusted accordingly. Similar updates to this structure take place on the deallocation of nodes.

The set of all processors that form a subcube of any size in a hypercube can be identified by a unique *cube identifier*. A cube identifier is an address mask. For example the 2-cube consisting of processing elements 0,2,8 and 10 are uniquely identified by the mask (*0*0), where * represents a don't care. Subcubes that can be allocated under the binary buddy system can be identified with cube identifiers of the form $p_{d-1}, p_{d-2}, \ldots, * * \ldots * *$, where the number of *'s is equal to the dimension of the subcube. This assumes that the recursive decomposition proceeds highest dimension first, followed by the next highest dimension, and so on.

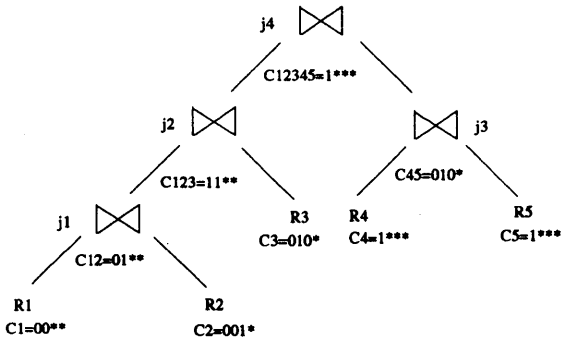


Figure 1: The Tree Representation of a Parallel Plan.

The following definitions are necessary to describe the dependencies between two joins.

Definition 4.1 *A join i is* **data-dependent** *on a join j, where $i \neq j$, if at least one of the operands in i depends on the result of j.*
A join i is **immediately-data-dependent** *on a join j, where $i \neq j$, if the following two conditions are true:*

1. *i is data-dependent on j.*
2. *j is the most immediate join prior to i that produces one of the operands in i.*

Definition 4.2 *A join i is* **location-dependent** *on a join j, where j is to be performed before i, if $C_i \cap C_j \neq \emptyset$, and i (j) is performed on cube C_i (C_j).*
A join i is **immediately-location-dependent** *on a set of joins J, where $i \notin J$, if the following two conditions are true:*

1. *i is location-dependent on k, $\forall k$, where $k \in J$.*
2. *$\nexists k_1$ that is location-dependent on k_2, where $k_1 \in J$ and $k_2 \in J$.*

Definition 4.3 *A join i is* **transfer-dependent** *on a cube C_j if all of the following conditions are true:*

1. *At least one of the operands for i, R_{1i}, is currently stored on C_j.*
2. *$C_i \neq C_j$, where join i is to be performed on C_i.*
3. *Join i is the first join that involves R_{1i} in the plan.*

As an example, consider the following query for a 4-cube, q_i,

$$R_1 \bowtie_{R_1.A=R_2.A} R_2 \bowtie_{R_2.B=R_3.B} R_3$$

$$\bowtie_{R_3.C=R_4.C} R_4 \bowtie_{R_4.D=R_5.D} R_5$$

Figure 1 is an arbitrary parallel plan for q_i. C_1 refers to the cube where relation R_1 is currently stored. C_{12} refers to the cube where the join between R_1 and R_2 is to be performed. The following dependencies exist:

1. j_1 is transfer-dependent on C_1 and C_2.
2. j_2 is immediately-data-dependent on j_1 and transfer-dependent on C_3.
3. j_3 is immediately-location-dependent on j_1 and transfer-dependent on C_4 and C_5.
4. j_4 is immediately-location-dependent on j_2 and immediately-data-dependent on j_2 and j_3.

5 The Optimization Process

The optimization process on every node consists of two steps as in [Swa89]. First, every node applies the specified heuristic to produce a feasible plan. Each of these initial plans is subject to combinatorial optimization in the second step.

In the following discussion, we represent a join plan as a binary tree, where all the non-leaf nodes represent join operations and the leaf nodes represent base relations. The cost of a plan refers to the execution time to complete this plan. The height of a plan (tree) refers to the number of join operations on the longest path from a node to the root.

5.1 Evaluating a Parallel Large Join Plan

This algorithm is used in the generation of an initial plan, and is used iteratively in the subsequent optimization of this plan.

We provide an algorithm for estimating the cost of a parallel large join plan. We require that all transfer-dependencies associated with every node be resolved before any join can be performed on this node.

To expedite the evaluation of a parallel large join plan, the following information is necessary: i, R_{1i}-ready, R_{2i}-ready, C_{12i}-ready, Start_time and Complete_time. The first two ready fields are used to indicate the existence of any immediate-data-dependency or transfer-dependency. R_{1i}-ready is set to j if i is immediately-data-dependent on j. R_{1i}-ready can also be set to C_k, if i is transfer-dependent on the cube

C_k. This means that i cannot be started before relation R_{1i} is transferred from C_k, and that i is the first join that involves R_{1i}. In this case, we require that R_{1i} be transferred to C_{12i} before any join can be performed on C_k. C_{12i}-ready is set to J if i is immediately-location-dependent on a set of joins J. When a join i is free of any immediate dependencies, all three ready fields are set to -1.

Start_time(i) is used to record the time at which i is started. This is set to the earliest time that i is free of any dependency. If i is immediately-data-dependent on j, its earliest Start_time will be the Complete_time of j plus the time to transfer the result of j. If a join i is transfer-dependent on a cube C_k because of R_{1i}, its earliest Start_time will be after R_{1i} is transferred from C_k. The earliest Start_time for a join j on a cube C_j is the earliest time when any of the nodes in C_j is done transferring data for any transfer-dependency. The time to complete this plan is thus the largest Complete_time among all joins.

The following notation is used in the following discussion, for a join i :

- i is immediately-data-dependent on joins $d1$ and $d2$.
- Transfer_time(R_{d1}) stands for the time to transfer the join result of $d1$ for i after $d1$ is completed. Notice that Complete_time(-1) and Transfer_time(R_{-1}) are both 0.
- i is immediately-location-dependent on the set of joins J.
- i is transfer-dependent on a cube C_{1k} because of R_{1k}, and on C_{2k} because of R_{2k}. The first join to occupy C_{1k} is the join $1k$, and the first join to occupy C_{2k} is the join $2k$.
- i is to be performed on C_i.
- Transfer_time(R_{1k}) stands for the time to transfer the relation $1k$ from C_{1k} to C_i.

Algorithm 5.1 *This algorithm is used to estimate the cost of a parallel large join plan.*

1. *Set all the ready fields by checking if there is any immediate data or location dependency between a join and all other joins prior to it, or any transfer-dependency between a join and any cube.*
2. *Initialize all Start_time's to* 0.
3. *Initialize all Complete_time's to infinity.*
4. *For every join i , $1 \leq i \leq n-1$, i in ascending order, n is the number of relations :*
   For every transfer-dependency of i on C_{jk}, $1 \leq j \leq 2$, do the following :
   (a) *Update the Start_time of i, to be* $Transfer\_time(R_{jk}) + \max(Start\_time(i), Start\_time(jk))$.
   i cannot start before the transfer of R_{jk} is completed. Notice that a previously positive Start_time(i) indicates that i was previously transfer-dependent on another cube or C_i was previously involved in some transfer-dependency.
   (b) *Update the Start_time of jk, to be* $Start\_time(i)$.
   We assume that jk cannot start before the transfer of R_{jk} is completed.
   (c) *Update the corresponding ready field for i and the locations of the associated relations.*
5. *Repeat Steps 6 and 7 until all joins are completed :*
6. *Compute the Complete_time for every uncompleted join that is dependency-free. Update the locations of the involving relations.*
7. *For every join i that has not been completed, $1 \leq i \leq n-1$, i in ascending order*
   For every immediate-data-dependency on dj, $1 \leq j \leq 2$,
   If dj has been completed, do the following :
   (a) *Reset the corresponding ready field.*
   (b) *Update the Start_time of i, to be* $\max(Start\_time(i), Complete\_time(dj)) + Transfer\_time(R_{dj})$.

   If i is free of any immediate-data-dependency, and i has immediate-location-dependencies on the set of joins J,
   if every join in J has been completed, do the following :
   (a) *Reset the corresponding ready field.*
   (b) *Update the Start_time of i, to be* $max(Start\_time(i), \forall_{l \in J} Complete\_time(l))$.

5.2 Initial Plan Generation

The complexity of large join optimization on a hypercube multiprocessor does not only involve finding

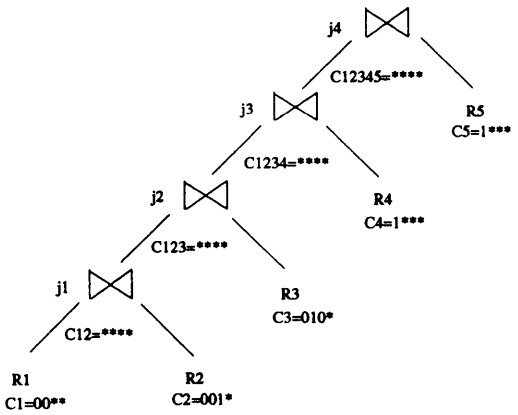


Figure 2: An Example of a Parallel Plan using the Maximum Intra-Join-Parallelism Heuristic.

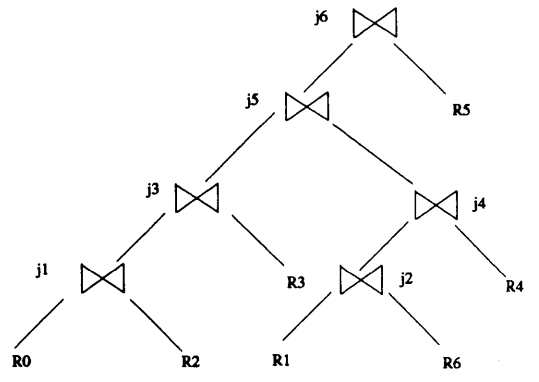


Figure 3: A Plan Generated by the Maximum Inter-Join-Parallelism Heuristic for q_j.

the best order for executing the join operations, but also the best mapping between a join and the subcube where it is to be performed. We provide the following two heuristics for mapping join operations to subcubes in order to produce an initial solution. The Maximum Inter-Join-Parallelism heuristic tries to reduce the height of the tree as much as possible. The Maximum Intra-Join-Parallelism heuristic always uses the largest cube to perform each join. These heuristics can be categorized as *greedy* heuristics.

5.2.1 Maximum Inter-Join-Parallelism

This heuristic tries to invoke as many joins in parallel as possible at the same time, and therefore, increases the degree of inter-join-parallelism. By invoking more joins in parallel, each join is allocated a smaller cube, however the height of the plan is reduced. Even though this plan may have a smaller height, each join may incur a higher cost.

The parallel plan for query q_i in Section 4, which is shown in Figure 1, could be produced by this heuristic. Due to constraints in the query, at most two joins can be performed in parallel. Therefore, j_1 and j_2 share the cube. Since the following two joins have to be performed sequentially, the largest cube is allocated to each join.

5.2.2 Maximum Intra-Join-Parallelism

The maximum degree of parallelism is applied to every join to reduce the individual cost, but at the cost of redistributing the relations prior to all the joins. Since every join uses the same set of nodes, a chain of immediate-location-dependencies is formed. As a result, all joins are forced to be performed sequentially.

Figure 2 is a possible parallel plan for the query q_i in Section 4 using this heuristic. Every join is allocated the largest cube, i.e., the whole system. Note that not all plans generated by this approach are necessarily binary linear processing trees[KBZ86] as shown in Figure 2, in which at most one intermediate relation is used as an input to subsequent join operation.

5.3 Combinatorial Optimization

Any plan for a large join query can be thought of as a *state* in a solution space which includes all possible plans. The ultimate goal of any optimization process is to find a state with the globally optimum cost.

We use a simple combinatorial optimization technique, *Iterative Local-Improvement*. In this technique, the current plan is transformed into a new plan by performing one move such as swapping the relative orders of two joins. If the new plan has a lower cost (as computed by Algorithm 5.1), it becomes the current plan. This process in general continues until a local optimum is found.

We now discuss how we optimize plans generated by each of the heuristics described in the previous section. Note that a local optimum is reached by a node when further local improvement is not possible. The best plan chosen by the nodes is selected as the parallel large join plan.

5.3.1 Maximum Inter-Join-Parallelism

Following the initial application of this heuristic, there is a limit to what extent the height of the plan (tree) can be reduced for a given query. Consider the query q_j, joining 7 relations, R_0 through R_6, where R_6 joins with relations R_1,R_3,R_4 and R_5; and R_2 joins with R_3 and with R_0. Since R_6 is to be joined with R_1,R_3,R_4 and R_5, only one of these four joins can be performed at a time. Figure 3 is a possible plan which has j_1 and j_2 performed in parallel. Following that, j_3 and j_4 are performed in parallel. Finally j_5 is performed followed by j_6. We optimize plans generated by the application of the Maximum Inter-Join-Parallelism heuristic as follows :

- Globally, each node chooses a different maximal independent set of relations, such that if two joins i and j are in the same independent set, we can perform i and j at the same time on two disjoint cubes.
- Locally, each node can swap the join locations of two randomly chosen joins for every independent set until a local optimum is reached.

5.3.2 Maximum Intra-Join-Parallelism

To optimize plans generated by this heuristic, we take an approach similar to that described using the Maximum Locality heuristic. However, since the join locations are fixed, i.e., the entire system, there is no need to alter the join locations.

6 Performance Evaluation

In this section, we present our experimental evaluation of the two heuristics for parallel large join query optimization.

6.1 System Description

The different heuristics and the entire process were coded in C and the experiments were run on a 16-node Intel iPSC/2 hypercube. To simulate a disk per node, we implemented a disk module for every node based on the single MAXTOR XT-8760S disk in our hypercube.

Since the Intel hypercube uses the circuit switching approach, we had to alter our cost model[OL89] used in estimating the cost of a plan. The model[OL89] assumes a packet switching approach. In addition, to simplify the evaluation, attributes are not added with each successive join and sufficient main memory is assumed to be available to guarantee that hash table overflow will not occur.

6.2 Query Characteristics

We categorize our queries into four groups based on the number of tuples per relation (relation cardinality), the relative locations of the relations, and the join selectivity factors:

1. All relation cardinalities are uniformly distributed between 100 and 625 so that every relation is stored on only one node. All join selectivity factors are uniformly distributed between 10^{-4} and 10^{-3}.
2. All relation cardinalities are uniformly distributed between 5001 and 10,000 so that every relation is stored on the entire 4-cube. All join selectivity factors are uniformly distributed between 10^{-5} and 10^{-4}.
3. All relation cardinalities are uniformly distributed between 100 and 10,000. All join selectivity factors are uniformly distributed between 10^{-4} and 10^{-3}.

6.3 Experimental Results

In order to compare the average performance of the two heuristics, we use the *scaled cost* instead of the real cost measured in seconds. The scaled cost is the ratio of the cost of the best plan produced by a heuristic for a given query, to the minimum cost of plans produced by the two heuristics for the same query. The reason we do not compare the actual timing is that the order of the costs for different queries can vary greatly and a scaled cost provides an objective measure of the relative advantage of a specific heuristic. A scaled cost of 1.0 means that this plan has the lowest cost among the plans produced by the two heuristics.

The execution costs of these plans on the hypercube are measured and the average scaled costs are compared to see if the result is consistent with the estimate produced by Algorithm 5.1. For simplicity, the execution cost only reflects the duration from the time a node receives a plan, until it finishes performing the entire join plan. This is the cost predicted by Algorithm 5.1.

Optimization of each of the individual plans will take a varying amount of time. The overhead in initiating the optimization process at each of the nodes, and the transfer of the results back to the host are approximately equal. Therefore, we only consider the longest and shortest durations of the optimization algorithms performed by the nodes.

For each table of results, several experiments were run. We compare the average scaled cost of the initial

Table 1: Performance of Queries in Category 1 for 20 Relations.

Cost	Inter-Join-Par.	Intra-Join-Par.
Best Initial	1.00	1.52
Best Optimized	1.00	1.50
Real Execution	1.00	2.20
Total	1.03	1.14

Table 2: Performance of Queries in Category 2 for 20 Relations.

Cost	Inter-Join-Par.	Intra-Join-Par.
Best Initial	3.42	1.00
Best Optimized	3.28	1.00
Real Execution	2.33	1.00
Total	1.13	1.16

plan, the average scaled cost of the optimized plan, the average scaled execution cost, and the scaled total cost which includes both the optimization time and execution time for the two heuristics.

6.3.1 Comparison of Algorithms

Table 1 shows the performance of the three heuristics when all relations are very small and scattered. The Maximum Inter-Join-Parallelism heuristic has the best performance since it enables many joins to be performed in parallel. In addition, for the Maximum Intra-Join-Parallelism heuristic, the longest time spend for query optimization was 93.17 seconds and for the Maximum Inter-Join-Parallelism heuristic it was only 87.34 seconds. The shortest time spend for query optimization was 31.49 seconds for the Maximum Intra-Join-Parallelism heuristic and 45.24 seconds for the Maximum Inter-Join-Parallelism heuristic. By assigning the entire cube to every join, the Maximum Intra-Join-Parallelism heuristic has to spend more time in resolving all the transfer-dependencies in the beginning since all relations have to be re-distributed over the entire cube.

When all relations are very large and stored on the entire cube, the performance of the two heuristics is summarized in Table 2. The Maximum Intra-Join-Parallelism heuristic is superior to the Maximum Inter-Join-Parallelism heuristic. Although the Maximum Inter-Join-Parallelism heuristic provides a higher degree of inter-parallelism, for this type of queries, each join takes a longer time in addition to the overhead in resolving the transfer-dependencies. In addition, for the Maximum Intra-Join-Parallelism heuristic, the longest time spent for query optimization was 94.11 seconds and for the Maximum Inter-Join-Parallelism heuristic it was only 79.77 seconds. The shortest time spend for query optimization was 25.51 seconds for the Maximum Intra-Join-Parallelism heuristic and 40.60 seconds for the Maximum Inter-Join-Parallelism heuristic.

Table 3: Performance of Queries in Category 3a for 20 Relations.

Cost	Inter-Join-Par.	Intra-Join-Par.
Best Initial	2.42	1.00
Best Optimized	4.65	1.00
Real Execution	6.35	1.03
Total	2.15	1.17

Table 3 summarizes the general case where the relation cardinalities are uniformly distributed. In general, the Maximum Intra-Join-Parallelism heuristic has the best initial and optimized costs. With respect to the longest and shortest optimization times, a similar trend appeared as with the previous experiments.

6.3.2 Query Optimization Time

In general, the longest time and the shortest time to reach a local optimum among the different starting solutions generated by different nodes are quite far apart. This makes it possible to improve the performance with the 2PO (Two Phase Optimization) method described in [IK90]. Those nodes that have reached a local optimum earlier can use the current best solution as the input to the second phase, which uses a modified simulated annealing method. This can better utilize the idle nodes and further improve the quality of their solutions.

6.3.3 Query Execution Time

Most of the average scaled costs for the query execution time on the hypercube are shown to be consistent with the estimated plan costs. That is, if a plan produced by a heuristic is shown to have the best estimate, its actual execution cost is most likely to be the best as well.

This is mainly due to the fact that for these two categories, every relation has to be re-distributed over the entire cube. This in turn results in increased link contention, and therefore communication delays.

6.3.4 Total Time

By examining the total time spent in both optimization and execution, we have a better picture of the performance of a heuristic. For category 1 in Table 1, the Maximum Inter-Join-Parallelism heuristic not only has the best execution cost but also the best total time. However in Table 2, the Maximum Inter-Join-Parallelism heuristic has the best total cost although it has the worst execution cost. One possible reason is that for this heuristic, the difference between the time that the earliest and latest node reaches a local optimum is significantly smaller than the other two heuristics; this compensates for the inferior quality of its optimized plan. Another possible explanation is that this heuristic may be able to handle queries whose relations are of similar sizes better than the other two heuristics. From Table 3 we can see that the Maximum Intra-Join-Parallelism heuristic has the best total time in general together with the best execution time.

7 Summary

In this paper, we examine the issue of optimizing large join queries on a hypercube multiprocessor. Two heuristics are proposed to produce an initial plan for a given query. We adapt the iterative local improvement procedure to the query optimization process in a hypercube multiprocessor to improve the plan produced by the application of the two heuristics. Our simulation of the query optimization and the query execution process show that the performance of these heuristics depends on the characteristics of the queries.

Optimizing complex queries in parallel will reduce the bottleneck in the query processor and will also improve the quality of the query execution plan. However, it is important to realize that the overhead can be substantial, and therefore the number of processors used in optimizing a query should depend on each individual query.

References

[DKT90] S. M. Deen, D. N. P. Kannanagara, and M. C. Taylor. Multi-Join on Parallel Processors. In *Proceedings of Second International Symposium on Databases in Parallel and Distributed Systems*, pages 92–102, 1990.

[IK84] T. Ibaraki and T. Kameda. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Transactions on Database Systems*, 9(3):482–502, September 1984.

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proceedings of ACM SIGMOD-International Conference on Management of Data*, pages 312–321, May 1990.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 128–137, 1986.

[OL89] E. Omiecinski and E. T. Lin. Hash-Based and Index-Based Join Algorithms for Cube and Ring Connected Multicomputers. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):329–343, September 1989.

[SD90] D. Schneider and D. J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *Proceedings of the 16th VLDB Conference*, August 1990.

[SG88] A. Swami and A. Gupta. Optimizing Large Join Queries. In *Proceedings of ACM SIGMOD- International Conference on Management of Data*, pages 8–17, September 1988.

[Swa89] A. Swami. Optimizing Large Join Queries : Combining Heuristics and Combinatorial Techniques. In *Proceedings of ACM SIGMOD- International Conference on Management of Data*, pages 367–376, June 1989.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases

Alexandre Lefebvre[1]

C.I.T., Griffith University
Nathan, QLD 4111, Australia
ajl@cit.gu.edu.au

Abstract

This paper is devoted to the evaluation of aggregates (*avg, sum, ...*) in deductive databases. Aggregates have proved to be a modeling tool necessary for a wide range of applications in non-deductive relational databases. They also appear to be important in connection with recursive rules, as shown by the *bill of materials* example. Several recent papers have studied the problem of semantics for aggregate programs. As in these papers, we distinguish between the classes of stratified (non-recursive) and recursive aggregate programs. For each of these two classes, the declarative semantics is recalled and an efficient evaluation algorithm is presented. The semantics and computation of aggregate programs in the recursive case are more complex: we rely on the notion of graph traversal to motivate the semantics and the evaluation method proposed. The algorithms presented here are integrated in the QSQ framework. Our work extends the recent work on aggregates by proposing an efficient algorithm in the recursive case. Recursive aggregates have been implemented in the EKS-V1 system.

1 Introduction

This paper examines an advanced functionality of deductive database systems, namely the ability to express programs involving both recursion and aggregate computations in a declarative manner. The *bill of materials* application (compute the total cost of a composite part built up recursively from basic components) shows the importance of this feature in real life databases. It is well known that such programs are not expressible in Datalog. We discuss semantics, evaluation model and implementation of aggregates in the EKS-V1 system [VBKL90].

The recursive aggregate facility is one of the innovative features of the declarative language of EKS-V1, in addition to more standard features like recursion, negation and universal and existential quantifiers. EKS-V1 also provides an extensive integrity checking facility and sophisticated update primitives (hypothetical reasoning, conditional updates). EKS-V1 was developed mainly in 1989 and demonstrated at several database conferences (EDBT, Venice, March 1990 - SIGMOD, Atlantic City, May 1990, ICLP, Paris, June 1991 - VLDB, Barcelona, September 1991, ...).

The aggregate capabilities we consider are essentially those of SQL: a grouping primitive (*group_by*) is used in association with scalar functions (such as *sum*, *avg*, *min*) computing aggregate values for each group of tuples. Adding aggregate capabilities to a recursive language causes different problems, depending on the class of programs accepted. We will consider two such classes: *stratified aggregate programs* and *non-stratified aggregate programs* (this terminology builds on an analogy with negation that will be explained below).

Our aim here is *to provide efficient evaluation algorithms* which can be integrated in the general evaluation frameworks such a QSQ or Magic Sets. In the case of EKS-V1, this is performed within the top-down QSQ/DedGin* framework which was developed in [Vie86, Vie88, Vie89] and for which compilation and implementation techniques in a set-oriented way were developed in the DedGin* prototype [LV89]. Studying evaluation in this framework does not limit its scope. Indeed, it is now accepted that there is a canonical mapping between an evaluation performed along a Magic Sets like strategy [RLK86, BR87, SZ87] and a "top-down" strategy [Vie86, Vie88, TS86] (see [Bry89b, Sek89, Ull89, Vie89] for a comparison). Hence, anything that we develop here can be adapted to Magic Sets (and vice-versa).

In *stratified aggregate programs*, aggregate operations and recursion are not allowed to be interleaved. In other words, an aggregate value may be specified over the result of a recursive query, or a recursive query may be specified over the result of an aggregate operation. However, an aggregate operation may not be part of a recursive cycle, i.e. one aggregate predicate can not recursively refer to itself.

[1]This work was achieved while the author was at the European Computer-Industry Research Centre in Munich.

For stratified aggregate programs, both semantics and evaluation issues are readily solved: 1) the semantics can be defined in a standard proof-theoretic way and 2) the evaluation problems are essentially those of top-down *constant propagation* and of *coordination* on the strata. The constant propagation issue is the (classical) problem of making use of constants given in the query to limit the search space. For a query like *"Give me the average salary for the sales department"*, one does not need to consult the entire employee relation. As for coordination, one has to make sure that all relevant tuples have been computed before performing the aggregate operation: again, this is a classical and relatively easy problem, which can be solved by appropriately extending the query evaluation method of the respective system.

In the case of *non-stratified aggregates*, interaction of recursion and aggregate computation raises more difficult problems. As a motivating example, consider the classical *bill of materials* application for a bicycle. In order to compute the total cost of a bicycle, one has to 1) compute the total costs of all its direct subparts (e.g. a wheel), 2) multiply these costs by the number of occurrences of these subparts (e.g. 2 wheels in a bicycle) and 3) sum up the resulting costs (aggregate computation). Step 1 consists in a recursive invocation of the *bill of materials* query, implying a recursive invocation of step 3 (aggregate computation). Clearly, aggregate computation and recursion are intertwined. In the following, we refer to this more general class of programs either as *non-stratified aggregate programs* or as *recursive aggregate programs.*

The first difficulty concerns semantics. For instance, suppose that, in the *bill of materials* example, a composite part is defined in terms of itself (cyclic data). Clearly, the cycle problem has to be solved in order to provide semantics for such queries. Our definition of the semantics of recursive aggregate queries relies on the two following intuitive choices. 1) We regard recursive aggregate computations as operations on top of the evaluation of a Datalog program. This underlying program represents a generalized graph (Datalog allows more than just transitive closure) being traversed during evaluation [RHDM86]. 2) Semantics should be definable in a way orthogonal to the semantics of the aggregate operations themselves: for example, the semantics of a query should be definable whenever *min* is replaced by *max* or vice-versa (of course, the result of the evaluation would be different!).

In order to give semantics to recursive aggregate programs, we consider the subclass of programs for which it is possible to associate a *reduced program* leaving out the associated computation of aggregates. This program conceptually represents the graph being traversed. We call such programs *reducible aggregate programs.* A query on a reducible program is meaningful only if there is no cycle in the derivations on the associated reduced program (we speak then of *group stratification*). Its semantics can then be defined in a classical proof-theoretic manner.

The second difficulty is the evaluation of recursive aggregate queries. As in the stratified aggregate case, this issue is two-fold: constant propagation and coordination. *Constant propagation* is done in the same way as in the stratified aggregate case. *Coordination* is more difficult than in the stratified aggregate case as one has to rely on *data stratification* (there is no predicate stratification any more). Hence, one has to ensure that the whole group of tuples for a *given input value* has been computed before performing the corresponding aggregate operation. However, we are manipulating sets of tuples: in a given set of tuples at a given time, there might be a group that has been fully computed, and another one for which only a partial set of tuples have been produced. This makes the control over the order of evaluation more complicated as it now has to be performed at the data level.

In the top-down evaluation scheme of EKS-V1, we introduce the notion of *subquery completion.* We rely on dependencies between subqueries in order to check whether the evaluation of a given group has been completed. A general solution is proposed which makes use of the reduced associated program in order to provide ranges for the subqueries, so that the resulting subquery dependencies correspond to the group dependencies. In the case of tail-recursive programs, including the *bill of materials* program, a simplification is possible.

The main contribution of our work is the integration of recursion and aggregates in a general query evaluation framework. Two independent studies on recursive aggregates [MPR90, CM90] have been developed in parallel to our work. They take a model-theoretic approach, as we consider a proof-theoretic approach to the semantics of aggregate programs. [MPR90] describes an algorithm extending the Magic Sets technique to stratified aggregate programs (in fact *Magic Stratified* aggregate programs). In this paper, *we extend the evaluation algorithm based on QSQ to group stratified aggregate programs* of which the *bill of materials* program is an example.

The structure of this article is as follows. The remainder of this section introduces some definitions and notations. Section 2 examines semantics and evaluation of stratified aggregates. For the recursive aggregate case, we first analyze the semantics problem in section 3 where we define the class of reducible aggregate programs. We then propose an evaluation method in section 4 which relies on the notion of subquery completion. Section 5 discusses related work, summarizes the paper and opens towards future work.

1.1 Definitions and Notations

We assume that a database is composed of base relations and of deduction rules of the form *Head* ← *Body* where the *Body* is a conjunction of positive and negative literals. All the variables in the *Head* should appear in a positive literal in the body. Deduction rules define virtual predicates, which are also commonly called *views* in the classical relational terminology.

Definition 1.1 Aggregate rule

An **aggregate predicate** agg_pred *is syntactically defined, as in [MPR90], by an* **aggregate rule** *in the following way:*

agg_pred($\vec{\text{Out}}$) ← **group_by**(
group_pred($\vec{\text{In}}$),
List_of_Grouping_Variables,
List_of_Aggspecs
).

where:

- List_of_Grouping_Variables *is a list of variables.* $\vec{\text{Out}}$ *and* $\vec{\text{In}}$ *are sequences of variables. They are called* **grouping, output and input variables** *respectively;*
- group_pred *is any virtual or base predicate and is called the* **grouping predicate;**
- List_of_Aggspecs *is a list of aggregate specifications of the form* A **isagg** func_agg(B) *or* A **isagg** count *where* func_agg *can be 'sum', 'min', 'max' or 'avg',* A *must be an output variable and* B *must be an input variable. The variable* A *is called an* **aggregate variable** *and* B *a* **variable to-be-aggregated;**
- *an output variable must either be a grouping variable or an aggregate variable.*

Without loss of generality, we assume that an aggregate predicate is defined by one aggregate rule only. □

Note that the aggregate function *count* has no argument, as it simply counts the number of tuples for a given group.

We allow the use of e.g. arithmetic predicates in the body of Datalog rules. Such predicates, not computable by the basic relational operations, are called *external predicates.* We suppose that the external predicates are used in a *safe* way (as in [BS89] - finite set of answers and finite top-down evaluation). As an example, the *bill of material* example uses an external predicate performing a multiplication (see section 3). The use of this predicate is safe as soon as the data is acyclic.

Definition 1.2 Grouping subtuples and groups of tuples

Given a tuple for the grouping predicate, its **grouping subtuple** *is its projection over the grouping arguments.*

Given a set of tuples S *for a grouping predicate, we partition* S *into* groups of tuples*: there is one* **group** *for each different grouping subtuple* GST *in* S. *A group contains those and only those tuples of* S *having* GST *as grouping subtuple (and no other tuple).* □

We say that a predicate $pred_1$ depends directly (resp. indirectly) on the predicate $pred_2$, if $pred_2$ appears in the body of a rule defining $pred_1$ (resp. if there is a predicate $pred_3$ such that $pred_1$ depends directly on $pred_3$ and $pred_3$ depends indirectly on $pred_2$). We can now give the following definition, inspired by the terminology used in the case of Datalog queries with negation.

Definition 1.3 Stratified aggregate program

An aggregate program is **stratified** *if no aggregate predicate depends directly nor indirectly on itself.* □

Note that aggregate programs having recursive predicates which are not mutually recursive with aggregate predicates are indeed aggregate stratified.

A simple example of a stratified aggregate program is the following:

Example 1.1 Aggregate on a base relation

Suppose that the database contains a base relation employee *with tuples of the form* employee(Name, Dept, Salary). *One can define a virtual predicate* avg_salary_per_dept *by the following rule:*

```
avg_salary_per_dept(Dept, AvgSal) <-
      group_by( employee(Name, Dept, Salary),
                [Dept],
                [AvgSal isagg avg(Salary)] ).
```

If the predicate avg_salary_per_dept *is queried with the argument* Dept *instantiated, it returns one single value. If the query is fully uninstantiated, the result is a binary table with one value per department.* ▽

2 Stratified Aggregates

In this section, we first recall the natural semantics of stratified aggregate programs, which rely on the stratification of rules. We then describe their evaluation by extending the QSQ framework.

2.1 Semantics

The stratification of a database ensures the soundness of the following extension of the classical proof-theoretic definition of semantics for stratified aggregate programs.

Similarly to Datalog programs with stratified negation, a stratified aggregate program P can be divided into *strata* S_i, $i = 1,\ldots,n$.

Consider a predicate p appearing in the body of a rule $R \in S_i$. If R is an aggregate rule and p appears as a grouping predicate in R, then the definition of p is contained in $\bigcup_{j<i} S_j$. Else, its definition is contained in $\bigcup_{j\leq i} S_j$.

Definition 2.1 Semantics of a stratified aggregate program P

Facts derivable for P *from the database are obtained by saturation of the immediate consequence operator, consecutively on each stratum* S_i, *starting from* i = 1 *up to* i = n. *Facts for aggregate predicates are defined as follows.*

For an aggregate predicate agg_pred, *there is one tuple* T_G *for each group* G *of the corresponding grouping predicate* group_pred *such that:*

1. If an attribute of T_G corresponds to a grouping variable, *its value is the value of the same variable in* G.

2. If an attribute of T_G corresponds to an aggregate variable, *its value is the result of the aggregate operation performed on the corresponding values of* G *to be aggregateed.*

□

Note that this proof-theoretic definition of the semantics is equivalent to the model-theoretic one given in [MPR90, CM90] (see proof in [Lef91]).

2.2 Evaluation

We present here an evaluation algorithm integrated in the QSQ framework. [MPR90] extend the Magic Set formalism to the stratified aggregates in a similar way.

2.2.1 Constant Propagation

The propagation of constants (i.e. taking advantage of the constants appearing in the query in order to reduce the search of the database) is addressed by adapting the QSQ framework: the top-down generation of subqueries is used for focusing on relevant data while answers are propagated bottom-up.

We first describe this adaptation on a tuple-at-a-time basis. Let Q be a query over the aggregate predicate *agg_pred* defined by an aggregate rule as in definition 1.1. Answering Q consists in the following steps:

1. If Q matches the head *agg_pred(Out)* of the aggregate rule, then generate a subquery SQ on *group_pred* by binding each variable X of *group_pred* which is also present in *agg_pred* (X must be a grouping variable) to its value in Q (either a variable or a constant).

2. Answer the subquery SQ.

3. Partition the answers to SQ into groups of tuples and perform the aggregate operations for each group.

4. Project the results over the arguments of *agg_pred.*

Note that only the bindings of grouping variables are propagated downwards. If some aggregate variables of *agg_pred* are bound in SQ, then their bindings are not propagated to SQ (e.g. if a value for the *AvgSal* argument of example 1.1 is provided in the query, then this binding is not propagated). The gain obtained by using such bindings in order to reduce the search space depends on the nature of the aggregate and can require a complicated mechanism.

2.2.2 Set-Oriented Evaluation in EKS-V1

The evaluator/compiler of EKS-V1 derives from the DedGin* prototype. The above computational scheme is implemented in a set-oriented way by a simple adaptation of the DedGin* query answering mechanism. The following operations correspond to the previously described steps:

1. A selection/projection selects from a set of queries Q those queries matching the head of the aggregate rule, and projects the resulting tuples over the relevant arguments of *group_pred.* This results in a set of subqueries SQ over *group_pred.*

2. The standard set-oriented evaluation of DedGin* is used to answer the subqueries in SQ.

3. The grouping and aggregate operations are implemented in one pass, by an extended operator. This results in an intermediate relation *tmp* containing one attribute for each grouping variable and for each aggregate variable.

4. A projection of the tuples in *tmp* over answer tuples for *agg_pred* is finally performed.

2.2.3 Coordination Aspects

In general, the evaluation of deductive queries can be viewed as a saturation both on the top-down propagation of (non-redundant) subqueries and on the bottom-up generation of answers. In the case of recursion without negation or aggregates, there is total freedom as far as the order of propagation is concerned. In particular, answers can be propagated bottom-up even if they represent only a partial set of answers to the corresponding

subqueries. In case of aggregates (also in case of negation), however, *subqueries must be completely answered before their answers can be used or propagated further.* If one did not stick to this strategy, wrong inferences could be made: for instance, one could propagate an intermediate count different from the final count.

In EKS-V1, in order to implement this strategy, we make use of a run-time structure described in [Vie88, LV89] called the *data-flow graph* (DFG). Nodes of this graph essentially represent (occurrences of) virtual predicates and the graph serves to monitor the sets of data (essentially subqueries, environments and answers) manipulated for these (occurrences of) predicates. The nodes are linked according to their relative positions in rules: the *brother* of a node corresponds to the immediately next literal in the body of a rule; predicates in the body of a rule defining a virtual predicate p form *children* nodes with respect to the node corresponding to p. Please refer to [Vie88, LV89] for a precise definition of the DFG. This structure is quite adequate for coordination aspects since it gives, at any time, a "map" of the rules that have been evaluated or remain to be evaluated to fully answer a virtual predicate. The coordination strategy described above can be formulated in the case of aggregate predicates as follows:

> *For each node* N *of the DFG corresponding to an aggregate predicate, saturate the descendants of* N *before performing the aggregate operation associated to* N.

3 Semantics of Recursive Aggregates

In order to introduce problems arising in case of recursive aggregate programs, we discuss the classical *bill of materials* example, also presented in [MPR90, CM90].

Example 3.1 Bill of materials

Suppose that the database contains the following information: basic parts *and their cost and* assembly links *to make up composite parts are stored in two base relations*

```
basic_part(Part, Cost).
assembly(Part, SubPart, Qty).
```

The bom *predicate computes the total cost of a given part by summing up the costs of all its direct subparts, computed by the grouping predicate* subpart_cost.

```
bom(Part, TotalCost) <- group_by(
        subpart_cost(Part, SubPart, Cost),
        [Part],
        [TotalCost isagg sum(Cost)] ).
```

The non-recursive rule of subpart_cost *returns the cost for a basic part. The recursive rule computes the cost which a direct subpart* SubPart *accounts for in the total cost of* Part *by recursively computing its cost and multiplying it by the number of occurrences of* SubPart *in* Part.

```
subpart_cost(Part, Part, Cost) <-
        basic_part(Part, Cost).
subpart_cost(Part, SubPart, Cost) <-
        assembly(Part, SubPart, Quantity)
        and bom(SubPart, TotalSubCost)
        and Cost is Quantity * TotalSubCost.
```

As an example, if Part *is "bicycle", and if "bicycle" is made up of two wheels (each costing 10) and of one frame (costing 100), then the subquery* subpart_cost(bicycle, Subpart, Cost) *will return two tuples:*

```
(wheel, 20)          % 20 is  2 * 10
(frame, 100)         % 100 is 1 * 100
```

The aggregate computation performed in the rule defining bom *then returns 120 as the total cost for a "bicycle".* ∇

What would be the semantics of the *bill of materials* example if there were a cycle in the data: which would be the cost of a recursively defined composite part (its value depending on itself)? In order to solve this problem, we rely on the following two choices:

1. We intuitively view recursive aggregate computations as *generalized graph traversals.* In this framework, computations are performed both along deduction paths (e.g. multiplying by the number of occurrences) and by aggregating the values associated with several paths (summing up costs). However, recursive aggregate computations go beyond graph traversal as they require 1) more complex structures than graphs to be searched (n-ary relations correspond to hypergraphs), 2) the combination of several "graphs" in the search (several, different predicates) and 3) more general search than transitive closure (e.g. non-linear recursion).

   To each recursive aggregate program, we conceptually associate a so-called *reduced program.* Intuitively, the reduced program captures the essence of traversal, while leaving out the associated computation of aggregates. We provide a rewriting method which, given a recursive aggregate program, obtains its reduced program, if one exists.

   A recursive aggregate program is then acceptable if it is syntactically correct, i.e. if there exists a reduced program attached to the original aggregate program. In such a case, the program is said to be *reducible.*

2. Moreover, we consider that the semantics should be definable in a way orthogonal to the semantics of the aggregate operations: for example, the cases where the semantics of a query is defined should be the same whenever *min* is replaced by *max* or vice-versa (however, the result of the evaluation would be different). As a consequence, we give semantics to recursive aggregates only when *the data is acyclic*, i.e. if the proof trees generated from the database for the reduced query are *acyclic*. The actual semantics of meaningful recursive aggregates queries is then defined in a classical bottom-up manner.

   Indeed, although one could compute the shortest path between two nodes of a cyclic graph, one can not compute the *maximal* length of a path in such a case. But, accepting the first case without accepting the second one would violate this principle.

3.1 Reducible Aggregate Programs and Group Stratification

We conclude the semantics chapter by giving more precise definitions of the notions "reduced", "reducible" and "acyclic" introduced above.

Consider the program P consisting of the set of rules mutually recursive with an aggregate predicate *agg_pred*. We first say that two variables X and Y are *directly connected* if they appear in the same external predicate. Furthermore, consider a predicate *pred* mutually recursive with *agg_pred*. If X appears as the i^{th} argument of *pred* in the head of a rule defining *pred*, and Y appears as the i^{th} argument in a body occurrence of *pred*, then X and Y are also directly connected. The *connected* relationship is finally the transitive closure of the *directly connected* relationship. A variable X in P is said to be **aggregate connected** if X is connected to an aggregate variable or a variable to-be-aggregated.

Obtaining a reduced program from an original program P will be possible if the grouping variables, representing the essence of the program, can be isolated from the aggregate connected variables.

Definition 3.1 Reducible aggregate program
The program P *is said to be* **reducible** *if no grouping variable is aggregate connected. If* P *is reducible, its reduced program* reduce(P) *is obtained by 1) deleting from any rule of* P *any external predicate containing aggregate connected variables and 2) replacing each literal mutually recursive with* agg_pred *by a new predicate where the aggregate connected variables have been omitted (hence, reducing its arity).* □

Indeed, if P was not reducible, then the transformation *reduce* would also remove some grouping variables carrying the essence of the program.

The full definition of the transformation can be found in [Lef91]. From now on, we consider only reducible aggregate programs[2].

In order to illustrate the concepts defined here, let us introduce the *parts explosion* example, which computes the total amount *Qty* of a given subpart *SP* involved in the construction of a given part *P*. The definition of *part_subpart_qty* has the same structure as the definition of *bom*. It uses a grouping predicate *int_subpart_qty* which gives, for each direct intermediate component *IP* of *P*, the quantity of *SP* involved through *IP*. Note that the predicate *part_subpart_qty* is an extension of the *bom* predicate having thus more didactic properties.

Example 3.2 Parts explosion and reduced program

```
part_subpart_qty(P, SP, Qty) <-
        group_by(
          int_subpart_qty(P, IP, SP, IQty),
          [P, SP],
          [Qty isagg sum(IQty)]
                ).

int_subpart_qty(P, P, SP, Qty) <-
        assembly(P, SP, Qty).

int_subpart_qty(P, IP, SP, IQty) <-
        assembly(P, IP, Qty) and
        part_subpart_qty(IP, SP, IQty1) and
        IQty is Qty * IQty1.
```

The aggregate connected variables in this program are Qty, IQty *and* IQty1. *No grouping variables are aggregate connected therefore the program is reducible. The reduced program is:*

```
r_part_subpart_qty(P, SP) <-
        r_int_subpart_qty(P, IP, SP).

r_int_subpart_qty(P, P, SP) <-
        assembly(P, SP, Qty).
r_int_subpart_qty(P, IP, SP) <-
        assembly(P, IP, Qty) and
        r_part_subpart_qty(IP, SP).
```

∇

We now define precisely what we mean by "cyclic data".

Definition 3.2 Fact and Group Dependencies
A fact F *derivable from* DB *is* **directly dependent** *on a fact* F′ *if there is a ground instance* I *of a clause*

[2] In practice, the only reasonable recursive aggregate programs we could think of are reducible. This is also the case of all examples treated in the related work.

such as I : F $\leftarrow \ldots, F', \ldots$ *and such that all the ground literals of the body of* I *are derivable from* DB. *The dependency relationship is the transitive closure of the direct dependency relationship.*

The **group dependency** *relationship is the fact dependency relationship induced by* reduce(P) *over* DB. □

Definition 3.3 Group stratified program

A recursive aggregate program P *is* **group stratified** *over a database* DB *if the group dependency relationship introduced by* P *over* DB *is acyclic.* □

We can now define the semantics of a group stratified program P over DB, by refining the definition 2.1. Again, the notion of group stratified programs here is identical to the one proposed in [MPR90].

This time, we note that the facts in *reduce(P)* can be divided along *group strata* GS_i, $i = 1,\ldots,n$, such that, if a fact $F_i \in GS_i$ depends on a fact $F_j \in GS_j$, then $j < i$. In addition, grouping and aggregate facts in P will be given the group stratum level of the corresponding reduced facts.

Definition 3.4 Semantics of a group stratified aggregate program P

Facts derivable for P *from the database are obtained by saturation of the immediate consequence operator using consecutively facts belonging to the group strata* $GS_{j \leq i}$, *starting from* i = 1 *up to* i = n. *Facts for aggregate predicates are derived as in definition 2.1.* □

4 Evaluation of Reducible Group Stratified Aggregate Programs

The evaluation problems in the recursive aggregate case are, like in the aggregate stratified case, those of constant propagation and coordination. As far as *constant propagation* is concerned, the problem is solved in the recursive aggregate case as described in section 2.2.1.

The *coordination* problem is now different. The goal is still to perform the aggregate operations only on *complete groups*. However, there is no predicate stratification in the recursive case, and a control as described in section 2.2.3 cannot be performed any more. Instead, the group stratification that the program is supposed to enforce is data dependent and not predicate dependent. Hence, the coordination will have to be brought at the data level instead of at the predicate level. [MPR90] remark that group stratified programs can be evaluated in the order of the groups. We give in this section a precise algorithm performing this evaluation.

Theoretically one could first generate the group dependency graph and base the computation on this graph. However, the representation and analysis of such a graph is likely to be expensive.

The solution proposed in EKS-V1 relies on the top down character of the evaluation: there exist natural dependencies between the *subqueries* (a subquery SQ is said to **directly depend** on the subqueries derived during the evaluation of the rules invoked for answering SQ: a formal description of these dependencies can be provided based on SLD-AL trees - see [Vie88]). In section 4.1 we first present the *subquery completion* mechanism: the evaluation of a program under subquery completion ensures that *the set of answer tuples to a subquery is propagated only when it is complete.* In section 4.2 we apply this technique to recursive aggregates. Modification of the original program using reduced literals is proposed in order to make subquery dependencies and group dependencies correspond. The subquery completion mechanism can then be applied to the modified program. Section 4.3 is concerned with tail-recursive rules. In such a case, the subquery dependencies naturally correspond to the group dependencies and the original program can be evaluated under subquery completion.

4.1 Subquery Completion

We consider that a subquery has been completed during evaluation if its *complete* set of answers has been generated.

Definition 4.1 Subquery Completion

A given subquery SQ *has been* **completed** *if one of the two following conditions holds:*

- for a subquery on a base predicate: *the join with the corresponding base relation has been performed;*
- for a subquery on a virtual predicate: *all the rules have been fired, and recursively all the subqueries on which* SQ *directly depends have been completed.*

We say that a program is evaluated under **subquery completion** *if the set of answers to each subquery* SQ *is propagated only when* SQ *has been completed.* □

The subquery completion mechanism can be implemented as follows:

1. When a subquery is derived, it is originally marked as non-completed.
2. When answering a set of subqueries for which all the rules have been triggered, the subqueries having non-completed direct descendants are left out. The other subqueries are marked as completed and the join with their corresponding answer tuples can take place.

4.2 Evaluation with the Reduced Program

Our goal is now to use the subquery completion mechanism in order to solve the problem of recursive aggregate evaluation. However, the subquery completion mechanism ensures that answers to a *subquery* are used when it has been completed, but not when a given *group* of tuples has been completed. We use calls to the reduced program in order to generate bindings for the grouping variables: this way, all grouping variables are instantiated and *the subquery tuples are identical to the grouping subtuples.* It follows that the subquery dependencies and the group dependencies coincide.

Consider a recursive aggregate program P. The algorithm can be formalized as follows.

Algorithm 4.1

1. *Produce the corresponding reduced program* reduce(P).
2. *Modify* P *by introducing, in front of each grouping and each aggregate literal in the body of rules, the corresponding reduced literal. The evaluation of the reduced literal will provide bindings for all the grouping variables. Let P' be the obtained program.*
3. *Modify the query by adding the corresponding reduced literal.*
4. *Evaluate the modified query under subquery completion over reduce(P)$\bigcup P'$.*

Thanks to the instantiations of all the grouping arguments by the reduced literals, the *subquery dependencies* correspond exactly to the *group dependencies*: the completion mechanism applied to the modified program guarantees that a given group is used for aggregate operations only when it is complete (see proof in [Lef91]).

Note that the evaluation of reducible aggregate programs which are *not* group stratified stops and returns a negative answer. As there are cycles in the dependencies, there always exists a non-completed subquery (which depends on itself), and the evaluation stops.

Example 4.1 (example 3.2 continued)
Consider a query part_subpart_qty(P, *SP, Qty) *(where "*" marks an argument which is instantiated when the literal is consulted during evaluation). Suppose that the compiler chooses the following ordering of the subqueries for the recursive rule of* int_subpart_qty.

```
int_subpart_qty(P, IP, *SP, IQty) <-
        part_subpart_qty(IP, *SP, IQty1) and
        assembly(P, *IP, Qty) and
        IQty is *Qty * *IQty1.
```

The evaluation of the recursive rule for int_subpart_qty *immediately generates subqueries on* part_subpart_qty *which are redundant w.r.t. the initial query on* part_subpart_qty*: they have the same argument* *SP *carrying the same value. However, the group dependencies are cycle free for this example as soon as the relation* assembly *is not cyclic.*

*Using the reduced literals for generating bindings for the grouping variables has the following effect on our example. The call to the query literal is replaced by "*r_part_subpart_qty(P, *SP) and part_subpart_qty(*P, *SP, Qty)*". The modified version of the program is:*

```
part_subpart_qty(P, *SP, Qty) <-
       r_int_subpart_qty(P, IP, *SP) and
       group_by(
         int_subpart_qty(*P, *IP, *SP, IQty),
         [*P, *SP],
         [Qty isagg sum(IQty)]
               ).

int_subpart_qty(*P, *P, *SP, Qty) <-
        assembly(*P, *SP, Qty).

int_subpart_qty(*P, *IP, *SP, IQty) <-
        assembly(*P, *IP, Qty) and
        r_part_subpart_qty(*IP, *SP) and
        part_subpart_qty(*IP, *SP, IQty1) and
        IQty is *Qty * *IQty1.
```

As one can see, the reduced literal r_part_subpart_qty(*IP, *SP) *in the recursive rule is superfluous as the two grouping arguments* *IP *and* *SP *would have been instantiated anyways. It can be removed.* ∇

4.3 Simplification in the Tail-Recursive Case

The mechanism we have just presented has a main drawback. For the evaluation of a query on an aggregate predicate the evaluator performs the search through the relevant data twice: once during the evaluation of the reduced predicates, and once during aggregate computation. There is a case however where the subquery dependencies naturally correspond to the group dependencies, even though some of the grouping arguments can be uninstantiated in the subqueries. In such a case, it is sufficient to evaluate the original aggregate program under subquery completion, therefore searching the data only once.

This case has been called *tail-recursive* in [LV89], and also corresponds to the *right- and left-linear recursive case* as in [NRSU89]. A tail-recursive program is characterized by the following property: for a given subquery, the variables not shared between the head and the body

literal for the recursive predicate are instantiated, and the free variables of the head and the body literal have the same positions in those literals.

Algorithm 4.1 on reducible aggregate programs in the tail-recursive case has been implemented in the EKS-V1 prototype. This includes the *bill of materials* and the *parts explosion* examples.

Example 4.1 (continued)

In case the first variable Part *is instantiated in the query literal, the program is tail-recursive and there is no need to add any reduced literals. During the evaluation of a query* ?- part_subpart_qty(*P, SP, Qty), *a subquery* part_subpart_qty(*IP, S_i, Q_i) *may depend on itself (actually on a variant of itself) if and only if there is a cycle in the* assembly *relation. These dependencies correspond to several group dependencies with the same value* *IP *for the first argument. The evaluation of queries for this pattern under subquery completion is complete and correct in the acyclic case, and fails if the* assembly *relation is cyclic.* ∇

5 Related Work

[Klu82] has first formalized aggregates in relational algebra and calculus, and argued that the notion of duplicates (multi-sets) was not needed for the expressivity of aggregates. We also think that the notion of multi-sets is not needed for specifying semantics. We regard the problem of being able to handle full duplicates within sets as an issue independent from aggregate computation. It is rather a modeling issue, as to how one may want to represent the data for a given application. Our standpoint however still permits a correct solution to the duplicate issue in the computation of aggregates: it can be performed by choosing the arguments being present in the grouping predicate. The model that we consider remains a *flat* model: it does not allow set-valued (or nested) attributes. In other words, sets are not first-class objects in EKS-V1. Hence, we are not following here the research trend around nested relations, NF2 models, represented for instance by research projects such as COL or LDL [AG91, TZ86]. In these approaches, a more general grouping (or nesting) facility is provided allowing aggregate functions to be simply expressed as functions applied to set-valued attributes. We believe that the extension of a flat model with (scalar) aggregate facilities (chosen here as in [MPR90, CM90]) remains worth investigating because its requirements on the physical level (storage and manipulation) are less stringent, it represents a natural extension of Datalog systems and despite its restrictions, it may well cover an important part of the application requirements.

Our work is close to that on Traversal Recursion by [RHDM86] in the way we consider aggregate operations as operations on top of graph traversals. However we generalize graph traversal to more complex structures than graphs and we do not incorporate the semantics of the particular aggregate function (min/max) and thus never allow cyclic graphs. Although this leads to some restrictions, we believe that, if one takes semantics of the aggregate functions into account, this should be done within as formal and as general a framework as possible.

Several recent papers [MPR90, CM90] [KS91, RS92] also consider aggregates in Datalog programs. These papers take a model theoretic approach for defining the semantics of aggregate programs.

In the *stratified aggregate* case, the semantics and evaluation methods proposed are equivalent to ours. [MPR90] extend the Magic transformation producing so-called *magic stratified* programs. The evaluation of such programs can be performed in an order corresponding to the stratification order of the original program by a modification of the bottom-up fixpoint.

For defining the *semantics of non-stratified aggregate programs*, the approach taken in [MPR90, CM90] [KS91, RS92] is different from ours: they do not consider separately the underlying reduced program. Instead, they take into account the semantics of the aggregate operations, as well as the other arithmetic constructs appearing in an aggregate program, in order to define semantics. This allow them to treat the class of *monotonic aggregate* programs (like the *minimal length path* program or the so-called *corporate takeover* program) for which natural semantics exists. [CM90] also treat *closed semiring* programs as a special case of recursive aggregate programs having natural semantics.

The *evaluation* of recursive aggregate programs is not addressed by [MPR90]. It is simply mentioned there that an evaluation following the order of the groups would be possible (which seems to be quite easy to realize). [CM90] propose a general algorithm applying to closed semiring or to monotonic aggregate programs. Closed semirings are also interesting because specialized algorithms relying on graph traversal (such as in [CN89]) can be used for their evaluation. The case of monotonic programs involving minimum and maximum predicates has been the object of another recent paper [GGZ91], proposing a bottom-up evaluation mechanism called *greedy fixpoint*. The *parts explosion* example 3.2 is also treated in [Phi90]. They use a procedural language, where the control of the completion for each subquery during query evaluation is expressed in the program by the user.

The more recent work of [KS91, RS92] is concerned with the model-theoretic semantics of aggregate programs, and unifies all the other approaches in a more

general framework.

We have also pointed out the analogy between aggregates and negation. There is a correspondence between the two notions of stratification in both areas, and between group stratification on the one hand and dynamic stratification [Prz90] (or effective stratification [BL90] or constructive consistency [Bry89a]) on the other. The coordination issue is essentially the same for negation (stratified case) as for aggregates (stratified case). Indeed, just as for aggregate predicates, negated subqueries must be fully answered before negated facts can really be inferred.

Our contribution has been to provide an efficient evaluation mechanism for group stratified reducible aggregate programs. An extension to the work presented here would be to extend our solution to the classes of closed semirings and monotonic aggregate programs, which indeed have "natural" semantics. For this class of programs, only bottom-up algorithms have been proposed yet, thus unable to focus on relevant data.

6 Acknowledgements

The author is indebted to Laurent Vieille for many constructive comments on earlier versions of this paper. I have also benefited from comments from several colleagues at ECRC, as well as useful remarks from anonymous referees.

References

[AG91] S. Abiteboul and S. Grumbach. A Rule-Based Language with Functions and Sets. *TODS*, 16(1):1–30, March 1991.

[BL90] N. Bidoit and P. Legay. *WELL!* An Evaluation Procedure for All Logic Programs. In *Proc. of the 3rd Int. Conference on Database Theory (ICDT)*, Paris, France, December 1990.

[BR87] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proc. of the 6th ACM Symposium on Principles of Database Systems (PODS)*, San Diego, California, March 1987.

[Bry89a] F. Bry. Logic Programming as Constructivism: A Formalization and its Application to Databases. In *Proc. of the 8th ACM Symposium on Principles of Database Systems (PODS)*, pages 34–50, Philadelphia, Pennsylvania, March 1989.

[Bry89b] F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. In *Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 95–112, Kyoto, Japan, 1989.

[BS89] A. Brodsky and Y. Sagiv. On Termination of Datalog Programs. In *Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 95–112, Kyoto, Japan, 1989.

[CM90] M. P. Consens and A. O. Mendelzon. Low Complexity Aggregation on GraphLog and Datalog. In *Proc. of the 3rd Int. Conference on Database Theory (ICDT)*, Paris, December 1990.

[CN89] I. Cruz and T. Norvell. Aggregative Closure: An Extension of Transitive Closure. In *Proc. IEEE 5th International Conference on Data Engineering*, pages 384–391, February 1989.

[GGZ91] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and Maximum Predicates in Logic Programming. In *Proc. of the 10th ACM Symposium on Principles of Database Systems (PODS)*, Denver, Colorado, May 1991.

[Klu82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, 29(3):699–717, July 1982.

[KS91] D. Kemp and P. Stuckey. Semantics of Logic Programs with Aggregates. In *Proc. of the International Logic Programming Symposium*, 1991.

[Lef91] A. Lefebvre. Recursive Aggregates in the EKS-V1 System. Technical Report TR-KB-34, ECRC, February 1991.

[LV89] A. Lefebvre and L. Vieille. On Deductive Query Evaluation in the DedGin* System. In *Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 95–112, Kyoto, Japan, 1989.

[MPR90] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The Magic of Duplicates and Aggregates. In *Proceedings of the 16th VLDB Conference*, pages 264–277, Brisbane, Australia, August 1990.

[NRSU89] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient Evaluation of Right-, Left-, and Multi-Linear Rules. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 235–242, Portland, Oregon, June 1989.

[Phi90] G. Phipps. Glue: A Deductive Database Programming Language. In Jan Chomicki, editor, *Proc. of the NALCP '90 Workshop on Deductive Databases*, pages 1–6, October 1990. Extended Abstract.

[Prz90] T. C. Przymusinski. Every Logic Program has a natural Stratification and an Iterated Fixed Point

Model. In *Proc. of the 9th ACM Symposium on Principles of Database Systems (PODS)*, pages 11–21, Philadelphia, Pennsylvania, April 1990.

[RHDM86] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal Recursion: a Practical Approach to Supporting Recursive Applications. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Washington DC, May 1986.

[RLK86] J. Rohmer, R. Lescoeur, and J.-M. Kerisit. The Alexander Method: a Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing*, 4(3):273–285, 1986.

[RS92] K. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. In *Proc. of the 11th ACM Symposium on Principles of Database Systems (PODS)*, 1992. Also presented at the ILPS'91 workshop on deductive databases.

[Sek89] H. Seki. On the Power of Alexander Templates. In *Proc. of the 8th ACM Symposium on Principles of Database Systems (PODS)*, pages 150–159, Philadelphia, Pennsylvania, March 1989.

[SZ87] D. Sacca and C. Zaniolo. Magic Counting Methods. In U. Dayal and I. Traiger, editors, *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 49–59, San Francisco, USA, May 1987.

[TS86] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proc. of the 3rd Int. Conference on Logic Programming*, pages 84–98, London, UK, June 1986.

[TZ86] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data-Language. In *Proc. of the 12th VLDB Conference*, pages 33–41, Kyoto, Japan, August 1986.

[Ull89] J. D. Ullman. Bottom-Up Beats Top-Down for Datalog. In *Proc. of the 8th ACM Symposium on Principles of Database Systems (PODS)*, pages 140–149, Philadelphia, Pennsylvania, March 1989.

[VBKL90] L. Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, A Short Overview. In E. Mays, editor, *AAAI Workshop on Knowledge Base Management Systems*, Boston, USA, July 1990.

[Vie86] L. Vieille. Recursive Axioms in Deductive Databases: the Query/SubQuery Approach. In L. Kerschberg, editor, *Proc. 1st Int. Conference on Expert Database Systems*, pages 179–193, Charleston, SC, USA, April 1986.

[Vie88] L. Vieille. From QSQ towards QoSaQ: Global Optimization of Recursive Queries. In L. Kerschberg, editor, *Proc. 2nd Int. Conference on Expert Database Systems*, pages 421–434, Tysons Corner, Virginia. April 1988.

[Vie89] L. Vieille. Recursive Query Processing: the Power of Logic. *Theoretical Computer Science*, 69(1), December 1989.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Distributed Programming Environment based on Logic Tuple Spaces

Paolo Ciancarini
Dipartimento di Informatica
University of Pisa - Italy
E-mail: cianca@di.unipi.it

David Gelernter
Dept. of Computer Science
Yale University - USA
E-mail: dhg@cs.yale.edu

Abstract

In this paper we describe PoliS, a coordination model based on Multiple Tuple Spaces. PoliS addresses the specification and coordination of logically distributed systems. We show that it can be used as a basic model for designing distributed and rule-based software development environments. In fact, PoliS has been used in the design of Oikos, a distributed software development environment. It has been specified and implemented using Extended Shared Prolog, a parallel logic language that smoothly combines the PoliS approach, to deal with concurrency and distribution, with Prolog, to deal with rules and deduction. Such a combination of blackboard-based communications and logic programming provides a powerful framework in which experiments about different environment architectures can be performed and evaluated.

1 Introduction

The concept of *software development environment* is a key issue in software engineering. Logic programming was proposed as an interesting technology for designing and implementing innovative environments since the first FGCS conference [Furukawa *et al.*, 1984]. However, only recently a theory for abstractly studying and comparing different software development environments has been developed [Perry and Kaiser, 1991]. Perry and Kaiser introduced a hierarchy of classes of software development environments. Their hierarchy is roughly based on the number of programmers involved and includes four classes: individual, family, city, and state. Each class is characterized by three interrelated components: policies, mechanisms, and structures. *Policies* are the strategies and the constraints imposed on the programmer by the environment; *mechanisms* are the tools supported by the environment; *structures* are the objects on which mechanisms operate.

The main contribution of this paper is the definition of an abstract paradigm for modeling and implementing a software development environment at the "city" level. The paradigm is called *PoliSpaces*, because it is based on Multiple Tuple Spaces [Gelernter, 1989, Matsuoka and Kawai, 1988], and it is shortened to *PoliS*, from the Greek word for "town". Using the terminology introduced by Perry and Kaiser, our model allows programmers to express different coordination policies simply and consistently, giving to the environment designer a powerful tool for structuring distributed software development environments.

Our proposal is twofold. Firstly, we define an abstract coordination model that can be used as a tool in the design of a distributed software development environment supporting activities by many agents. A coordination model is a set of mechanisms for expressing and controlling distributed activities [Ciancarini, 1990b, Carriero and Gelernter, 1991]. The activities themselves can be expressed in any sequential language; their interaction with respect to other activities is defined using the coordination model. To make clear the coordination issues, we have introduced Extended Shared Prolog (ESP for short) [Bucci *et al.*, 1991], a parallel logic language based on PoliS.

Secondly, we show how a software development environment can be specified using ESP. The idea is that the environment enforces protocols that specify goals, duties, and constraints of the agents involved in the software development process. We show how ESP can be used to specify simple programming environments corresponding to simple software development processes. The power of this method has been tested in the design of Oikos, a fully-fledged distributed environment [Ambriola *et al.*, 1990b]. Oikos offers a number of services giving some basic facilities, like access to databases and private workspaces, activation of shells, etc. ESP can be used to reconfigure and customize the environment.

The paper is organized as follows: Section 2 describes PoliS. Section 3 introduces Extended Shared Prolog, a programming notation based on PoliS. Section 4 shows how ESP can be used in the design of

simple software development environments and processes. Section 5 summarizes the main design principles underlying Oikos.

2 PoliSpaces: A Model for Coordination

Intuitively, a *PoliSpace* is like an abstract town where there are many places; in each place many agents cooperate. In the town many activities take place simultaneously, mostly independently; however, they are ruled by constraints that are either physical (*e.g.,* the available resources, like space and time) or abstract (*e.g.,* a set of laws that prohibit some behavior).

Formally, a PoliSpace is a distributed system that is a collection of *tuple spaces*. A *tuple space* is a multiset of tuples; a *tuple* is simply a sequence of fields. More precisely, in PoliS three concepts are important: tuples, agents, and places.

- A *tuple* is a structured data object that is a sequence of values. It is produced by some agent in some space, and it remains there until some agent consumes it. A tuple can be "copied" (read) or "consumed" (read and deleted) only by an agent included in the same place. Access to a tuple is associative, *i.e.,* it is done "by contents". The particular access mechanism chosen is a degree of freedom: *e.g.,* PoliS can accommodate either a mechanism based on typed pattern matching, as in Linda [Gelernter, 1985], or a mechanism based on unification, as in a logic language.

- An *agent* is an execution thread, *i.e.,* it is an abstraction of a running program completely independent of other agents. An agent is contained in a particular place and is able to perform some operations on the tuples that it contains. The semantics of an agent can be described as follows: an agent looks continously for some tuples; when they are found, it executes a computation consisting of instructions written in some sequential programming language; finally, it creates new entities (tuples or places). The sequential language chosen for programming the internal working of the agent is left outside the scope of the model as a degree of freedom, so that agents written in many different sequential languages can coexist.

- A *place* is a named multiset of tuples (in this paper we will use as synonyms for "place" the terms *tuple space* and *blackboard*). Places are containers in the sense that the universe of tuples and agents is partitioned in a number of places. Places can be dynamically created by agents. A place is both a computing space and a communication channel, *i.e.,* a shared data structure on which agents read and write data; in fact, an agent can produce a tuple inside a place and it has access to every tuple in its own place. An agent cannot directly read the contents of an external place.

The PoliS model is enforced by a notation whose syntax is informally described below.

2.1 Places

A place is a named multiset of tuples. Syntactically, we will write places as braced sequences of tuples.

Example: For instance, we write

```
place1{ (a) (b,X) }
```

to describe a place named `place1` containing two tuples. □

An interesting feature of PoliS places is that they have *names*. Agents can send tuples outside their own place using the name of another place. The name system of places is an interesting design choice that has been left out of PoliS: it is another degree of freedom, just like the choice of the matching mechanism to access the tuples, and the sequential language for expressing local computations. For instance, in ESP the names are structured: they are paths in Unix-like style.

2.2 Tuples

Tuples are sequences of variables and values. Values obviously depend on the chosen sequential component, *i.e.,* the sequential programming language adopted for agents. However, in PoliS a number of basic value types, as well as lists of these values, are allowed. Tuples denote themselves; they are simply data objects that exist in a place, produced by some agent and possibly in the future consumed by some agent. An important topic is the scope of variables inside tuples contained in a place: the scope of these variables spans only the tuple to which they belong. This means that each tuple inside a Tuple Space is completely independent from other tuples.

2.3 Agents

Abstractly, agents are execution *threads*, *i.e.,* an agent is a process executing some program. Syntactically, an agent is represented by a tuple and executes the program contained in another (special) tuple, called *program-tuple*. An agent can use the following *abstract tuple operations* for its interaction with the landscape it lives in:

- associative *test* of a tuple contained in the same place the agent is;
- associative *consumption* of a tuple from the same place the agent is;
- asynchronous *creation* of a place or a tuple inside the landscape the agent knows.

These operations are borrowed from Linda. Actually, Linda offers an intuitive syntax for PoliS operations, introducing two different "flavors" for the test and consumption operations (they can be either blocking or not-blocking), and two not blocking operators for the creation of entities. The blocking *test* operation in Linda is written `read(`*Tuple_schemata*`)`, the non-blocking *test* operation is written `readp(`*Tuple_schemata*`)`, the blocking *consumption* operation is written `in(`*Tuple_schemata*`)`, the non- blocking *consumption* operation is written `inp(`*Tuple_schemata*`)` (a Tuple_Schemata is simply a tuple containing variables, *i.e.*, wild cards that match any actual argument inside a tuple contained in the Tuple Space).

The *creation* operation is written as `out(`*Tuple*`)` in the case of tuples, and `Name.tsc()` in the case of places (in this paper we assume no structure on the set of names of Tuple Spaces).

An agent can output any of these entities:

- a tuple; the operation is written `out(Tuple)` in case of local writing, `name.out(Tuple)` in case of outside writing;
- a Tuple Space (*i.e.*, it creates a new place); the operation is written `name.tsc()`.

The destination of such operations is always a place. The target of an `out` operation is specified using a record-like notation. If no target is specified, the Tuple Space of the agent is used by default. What happens if an `out` operation targets an external tuple space that does not exists? PoliS tries to follow the Linda semantics: `out` is a non-blocking operation (*i.e.*, the agent that issue it does not wait for any result or error code), that never fails. Thus, communications among places are supported by a meta Tuple Space where undelivered tuples remain deposited; whenever a place comes into existence, the undelivered tuples "pop up" in the tuple space.

If an agent needs to be certain that a message arrived somewhere, it must explicitly use some protocol. For instance it could send the message and an agent that, upon arrival in the target Tuple Space, sends back an ack.

Finally, we note that an agent can *test* or *consume* tuples representing other agents. Such operations are useful to build agents that schedule agents. Places cannot be operands neither for testing nor for consuming, because the obvious semantics for such operations (test a whole place, delete a whole place) should necessarily manipulate the global state of a place, sharply contrasting with the asynchronous nature of its internal activities.

PoliS agents have a reactive semantics defined by a fixed protocol of tuple operations. The basic protocol is the following (we borrow some syntax from regular expressions: with *op** we intend a sequence of indefinite length of tuple operations):

*test**; *consume**; *loc_eval*; *out**

Syntactically, such a protocol is written inside a program-tuple.

(Heading: (Test; Consume; Loc_Eval; Out))

The `Heading` is a normal tuple. Instead, `Test`, `Consume`, and `Out` are actually sequences of tuple operations, whereas `Loc_Eval` is a sequential computation that has no side effect on the place to which the agent belongs. An agent is activated when the place contains both a program-tuple and a normal tuple matching the heading in the program-tuple. The second component of a program-tuple is also called a *pattern*. Executing a pattern, an agent will do the following actions:

- it reads associatively something from its place using any number of *test* operations; actually the PoliS *test* operation has a broader semantics than `read` in Linda: a number of predefined tests on the place are allowed, depending on the chosen type system for tuple arguments. Some useful general predefined tests are: relational (binary) predicates, a `var` predicate to check if an argument inside a tuple is a variable, and a `self` predicate returning the name of the place in which an agent is located.
- it deletes some tuples using any number of *consume* operations.

When an agent has finished testing and deleting tuples from the place, it "reacts" and starts a computation that ends by creating some new objects in the landscape.

- it executes a "local evaluation" that has no effect on the place and is invisible from outside the agent insofar as no operations on the place are allowed; this local computation is expressed in a sequential programming language,
- it outputs the results obtained in a number of places it "knows"; these outputs can consist of tuples or places;

- at the end of the sequence the agent "dies", terminating its thread of evaluation; however, we can specify an ever-lasting agent by inserting among its outputs the creation of a copy of itself.

Which is the computing model underlying agents' computations? The idea is that agents are stateless and reactive, *i.e.*, they compute when a "molecule" can be built inside the Tuple Space. A molecule is composed of a program-tuple, a normal tuple matching the first field of a program-tuple, and all the tuples to be consumed as specified by the *consume* section in the program-tuple. The agent "reacts" to its environment, "burning" the molecule, and as a result creates new entities as specified in the *create* section. This "chemical" model is also used in GAMMA [Banatre and LeMetayer, 1990].

Example:
An ever-lasting chemical reaction can be seen in this Tuple Space containing two table tennis players:

```
{(a)(b)(ping)
 ((a) : (in(ping);out(pong);out(a)))
 ((b) : (in(pong);out(ping);out(b)))}
```

Agent a begins building a molecule with tuple (ping); it consumes that tuple and produces tuple (pong) and a copy of itself (a). Then it is the turn of agent b which can react and consume tuple (pong) to produce tuple (ping) and a copy of itself (b), and so on, either forever or until something from outside comes to alter this "chemical solution". For instance, suppose that an external agent sends a new ping tuple in the above Tuple Space; as soon as the new tuple is noticed by agent a, the two agents are no longer serialized. □

Even if the relationship among places, agents, program-tuples, and local evaluations can look slightly contrived, actually their relative meaning is quite simple: a place defines an AND-parallel computation of agents; an agent executes the computation defined by a program-tuple; the agent reacts to the contents of its place with a local evaluation followed by the creation of new entities, either tuples or places.

3 ESP: A programming notation based on PoliS

PoliS is a coordination model that could accommodate any sequential language as sequential component for local computations inside agents. For example, C-Linda can be considered an instance of PoliS where the sequential language is C, tuples are built using the C data types and a unique place is allowed for every program. In Linda tuple operations inside agents are not structured (*i.e.*, you can have in, read, and out in any order), but any sequence of Linda operations can be split in a number of subsequences such that each begins with read/in operations and terminates with out operations.

PoliS is a model for designing coordination of distributed systems. It is introduced as a paradigm for explorative distributed programming, and can be considered a useful prototyping model for distributed applications. In order to explore its usefulness for this task, we have defined Extended Shared Prolog (ESP), a programming language that embeds its main features.

ESP is a logically distributed extension of the parallel logic language Shared Prolog, which is a logic language that uses the blackboard model for interprocess communication [Brogi and Ciancarini, 1991]. With some approximations, Shared Prolog can be considered a logic counterpart of the Linda family of physically distributed programming languages [Gelernter, 1985]. The main difference is that Shared Prolog gains in expressive power with respect to Linda by exploiting unification and backtracking during synchronization with the blackboard (Linda uses pattern matching, and no backtracking is allowed). ESP generalizes Shared Prolog allowing multiple blackboards using a hierarchical name system.

3.1 Theories

An ESP program is composed of a set of theories. Each *theory* has the following syntactical structure:

theory $name(V1, \ldots, Vn)$:-
eval $pattern_1 \# \ldots \# pattern_k$
with *Prolog_program*

A theory is identified by a name and zero or more arguments V_i that are logic variables that scope over the patterns. The *theory interface* follows the keyword eval and includes a number of patterns, separated by the symbol #; the *theory implementation* is the Prolog program that follows the keyword with. If we compare ESP with other languages for programming in the large, the set of patterns of a theory can be considered the interface of a module, while the Prolog program is the private implementation of the module.

Logic patterns are clauses that include *test* and *consume* operations, a *loc_eval* that is a goal to be evaluated with respect to the *Prolog_program*, and finally some *out* operations. For simplicity and consistency with the logic paradigm, test operations are written as goals, whereas consumption and creation operations are put between braces.

```
Test {Consume} → Goal {Success} fail
{Failure}
```

The combination of `Test` and `Consume` operations is a *guard*: when such a guard is satisfied, *i.e.*, when all its test and consume operations are completed, the pattern commits and the Prolog goal is evaluated. To deal with the possibility of a failure of such a Prolog goal, creation operations are partitioned in two sets separated by the keyword `fail`: if the goal evaluation succeeds the `Success` out-set is produced, else the `Failure` out-set is produced. Thus, an ESP pattern is similar to a Concurrent Prolog clause (but ESP clauses are failure-free), and a theory corresponds to the definition of a CP predicate.

As a simple example, we show a simple theory including one pattern: it defines an agent computing a value as a function of some input, or outputs an `error` tuple if the evaluation fails.

```
theory agent(State) :-
eval
    {tuple(Input)} % consume
    →
    f(Input, State, Output, NewState)
    {tuple(Output), agent(NewState)} % Success
    fail {error(f(Input,State)), agent(State)}
with
      f(I,S,O,NS):- ... % Prolog_program
```

3.2 Agents

Logic agents are represented by active tuples; they react to the presence of other tuples in their blackboard. They can read and delete tuples from their blackboard; they answer by writing tuples in any blackboard they know. The relation between input and output is defined by a Prolog program (with a slight abuse of language, we will say sometime that the behavior of an agent is defined by a theory). Several agents with the same theory can be active at the same time, in the same blackboard or in different ones.

A notable feature of ESP is that control flow of *test* and *consume* operations is ruled by backtracking. Each *test* or *consume* operation either is successful or fails; a failure activates backtracking to the preceding operation. The formal semantics of such a mechanism has been studied in [Brogi and Ciancarini, 1991].

3.3 Blackboards

For reasons that will become clear in Sect.5, the name system chosen for ESP blackboards defines a hierarchical system. In fact, blackboard names are paths in the style of a Unix file system. Such a hierarchy is not limiting the communication patterns among the agents, since blackboard names can be exchanged in tuples, and an agent can put tuples in any blackboard, provided that it knows the name of the destination. Therefore, highly dynamic communication patterns can be set up, even connecting blackboards at different levels of the hierarchy, if this is convenient.

Blackboards can be dynamically created by agents simply outputting an *activation goal* that specifies a number of agents. This is the syntax of an activation goal.

```
?- child{agent_1, ... , agent_n } @ parent.
```

This goal creates a blackboard named *child* as offspring of blackboard *parent*.

In general, the execution of an ESP goal builds a tree of blackboards. Syntactically, a blackboard is a multiset of tuples that are Prolog terms. Semantically, a blackboard defines an AND parallel evaluation that transforms the contents of the blackboard itself. The actors of such an evaluation are logic agents, whose evaluations are defined by Prolog programs.

4 Programming with ESP

The activity of programming with ESP consists of building distributed systems; this topic has been explored in another paper [Ciancarini, 1990a]. Here we will show how ESP can be used as a specification and design language for software development environments.

4.1 A Tiny Programming Environment

A rule-based distributed software development environment can be easily specified in ESP. Rule-based software development environments have recently become popular [Barghouti and Kaiser, 1990] because they can be used to support process programming, *i.e.*, the activity of specifying multiagent software development.

A very simple programming environment can be set up including an editor and a compiler. Suppose we have to specify a software development process that consists of editing a file, then compiling it as soon as the editing by a programmer is terminated; Fig.1 depicts such an environment as a PoliSpace. If the compilation gives no errors, the object program has to be invoked and executed using some test data.

In order to build the ESP program that implements such a PoliSpace we need three theories: one for an editing agent, one for a compiler agent, and one for an executing agent. We show the code for the `compiler` theory.

```
theory compiler:-
eval
    {compile(File)}
    →
    call_compiler(File),
```

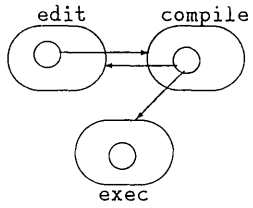


Figure 1: *A PoliSpace Coordinating a Simple Programming Environment*

```
    {compiled(File)}@exec
    fail  {do_edit(File)}@edit
with.
    call_compiler(File):- ...
     % invoke Prolog-Unix envelope for cc
     % fails if compilations fails for errors
```

Such a theory is called *envelope* because they incapsulate external software tools [Kaiser *et al.*, 1987]. Envelopes are useful to introduce non-declarative operators inside a declarative framework, because they are able to call standard Unix tools via some system predicates that return a logic result (*i.e.*, success or failure).

This minimal programming environment enforces a simple *edit-compile-exec* programming model. Admittedly, something similar is not difficult to do with sophisticated editors like GNUEmacs, however in ESP distribution and remote evaluation are very easy to deal with. Moreover, it is easy to specify different interaction paradigms. For instance the three agents `editor`, `compiler`, and `executor` are easily integrated in a unique blackboard, or can be separated in different blackboards, as in Fig.1, aiming at enforcing distribution and protection.

4.2 A Multiuser Environment enforcing an Access Protocol

A software project is composed of a set of modules on which a team of programmers operate. The updated public version of the whole project is stored within a *main database*. Users can access the main database in *read mode*. It is not possible to directly change the main database. The core of the environment is a reserve/deposit access protocol to the main database which guarantees mutual exclusion and consistency: the main database always contains a consistent and updated version of the project. To modify the contents of a module, the user must reserve the module to gain *write access*. Obviously, at any time a module can be reserved just once.

A reserved module is copied into the *user database*, where the user can modify it at will. While a reserved module is being edited in a user database, other users

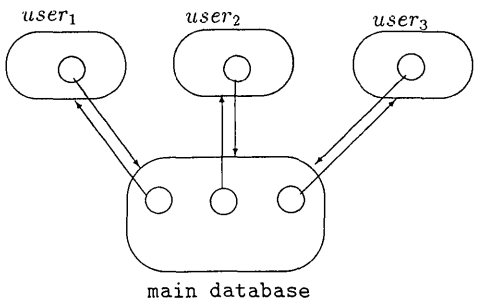


Figure 2: *A PoliSpace Coordinating a Multiuser Environment*

can access in read mode the old public version of the module stored in the main database. When the changes to the module are completed and tested, the user will deposit the new version back into the main database. The updated version is then readily accessible by all other users.

The PoliSpace realizing such an environment is showed in Fig.2.

```
theory user_database_manager:-
eval
    self(Udb), {check_in(File, Dbmain)}
    →
    {check_in(File,Udb)}@Dbmain
#
    self(Udb), {check_out(File,Dbmain)}
    →
    {check_out(Udb,File)}@Dbmain

theory main_database_manager:-
eval
    file(F), not reserved(F,_), {check_out(P,F)}
    →
    {reserved(F,by(P)}, {file(F)}@P
#
    not file(F), {check_out(P,F)}
    →
    {error(nofile(F))}@P
#
    reserved(F,by(OP)), P ≠ OP, {check_out(P,F)}
    →
    {error(is_locked(F,by(OP)))}@P
#
    file(F), not reserved(F), {check_in(F,P)}
    →
    {error(file_exists(F))}@P
#
    file(F), reserved(F,by(F,OP), P ≠ OP,
{check_in(F,P)}
    →
    {error(is_locked(F,by(OP)))}@P
#
    not file(F), {check_in(F,P)}
```

→
{file(F)}, {created(F)}@P)

We show only the code of the theories user_database_manager, that handles the user's requests in a user database, and main_database_manager, which guarantees the consistency of the main database.

5 Oikos

Oikos is a distributed software development environment based on PoliS and written in ESP [Ambriola *et al.*, 1990b]. Oikos provides a number of standard facilities that can be easily configured using ESP itself. The overall approach consists of offering mechanisms that can be easily composed, in order to easily explore different environment designs.

The ESP blackboard hierarchy offers a natural way of structuring a software development environment. It is used to reflect its decomposition in subenvironments, according to a top-down refinement strategy. The blackboard hierarchy is not really constraining the communication patterns among the agents participating in a software development process, since blackboard names can be exchanged in tuples, and an agent can put tuples in any blackboard, provided that it knows the name of the destination. Therefore, highly dynamic communication patterns can be set up, even connecting blackboards at different levels of the hierarchy, if this is convenient.

5.1 A Prototype Implementation of Oikos

The Oikos prototype has been implemented on top of a local network connecting some Sun workstations and a Vax mainframe. Oikos is written is ESP, that provides the basic mechanisms for physical distribution and dynamic activation of communicating processes. ESP itself is implemented partly in C and partly in Prolog [Bucci *et al.*, 1991]. The standard set of Prolog system predicates has been augmented with IPC mechanisms using Unix Internet sockets [Ambriola *et al.*, 1990a].

The three layers of the Oikos architecture are: the Oikos runtime support, which is written in ESP and provides escapes to the underlying operating system; a collection of separate processes, that implement a distributed ESP run-time system; the underlying operating system, UNIX in this case. The processes in the second layer are depicted by circles: an ESP process is the local interpreter of the ESP language, and there are as many of them as machines in the network, eager to interpret pieces of the ESP program. For a more detailed exposition see [Bucci *et al.*, 1991].

5.2 Oikos Services

Oikos provides a set of basic services. A *service* offers access to shared resources according to a given protocol. The public interface of a service specifies the protocol of interaction with the service, *i.e.,* which tuples must be put into its blackboard to obtain its service. For lack of space, we simply summarize the Oikos *standard services*, which play the role that primitive operators and data types play in a programming language. We discuss here the most meaningful only, *i.e.,* those that are fundamental in any software development process.

The Tool Kit Server (TKS), the Service and Theory Server (STS) and the History Server (HS) offer restricted access to databases of system data, *e.g.,* those modeling the predefined documents. A User Interface Service (UIS) is used to interact with running software process programs, whereas the Workspace Server (WS) allows users to run the tools and the executable products of the software process. The DataBase Server (DBS) offers unrestricted access to a general purpose project database, and is therefore used to set up specific project databases. Finally, the Oikos Run Time System (ORTS) can also be seen as a server offering essential services, like escapes to the underlying operating system. All these services, except ORTS, can be simultaneously activated several times in different blackboards.

The user accesses Oikos through a special interactive service called User Interface Service (UIS). It is a service because several different UIS can coexist, and their definitions are ESP programs found in STS. A UIS shows the user the contents of its blackboard in a window, and acts according to the user's input. A UIS offers also a flexible way to monitor a software process, since the user can activate it on a blackboard, looking at the tuple flow, and even saving some tuples with the history server HS. UISs are the basic blocks of the role services, *i.e.,* those parts of the process program that allows users to interact with the software process.

For lack of space, here we do not show how Oikos is used in a real software development process. The interested reader can see the example contained in [Ambriola *et al.*, 1990b].

6 Conclusions

In this paper we have introduced PoliS, a coordination model useful for designing distributed systems. A programming notation based on PoliS, ESP, has been used to illustrate the design of Oikos, a distributed software development environment. The goal of the ESP/Oikos project is to assess the combination of the blackboard model with logic programming in the de-

sign of distributed programming environments. While the blackboard model is well known in Artificial Intelligence, its use in Software engineering is quite novel.

After completing the implementation of Oikos, our future plans include the study of the impact of different models of tool coordination in the definition of planning tools for assisting users in the software process, and the analysis of the role interplay in dealing with the software process itself.

Acknowledgements P.Ciancarini has been partially supported by C.N.R. Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, and M.U.R.S.T. The authors are very grateful to N.Carriero at Yale, for many discussions on Linda and PoliS, and to the Shared Prolog and Oikos groups in Pisa, including V.Ambriola, A.Bucci, T.Castagnetti, M.Danelutto, and C.Montangero.

References

[Ambriola *et al.*, 1990a] Vincenzo Ambriola, Paolo Ciancarini, and Marco Danelutto. Design and distributed implementation of the parallel logic language Shared Prolog. In *Proceedings of ACM Symp. on Principles and Practice of Parallel Programming*, volume 25:3 of *SIGPLAN Notices*, pages 40–49, 1990.

[Ambriola *et al.*, 1990b] Vincenzo Ambriola, Paolo Ciancarini, and Carlo Montangero. Enacting software processes in Oikos. In *Proceedings of ACM SIGSOFT Conf. on Software Development Environments*, volume 15:6 of *ACM SIGSOFT Software Engineering Notes*, pages 12–23, 1990.

[Banatre and LeMetayer, 1990] Jean-Pierre Banatre and Daniel LeMetayer. The gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[Barghouti and Kaiser, 1990] Naser Barghouti and Gail Kaiser. Modeling concurrency in rule-based development environments. *IEEE Expert*, 5(6):15–27, December 1990.

[Brogi and Ciancarini, 1991] Antonio Brogi and Paolo Ciancarini. The concurrent language Shared Prolog. *ACM Trans. on Programming Languages and Systems*, 13(1):99–123, 1991.

[Bucci *et al.*, 1991] Annamaria Bucci, Paolo Ciancarini, and Carlo Montangero. Extended Shared Prolog: A multiple tuple space logic language. In *Proceedings of the 10^{th} Japanese Logic Programming Conference*, 1991.

[Carriero and Gelernter, 1991] Nick Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 1991.

[Ciancarini, 1990a] Paolo Ciancarini. Blackboard programming in Shared Prolog. In David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 170–185. MIT Press, 1990.

[Ciancarini, 1990b] Paolo Ciancarini. Coordination languages for open system programming. In *Proceedings IEEE Conf. on Computer Languages*, pages 120–129, 1990.

[Furukawa *et al.*, 1984] K. Furukawa, A. Takeuchi, S. Kunifujii, H. Yasukawa, M. Ohki, and K. Ueda. Mandala: A logic based knowledge programming system. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, pages 613–622, 1984.

[Gelernter, 1985] David Gelernter. Generative communication in Linda. *ACM Trans. on Programming Languages and Systems*, 7(1):80–112, 1985.

[Gelernter, 1989] David Gelernter. Multiple tuple spaces in Linda. In *Proceedings of PARLE '89*, volume 365 of *Lecture Notes in Computer Science*, pages 20–27, 1989.

[Kaiser *et al.*, 1987] Gail Kaiser, Simon Kaplan, and Josephine Micallef. Multiuser, distributed language based environments. *IEEE Software*, 4(11):58–67, 1987.

[Matsuoka and Kawai, 1988] S. Matsuoka and S. Kawai. Using tuple-space communication in distributed object-oriented architectures. In *Proc. OOPSLA '88*, volume 23:11 of *ACM SIGPLAN Notices*, pages 276–284, November 1988.

[Perry and Kaiser, 1991] Dewayne Perry and Gail Kaiser. Models of software development environments. *IEEE Trans. on Software Engineering*, 17(3):283–295, 1991.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Visualizing Parallel Logic Programs with VISTA

E. Tick
Dept. of Computer Science
University of Oregon
Eugene OR 97403

ABSTRACT

A software visualization tool is described that transforms program execution trace data from a multiprocessor into a single color image: a *program signature*. The image is essentially the program's logical procedure-invocation tree, displayed radially from the root, with possible radial and lateral condensation. An implementation of the tool was made in X–Windows, and experimentation with the system was performed with trace data from Panda, a shared-memory multiprocessor implementation of FGHC. We demonstrate how the tool helps the programmer develop intuitions about the performance of long-running parallel logic programs.

1 Introduction

Parallel programming is difficult in two main senses. It is difficult to create correct programs and furthermore, it is difficult to exploit the maximum possible performance in programs. One approach to alleviating these difficulties is to support debugging, visualization, and environment control tools. However, unlike tools for sequential processors, parallel tools must manage a distinctly complex workspace. The numbers of processes, numbers of processors, topologies, data and control dependencies, communication, synchronization, and event orderings multiplicatively create a design space that is too large for current tools to manage.

The overall goal of our research is to contribute to processing this massive amount of information so that a programmer can understand it. There is no doubt that a variety of visualization tools will be needed (e.g., [6, 9, 12, 5]): no one view can satisfy all applications, paradigms, and users. Yet each view should be considered on its own merits: what are its strong and weak points, how effective is it in conveying the information desired, and hiding all else. In this paper we introduce one view in such a system: based on a new technique, called "kaleidescope visualization," that summarizes the execution of a program in a single image or signature.

Unlike scientific visualization, i.e., the graphical rendering of multi-dimensional physical processes, in parallel performance analysis there are no "physical" phenomena; rather, abstract interactions between objects. Thus renderings tend to be more abstract, are less constrained by "reality," and are certainly dealing with many interacting parameters controlling the design space. Kaleidescope visualization is the graphical rendering of a dynamic call tree of a parallel program in polar coordinates, to gain maximum utilization of space. To fit the entire tree into a single workstation window, condensation transformations are performed to shrink the image without losing visual information.

This paper concentrates on the analysis of parallel logic programs with VISTA, an X–Windows realization of kaleidescope visualization. Although we concentrate on committed-choice reduction-based languages, VISTA is applicable to a wider class of procedure-based AND-parallel languages. The paper is organization as follows. Section 2 summarizes similar types of visualization tools, and Section 3 reviews the VISTA algorithms (summarizing [14]). In Section 4, we describe the parallel logic programming platform upon which VISTA experimentation was conducted, and analyze the performance of logic programs to illustrate the power of the tool. In Section 5, conclusions and are summarized.

2 Literature Review

Earlier work on WAMTRACE [2, 3], a visualization tool for OR-parallel Prolog, has influenced our work a great deal. WAMTRACE is a trace-driven animator for Aurora Prolog [8] (originally for ANLWAM [2]). Aurora creates a proof tree over which processors ("workers") travel in search of work. WAMTRACE shows the tree, growing vertically from root (top) to leaves (bottom), with icons representing node and worker types (e.g., live and dead branchpoints, active and idle workers). The philosophy of WAMTRACE was that an experimental tool should present as much information to the programmer as is available. This often results in information overload, especially because the animation progresses in time, leaving only the short interval of the near-present animation frames in the mind of the viewer.

With comparison to WAMTRACE, our goals in VISTA were: (1) generalize the tool for other language paradigms. Specifically, AND-parallel execution is more prevalent in most languages, and needed to be addressed; (2) to summarize the animation; (3) abstract away information so as not to detract the viewer from understanding one thing at a time. Thus we introduce different views of the same static image, to convey different characteristics; (4) more advanced use of color to reduce image complexity and increase viewer intuitions.

Note that the emphasis of WAMTRACE on animation is a feature, not a bug — the animation enables the gross behavior of the dynamic scheduling algorithms to be understood. Animation is implemented, but not stressed in VISTA. In this paper, we analyze a system with simple on-demand scheduling [10], and so animation is not critical to understanding program behavior.

There are numerous views of performance data, quite different than WAMTRACE, e.g., [6, 9, 12, 5]. In general, these methods are effective only for large-grain processes, and either do not show logical (process) views of program execution, or cannot show such views for large numbers of processes. Voyeur [12] and Moviola [5] are closest in concept to VISTA. These animators have great benefit, but this limits the complexity that can be realistically viewed. Related research concerns a visual representation [7] and visual debugger [4] for committed-choice languages, but these are not for performance analysis.

3 Inside VISTA

The main goal of VISTA is to give *effective* visual feedback to a programmer tuning a program for parallel performance. To achieve this goal, VISTA displays an entire reduction tree in one (workstation) window, with image *condensation* if needed. Two types of condensation are performed: level and node condensing (described below). In addition, VISTA enables a user to view the tree from different perspectives (PE, time, or procedure) and zoom-up different portions of the tree. Since the tree is usually dense for small-grain parallel programs (even after condensation), and the tree must be redisplayed when the user desires different views, the tree-management algorithm must be efficient and the window space must be utilized effectively.

We now define terminology for describing logical call trees. The level of a node is the path length from the root to the node. The root level is zero. The root is the initial procedure invocation, i.e., the query. The height of a tree is the longest path from the root to a leaf, plus one. The height of a node is the height of the tree minus the level of the node. Level condensing is a mapping from a tree T to a tree T' with the same ancestor and descendant relationships, such that a node n at level l in T is mapped into a node n' at level l' in T', where $l' = \lfloor l/c \rfloor$ and c is level-condensing ratio (defined below). Node condensing is the removal of all the descendants of the node n from the tree, if the allocated sector (defined below) for n in the window space is less than one pixel.

With these definitions in hand, VISTA management is now reviewed. There are two inputs to the algorithm: a trace file and a source program. A trace file entry consists information corresponding to a time-stamped procedure reduction. Although not currently implemented, VISTA could easily be extended to accept arbitrary events logged in the trace, as does WAMTRACE.

There are alternative ways to map an arbitrarily large tree onto a limited window space. We employ an abstraction requiring two passes over the trace: finding the tree height and creating a level-condensed tree. The condensed tree keeps the shape of the original tree (although scaling is not precise). This original shape allows us to carry our intuitions over from the tool's view to tuning performance of actual programs. In order to calculate the level-condensing ratio, c, the maximum tree height to be displayed (i.e., limitation of the window space) is needed: $h_{max} = \lfloor w/2d \rfloor$, where w is the maximum window width, and d is the distance between two adjacent levels. If tree height $h \leq h_{max}$, level condensing is not needed. If $h > h_{max}$, level condensing is performed with the c ratio calculated as follows:

$$
\begin{aligned}
c_0 &= \lfloor h/h_{max} \rfloor \\
t &= c_0 h_{max} - c_0(h - c_0 h_{max}) \\
c &= \begin{cases} c_0 & l \leq t \\ c_0 + 1 & l > t \end{cases}
\end{aligned}
$$

where l is the node level. This condensation scheme puts more emphasis (space) on the levels closer to the root because earlier reductions are generally more "important" than later reductions. The heuristic corresponds to the user's intuition that processes responsible for distribution of many subprocesses should appear larger.

An open question is the categorization of programs into those which abide by this heuristic, and those which do not. A program that would "frustrate" VISTA heuristics has the most significant computation near the leaves, where distribution of this work (near the root) is less important. A trivial example is a tree of parallel tasks, each a very heavy sequential thread of computation. VISTA will condense the graph to fit within the window, in the limit (of very long threads) producing a star shape. Although this may be considered "intuitive," it is abstracts away all information except the threads, which themselves are difficult to view against the background. Alternative views, such as condensing each thread into a polygon, perhaps colored as a function of the condensation, may be more informative because the freed-up window space would allow the work distribution at the root to be viewed also. How this and other types of con-

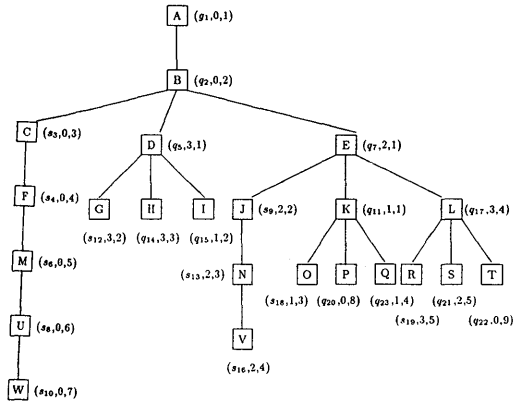


Figure 1: Whole Tree for Qsort: "?- qsort([2,1,4,5,3],X)"

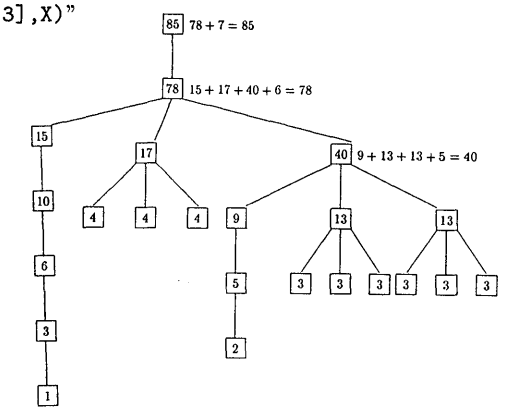


Figure 2: Weight Calculation for Qsort

densation, while not scaling the tree linearly, can lead to better understanding of certain programs, is a topic of future research.

At this stage of the algorithm, the levels to be displayed, and to be discarded, are decided. To illustrate, consider Fig. 1 showing the original tree for a Quick Sort program (the trace executed on four PEs and consists of 23 records). Each node is labeled with a triple, $(s_i, pe, index)$, where s_i is procedure s (abbreviated) invoked at trace index i, pe is the PE number, and $index$ is the sequence index of that PE. For example, $(q_5, 3, 1)$ of node D denotes that procedure *qsort* was invoked as the 5^{th} trace record, and reduced on PE = 3 as the first goal executed by that processor. After level condensing with c=2, nodes B, F–L, U, and V are contracted into their parents. These odd-level nodes are removed because if (l mod c) $\neq$ 0, all nodes in level l are condensed.

Each node at level l in the logical tree is displayed in the window at a locus defined by radius $r = d \times l$, where d is the (constant) distance between two adjacent levels. The node is illustrated by a point, however it is connected to its children (at the next level) by a closed polygon around the "family" (the polygon degenerates into a line if there is one child). The polygon itself is colored, representing an attribute of the parent. After

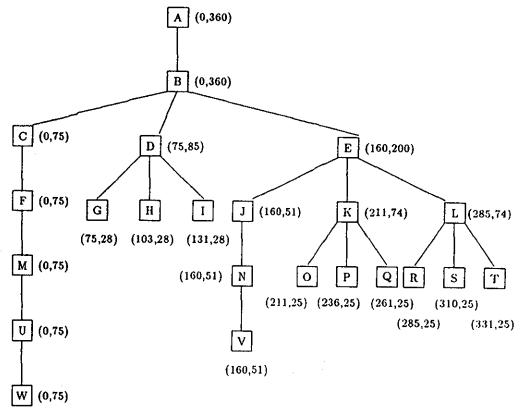


Figure 3: Node Allocation for Qsort

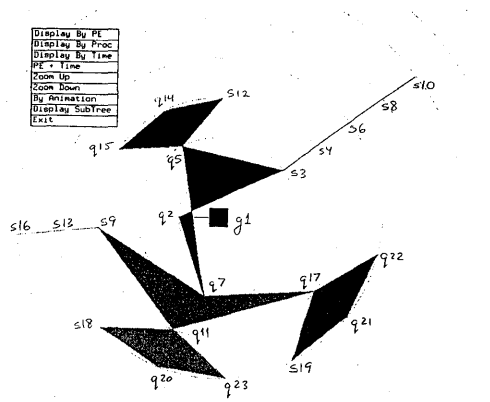


Figure 4: Execution Graph of Qsort: PE View (4 PEs)

level condensing has completed, the nodes at each level are allocated to the corresponding locus (a concentric circle). This is analogous to the pretty printing problem for text. We solve this problem heuristically by allocating a *sector* to each node depending on its *weight*. The node and its children are then displayed within the range of the sector only. The weight w for each node is heuristically defined as the sum of the weights of its children plus the height of the node. Thus more weight is put on nodes closer to the root because, the closer the node is to the root, the fewer nodes the corresponding circle can contain. Fig. 2 shows an example of the weight calculation for the Quick Sort program.

A sector is defined as the subset of the concentric circle within which a node can be displayed. To formalize the sector calculation, consider a unique labeling of each node by a path from the root $\{x_1, x_2, ..., x_k\}$, where x_i is the sibling number traversed in the path. For example, in Fig. 2, node J with weight 9 has label $\{1, 3, 1\}$. The sector of a node at path p is represented as a pair (s_p, a_p), where s_p and a_p are the starting degree and the allocation degree of the node, respectively. The sector of the root is defined as (0, 360). The starting degree s_p for a node at level k is calculated as follows:

$$s_{x_1,x_2,\ldots,x_k} = \begin{cases} s_{x_1,x_2,\ldots,x_{(k-1)}} & \text{leftmost child} \\ s_{x_1,x_2,\ldots,(x_k-1)} + a_{x_1,x_2,\ldots,(x_k-1)} & \text{otherwise} \end{cases}$$

In other words, if the node is the leftmost child, then the starting degree is equal to the starting degree of the node's parent. Otherwise, the starting degree of the node is equal to the sum of the starting and allocation degrees of its left sibling. The allocation degree a_p for a node at level k is calculated as follows:

$$a_{x_1,x_2,\ldots,x_k} = \frac{w_{x_1,x_2,\ldots,x_k}}{\sum_{j=1}^{m} w_{x_1,x_2,\ldots,(x_k-j)}} \times a_{x_1,x_2,\ldots,x_{(k-1)}}$$

where w_p is the node's weight, the summation is the total weight of all m siblings (including the node itself), and the final factor is the allocation degree of the parent. Fig. 3 shows the sectors of the nodes for Fig. 2.

After the previous steps, the execution graph is ready for display in the X–Window System [11] with VISTA as a client. When drawing the graphic, if the sector calculated is less than one pixel, node condensing is done, i.e., the node and its children are not displayed. The exact position for each node in the window space isn't calculated until the tree is displayed, since the size and the center of the tree may be changed. The exact node position (x, y) in the window is calculated in the next step as $x = d \times l \times cos(s + a/2)$ and $y = d \times l \times sin(s + a/2)$, where, d is the level distance, l is the level of the node, and (s,a) are the start/allocation degrees of the node. A complete description of the internal algorithms is given in [14]. To put the algorithms into perspective, Fig. 4 shows the VISTA display showing a PE view of the Quick Sort program (corresponding to Figs. 1–3).

4 Program Analysis with VISTA

Our initial experimental testbed for VISTA is an instrumented version of the parallel FGHC system, Panda [10, 13]. Tuning a fine-grain parallel FGHC program for increased performance involves understanding how much parallelism is available and what portion is being utilized. In experimenting with parallel logic programs using VISTA, we have found a number of approaches useful for understanding performance characteristics. Our experiments consisted of a set of execution runs on a Sequent Symmetry, and involved both modifying the benchmarks and varying the numbers of PEs. We examine three such benchmarks here.

4.1 Pascal's Triangle Problem

Pascal's Triangle is composed of the coefficients of $(x + y)^n$ for $n \geq 0$. The binomial coefficients of degree n are computed by adding successive pairs of coefficients of degree $n - 1$. A set of coefficients is defined as a *row* in Pascal's Triangle. Our first benchmark [13] computes the 35^{th} row of coefficients, with bignum arithmetic.

The easiest way to understand a program in VISTA is with a procedure view or graph. Fig. 5 shows the reduction tree from the procedure view. This graph, displayed here without any condensation, has 2,235 nodes and a height of 56. The interesting snail shape, where the radial arms correspond to row calculations, indicates that the rows, and therefore the computations, are growing in size. Near the root, a cyan distribution procedure spawns the rows, and near the leaves, a sky-blue bignum procedure adds coefficients. The size of the subtree (i.e., one row) is increased by one for every two rows. This means that $\frac{n}{2}$ and $\frac{n}{2} + 1$ rows have the same number of coefficients (because only the first half of the row is ever computed, taking advantage of the symmetry of a row). The two lines at the east side represent the expansion of the final half row into a full row. This program illustrates how the user can roughly understand execution characteristics from the procedure view, even without knowing the precise details of the source code.

To analyze the parallelism of the execution graph, we first examine load balancing among PEs. Good load balancing among PEs does not necessarily mean efficient exploitation of parallelism. However, without fair load balancing, full exploitation of parallelism cannot be achieved. In VISTA, a fair color distribution in the PE view or graph represents good load balancing. Figs. 6 (PE graph) and 7 (time graph) represent the execution on five PEs. In the time view, the RGB color spectrum from blue to magenta represents the complete execution time. Because there are few visibly distinct colors in this range, the same color in the time graph does not necessarily represent the same time. If some nodes are represented with the same color within the time graph, and by the same PE color within the PE graph (i.e, all the nodes are executed by the same PE), then the reductions were executed sequentially.

All five colors are distributed almost evenly in the PE graph for Pascal, representing good load balancing. To further analyze parallelism, both PE and time graphs are used in conjunction. In the time view, the spectrum is distributed *radially*, although not perfectly so. This indicates that most rows were executed in parallel. Although the maximum parallelism is limited by the PEs at five, again the vagueness of the RGB spectrum can be misleading, making it appear as if there is *more* parallelism. This problem can be overcome to some extent with a subtree display, where the spectrum is recycled to represent time relative to the selected root.

Fig. 8 shows the single-PE time graph for Pascal. In this graph, the spectrum is distributed laterally, around the spiral. This distribution indicates that the nodes were executed by depth-first search, the standard Panda scheduling when no suspensions occur. By comparing the two time graphs, we can infer the manner of scheduling: breadth-first on five PEs, and depth-first on one PE, but without a PE view, we cannot conclusively infer parallelism. Fig. 6 shows that some rows are not executed entirely by the same PE (i.e., task switches

occur within some rows). These characteristics indicate that suspensions are occurring due to data dependencies between successive rows of coefficients. All three figures in conjunction indicate the "wavelike" parallelism being exploited as the leftmost coefficients of the Triangle propagate the computation down and to the right.

4.2 Semigroup Problem

The Semigroup Problem is the closure under multiplication of a group of vectors [13]. The benchmark uses an unbalanced binary hash tree to store the vectors previously calculated so that lookups are efficient when computing the closure. Fig. 9 (PE graph) and Fig. 10 (time graph) were executed on five PEs. The total number of nodes in the reduction tree is 15,419, and the tree height h=174. In this experiment, the window size was 850×850 and the level distance was four. The maximum tree height to be displayed is calculated as $h_{max} = \lfloor \frac{850}{4} \times 2 \rfloor = 106$. Level condensing is performed because $h > h_{max}$. The first 38 levels are not condensed, but the remaining 136 levels are condensed by 2:1. This example demonstrates some strong points of VISTA: (1) After level condensing, the tree keeps its original shape; (2) The window-space efficiency is very good. If the tree were represented in a conventional way (propagating from the top of the window), representation would be difficult, and space efficiency would be poor.

To understand the parallelism characteristics of Semigroup, load balancing among processors is analyzed first. The most immediate characteristic of the PE graph is that the reductions form the shape of many spokes or *threads* of procedure invocations. Near the root are distribution nodes. Each thread represents a vector multiplication. As the graph shows, almost all threads were executed without task switch. This indicates few suspensions due to lack of data dependencies, i.e., the vectors are *not* produced in the pipelined fashion of the Pascal program. By eye, we judge that the five colors in the PE graph are evenly distributed, indicating that load balancing is good.

Lack of data dependencies between nodes is confirmed by a single-PE time graph (not shown). The color distribution of this graph is similar to that of Fig. 10, indicating that as soon as the first node of the new thread is spawned, both the child and parent threads were executed in parallel, without any suspensions. The reason that the threads are not executed clockwise or anticlockwise in Semigroup, as in Pascal, is that there were some initial data dependencies near the root. These dependencies, caused by hash-tree lookups for avoiding recomputation of a semigroup member, cause critical suspensions that "randomize" the growth pattern.

When the PE graph (Fig. 9) is viewed in conjunction with the time graph (Fig. 10), parallelism can be analyzed in more detail. Threads with the same colors in the time graph, and different colors in the PE graph, are executed in parallel. The PE graph still has a fair number of threads per PE, indicating that not all potential parallelism has been exploited and additional PEs will improve speedup. These approximations can be refined by examining subtree displays.

Historically, the Semigroup program analyzed above was the result of a number of refinements from an original algorithm written by N. Ichiyoshi [13]. Earlier algorithms utilize a pipeline process structure, wherein new tuples are passed through the pipe, and duplicates are filtered away. Any tuple surviving the pipe is added as a new filter at the end, and all of its products with the "kernel" tuples (the program's inputs) are sent through the pipeline. Although these algorithms are elegant, the pipeline structure is a performance bottleneck. The version analyzed above utilizes a binary tree instead of a pipeline, increasing the parallelism of the checks.

In retrospect, we see how VISTA could have helped in developing these successive algorithms. Fig. 11 shows the time graph for an older version of the program that has the same complexity as the program analyzed above. Thus the main difference is the pipeline bottleneck, which is clearly indicated by the signature's snail shape. Unlike Pascal, time is not projecting radially, indicating lack of wave parallelism. Successive tuples are dependent on previous tuples surviving the pipeline, and this dependency is seen in the coloring (it could be better viewed if the RGB spectrum were more distinguished). The dependency is made explicit by clicking on nodes to indicate the corresponding procedures. Fig. 10 radiates from the query, indicating the potential parallelism afforded by a tree vs. a pipeline. The coloring further indicates that the tree is not bottlenecked.

4.3 Instant Insanity

The Instant Insanity problem is to stack four four-colored cubes so that the faces of each column of the stack display all four colors. This is a typical all-solutions search problem with eight solutions. There are several methods for doing the search in a committed-choice language: most notable are candidates/noncandidates and layered streams [13]. The candidates method builds an OR-tree where each node concerns whether the current candidate is consistent with the current partial solution. At the root, all orientations of all cubes are candidates and the partial solution is empty. At the leaves, no candidates remain and the partial solutions are complete. Each node has two branches: one branch contains the solutions that include the current candidate, and the other branch contains solutions that do *not* include the candidate. Layered streams is a network of filters that eagerly attempt to produce a stream of solutions of the form `H*T`. Here `H` is the first element shared by a set of solutions, and `T` are the tails of these solutions. To throttle excessive speculative parallelism, a "nil check" is inserted at each filter to ensure that `T` must have at least one element.

Layered streams has 9,094 reductions (nil check) and 9,775 reductions (without nil check). This increase of 7% reductions is because of two factors: the additional speculative execution and the bloated conversion of the now largely incomplete layered stream back into normal form. Both of these effects are seen by comparing the VISTA graphs (Fig. 12 and 13). The conversion routine is clearly viewed as a significant subtree without the nil check, compared to a single thread with checking. The user can now appreciate the relative weight of the conversion with respect to the entire search. The speculative branches, however, do not stand out. This would be an interesting application of user-defined trace records, where a "trace dye" could be introduced with a nil check that does not throttle the speculation.

The candidates program has 37,687 reductions, so that VISTA must condense the image. The final image, shown in Fig. 14, has 25,127 nodes. Examining the structure and coloring of the layered-streams and candidates programs, there are no obvious parallelism bottlenecks in either (measurements of all three programs showed equal PE utilization of 93–95%). From the time graph coloring, the fine-grain parallelism of the filter structure is apparent in the layered streams program. The candidates graph shows large-grain structure, although we must view the time and PE graphs together to ensure that PEs are equally distributed across time.

The simple examples analyzed here facilitate the exposition of VISTA. Intuitions gained for these programs have been confirmed by timing measurements [13]. Programs without as much parallelism, and on larger numbers of PEs, can be similarly analyzed. As the number of PEs grows, however, the tool approaches its limitations because the user can no longer distinguish between the multiple colors representing the PEs. This is an important area of future research.

5 Conclusions and Future Work

This paper described the performance analysis of parallel logic programs using "kaleidescope visualization." The VISTA system is an X–Windows realization of the method, and is demonstrated in the context of parallel FGHC programs. We showed how the user can tune a large-trace program for performance by examining alternative abstract views of the execution. VISTA, because of its efficient implementation, proved its merit in enabling rapid analysis of views. This tool complements, but by no means replaces, other visualization methods, e.g., animation of PE activity and message passing.

We are currently extending this research in several areas. First, we need to experiment more with the current VISTA prototype, for various programming languages, to determine its utility. Second, coloration methods for combining the time and processor views need exploration, e.g., a method of spectral superposition [1].

The author was supported by an NSF Presidential Young Investigator award, with funding from Sequent Computer Systems Inc. Computer resources were supplied both by OACIS and Argonne MCS. D.-Y. Park, in an outstanding effort, implemented VISTA.

[1] J. A. Berton. Strategies for Scientific Visualization: Analysis and Comparison of Current Techniques. *Proceedings of Extracting Meaning from Complex Data: Processing, Display, Interaction*, SPIE vol. 1259, pages 110–121. February 1990.

[2] T. Disz and E. Lusk. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In *Inter. Symp. on Logic Prog.*, pages 46–53. IEEE Computer Society, August 1987.

[3] T. Disz *et al.* Experiments with OR-Parallel Logic Programs. In *Inter. Conf. on Logic Prog.*, pages 576–600. MIT Press, May 1987.

[4] Y. Feldman and E. Shapiro. Temporal Debugging and its Visual Animation. In *Inter. Symp. on Logic Prog.*, pages 3–17. MIT Press, November 1991.

[5] R. Fowler *et al.* An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. *SIGPLAN Notices*, 24(1):163–173, January 1989.

[6] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[7] K. M. Kahn and V. A. Saraswat. Complete Visualization of Concurrent Programs and their Executions. In *IEEE Visual Language Workshop*. IEEE Computer Society, October 1990.

[8] E. Lusk *et al.* The Aurora Or-Parallel Prolog System. In *Inter. Conf. on Fifth Gen. Comp. Systems*, pages 819–830, Tokyo, November 1988. ICOT.

[9] A. D. Malony and D. Reed. *Visualizing Parallel Computer System Performance*, pages 59–90. Addison-Wesley, 1990.

[10] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

[11] R. Scheifler and J. Gettys. The X Window System. *ACM Trans. on Graphics*, 5:79–109, April 1986.

[12] D. Socha *et al.* Voyeur: Graphical Views of Parallel Programs. *SIGPLAN Notices*, 24(1):206–215, January 1989.

[13] E. Tick. *Parallel Logic Programming*. MIT Press, Cambridge MA, 1991.

[14] E. Tick and D.-Y. Park. Kalaidescope Visualization of Fine-Grain Parallel Programs. In *Hawaii Inter. Conf. on System Sciences*, vol 2, pages 137–148. Kauai, IEEE Computer Society, January 1992.

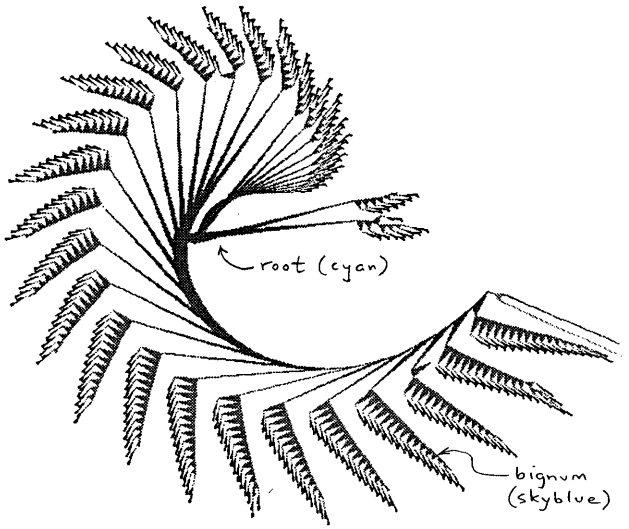


Figure 5: Graph of Pascal from Procedure View (5 PEs)

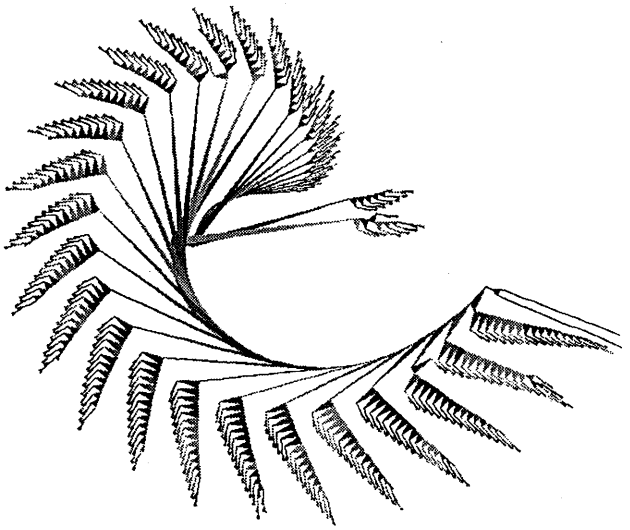


Figure 6: Graph of Pascal from PE View (5 PEs)

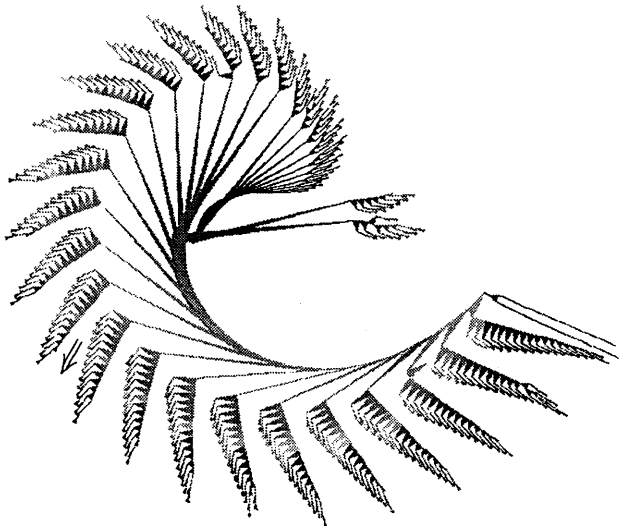


Figure 7: Graph of Pascal from Time View (5 PEs)

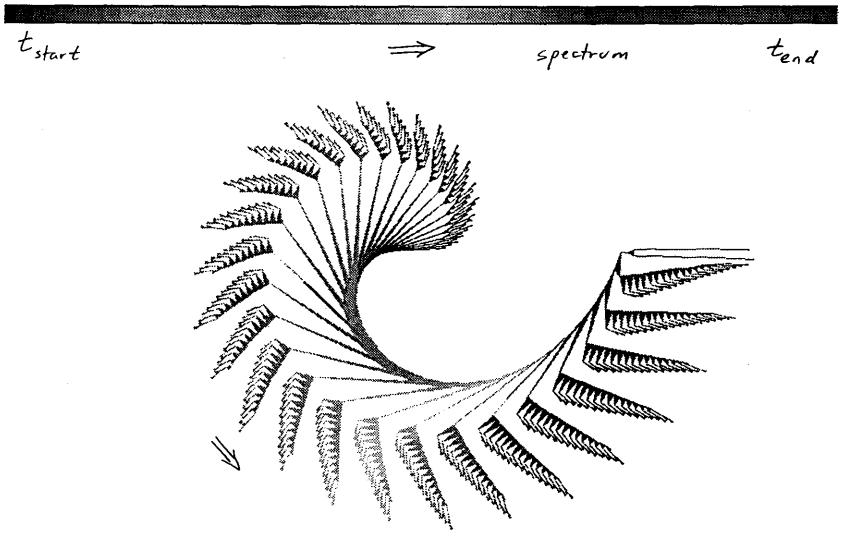


Figure 8: Graph of Pascal from Time View (1 PE)

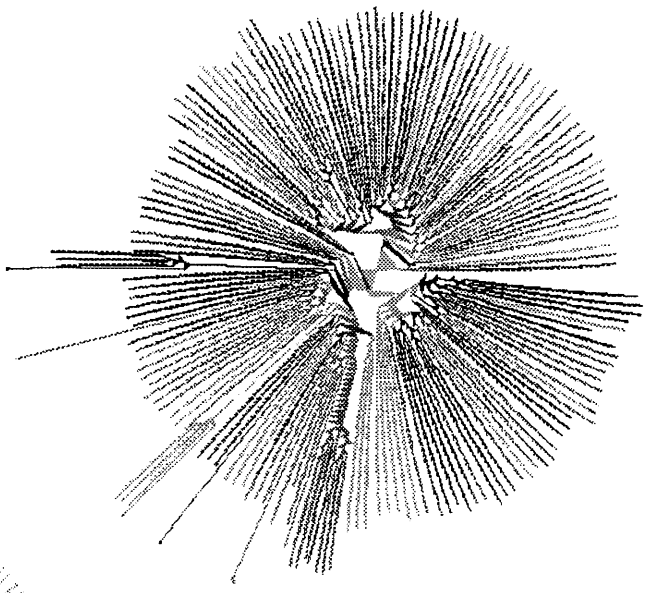


Figure 9: Graph of Semigroup from PE View (5 PEs)

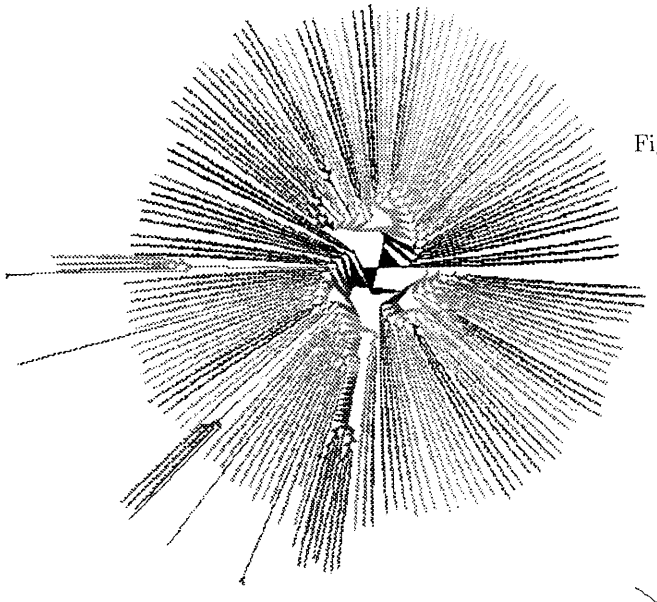


Figure 10: Graph of Semigroup from Time View (5 PEs)

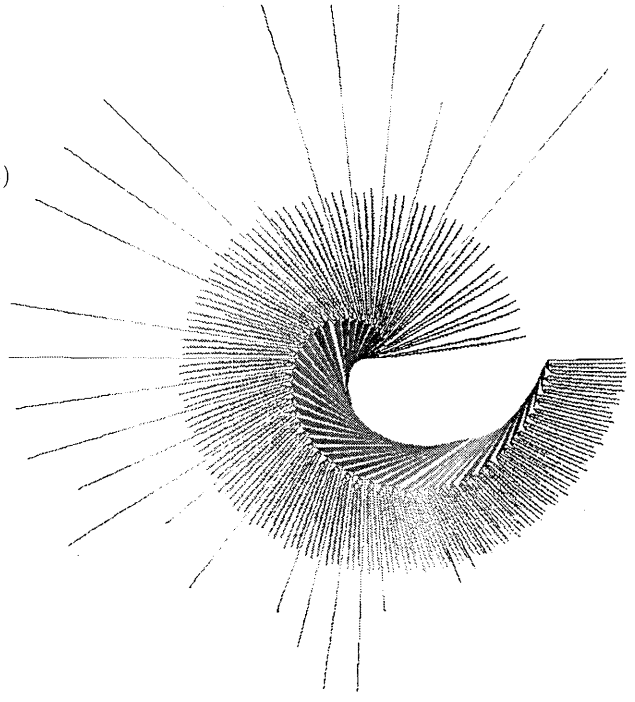


Figure 11: Graph of Old Semigroup Algorithm from Time View (5 PEs)

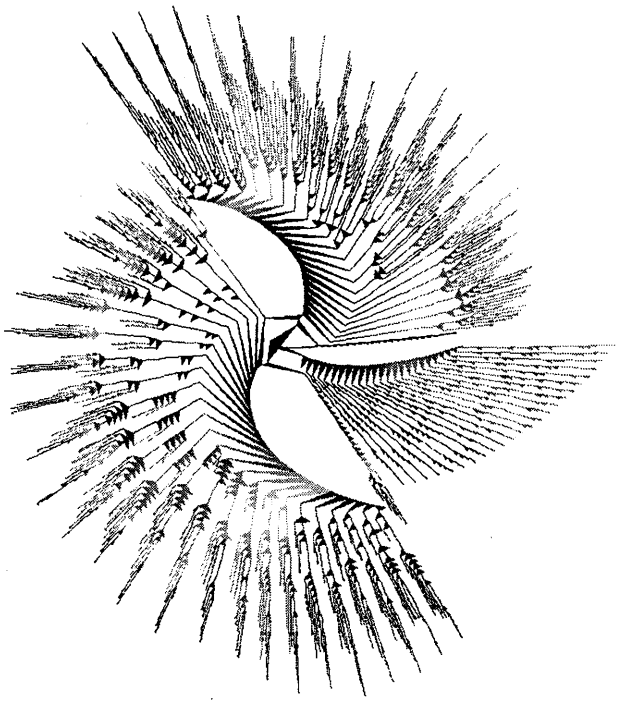


Figure 12: Graph of Layered Stream Cubes with Nil Check from Time View (5 PEs)

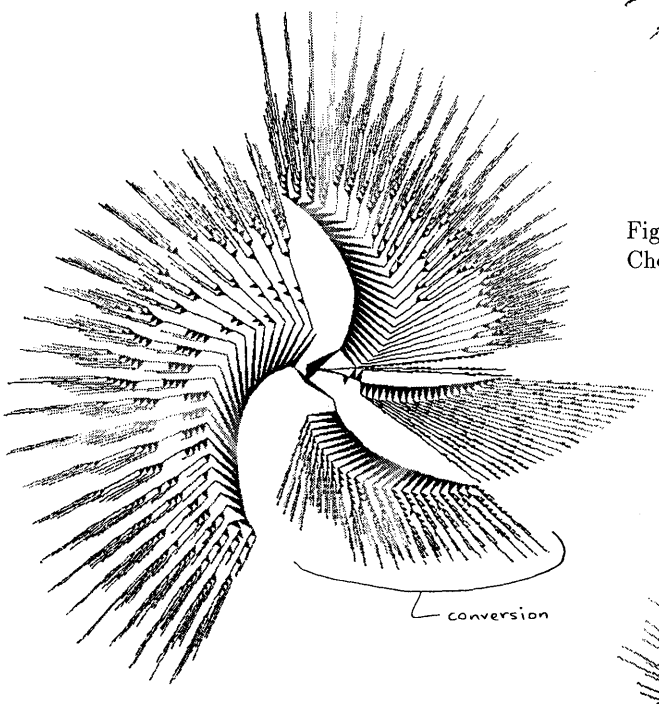


Figure 13: Graph of Layered Stream Cubes without Nil Check from Time View (5 PEs)

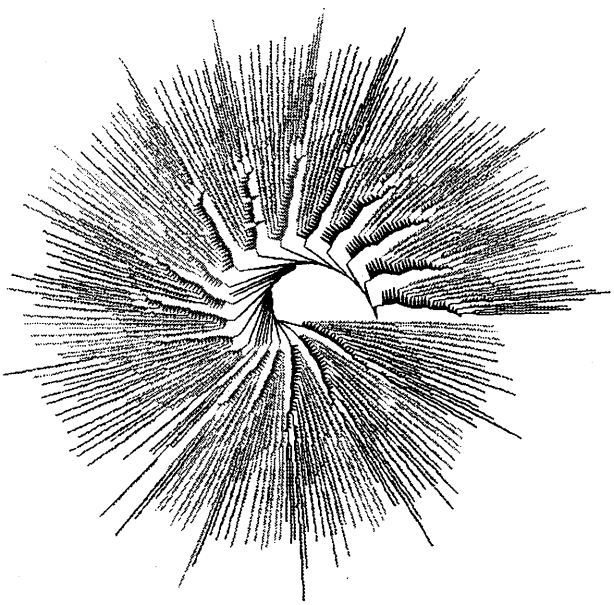


Figure 14: Graph of Candidates Cubes from Time View (5 PEs)

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs

Kenneth M. Kahn
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

kahn@parc.xerox.com

Abstract

The design and implementation of a visual programming environment for concurrent constraint programming is described. The system is implemented in Strand, a commercially available concurrent logic programming language. Three components are now operational and are described in detail in this report; they are a parser, fined-grained interpreter, and animator of Pictorial Janus programs. Janus, a concurrent constraint programming language designed to support distributed computation, has much in common with concurrent logic programming languages such as FGHC. The design of a visual syntax for Janus called Pictorial Janus is described in [KS90].

Visual programs can be created using any illustration or CAD tool capable of producing a PostScript description of the drawing. The Pictorial Janus Parser interprets traces of PostScript executions and produces a textual clausal version of the parsed picture which can be converted to Strand and run as an ordinary Strand program.

The parser can also produce input to the Pictorial Janus Interpreter. The interpreter accepts as input a term representing the program clauses and query. This term is annotated with the colors, shapes, fonts, etc. used in the original drawing. It spawns recurrent agents corresponding to each agent (i.e. process or goal), rule (clause), message (term), port (variable), link (equality relation), and channel. These agents interact to do the equivalent of clause reduction. The agents also produce streams of major events (e.g. that some message moved and rescaled to the location and size of some other message). These streams are merged and fed into the Pictorial Janus Animator.

The animator generates a stream of animation frames and associated sounds. The resulting frames can then be printed; more importantly, they can be converted to a raster format and recorded on video tape or animated on a work station. The colors, shape, fonts, line weights, used in the original drawing are preserved so that the animation displays these elements in the same graphical terms as they were conceived and created.

Various lessons were learned in the process of constructing the system, ranging from parallel performance issues, to deadlock, to trade-offs between the use of terms and agents.

1 Introduction

This paper presents a software architecture in which concurrent constraint (or logic) programming plays a predominant role. The structure of this software and the programming techniques used are described, and problems that arose and the resulting redesigns are discussed. This paper is primarily about a large concurrent logic program and the fact that the program is one which supports a programming environment for concurrent constraint programming is unimportant to this paper. The purpose of this paper is rather to relate experiences in writing a large, complex, and somewhat unusual application in a concurrent logic programming language. Much of the discussion centers around difficulties in applying, adapting, and choosing between well-known concurrent logic programming techniques. Other papers are in progress which present the visual programming environment.

The software described is part of a "grand plan" in which parsers, editors, source transformers, visualizers, animators, and debuggers all work together to support a programmer in constructing, maintaining, and understanding concurrent constraint programs in a completely visual manner. This work is driven by the belief that such an environment can have a dramatic impact on the way in which software is developed.

The grand plan is to support whole families of concurrent constraint languages, including the familiar Herbrand family, which includes FGHC [Ued85], Strand [FT89], and Andorra Prolog [HJ90]. We also anticipate supporting constraint systems other than the traditional Herbrand constraints of logic programming. Initially the system is being built to support only a pictorial syntax for Janus [SKL90], a concurrent constraint language designed to address some of the needs of distributed computing. Janus most closely resembles DOC [Hir86], Strand, and FGHC.

An important aspect of the pictorial syntax of Janus is that it is a complete syntax (i.e. anything expressible in textual Janus is expressible in Pictorial Janus) and that the syntax is based upon the *topology* of pictures. For example, a port (i.e. a variable occurrence) is represented by any closed contour which has no other elements of the picture inside. A programmer is free to choose any size, color, shape, etc. for the elements of the program. The syntax of Pictorial Janus is discussed in greater detail in [KS90]. A simple example

program that appends two lists is shown in Figure 1.

Computation is visualized as the reduction of asynchronous agents. The rules for each agent are inside it. If at least one of these rules can match, then the agent reduces. A matching rule expands and its "ask" devices (its head and guard) match the corresponding devices attached to the agent. The rule is then removed, and its body remains connected to the pre-existing configuration by links. These links represent equality relations and are collapsed bringing the ports at each end together.

The matching of an append agent with its recursive rule is shown in Figure 2. The matching rule contour expands to match the contour of the agent. The messages and ports rescale and translate to match the corresponding ports and messages of the agents. In Figure 3 the commitment of a rule is depicted. It shows the agent and the matched elements dissolving away leaving the configuration in the body of the rule connected to the configuration of the computation. Figure 4 shows changes which have no semantic meaning and are performed to tidy up the picture. Links in the configuration establish equality relations between ports and can be shrunk to zero, thereby bringing the equivalent ports together. Newly spawned agents are scaled.

2 Pictorial Janus System Architecture

Figure 5 attempts to capture the essential modules and data of a complete Pictorial Janus programming environment. It depicts the various processing stages which take Pictorial Janus program drawings to either a textual form for ordinary compilation or to animations of its execution.

Source programs are drawings in PostScript. PostScript is well-suited for this because of its ability to describe curves, colors, fonts, etc. in a flexible and general manner. Since PostScript is a common page description language for laser printers, every modern illustration or computer-aided design program is capable of producing a PostScript description of a drawing. This is analogous to the situation in textual programming where the text file for a program can be produced by any text editor. An alternative to PostScript input yet to be explored is a custom structure editor that only allows the construction of syntactically correct pictorial programs and can maintain a semantic representation of the program. Another source of PostScript is from automatic tracing tools such as Streamline from Adobe which converts scanned images of hand-made drawings into PostScript strokes.

The problem of discovering the underlying program from a PostScript description is complicated by the fact that PostScript is a full programming language. This is analogous to the situation in conventional languages with sources which require pre-processing. Such sources are not parsed by a compiler; instead the output of a pre-processor run on those sources is. We handle this by executing the PostScript with an ordinary PostScript interpreter in an environment which redefines the graphical primitives that draw strokes,

```
rule(a57(272,485,308,521),
   append(port(p61(box(273,489,276,492))),
          port(p59(box(276,516,279,519))),
          port(p63(box(309,501,312,504)))),
   [equal(c6(312,504,324,516)
          port(p63(box(309,501,312,504))),
          $(port(p65(box(324,516,327,519))))),
    equal(m55(262,488,277,493),
          port(p61(box(273,489,276,492))),
          [])],
   [equal(l5(280,518,324,518),
          port(p59(box(276,516,279,519))),
          port(p65(box(324,516,327,519))))],
          [])
```

Figure 6: Annotated Janus Parse of the base case of Append

show text, etc. to, instead, print a trace of their calls to a file.

The Pictorial Janus Parser is the module which accepts such traces of calls to PostScript graphical primitives and produces a parse in a format called "annotated Janus". This format captures the parse tree of the program picture and maintains correspondences with the original graphical appearances. These correspondences consist of annotations which give the animator guidance in choosing the appearance, position, and scale of various program elements. They are ignored if the program is simply to be compiled and executed without the production of an animated trace. Annotated Janus is the "lingua franca" of the system. It can be produced by the parser, by a visualizer from textual Janus to Pictorial Janus, by a custom structure editor, or by a program transformation tool. It can be used by visual debuggers, animators, or program transformation tools, or it can be converted to textual Janus for ordinary compilation and execution. Figure 6 contains the annotated Janus for produced by parsing the base case rule of append in Figure 1. (Constants such as "p61" also name PostScript drawing procedures.)

A central component of the system is a fine-grained interpreter for annotated Janus. As it interprets the program it produces a stream of events describing activities for each element of the computation (i.e. each agent, rule, port, channel, message, etc.). The event descriptions include a start and end time. By default, the interpreter performs every reduction as soon as possible. This corresponds to a maximally concurrent scheduler. The scheduler can currently be customized to some extent. It can follow a schedule based upon the trace of real execution on a parallel machine or network.

The third major component is the Pictorial Janus Animator. It accepts the stream of event descriptions from the fine-grained interpreter and some layout control and produces PostScript describing each individual frame. This PostScript can be printed, converted to raster for viewing or video taping, or converted to film.

Other components of the system such as the "visualizer" which converts textual Janus to Pictorial Janus and a spe-

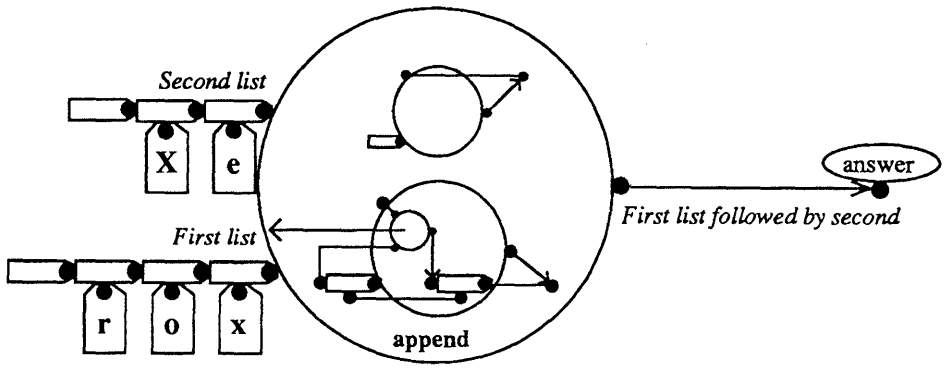


Figure 1: A Simple Example Program to Append Lists

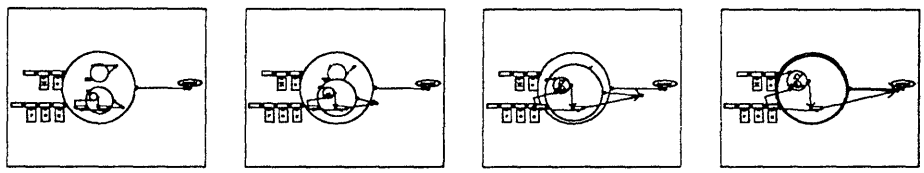


Figure 2: The Animation of a Sucessful Rule Match

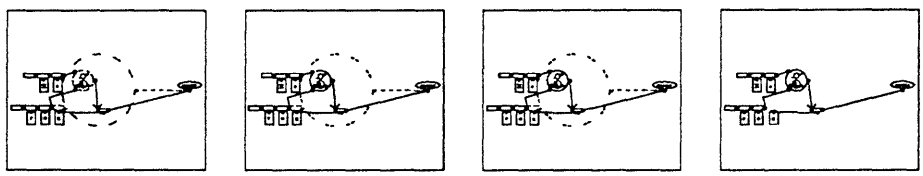


Figure 3: The Animation of a Rule Commitment

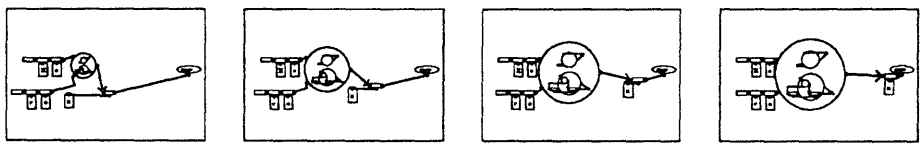


Figure 4: The Animation of Links Shrinking and Agents Rescaling

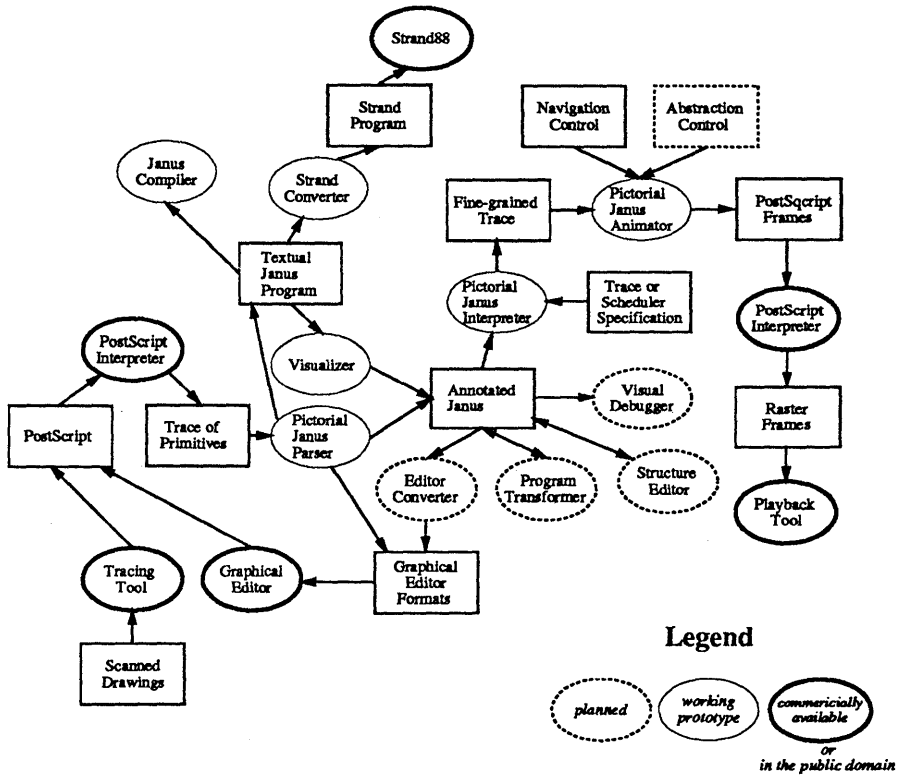


Figure 5: Overall Architecture of the System

cialized pen-based editor for Pictorial Janus are under development and will be discussed in future papers. Additional components such as an interactive visual debugger, a program transformation, and a partial evaluation system based upon Pictorial Janus are only in the planning stages.

The three major components (the parser, interpreter and animator) are operational prototypes and are discussed in further detail in the rest of this paper.

```
ps(stroke,4,
    [moveto(496,483),
     curveto(496,487,485,490,472,490),
     curveto(459,490,448,487,448,483),
     curveto(448,479,459,475,472,475),
     curveto(485,475,496,479,496,483)],
    box(448,475,496,490),
    [eofill,setrgbcolor(1,1,1),setdevicelinewidth(1)]).
```

Figure 7: Sample Trace of PostScript Execution

3 Pictorial Janus Parser

The parser begins with an unordered set of line and curve segments and located text which is the trace from executing a PostScript description of a Pictorial Janus drawing (see Figure 7. These elements are analogous to the "alphabet" of the language. The first phase of the parser is a sort of "tokenizing" where abutting curves are joined, closed curves detected, and arrows recognized. As with most of the phases, some tolerance for sloppy drawings is allowed.

To reduce the complexity of further operations, a containment tree of the elements is constructed. The containment tree associates with each closed curve the sets of closed curves directly contained within, as well as the end points of open curves and arrows and the text directly contained within. Since many parsing decisions depend upon which elements are closest to other elements in the same containment level, the containment tree reduces the amount of search necessary.

The parse proceeds by a series of phases.

- *Identification of Ports.* A port is defined as a closed contour with nothing inside of it. Once the containment tree is completely built this is a trivial test.

- *Classification of Arrows.* Arrows are used for two purposes in Pictorial Janus: channels between ports (i.e. distinguishing between the asker and teller occurences of variables) and the association of agents with rules. Once ports are identified it is easy to distinguish the two cases since channels are arrows between ports while definition arrows connect agents.

- *Attachment of Ports.* Ports cannot be free-standing; they must be attached to either an agent contour, a rule contour, a message contour, or the head of an channel arrow. Essentially the port attachs to the nearest syntactically correct element. This phase also identifies which ports are the internal ports of messages. Messages are identified by having only an internal port and possibly a label inside.

- *Connecting Links.* Open curves depict links which connect ports. This phase determines which port is closest to the end of a link. The connecting port can be up or down one containment level from the level of the link end.

After these phases have completed, the parser can generate Annotated Janus or textual Janus by descending from the top node in the containment tree and collecting information. Agents are distinguished from rules here by alternating containment levels (i.e. the top level contains agents which contains rules which contain agents and so on). Ports of agents, rules and messages are collected into a list by going clockwise from a distinguished port.

Early versions of the parser represented picture elements by terms. Initially, little was known about the terms, so they contained many unbound logic variables for their role, their attached ports, their label, etc.. Lists posed problems since it can't be known beforehand how many elements they have. If tails are left uninstantiated then at the phase where no more elements can be added, some process must find these tails and bind them to the empty list. The lists are constructed in the order the elements were discovered; another logic variable was needed to hold the sorted list. This implementation became more cumbersome and was forced to rely upon some questionable primitives.

Because of these problems, the parser was completely rewritten to represent picture elements by recurrent agents (processes). Lists are no longer a problem since agents can simply recur with a different list. Sorting the list is equally straight forward.

This use of recurrent agents is an object-oriented programming style [ST83]. Unlike traditional object-oriented programming systems, however, the underlying flexibility of concurrent logic programming can be used to incrementally refine the type or class of elements. This might be called "object-oriented recognition". Closed contours, for example, begin as generic "vanilla" nodes. As relationships are discovered, a node may specialize itself to a port or non-port. Figure 8 is a sketch of the code for determining whether a node is a port or not.

Similarly non-ports may be further specialized as messages, rule contours, or agent contours depending upon their local relationships. The locality in this case is between a contour and the elements directly contained within and elements contained in the same contour.

In most object-oriented systems an instance cannot easily (or at all) change to be the instance of another class, even if that class is subclass of the original class.

```
node(In,Contents,State) :-
        In = [identify_ports|In'],
        Contents = [] |
        port(In',State).

node(In,Contents,State) :-
        In = [identify_ports|In'],
        Contents ≠ [] |
        non_port(In',Contents,State).
```

Figure 8: An Example of Incremental Class Refinement

4 Pictorial Janus Interpreter

A detailed trace of every reduction in a computation is needed by the animator. A meta-level interpreter is too coarse for this purpose. Instead a fine-grained interpreter which can report events such as each subterm match is needed.

The fine-grained interpreter of Pictorial Janus is constructed out of recurrent agents. Agents are spawned which represent each element of a program or configuration (programs and configurations are treated identically). There are agents for each rule (clause), port (variable), message (term), link (equality relation), channel (asker and teller pairs) and agent (process). They emulate the ordinary execution at a message-passing level. An agent reduces by spawning an arbiter and sending a message to each of its rules. The rules reduce by sending match messages to each of their ports with streams to the corresponding ports of the agent. If all of the ports respond with a possible match then the rule sends a message to the arbiter. The first rule to send a message to the arbiter then commits and the others eliminate themselves. A committing rule spawns new agent, port, rule, message, channel and link agents.

The agents of the fine-grained interpreter also generate a stream of events. For example, when a rule commits it produces two event descriptions. The first indicates that the rule contour should transform to match the contour of the agent it is reducing. As with all event descriptions, it also indicates the start and stop time for this activity. These times are computed based upon a specification of the scheduler. The second event describes the removal of the rule. All the event streams are merged to produce a time-ordered stream of events.

One problem with the fine-grained interpreter is how it interprets pictorial programs which deadlock. Each rule agent suspends, waiting to hear from its ports how the match went. A port in turn passes the match request to its attached message. The message asks the corresponding port of the reducing agent for a description of its attached message. If there is no message there then the whole collection of rule, message and port agents suspend until a message is

connected to the port. In a deadlocked computation there never will be an attached message. These suspended agents are unable to produce events, which in turn prevents the ordered merge process from producing events; the whole production of the stream of events is cut off.

In Strand it is possible to work around this by relying upon the questionable "idle" guard that suspends until the whole system is idle. The message agents waiting for a response from the corresponding message which are also part of an idle system can then proceed to return a match failure or signal an exception and the interpretation can proceed. It is possible to detect deadlock in a more principled manner [SWKS88], but the price is a significantly more complex and verbose program.

A related problem is controlling the arbiter between competing rule commitments. For example, a merge agent with inputs on both incoming ports can reduce with either rule. Which rule is chosen depends upon which one is the first to get a message to the arbiter of the reduction. Consequently, the fine-grained interpreter selects between competing clauses depending upon the scheduler of the underlying Strand implementation. When run on a single processor this means that the same rule is always chosen. To make more interesting animations a random number generator was needed to remove these biases.

5 Pictorial Janus Animator

The Pictorial Janus Animator consumes the stream of events produced by the fine-grained interpreter. It also can be given layout and viewpoint instructions. It produces a stream of animation frames in PostScript. The animator currently models space as a sequence of ten planes. The graphics of lower planes can be obscured by the graphics of higher planes. The planes are infinite in extent but only a portion is "viewed" at any one time.

The animator accepts event descriptions describing events whose times are described by real numbers. Given a frame rate (i.e. a sampling rate), these are converted to frame numbers. The animator is like a discrete-time simulator where on every "tick" every component needs to compute its next state.

For each kind of event, the animator has methods for depicting it. A typical method might transform one element to gradually match another (currently the transform involves translation, scaling, and rotation). For example, a message matches another message by incrementally changing its position and size until it has the same bounding box as the other message. The other message may be changing and the animator needs to adjust the transformation accordingly. Furthermore, the method must maintain various constraints on the matching message contour so that it remains in contact with other elements. In order for a method to transform an element based upon the position of others, the animator maintains transformation "histories" for each picture element. The history of a visual element is a list of transform matrices, one for each frame. A frame is constructed by selecting from each history the appropriate transformation to apply to the appearance of each element.

The histories are also used to deal with graphical interactions between elements. For example, if a port is to transform itself to match another port which itself is moving, then on each frame the position of the tracking port is a function of where it is, where the other port is, and the amount of time before they meet. An interesting alternative is that it is a function, not of where the other port is on each frame, but where it will be at the time of the meeting. As illustrated in Figure 9, the former corresponds to one port chasing another, while the later is more like a rendezvous. Generally, the rendezvous looks better but it requires "knowledge of the future". With care it is possible to avoid cycles of such requirements of future knowledge that would lead to a deadlock.

The first time the animator was run on a large problem (i.e. one requiring several million reductions), it ran out of memory. Increasing the amount of available memory to 30 or 40 megabytes helped but then it ran out again for somewhat larger tasks. The cause of this kind of problem is very difficult to track down. After much experimentation it was discovered that the problem was that agents inside the animator were producing information faster than other agents were able to consume it. Memory was being used to "buffer" the messages from the producers to the lagging consumers.

This is a well-known problem and there is a well-known concurrent logic programming technique called "bounded buffers" [TF87] for dealing with it. The simple case of a single producer and a single consumer is rare in the animator and a more complex variant was needed to deal with consumers of streams that have multiple producers (typically combined by an ordered merge). This fixed the problem but significantly increased the complexity of the source code. Many subtle bugs cropped up which eventually were traceable to some piece of code not following the bounded-buffer protocol correctly. These were hard to debug because they resulted in deadlocks of thousands of agents.

Another shortcoming of using bounded-buffers is that it is difficult to tune for different language implementations and hardware platforms. Under some schedulers all this complexity is unneeded because the scheduler runs consumers first. To both simplify the code and increase its flexibility, the bounded buffer technique was abandoned and instead each agent was programmed to know the animation frame number it is contributing to and which was the last frame to be completed. Producers are now controlled by an integer indicating how many frames ahead they are allowed to proceed. If this is set to a number larger than the total number of frames in the animation, then buffering is effectively turned off. The use of frame numbers to control producers is easy to generalize to other problem domains such as simulation, but it is not as general as the bounded-buffer technique.

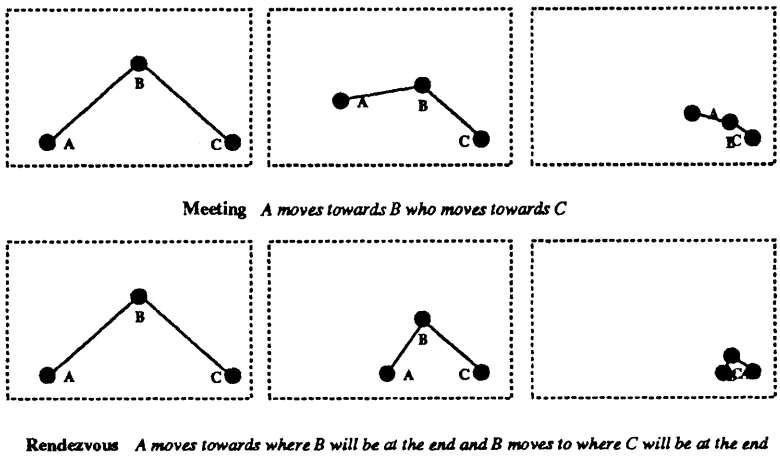


Figure 9: An Illustration of a Chase in Contrast to a Rendezvous

When bounded buffers were first introduced the animator would deadlock sometimes. After a few days of investigation it turned out that the problem was an interaction with the method for having two ports meet. Recall that there are two alternatives: "chase" and "rendezvous". Rendezvous requires knowledge of where the other port will be at the end of the event. It turned out that the system was deadlocking whenever the buffer size was smaller than the number of frames needed to animate a port meeting. Once discovered it was easy to conditionalize the meet method to use the rendezvous style if the buffers are large enough and otherwise use the chase style.

Concurrent logic programs can be written without carefully ordering events since the basic computational mechanism reorders events based upon data dependencies. This greatly simplified the construction of both the fine-grained interpreter and the animator. Each event can be handled independently regardless of whether the data it needs from other concurrent activities has been produced. In a sequential language the programs would have to have been carefully constructed so that, say, the agent contour changes are computed before the dependent changes on their ports.

6 Preliminary Performance Results

The parser, interpreter and animator are implemented in over 11,000 lines of Strand code. The only important component in C is a routine which finds the closest point on a Bezier curve to another point. A typical parse takes a few CPU minutes (on a SUN Sparc 2). The interpreter typically takes a few CPU minutes as well. The animator typically takes tens of CPU minutes (10 to 20 million reductions is not uncommon).

The sequential execution of the system cannot be sped up much by optimizing the Strand code or replacing it with C. For the parser nearly half of its time is spent in the C routine for finding the closest point to a curve. The Strand code of the animator takes a third of the total time to produce an animation; the PostScript rendering to raster takes up the rest.

It would seem that parallel execution should speed up the system significantly. Preliminary results have, however, been disappointing. Possible reasons include:

- *Communication costs.* The coding style used strived for maximal parallelism but little attention was paid to the amount of information passed between agents. On a good shared-memory implementation, this would not be a factor. There are many cases where much of this communication can be programmed away. For example, rather than communicate large shared structures between agents, each processing node could have its private copy and the messages between nodes would just contain tokens referring to elements of these structures. This rewriting has yet to be done. It would also be counter to the dream of concurrent constraint programming (including the special case of concurrent logic programming) that straight-forward high-level portable programs can run efficiently in different environments without major revision. Some rewriting has been done to enable experimentation with parallelism. For example, the output of the animator previously was a large PostScript file and now is a set of files, one for each frame.

- *Agent to Processor mappings.* Experiments to date have used agent-to-processor mapping annotations. While a few different mappings have been tried it is possible that a good one exists but has yet to be discovered. No experiments using a load balancing scheduler have been tried.

Speedups of a factor of 2 to 3 were easily obtained by spawning Unix-level processes to convert PostScript to raster format on separate processors in parallel.

7 Conclusions and Future Work

The building of a large prototype visual programming environment in a concurrent logic programming language was described. The architecture was presented and some experiences and lessons learned were described. These lessons range from the trade-offs between using messages (terms) and recurrent agents, to difficulties with producers getting too far ahead of consumers, to dealing with deadlock.

For sequential executions the overhead of using a concurrent logic programming language was small. For parallel executions on distributed memory machines, speedups are not readily available and appear to require program rewriting and/or very clever distributions of agents and data to processors.

The system is under development. Current plans include extending the animator to deal with both spatial and temporal abstractions. The animator needs to deal better with the layout of elements. The parser needs to be revised to deal robustly with hand-drawn input. Support for primitives and foreign procedure calls are needed. The interpreter needs to be able to accept general scheduler specifications. The animator is currently able to produce a simple sound track synchronized with the animation. The sounds depend upon the kind of activities occurring. This should be extended to differentiate between different elements involved in the activities.

A very challenging direction for future development is to build a "real-time" version of the system that the user can influence as the computation proceeds. This could lead to very powerful debugging tools. It could also be the basis for user interfaces that are simultaneously interactive visual programs. Such a system would need to run on platforms capable of many millions of reductions every second.

8 Acknowledgements

The design of this system benefited from discussions with Vijay Saraswat and Volker Haarslev. I am grateful to Mary Dalrymple, Vijay Saraswat, and Markus Fromherz for comments on earlier drafts of this paper.

References

[FT89] Ian Foster and Stephen Taylor. Strand: A practical parallel programming language. In *Proceedings of the North American Logic Programming Conference*, 1989.

[Hir86] Masahiro Hirata. Programming language doc and its self-description, or, x=x considered harmful. In *3d Conference Proceedings of Japan Society for Software Science and Technology*, pages 69–72, 1986.

[HJ90] Seif Haridi and Sverker Janson. Kernel andorra prolog and its computation model. In *Proceedings of the Seventh International Conference on Logic Programming*, June 1990.

[KS90] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the IEEE Visual Language Workshop*, October 1990.

[SKL90] Vijay A. Saraswat, Kenneth Kahn, and Jacob Levy. Janus—A step towards distributed constraint programming. In *Proceedings of the North American Logic Programming Conference*. MIT Press, October 1990.

[ST83] Ehud Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog. *New Generation Computing*, 1:25–48, 1983.

[SWKS88] Vijay A. Saraswat, David Weinbaum, Ken Kahn, and Ehud Shapiro. Detecting stable properties of networks in concurrent logic programming languages. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC 88)*, pages 210–222, August 1988.

[TF87] A. Takeuchi and K. Furukawa. *Concurrent Prolog: Collected Papers*, volume I, chapter Bounded Buffer Communication in Concurrent Prolog, pages 464–476. The MIT Press, 1987.

[Ued85] K. Ueda. Guarded Horn Clauses. Technical Report TR-103, ICOT, June 1985.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Logic Programs with Inheritance

Yaron Goldberg, William Silverman, and Ehud Shapiro

Department of Applied Mathematics and Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

Abstract

It is well known that while concurrent logic languages provide excellent computational support for object-oriented programming they provide poor notational support for the abstractions it requires. In an attempt to remedy their main weaknesses — verbose description of state change and of communication and the lack of class-like inheritance mechanism — new object-oriented languages were developed on top of concurrent logic languages.

In this paper we explore an alternative solution: a notational extension to pure logic programs that supports the concise expression of both state change and communication and incorporates an inheritance mechanism. We claim that combined with the execution mechanism of concurrent logic programs this notational extension results in a powerful and convenient concurrent object-oriented programming language.

The use of logic programs with inheritance had a profound influence on our programming. We have found the notation vital in the structuring of a large application program we are currently building that consists of a variety of objects and interfaces to them.

1 Introduction

We share with the Vulcan language proposal [8] the view on the utility of concurrent logic languages as object-oriented languages:

> "The concurrent logic programming languages cleanly build objects with changeable state out of purely side-effect-free foundations. [...] The resulting system has all the fine-grained concurrency, synchronization, encapsulation, and open-systems abilities of Actors [4]. In addition, it provides unification, logic variables, partially instantiated messages and data, and the declarative semantics of first-order logic.
>
> Abstract machines and corresponding concrete implementations support the computational model of these languages, providing cheap, light-weight processes [...] Since objects with state are not taken as a base concept, but are built out of finer-grained material, many variations on traditional notions of object-oriented programming are possible. These include object forking and merging, direct broadcasting, message stream peeking, prioritized message sending, and multiple message streams per object."

See also [7] for a more recent account of the object-oriented capabilities of concurrent logic programs.

We also share with the designers of Vulcan the conclusion that:

> "While [concurrent logic languages] provide excellent computational support [for object-oriented programming], we claim they do not provide good notation for expressing the abstractions of object-oriented programming."

However, we differ in the remedy. While Vulcan and similar proposals [2,13] each offer a new language, whose semantics is given via translation to concurrent logic languages, we propose a relatively mild notational extension to pure logic

programs, and claim that it addresses quite adequately the needs of the object-oriented programmer. Specifically, our notation addresses the main drawbacks of logic programs for object-oriented programming: verbose description of objects with state, cumbersome notation for message sending and receiving, and the lack of a class-like inheritance mechanism that allows the concise expression of several variants of the same object. We explain the drawbacks and outline our solutions.

Inheritance

In certain applications, most notably graphical user interfaces, many variants of the same object are employed to cater to various user needs and to support smooth interaction. In the absence of an inheritance mechanism, a variant of a given object must be defined by manually copying the description of the object and editing it to meet the variant specification. Both development and maintenance are hampered when multiple copies of essentially the same piece of code appear within a system. Class-based inheritance mechanisms provide the standard solution for defining multiple variants of the same basic object in a concise way, without replicating the common parts. In this paper we propose an inheritance mechanism for logic programs, called *logic programs with inheritance*, or *lpi* for short.

The idea behind *lpi* is simple. When a procedure p inherits a procedure q via the inheritance call $q(T_1, \ldots, T_k)$, add to p's clauses all of q's clauses, with the following two basic modifications:

1. Replace the head predicate q by the head predicate p, with "appropriate arguments".

2. Replace recursive calls to q by "corresponding calls" to p.

The formal definition of *lpi* will make precise the meaning of the terms "appropriate arguments" and "corresponding calls". The effect of inheritance is that behaviors realizable by q are also realizable by p, as expected.

Using common object-oriented terminology, *lpi* can be characterized as follows:

- *Predicates are classes*: Logic program predicates are viewed as *classes*, and procedures (*i.e.*, predicate definitions) as *class definitions*. Classes can be executed as well as inherited; that is, superclasses are executable in their own right.

- *Clauses are methods*: In the concurrent reading of logic programs, each clause in a procedure specifies a possible process behavior. Inheriting a procedure means incorporating appropriate variants of the clauses of the inherited procedure into the inheriting procedure.

- *Multiple inheritance*: A procedure may inherit several other procedures.

- *Parameterized inheritance*: Inheritance is specified by "inheritance calls", which may include parameters. Hence an inheriting procedure may inherit the same procedure in several different ways, using different parameters in the inheritance calls.

We shall see examples for these features in the following sections.

We note that the inherent nondeterminism of logic programs (and the inherent indeterminacy of concurrent logic programs) can accommodate conflicts in inherited methods with no semantic difficulty. If necessary, an overriding mechanism can be incorporated to enforce a preference of subclass methods over superclass ones.

Implicit Arguments

Objects with state, accessible via messages, are realized in concurrent logic programs by recurrent processes. Typically, such a recurrent process has one or more shared variables and zero or more private state variables. The expression of a recurrent process by a concurrent logic program has the general form:

$p(\ldots$ old state of process $\ldots) \leftarrow$
$\ldots$ message receiving and sending, $\ldots$
$p(\ldots$ new state of process $\ldots)$.

When a process has several variables, where only a few of them are accessed or changed on each process reduction, then the notation of plain logic programs becomes quite cumbersome. This is due to the need to specify explicitly all the variables that do not change in a process's state transition twice: once in the "old" state in the clause head, and once in the "new" state in the clause body. In addition, different names need to be invented for

the old and new incarnations of a state variable that did change in the transition.

This verbose syntax introduces the possibility of trivial errors and reduces the readability of programs, since it does not highlight the important parts (what has changed) from other details (repetition of the unchanged part).

We define a notational extension, independent of the inheritance notation, called *implicit arguments*, to support the concise expression of recurrent processes. The notation allows specifying what has changed in the process's state during a state transition, rather than the entire old and new states required by a plain logic program, thus effectively providing a frame axiom for the state of recurrent processes. The semantics of the extended notation is given in terms of its translation to plain logic programs.

Streams are the most commonly used data structure in concurrent logic programs. To describe sending or receiving a message M on a stream Xs one equates Xs with a list (stream) cell whose head is M and tail is a new variable, say Xs', as in $Xs = [M—Xs']$. In this notation the "states" of the stream before and after the communication have to be named and referred to explicitly. We propose a notation that, by exploiting the implicit arguments notation, refers only once to the stream being used.

In practice we combine the two notational extension, inheritance and implicit arguments, into one language. We find the resulting language greatly superior to the "vanilla" syntax of (concurrent) logic programs.

In the rest of the paper we formally define the inheritance notation, the implicit arguments notation and give examples of their use.

2 Logic Programs with Inheritance

2.1 An Example

The inheritance notation is an extension to plain logic programs which allows *inheritance calls*, which are calls of the form $+p(T_1, \ldots, T_n)$. Using object-oriented terminology, we refer to a predicate as a *class* and to a procedure as a *class definition*. Each clause (or disjunct in a disjunctive form) of the procedure is viewed as a *method* which manipulates the class's state.

As an example of inheritance consider the following well known logic program which manipulates a simple counter:

```
counter(In) :-
    counter(In,0).

counter([clear|In],_) :-
    counter(In,0).
counter([add|In],C) :-
    C' := C + 1, counter(In,C').
counter([read(C)|In],C) :-
    counter(In,C).
counter([],_).
```

An alternative representation of a logic program can be in a disjunctive form, where all clauses of a predicate are written with separating semicolons and are under a single, simple head (an atom is *simple* if its arguments are distinct variables). The translation between plain logic programs and logic programs in disjunctive form is trivial.

Put in disjunctive form, the definition of *counter/2* would appear as:

```
counter(In, C) :-
    In = [clear|In'], counter(In',0);

    In = [add|In'], C' := C + 1,
        counter(In',C');
    In = [read(C)|In'], counter(In',C);

    In = [].
```

We illustrate the inheritance notation by adding a feature to *counter*, which enables us to retain a backup value of the counter, named *BackUp*. The *checkpoint* method backs up the counter value and *restore* restores its value from the backup. The syntactic changes from the previous *counter* version are an added argument and two new disjuncts. Using the inheritance notation we would write this as:

```
counter2(In) :-
    counter2(In, 0, 0).

counter2(In, C, BackUp) :-
    +counter(In, C);

    In = [checkpoint|In'],
        counter2(In', C, C);
    In = [restore|In'],
        counter2(In', BackUp, BackUp).
```

This procedure stands for:

```
counter2(In, C, BackUp) :-
    In = [checkpoint|In'],
        counter2(In', C, C);
    In = [restore|In'],
        counter2(In', BackUp, BackUp);
    In = [clear|In'],
        counter2(In',0, BackUp);
    In = [add|In'], C' := C + 1,
        counter2(In',C', BackUp);
    In = [read(C)|In'],
        counter2(In',C, BackUp);
    In = [].
```

2.2 Syntax

A *logic program with inheritance*, *lpi*, is a set of procedures, each having a unique head predicate. Each procedure is a disjunctive clause of the form:

$$p(X_1, \ldots, X_n) \leftarrow \alpha_1; \ldots; \alpha_k.$$

where $n, k \geq 0$, X_i's are distinct variables and each α_i is either a conjunctive goal or an *inheritance call* of the form $+q(X_{i_1}, \ldots, X_{i_m})$, where the i_j's are distinct and $1 \leq i_j \leq n$ for every j, $1 \leq j \leq m$.

Note that if p/n inherits q/m the definition implies that $m \leq n$.

An *inheritance graph* for an *lpi* program P is a directed graph (V, E) where V is the set of predicates defined in P and for every inheritance call to a predicate q in the procedure of p in P, $(p, q) \in E$.

An *lpi* program P is *well-defined* if the graph (V, E) is well-defined (*i.e.*, every predicate that occurs in E is a member of V) and acyclic.

For convenience, we employ the following syntactic default. Suppose the predicate q is defined by:

$q(Y_1, \ldots Y_m) \leftarrow \beta_1; \ldots; \beta_l.$

Then the inheritance call $+q$ is a shorthand for $+q(Y_1, \ldots, Y_m)$.

2.3 Semantics

The semantics of a well-defined logic program with inheritance P is given by the following unfolding rule, whose application to completion to P results in a logic program in disjunctive form. In the following rule p, q are predicates, the S's are terms, X and Y are logic variables, α_i, β_i are disjuncts.

Lpi Rule: Replace the clause:

$$p(X_1, X_2, \ldots, X_n) \leftarrow \alpha_1; \ldots; +q(X_{i_1}, \ldots, X_{i_m}); \ldots; \alpha_k.$$

where q is defined by the (renamed apart) clause:

$$q(Y_1, \ldots Y_m) \leftarrow \beta_1; \ldots; \beta_l.$$

with the clause:

$$p(X_1, X_2, \ldots, X_n) \leftarrow \alpha_1; \ldots; \beta'_1; \ldots; \beta'_l; \ldots; \alpha_k.$$

where β'_i is obtained from β_i by the following transformation:

1. Apply the substitution $\theta \stackrel{def}{=} \{Y_j \rightarrow X_{i_j} \mid 1 \leq j \leq m\}$.
2. Replace each recursive call $q(S_1, \ldots, S_m)$ with the call $p(X_1, \ldots, X_n)\sigma$, where $\sigma \stackrel{def}{=} \{X_{i_j} \mapsto S_j \mid (1 \leq j \leq m)\}$.

This completes the definition of logic programs with inheritance.

Assume some fixed first-order signature L. Let $\mathcal{P}$ be the set of all well-defined logic programs with inheritance over L. Let $\rightarrow: \mathcal{P} \times \mathcal{P}$ be the relation satisfying $P \rightarrow P'$ iff P' can be obtained from P by an application of the *lpi* rule to a clause in P.

The pair $\langle \mathcal{P}, \rightarrow \rangle$ is not strictly a rewrite system according to the standard definitions [3], since logic programs with inheritance are sets, not terms, and since they are not closed under substitution. However, these differences do not affect the applicability of the relevant tools of rewrite systems, so we ignore them.

Lemma 1 *The rewrite system $\langle \mathcal{P}, \rightarrow \rangle$ is terminating and confluent up to clause renaming, i.e. if P' and P'' are two normal forms of P then they are equivalent up to clause renaming. Furthermore, all normal forms are ordinary logic programs.*

Proof outline: Termination follows from the fact that any application of the *lpi* rule eliminates one inheritance call. Normal forms don't have inheritance calls since they can all be reduced by the requirement that $\mathcal{P}$ contains only well-defined logic programs with inheritance. Confluence follows from the associativity of substitution composition.□

Corollary 1 *The semantics of* lpi *is well defined.*

2.4 An Example of Parameterized Multiple Inheritance

As an example of parameterized inheritance, suppose we have a "show_id" feature which waits on an input port *In* for a message *show_id* and then fills the incomplete message with the value *Id*:

```
id(In, Id) :-
    In = [show_id(Id)|NewIn],
    id(NewIn, Id).
```

And suppose we have a class containing two input ports *In1* and *In2*, where on each port the class can receive requests to show its id. Instead of copying the method twice, we shall write:

```
class(In1, In2, Name) :-
    +id(In1, Name);
    +id(In2, Name);

    <<class body>>
```

The result of applying the *lpi* rule would be:

```
class(In1, In2, Name) :-
    In1 = [show_id(Name)|NewIn],
        class(NewIn, In2, Name);
    In2 = [show_id(Name)|NewIn],
        class(In1, NewIn, Name);

    <<class body>>
```

The same *id* feature could be used when an object wishes to have different id's on different ports, *i.e.*, return different answers on different ports for the same incomplete message *show_id*, as in:

```
class(In1, In2, Name1, Name2) :-
    +id(In1, Name1);
    +id(In2, Name2);

    <<class body>>
```

The expansion is as follows:

```
class(In1, In2, Name1, Name2) :-
    In1 = [show_id(Name1)|NewIn],
        class(NewIn, In2, Name1, Name2);
    In2 = [show_id(Name2)|NewIn],
        class(In1, NewIn, Name1, Name2);

    <<class body>>
```

2.5 Integration with a Module System

The power of logic programs with inheritance is enhanced when integrated with a module system. We have integrated *lpi* with the hierarchical module system of Logix [11]. To simplify the description, we outline the principles behind the integration for a non-hierarchical module system.

When p in module M inherits q from another module M', the semantics of inheritance is that the definitions of all predicates in M', called or inherited directly or indirectly by q, are incorporated in M, unless they are already defined in M.

This overriding capability, which gives some of the effects of higher-order programming, proves to be invaluable in practice. One can easily specify a variant of a module M by inheriting its top-level procedure and overriding the definition of one or more of its subprocedures. For example, by inheriting a sorting module and overriding the comparison routine, one can turn an ordinary sort routine to a sort routine that operates on records with keys.

We note that although the semantics specifies "code copying", the following semantics-preserving optimization may apply. If M' inherits from M, P is a set of procedures in M that do not call or inherit procedures outside of P, and none of the procedures in P is redefined in M', then the code for P need not be included in M', and any call to a procedure in P may be served by M. This optimization achieves runtime code sharing among several modules inheriting from the same module.

3 Logic Programs with Implicit Arguments

3.1 Example

We illustrate the notation of implicit arguments via an example. The *counter* program (section 2.1) is a typical logic program specifying a recurrent process. A logic program with implicit arguments that corresponds to the plain logic program for *counter* is:

```
counter(In) + (C=0) :-
    In = [clear|In'], C' = 0, self;

    In = [add|In'], C' := C + 1, self;
```

```
    In = [read(C)|In'], self;

    In = [].
```

Similarly, a binary merge can be defined using implicit arguments by:

```
merge(In1,In2,Out) :-
    In1 = [X|In1'], Out=[X|Out'], self;
    In2 = [X|In2'], Out=[X|Out'], self;
    In1 = [], Out = In2;
    In2 = [], Out = In1.
```

3.2 Syntax

A logic program with implicit arguments is a set of clauses. A clause is composed of a predicate declaration and a disjunctive body. The predicate declaration has the form:

$$p(X_1,\ldots,X_n) + (X_{n+1} = V_1,\ldots,X_{n+k} = V_k) \leftarrow$$

where n, $k \geq 0$, the X's are distinct variable names and the V's are terms. We say that the predicate p has n global and k local arguments, denote it by $p/n{+}k$ (or p/n if $k = 0$), and call $V_1,\ldots,V_k$ the initial values of the local arguments of $p/n{+}k$. There can be at most one clause for any predicate p.

A call to $p/n{+}k$ in a procedure other than that of $p/n{+}k$ has the form $p(T_1,\ldots,T_m)$, where $m = n$ or $m = n + k$, where the T's are terms. A call to $p/n{+}k$ that occurs in its own procedure may also have the form p, *i.e.*, the call may have no arguments whatsoever. Such a recursive call is called *implicit*. In addition, any call to $p/n{+}k$ that occurs in its own procedure may use the predicate name *self* as a synonym for p.

Variable names may be suffixed by a prime, e.g. X', Y'. X^n denotes the variable name X suffixed by n primes, $n \geq 0$. A primed version of a variable name denotes a new incarnation of the variable in the sense that the "most primed" occurrence of a variable name is considered the most updated version of the variable and hence is used in implicit recursive calls as explained below. We assume that the predicate $=/2$ is defined via the single unit clause $X = X$.

3.3 Semantics

The semantics of a logic program with implicit arguments P is given by the following rewrite rules, whose application to completion results in a disjunctive logic program P'.

Rule 1: Expand local argument of calls. Replace each procedure call:

$$p(T_1,\ldots,T_n)$$

to a procedure $p/n + k$, by the call:

$$p(T_1,\ldots,T_n,V_{n+1},\ldots,V_{n+k})$$

where $V_n,\ldots,V_{n+k}$ are the initial values of the local arguments of $p/n{+}k$.

Rule 2: Expand implicit recursive calls. Replace each procedure call:

$$p \text{ or } \mathit{self}$$

in the clause of $p/n + k$ by the call:

$$p(U_1,\ldots,U_{n+k})$$

where U_i is the most primed version of X_i in the clause. We say that X^n is *the most primed occurrence of X in a clause C* if X^n occurs in C, and for no $k > n$, does X^k occur in C.

For example, applying the rewrite rules to the merge procedure results in:

```
merge(In1,In2,Out) :-
    In1 = [X|In1'], Out=[X|Out'],
        merge(In1',In2,Out');
    In2 = [X|In2'], Out=[X|Out'],
        merge(In1,In2',Out');
    In1 = [], Out = In2;

    In2 = [], Out = In1.
```

3.4 Special Notation for Stream Communication

Streams are the most commonly used data structure in concurrent logic programs. Recurrent processes almost always have one input stream and often have several additional input and/or output streams. Sending and receiving messages on a stream Xs by a process p can be specified by the clause schema:

```
p(...Xs...) :-
    Xs = [Message|Xs'],
    ...,
    self.
```

where the difference between sending and receiving is expressed using a language-specific synchronization syntax. Since this is such a common case, we found it worthwhile to provide it with a special notation. Our notation is reminiscent of CSP [5] and Occam [6] (and in logic programming the Pool language [2]):

```
Xs ! Message,
Xs ? Message,
```

These constructs specify, respectively, sending and receiving a message *Message* on a stream *Xs*. Each is equivalent to *Xs = [Message—Xs']* with the appropriate language-specific synchronization syntax added. The construct requires *n+4* fewer characters, where *n* is the length of the stream variable name, and hence is less liable to typing errors and probably also more readable.

Using this notation, a binary stream merger can be specified by:

```
merge(In1,In2,Out) :-
    In1 ? X, Out ! X, self;
    In2 ? X, Out ! X, self;
    In1 = [], Out = In2;
    In2 = [], Out = In1.
```

The predicate *append/3* can be specified using the first and third disjuncts of *merge/3*:

```
append(In1,In2,Out) :-
    In1 ? X, Out ! X, self;
    In1 = [], Out = In2.
```

It is interesting to note that this description of *append* facilitates its process reading . The program can be read as: "*append* is a process with three streams. Upon receiving an item on its first stream it sends that item on its third stream and iterates. If the first steam is closed, then the second and third streams are connected".

Using multiple primes allows multiple messages to be sent or received on the same stream, as the following example for the filtering of pairs of items on a stream shows:

```
remove_pairs(In, Out) :-
    In ? X, In' ? X,  Out ! X, self;
    In ? X, In' ? Y, X =\= Y,
        Out ! X, Out' ! Y, self.
```

3.5 Special Notation for Arithmetic

Arithmetic operations are quite common in ordinary and concurrent logic programs. Recurrent processes with a loop counter such as the following are abundant:

```
list(N,Xs) :-
    N > 0, Xs = [N|Xs'], N' := N - 1,
       list(N',Xs');
    N = 0, Xs = [].
```

Following C conventions we allow variable names to be suffixed by -- and ++, with the semantics of the expression `N--` given by replacing it with `N` and adding the conjunctive goal `N' := N-1`. Using the stream and arithmetic support, the above list generator can be written as:

```
list(N,Xs) :-
    N-- > 0, Xs ! N, self;
    N = 0, Xs=[].
```

Similarly, we define `+=` and `-=`, where `N += K` stands for `N' := N+K`, and
`N -= K` stands for `N' := N-K`.

4 Implicit Logic Programs with Inheritance

4.1 Concepts

The combination of inheritance and implicit arguments proves to be both highly succinct and more readable. For example, the program *counter2* of section 2.1 can now be rewritten as:

```
counter2(In) + (C=0, BackUp=0) :-
    +counter;
    In ? checkpoint, BackUp' = C, self;
    In ? restore, C' = BackUp, self.
```

The new style differentiates between global and local (hidden) arguments and also avoids copying *counter*'s code as well as specifying the arguments of the two recursive calls.

An implicit logic program with inheritance is translated into a pure logic program by applying the two previously defined rules. That of *lpi* (section 2.3) and that of implicit arguments (section 3.3). Minor changes in the rules are due. Those changes depend on the order in which we apply the two transformations.

4.2 Examples

A curious example in which *lpi* gives us some insight into a program, is the redefinition of *merge* (section 3.4) as:

```
merge(In1,In2,Out) :-
    +append(In1, In2, Out);
    +append(In2, In1, Out).
```

which means that merging is actually trying nondeterministically to append in both possible ways.

The following example implements a simple lookup table as a list of *key - value* pairs. The *create* predicate builds the list (named *Table*):

```
create(Table) :-
        Table=[].
```

i.e, a new table is an empty list. The following two predicates are not for direct usage. *search* iterates through the list as long as the key-value pair at the top of the list does not match a given *Key*. *find* inherits *search*, and adds a termination clause.

```
search(Key, Table, Table1) :-
    Table ? Key1 - Value1, Key=\=Key1,
        Table1 ! Key1 - Value1, self.

find(Key, Table, Table1, Ok) :-
    +search;
    Table = [],
        Table1 = [],
        Ok = false('key not found', Key).
```

The following *check* and *lookup* predicates inherit *find* and add a clause for the case where an identical key was found. The *replace* predicate inherits *search* directly since we want a different error message.

```
check(Key, Table, Table1, Ok) :-
    +find;
    Table ? Key - Value,
        Table1 = Table,
        Ok = true.

lookup(Key, Value1, Table, Table1, Ok) :-
    +find;
    Table ? Key - Value,
        Table1 = Table,
        Value1 = Value,
        Ok = true.

replace(Key, Value, NewValue, Table,
                            Table1, Ok) :-
    +search;
    Table ? Key - OldValue,
        Value = OldValue,
        Table1 = [Key - NewValue
                | Table'],
        Ok = true;
    Table = [],
        Table1 = [],
        Ok = false('key not found',
                Key - NewValue).
```

Finally *insert* and *delete* add and remove key–value pairs from the table.

```
insert(Key, Value, Table, Table1, Ok) :-
    +search;
    Table ? Key - Value1,
        Table1 = Table,
        Ok = false('key already exists',
                Key - Value);
    Table = [],
        Table1 = [Key - Value],
        Ok = true.

delete(Key,Value,Table,Table1,Ok) :-
   +find;
   Table ? Key - Value1,
        Value = Value1,
        Table1 = Table',
        Ok = true.
```

As a third example we demonstrate the capabilities of the inheritance mechanism in a graphical environment by rewriting the window handling class from [10].

The first class defines a rectangular area with methods *clear* for painting the area specified by *Frame*, and *ask* to retrieve the rectangular's dimensions. *In* is an input port and *Frame* is a four-tuple of rectangle coordinates.

```
rectangular_area(In) +
         (Frame = {X, Y, W, H}) :-
    In ? clear,
        clear_primitive(Frame),
        self;
    In ? ask(Frame),
        self.
```

The following class *frame* is a rectangular area with some content, which means that apart from the methods *clear* and *ask*, one can *draw* the area boundaries, and *refresh* it. Note that *refresh* is just a combination of two previously defined methods *draw* and *clear*. This also fixes a subtle synchronization bug in Shapiro and Takeuchi [10] where the two methods were simultaneously activated, one by the class process and one by the *independent* superclass process, which could have caused drawing before clearing.

```
frame(In) + (Frame = {X,Y,W,H}) :-
    +rectangular_area;
    In ? draw,
        draw_lines(Frame), self;
    In ? refresh,
        self([clear, draw|In']).
```

The final class *labeledWindow* adds two more methods: *change*, to change a label, and *show* to show it. In addition we redefine the *refresh* method to show the label after refreshing (we thus require a method override mechanism). Another local variable *Label* is added.

```
labeledWindow(In) + (Frame = {X,Y,W,H},
            Label = default) :-
    +frame;
    In ? change(Label'), self;
    In ? show,
        show_label_primitive(Frame),
        self;
    In ? refresh,
        self([clear, draw, show|In']).
```

After we have the class *labeledWindow* we can subclass it to define our own window as in:

```
my_window(In,....) + (Frame = ...,
        Label = ...) :-
    +labeledWindow;

  <<my_window_additional_methods>>.
```

The generated code derived from the semantics of *lpi* and implicit arguments is not shown here due to space limitations.

5 Conclusions

5.1 Implementation

Both notations, *implicit* and *lpi*, have been implemented in FCP within the Logix system [11] by adding language preprocessors. The *lpi* preprocessor implements the combined notation of Section 4; *i.e.*, it translates FCP with inheritance and implicit arguments to FCP with implicit arguments. Another preprocessor translates implicit FCP to pure FCP. Each of the preprocessors is about 1000 lines of code. The *implicit* preprocessor was first written in FCP(:,?) [12]. That initial version was then used to bootstrap a new version written in the implicit notation. The *lpi* preprocessor is also written using the notation of implicit arguments.

5.2 Further work

A certain form of overriding is already available via the integration of *lpi* and a module system, described in Section 2.5. However, one may find useful also the ability of a subclass's method to override a method of the superclass. This can be achieved, for example, by stating that if several methods apply, then textual order dictates precedence. By appropriately placing inheritance calls, one can achieve the desired override effect.

Additional clarity and conciseness could be achieved by enabling an overriding method to also execute the overridden method (apart from doing some processing of its own). This feature, called *send to super* in the object oriented terminology, was easily implemented with Shapiro and Takeuchi's scheme [10] of a subclass having also an output stream to its super, by putting the method on the output stream. As an example of the send to super feature, suppose in *my_window* (section 4.2) we need to add functionality to the *draw* routine (e.g. drawing a grid on the rectangular_area), which means overriding the current *draw* method. Instead of copying the whole *draw* method, we would write:

```
In ? draw, send_to_super,
  draw_grid(Frame), my_window;
```

where *send_to_super* is a macro which copies the necessary code from the appropriate superclass.

A redundancy problem occurs when we want to use multiple inheritance but the generated inheritance graph is not a tree. For example, classes b and c both inherit a, and d inherits both b and c. Applying the transformation would result in d having a's methods twice. This (harmless) redundancy could be optimized later, e.g. by the decision graph compilation method [9].

5.3 Experience

The implicit arguments notation was incorporated into Logix more than two years ago, and has been used extensively by all members of our group. All of us found it preferable to the notation of plain logic programs.

Logic programs with inheritance were incorporated as an extension to the implicit arguments notation less than a year ago. It has been used by all of us extensively, and it has had a major effect on our programming style. One notable effect is that inheritance allows us to specify in a modular way processes with a dozen of arguments and dozens of clauses, by specifying multiple methods, each referring only to a subset of the process's arguments, and using multiple inheritance to specify the final process. This programming style meshes well with the decision graph compilation method to produce code which is readable, maintainable, and efficient.

We have implemented two large systems using *lpi*, each having several thousand lines of FCP code, and we find it hard to imagine how we could have written them without an inheritance notation.

6 Acknowledgments

The notation of implicit arguments was first described in an unpublished paper by Kenneth Kahn and the last two authors. We thank Yael Moscowitz and Marilyn Safran for comments on previous drafts.

References

[1] Clark, K.L., Gregory, S., *PARLOG: Parallel Programming in Logic*, ACM Trans. on Programming Languages and Systems, 8(1), pp. 1-49, 1986.

[2] Davison, A., *POOL: A PARLOG Object Oriented Language*, Imperial College.

[3] Dershowitz, N., and Jouannaud, J.-P., Rewrite Systems, in *Handbook of Theoretical Computer Science*, J. van Leeuwen (Ed.), pp.243-320, Elsevier Science Publishing, 1990.

[4] Hewitt C., *A Universal, modular Actor formalism for artificial intelligence*, Proc. International Joint Conference on Artificial Intelligence, 1973.

[5] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, New Jersey, 1985.

[6] INMOS Ltd., *OCCAM Programming Manual*, Prentice-Hall, New Jersey, 1984.

[7] Kahn K. M., *Objects - A Fresh Look.* Proceedings of the European Conference on Object-Oriented Programming, Nottingham, England, July 1989.

[8] Kahn K. M., Tribble D., Miller M. S., Bobrow D.G. *Vulcan: Logical Concurrent Objects.* in Concurrent Prolog: Collected Papers, Vol 2, Chapter 30, MIT press, 1987.

[9] Kliger, S., and Shapiro, E., From decision trees to decision graps, *Proc. of the 1990 North American Conf. on Logic Programming*, S. Debray and M. Hermenegildo (Eds.), MIT Press, pp. 97-116, 1990.

[10] Shapiro E., Takeuchi A., *Object Oriented Programming in Concurrent Prolog.* in Concurrent Prolog: Collected Papers, Vol 2, Chapter 29, MIT press, 1987.

[11] Silverman, W., Hirsch, M., Houri, A., and Shapiro, E., *The Logix System User Manual, Version 1.21*, in Concurrent Prolog: Collected Papers, Vol 2, Chapter 21, MIT press, 1987.

[12] Yardeni, E., Kliger, S., and Shapiro, E., The languages FCP(:) and FCP(:,?), *New Generation Computing*, 7(2-3), pp.85-87, 1990.

[13] Yoshida K., Chikayama T. *A'UM - A stream based Concurrent Object-Oriented Language* FGCS, Vol 2, 1988.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Implementing a Process Oriented Debugger with Reflection and Program Transformation

Munenori MAEDA
International Institute for Advanced Study of Social Information Science,
FUJITSU LABORATORIES LTD.
17-25, Shinkamata 1-chome, Ota-ku, Tokyo 144, Japan

m-maeda@iias.flab.fujitsu.co.jp

Abstract

Programmers writing programs following a typical process and streams paradigm usually have some conceptual image concerning the program's execution. Conventional debuggers cannot trace or debug such programs because they are unable to treat both processes and streams directly. The process oriented GHC debugger we propose provides high-level facilities, such as displaying processes and streams in three views and controlling a process's behavior by interactively blocking or editing data in its input streams. These facilities make it possible to trace and check program execution from a programmer's point of view. We implement the debugger by adopting reflection and program transformation to enhance standard GHC execution and to treat extended logical terms representing streams.

1 Introduction

Debugging methods for programs in Guarded Horn Clauses(GHC)[Ueda 1985] are classified into those based on algorithmic debugging[Takeuchi 1986] under the denotational semantics of GHC programs, and those based on execution tracing [Goldszmidt *et al.* 1990][1] under the operational semantics. This paper proposes a debugging method belonging to the execution tracing class.

In GHC programming, object-oriented[Shapiro and Takeuchi 1983] and stream-based[Kahn and MacQueen 1977] programming focus on the notion of processes and streams. Individual abstract modules are regarded as processes, some of which are connected by streams, and communicate with each other concurrently. A typical process repeatedly consumes data from a stream, changes its internal state, and generates data for another stream.

In a conventional execution tracer, it is difficult to capture conceptual execution in terms of processes and streams, because they are decomposed into GHC primitives and never displayed explicitly. The tracer we propose fully reflects the notion of processes and streams, and enables both the specific control flow of processes and the data structure of streams to be processed, making the causality among processes explicit.

2 Process Oriented Programs and Debugging

2.1 Models of Processes and Streams in GHC

2.1.1 Process model

A process can be interpreted either as a goal or as a set of goals, e.g., an "object" in object-oriented programming[Shapiro and Takeuchi 1983]. The following sections discuss processes based on the latter.

A process consists of goals for the continuation of the process or goals for internal procedures defined in the process. The continuation goal accepts streams in its arguments one by one, and reserves its internal state in other arguments. The stream argument takes a role of an I/O port for the data migration. The internal state is not affected by other processes, but is calculated by the previous state and input data captured from streams.

A process features:

Creation: A process is created by the first call of the continuation goal.

One-step execution: Reading data from streams, writing data to other streams, and changing the inter-

[1] Even though the literature is concerned only with the execution tracing of Occam programs, its discussion is generally adaptable for most concurrent or parallel program debugging.

nal state using internal procedures are regarded as atomic actions in an execution step.

Continuation and termination: A process will carry on its computation with a new internal state when the continuation goal is invoked. Otherwise the process terminates its execution.

2.1.2 Stream model

A stream is a sequence of logical terms whose operations[Tribble *et al.* 1987] are limited to reading the first term of a stream and writing a term to the tail of a stream.

A simple notation for streams is first introduced. Streams are constructed by stream-variables SV, stream-functors $\langle SH \| ST \rangle$ and stream-terminators $\langle\rangle$, where SV is a variable constrained to become either a stream-functor or a stream-terminator, SH is an arbitrary term denoting the first data of the stream, and ST is a stream representing the rest of the stream.

A stream features:

Creation: Streams are created dynamically when a continuation goal of a process is invoked, where they are assigned to the arguments of the goal.

Data access: First data D is read from stream SX by unifying SX with structure $\langle D \| ST \rangle$ in the guard part of a clause at runtime. Data D is written to stream-variable SX by unifying SX with a structure $\langle D \| ST \rangle$ in the body, where ST, called the tail of stream SX, is a stream-variable. In reading or writing done several times, each operation is done recursively for the rest of stream, ST.

Connection: Streams S_a and S_b are connected if they are unified in the body. One of the connected streams is regarded as an alias of the other.

Equivalence relation $\cong$ is defined for the set of streams $\mathcal{S}$, used to visualize streams.

For substitution σ, relation $\simeq_\sigma$ is defined for $\mathcal{S}$, the set of streams consisting of terms obtained in the execution.

1. $S \simeq_\sigma S; {}^\forall S \in \mathcal{S}$
2. $\langle H \| S \rangle \simeq_\sigma S; {}^\forall S \in \mathcal{S}$
3. $S_1\sigma = S_2\sigma \Rightarrow S_1 \simeq_\sigma S_2, {}^\forall S_1, S_2 \in \mathcal{S}$

The first reflective rule implies that two lexically identical variables satisfy the relation. The second rule implies that a stream and its subpart are elements of the same equivalence class. The third rule means that connected streams are also elements of the same equivalence class. Relation $\cong_\sigma$ is defined as the symmetric and transitive closure of relation $\simeq_\sigma$. Below, relation $\cong$ is written in place of $\cong_\sigma$ if substitution σ is clearly understood from the context.

In GHC, a stream is actually implemented by a list in most programs, i.e. stream-functor $\langle D \| S \rangle$ and stream-terminator $\langle\rangle$ correspond to term $[D' \mid S']$ and atom $[]$.

2.2 Process Oriented Debugging

GHC programs based on the process model are called process oriented programs, each goal in the execution trace belongs to a process, which is either a continuation or a part of an internal procedure of the process. In tracing and checking process oriented programs, the goals belonging to a target process must first be extracted from the "chaotic" execution trace where these goals are interleaved.

The data flow must also be checked. Unless a process inputs intended data, the process outputs incorrect data to its output stream, or becomes permanently suspended. Intended data may not be sent to the process for two possible reasons. First, an adjacent process corresponding to the producer of the data malfunctions. Or, second, the input of the process is disconnected with the output of the producer, an error caused by misuse of a shared variable, in which case it is easier to detect the error if the stream connection between processes, called "a process network," is displayed.

To make process oriented programs execution traces easier to read, the process oriented debugger(**POD**) we propose, visualizes process and stream information structured from input/output data, internal state values, internal procedure traces and the stream connection between processes.

Programs can be debugged as follows:

Step 1 A user starts execution of a target program.

Step 2 The internal state and input/output data are displayed and checked at an appropriate interval. The process network is also checked.

Step 3 The program code corresponding to a process where an error occurs is checked in detail, with any adjacent processes possibly contributing to the anomaly also checked.

Step 4 Input/output data sequences are saved for checking an abnormal process because comparing the sequence of output data before and after a program is modified makes it easier to check the behavior.

If the process malfunctions in Step 3 and 4, it is forcibly suspended and overall execution is continued as far as possible because program reexecution takes much

time and costs, i.e., reexecution must be avoided if it will take too much data in streams up to a sufficient length. Otherwise the program will have nondeterministic transitions.

Reexecution can be avoided either by giving the debugger the functions to delete or to modify unexpected data and to insert data in a stream interactively, or by having functions preserve data in streams automatically and execute a process in the preserved environment,

Thus the POD requires the following execution control functions:

1. Forcibly suspending, resuming and aborting the execution of each process.
2. Buffering and modifying the data in streams interactively.
3. Reexecuting a process in the preserved environment.

3 Implementing the POD

3.1 Process Declaration

In our process model(Section 2.1.1), goals are classified into those for the continuation and those for internal procedures. They are syntactically the same, and are specified by the user in a process declaration.

The process declaration consists of a predicate specification and continuation marking. The predicate specification begins with the keyword `process` followed by the name of the predicate specifying the usage of each argument. The usage of each argument is specified by declaring keyword `state` or `port` in an appropriate order. Annotation `state` shows that the argument represents a part of the internal state. Annotation`port` shows that the argument represents a process's I/O port. The continuation mark consists of a @ preceding the goal in a clause. An example of the process declaration is given in Listing 1.

3.2 Stream Treatment

As mentioned previously, streams consist of special variables, functors, and terminators.

In the POD, streams are recognized and supported by introducing tagged data structures. Each variable, functor, and atom that makes up a stream has an auxiliary field to store the stream identifier. An identifier is associated with each stream equivalence class.

In implementing identifiers, note that if two streams with different identifiers are unified, their identifiers should be the same. This is achieved by assigning a variable to each identifier and unifying the identifiers if their streams are to be unified. The problem of whether two streams satisfy equivalence relation $\cong$ is solved in a variable equivalence check.

The POD recognizes and manages streams as follows: First, before starting the execution, a program translator, which is a subsystem of the POD, converts a target program to a canonical form as detailed in Section 3.4. Streams are replaced with special tagged terms, and extended unifications are placed for their unifications which causes accessing data in streams or connecting streams as described in Section 2.1.

All the parameters of each process are stored in its process table every own execution step. Process execution is visualized using these process tables.

3.3 Reflective Extension of Unifier

Section 3.2 addressed a need for extended unification, discussed in more detail together with its implementation with reflection.

Tagged structures must be implemented using wrapped terms 'Sm'(Var,ID), 'Sm'(Atom,ID), and 'Sm'({Cons,Head,Tail},ID). The first term represents the fresh variable of a stream whose first argument, Var, corresponds to the original variable. The second, ID, is a fresh variable that denotes the identifier of its stream. The second term represents the terminator of a stream whose first argument, Atom, abstracts $\langle\rangle$, while the third corresponds to $\langle$Head||Tail$\rangle$ and Cons is a functor for concatenation.

Terms are classified into six types: variable, atom, compound term[2], stream-variable, stream-functor, and stream-terminator.

New unification rules are needed for stream-term $\times$ stream-term and stream-term $\times$ regular-term. The following cases are representative of the extended unification X >< Y:

Case 1 X is stream variable 'Sm'(V,ID), Y is a variable.
Assign Y to X.

Case 2 X is stream variable 'Sm'(V_1,ID_1),
Y is stream variable 'Sm'(V_2,ID_2).
Assign V_2 to V_1 and ID_2 to ID_1.

Case 3 X is stream variable 'Sm'(V,ID),
Y is compound term {C,H,T}.
Assign {C,H,'Sm'(N,ID)} to V, and execute
'Sm'(N,ID) >< T, where N is a fresh variable.

Case 4 X is stream functor 'Sm'({C_1,H_1,T_1},ID).
Y is compound term {C_2,H_2,T_2}.
Assign H_2 to H_1 and C_2 to C_1.
Execute T_1 >< T_2 recursively. □

[2]In KL1, notation $\{F, A_1, ..., A_n\}$ is allowed to express compound term $F(A_1, ..., A_n)$. We follow the notation for convenience.

The remaining 17 possible cases are omitted here due to space considerations.

The variable check is essential when describing the unifier and is done by reflection [Smith 1984]. Because reflection provides functions to manage memory and goal queue, it becomes easy to implement streams.

Before developing the POD, we added a reflective feature to GHC similar to that for RGHC[Tanaka 1988]. When a user-defined reflective predicate is invoked, its arguments are automatically converted from internal representation to the meta-level ground form. Table 1 shows the correspondence between object-level and meta-level terms.

Case 3 is described using a reflective predicate whose second argument Gs is a stream connected with the goal scheduler.

```
reflect(vector({atom('Sm'),variable(V),ID}) ><
        vector({C,H,T}),Gs,Mm) :- true |
   Gs = [variable(V) = vector({C,H,vector(
                {atom('Sm'),variable(N),ID})}),
         vector({atom('Sm'),variable(N),ID}) >< T ],
   Mm = [ malloc(N)].
%% Gs: Goal scheduler, Mm: Memory manager.
```

The third argument Mm is a stream connected to the memory manager for the object program. Terms written in stream Gs are converted from meta-level ground terms to internal representation and placed in the goal queue. Terms written in stream Mm are understood as messages for memory access. Message *malloc*(N) invokes dynamic memory allocation, and the reference pointer to allocated memory is bound to variable N. Extended unification is defined similarly by the reflective predicate for all cases.

3.4 Tagged Term Transformation

As described above, tagged terms are represented as wrapped functors. The translator converts streams to tagged terms automatically. In the following, we show program examples before and after the conversion, then we explain the detail of the translating process. Furthermore we present additional transformation steps to direct the data migration, in other words, to detect the origin of data.

```
%Original program
process p(port,port).
boot :- true | p([1,2|_],X), q(X).
p([A|X],Y) :- true | Y=[A|Y1], @p(X,Y1).
%Conversion for streams
boot :- true | p([1,2|_],X,'Sm'(N1,ID1),'Sm'(N2,ID2)),
        q(X,'Sm'(N3,ID3)),
        [1,2|_] >< 'Sm'(N1,ID1), NX >< 'Sm'(N2,ID2),
        NX >< 'Sm'(N3,ID3).
p([A|X],Y,NA1,NA2) :- true | Y=[A|Y1],
        p(X,Y1,'Sm'(N1,ID1),'Sm'(N2,ID2)),
        NA1 >< [DA|DX], NA2 >< [DA|DY1],
        DX >< 'Sm'(N1,ID1), DY1 >< 'Sm'(N2,ID2).
```

The converted program differs from the original in the following ways:

1. The arity of predicate p doubles, i.e. the third and fourth arguments are new, and the parameters of p are converted to tagged terms for streams such as 'Sm'(N1,ID1).

2. The first and second arguments of the converted p are the same as those of the original, and the corresponding parameters are maintained.

3. Several extended unifications are added in the body.

The above points characterize the transformation: Two kinds of variable bindings are treated. One is the same as the original bindings and is used for the execution of the guard goal. The other consists of tagged terms for streams, and is used in extended unifications.

According to GHC semantics, unification invoked in a guard can not export any bindings to the caller. Furthermore user-defined predicates can not be placed in a guard. Because it is not easy to extend the guard execution rule of GHC, we follow the semantics as much as possible.

In our transformation, by maintaining the original bindings, the guard execution involving the parameter passing is independent of the term extension, and the extension never causes execution errors. The memory consumption by storing two kinds of bindings is, however, at least twice as much as that of the original.

Transformation processes are detailed as follows:

Step 1 Choose a clause, and erase all the guard unifications by partial evaluation[Ueda and Chikayama 1985]. Replace nonvariable argument Arg to fresh variable Var, and add goal Arg = Var in the guard. By applying the replacement for every argument, we get a canonical form such that every argument is a variable and every guard goal is ether a unification = of a variable and a nonvariable term, a difference \=, an arithmetic comparison or a type checker. We write a canonical clause as `P(A1,...,An):- G(A1,...,An) | Q(A1,...,An,B1,...,Bm)`, where `G(A1,...,An)` and `Q(A1,...,An,B1,...,Bm)` represent a conjunction of goals.

Step 2 Rename all variables in the clause and get a clause: `P(A1',...,An'):- G(A1',...,An') | Q(A1',...,An',B1',...,Bm')`. Extract all the unifications from `G(A1',...,An')`, and replace symbol = of unification with symbol >< of extended

Table 1: Representations of meta-level terms

Level	Term		
	Unbound variable	Atom	Compound
Object	*unobservable*	*Atom*	$\{C_1, \cdots, C_n\}$
Meta	variable(*Addr*)	atom(*Atom*)	vector($\{C'_1, \cdots, C'_n\}$)

unification. The obtained conjunction is written as `G'(A1',...,An')`.

Step 3 For goals defined as processes or as continuations in `Q(A1',...,An',B1',...,Bm')`, get conjunction `Q'(A1',...,An',B1',...,Bm')` by repeating follower. Replace port-declared parameter Param with stream 'Sm'(N,ID), where N and ID are fresh variables. Place extended unification Param >< 'Sm'(N,ID) in the body of the new clause.

Step 4 For two goals, `B(D1,...,Di)` in `Q(A1,...,An, B1,...,Bm)` where `Dj,1` $\leq$ `j` $\leq$ `i`, is ranged over {`A1,...,An,B1,...,Bm`}, and `B'(D1',...,Di')` in `Q'(A1',...,An',B1',...,Bm')`, goal `B''(D1,..,Di,D1',..,Di')` is defined as their concatenation. Conjunction `Q''(A1,...,An, B1,...,Bm,A1',...,An',B1',...,Bm')` is defined by combining every `B''`.

Step 5 An objective clause is obtained by combining `G`, `G'` and `Q''` as follows:

```
P(A1,...An,A1',...An'):- G(A1,..An) |
  G'(A1',..An'),
  C1 >< 'Sm'(S1,ID1),...,Ci >< 'Sm'(Si,IDi),
  Q''(A1,...,An,B1,..,Bm,A1',...,An',B1',..,Bm').
% Replace Cj in A1',...,Bm' to 'Sm'(Sj,IDj)
```

Detecting the origin of the data is achieved by using a tag similar to that stated above. A tagged functor 'Sb'(Term,PID) is introduced, where Term corresponds to the original term and may include other tagged structures, PID is an unbound variable used as a process identifier.

We show a modified example program using 'Sb' tag, then the additional transformation steps are detailed.

```
%Conversion for detecting the origin of the data
boot(PIDself) :- true |
        p([1,2|_],X,'Sm'(N1,ID1),'Sm'(N2,ID2),PID1),
        q(X,'Sm'(N3,ID3),PID2),
        'Sb'(['Sb'(1,PIDself)|
        'Sb'(['Sb'(2,PIDself)|_],PIDself)],PIDself)
        >< 'Sm'(N1,ID1),
        NX >< 'Sm'(N2,ID2), NX >< 'Sm'(N3,ID3).
p([A|X],Y,NA1,NA2,PIDself) :- true |
        Y=[A|Y1],
        p(X,Y1,'Sm'(N1,ID1),'Sm'(N2,ID2),PIDself),
        NA1 >< 'Sb'([DA|DX],PID1),
        NA2 >< 'Sb'([DA|DY1],PIDself),
        DX >< 'Sm'(N1,ID1), DY1 >< 'Sm'(N2,ID2).
%% PID1 specifies the origin of the input list.
```

Step 6 Add argument PID_{self} to the head of the selected clause.

Step 7 Select predicate p/n in the body of the clause and, if p/n is declared as a process, add new parameter $\text{PID}_{\text{p/n}}$ or else add parameter PID_{self}.

Step 8 Recursively replace every nonvariable term T_i except streams in `G'(A1',...,An',B1',...,Bm')` with term 'Sb'(T'_i,PID_i). Each PID_i is used to indicate the origin of corresponding data.

Step 9 Replace every nonvariable parameter T of the extended unifications with term 'Sb'(T',PID_{self}).

3.5 Execution Control

In the POD, the specific control of a process proposed in Section 2.2 is achieved by introducing a valve inserted into a stream. The valve serves as an intelligent data buffer having two input ports, one output port, and a programmable conditional switch to close the output port. One of the two input ports is connected to the original stream, and the other is connected to the user's console. The user can send commands to the valve. The amount of buffered data and the description of the type of storable data are programmable conditions.

The valve has three states, automatic migration mode, conditional migration mode, and manual edit mode, each changed by a user command or by evaluating programmable conditions. The valve operates as follows:

- In automatic migration mode, the valve receives data from its own input port and it stores the data in its own buffer. Once the buffer becomes full, the valve outputs the first data in the buffer through the output port.

- In conditional migration mode, The valve gets data then stores it in the buffer. Once the buffer becomes full or if data does not satisfy a condition, the valve displays an alert and changes to manual editing mode.

- In manual editing mode, the valve receives no new data. The number and the description of data to be stored, and data actually in the buffer can be referenced and modified using a text editor. After editing, the mode returns to the previous mode.

A data checking condition is provided as the conjunction of GHC goals. The goal is a built-in or user-defined predicate. Built-in predicates are classified into type check, arithmetic comparison, and guard unification. The type check goal is, e.g., $atom()$, $integer()$, or $float()$. the arithmetic one is, e.g., $>, \geq, <, \leq$. The user-defined goal is a combination of built-in goals.

4 Examples of Tracing

The POD is developed by extending the GHC interpreter with reflection in Prolog. A user can trace and debug a GHC program with a direct manipulation interface provided by the POD.

The interface provides several control facilities for the target program in a menu, enabling the user to easily manipulate the POD by selecting a facility from a menu with a mouse. The menu currently provides, (1) compulsive process suspension, (2) process resumption, (3) valve insertion, (4) valve control, and (5) terminated process deletion.

The POD provides different three views to visualize program execution: the stream graph, process char, and communication flow.

The stream graph uses animated icons and lines to show dynamic changes in a network graph of processes and streams.

The process chart displays, in a structured diagram, consumed and generated data from or to streams in a process and its subprocesses. More specifically, the diagram contains dots, two kinds of lines, and data. A dot represents a process's argument in each execution step. One kind of line connecting two dots is associated with relation $\cong$ between them. Consumed or generated data is located along this line. The other kind of line represents a subprocess fork point.

The communication flow[Shin 1991] shows I/O process causality. When a substitution generated in process X is referenced in a committed clause of process Y, a directed arrow from X to Y is displayed. we say in this case that data from process X makes Y active.

The usages of the menu and the views are described using a program in Listing 1, first suppose that the program and query `prime(10,Ps)` are given to the POD.

Listing 1: Primes generator program with process declaration

```
process gen(state,state,port), sift(port,port),
       filter(port,state,port).
prime(Max,Ps):- true | gen(2,Max,Ns), sift(Ns,Ps).
gen(N,Max,Ns):- N>=Max | Ns=[].
gen(N,Max,Ns):- N<Max | N1:=N+1, Ns=[N|Ns1],
       @gen(N1,Max,Ns1).
sift([],Ps):- true | Ps=[].
sift([P|Fs],Ps):- true | Ps=[P|Ps1],
       filter(Fs,P,Fs1), @sift(Fs1,Ps1).
filter([],P,Fs):- true | Fs=[].
filter([N|Ns],P,Fs):- true | sw(N,P,Fs1,[N|Fs1],Fs),
       @filter(Ns,P,Fs1).
sw(N,P,Fs1,Fs2,Fs):- N mod P=:=0 | Fs=Fs1.
sw(N,P,Fs1,Fs2,Fs):- N mod P=\=0 | Fs=Fs2.
```

Figure 1 shows the initial stream graph. Data in a stream that connects `gen` and `sift` can be checked in two ways, by setting a valve to either an output port of `gen` after suspending `gen` to prevent the creation of new data, or an input port of `sift` to avoid consuming data. Let item (1) be selected to `gen` suspend rather than `sift`. Selecting item (3), then (2), resumes `gen`. The generated valve is displayed as an icon in the window as for a process. Initially, the valve is in automatic migration mode and the default buffer is set to 100.

After process `gen` finishes generating data, information in the valve is displayed in a new dialog window if item (4) is selected. Figure 2 shows that buffer contents are modified by deleting the number 8. 0Assume, then, that the window is closed and flushes all buffer data flushed.

Flushing data causes `sift` to resume(Figure 3) with the stream graph eventually becoming stable.

Process charts for each process in a window are shown in Figure 4. In this figure:

- Process `gen` maintains an output stream specified by a vertical gray line at left in the window which connects all third arguments obtained at each execution step. Numbers generated by `gen` are aligned and displayed along this line.

- Process `filter` maintains both an input and an output stream specified by two vertical lines – black and gray in the middle of the window. The input sequence of numbers beside the black line, ranges from 3 to 9, with 8 deleted. Process `filter` generates or does not generate at each execution step when the output sequence on the gray line is referenced.

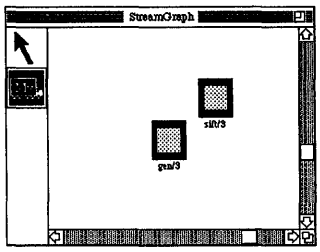


Figure 1: Initial stream graph

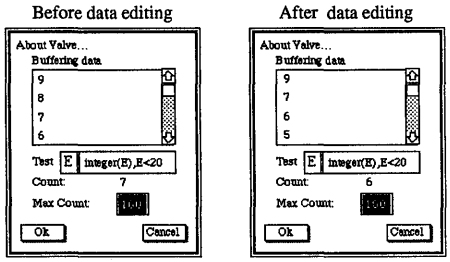


Figure 2: Valve controller display

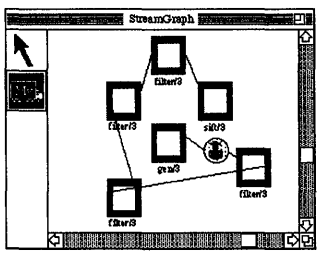


Figure 3: Final stream graph

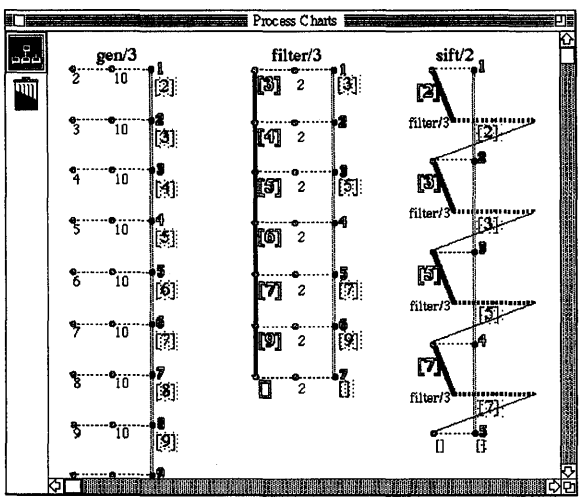


Figure 4: Process chart display

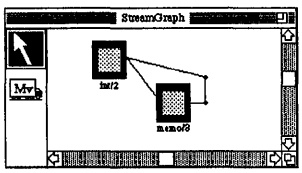


Figure 5: Stream graph for `int/2` and `memo/3`

- The difference between the process chart for `sift` and others is the presence of a process fork specified by a dashed line. Process `sift` also has both input and output streams. The output stream remains unchanged as the input stream is created dynamically. Process `sift` consumes a number from the input stream in the first argument, generating a `filter` and a prime number for the output stream in the second argument in an execution step. The input stream of the created `filter` is connected to the original input and the output stream to the new input stream of `sift`.

Listing 2: Bounded buffer program

```
process int(state,port), memo(port,port,state).
bb(N):- true | open(N,H,T), int(0,H), memo(H,T,C).
open(0,H,T):- true | H=T.
open(N,H,T):- N>0 | N1:=N-1, H=[_|H1],
        open(N1,H1,T).
int(N,[X|S]):- true | X=s(N), @int(s(N),S).
memo([s(X)|S],T,C):- true | T=[_|T1],
        @memo(S,T1,s(X)).
```

The bounded buffer program is shown in Listing 2. Assume that the program and query `bb(5)` are given. The query goal invokes processes `int` and `memo`, which are connected after internal procedure `open` terminates. Figure 5 shows the stable stream graph. The communication flow of these processes indicates the alternate transition of two states. At left in Figure 6, Process `memo` becomes active by consuming data derived by the inactive `int` and a stream functor derived by the previous `memo`. At right, data from the inactive `memo` activates `int`.

5 Conclusion

We have proposed a process oriented debugger(POD) for GHC programs based on a computation model for processes and streams. The POD enables

- Overall behavior of a process to be controlled by manipulating data in streams and arbitrary delaying the transmission and reception of data between processes,

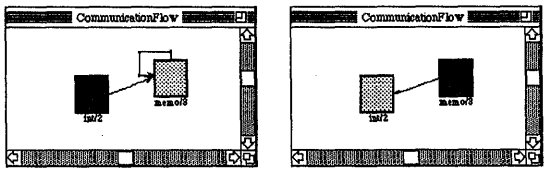


Figure 6: Communication flow transition

- Process causality to be shown using animated figures of processes and streams in both stream graph and communication flow displays,
- Stream connectivity to be organized and shown in a process chart, as a structure of lines connecting the arguments of a process.

Because individual goal execution is not a concern, our debugger gives some information such as input and output substitutions and timing in less detail, making it necessary to include a viewpoint in the future that interprets the original sequence of primitives in such a way that the user can follow it.

Our debugger is implemented using reflection and program transformation. Reflection makes it easy to describe extended unification, and program transformation guarantees the efficient execution of guard goals under the standard guard execution mechanism.

Acknowledgments

This research has been carried out as part of the Fifth Generation Computer Project of Japan. Dongwook Shin and Youji Kohda contributed insightful comments. Masaki Murakami assisted in formalizing streams. Simon Martin helped with the English. The author would like to express thanks to them.

The research originated in his postgraduate study, and he is indebted to Hirotaka Uoi and Nobuki Tokura of Osaka University for their invaluable advice.

References

[Goldszmidt *et al.* 1990] G.S.Goldszmidt, S.Yemini, S.Katz: "High-level Language Debugging for Concurrent Programs", ACM Transactions on Computer Systems, Vol.8, No.4, pp.311-336, November 1990.

[Kahn and MacQueen 1977] G.Kahn, D.B.MacQueen: "Coroutines and Networks of Parallel Processes", Information Processing 77, North-Holland, pp.993-998, 1977.

[Maeda *et al.* 1990] M.Maeda, H.Uoi, N.Tokura: "Process and Stream Oriented Debugger for GHC programs", Proceedings of Logic Programming Conference 1990, pp.169-178, ICOT, July 1990.

[Shapiro and Takeuchi 1983] E.Shapiro, A.Takeuchi: "Object Oriented Programming in Concurrent Prolog", New Generation Computing, Vol.1, No.1, pp.25-48, 1983.

[Shin 1991] D.Shin: "Towards Realistic Type Inference for Guarded Horn Clauses", Proceedings of Joint Symposium on Parallel Processing '91, pp.429-436, 1991.

[Smith 1984] B.C.Smith:"Reflection and Semantics in Lisp", Conference Record of the 11th Annual Symposium on Principles of Programming Languages, pp.23-35, ACM, January 1984.

[Takeuchi 1986] A.Takeuchi: "Algorithmic Debugging of GHC programs and its Implementation in GHC", ICOT Tech. Rep. TR-185, ICOT, 1986.

[Tanaka 1988] J.Tanaka: "Meta-interpreters and Reflective Operations in GHC", Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pp.774-783, ICOT, November 1988.

[Tribble *et al.* 1987] E.D.Tribble, M.S.Miller, K.Kahn, D.G.Bobrow, C.Abbot and E.Shapiro: "Channels: A Generalization of Streams", Proc. of 4th International Conference of Logic Programming(ICLP)'87 Vol.2, pp.839-857 (1987).

[Ueda 1985] K.Ueda:"Guarded Horn Clauses", ICOT Tech. Rep. TR-103, pp.1-12 (1985-06).

[Ueda and Chikayama 1985] K.Ueda, T.Chikayama: "Concurrent Prolog Compiler on Top of Prolog", in Proc. of Symp. on Logic Prog., pp. 119-126, 1985.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A New Parallelization Method for Production Systems

E. Bahr, F. Barachini, H. Mistelberger

Alcatel Austria–ELIN Research Center
Ruthnergasse 1–7, A–1210 Vienna, Austria

Abstract

The growing importance of expert systems in real–time applications reveals the necessity of reducing response times. Since uniprocessor optimizations of production systems have widely been explored, only multiple processor architectures appear to provide further performance gain. Efficient exploitation of the inherent parallelism of production systems, however, requires suitable algorithms for load balancing without simultaneously increasing organization or communication overhead. We present a novel parallel algorithm for PAMELA expert systems, based on dynamic distribution of data processing. The concept is supported by a transputer based architecture with an advanced interconnection structure.[1]

1 Introduction

PAMELA (PAttern Matching Expert system LAnguage) [Barachini and Theuretzbacher 1988, Barachini 1988] was originally designed as a high performance rule–based expert system language especially suited to treat real–time problems. PAMELA's inference engine is highly optimized and makes the language one of the most efficient platforms for rule–based systems on uniprocessors. Nevertheless, the computational complexity of rule–based programs leads to considerable response times. Significant additional speed–ups are expected from parallel execution of the inference engine.

Parallel PAMELA (P^2AMELA) uses a parallel matching scheme not restricted to a specific matching algorithm. The matches are performed concurrently on a number of identical processing elements, requiring only little communication. This is achieved by means of a special scheduling algorithm. The parallelization algorithm is able to incorporate all optimization techniques of the serial PAMELA version.

A transputer based architecture, the "Parallel PAMELA Research Engine" (PRE), has been developed to support the needs of the parallel version of PAMELA. PRE uses a personal computer as master processor, with a multicast interface from the PC to the processing elements [Kasparec *et al.* 1989]. PRE is a research architecture and scalable to 32 transputers. This limitation is not due to the parallelization algorithm but arises from intended cost and complexity restrictions for the hardware architecture. Moreover, it is well known from the literature that the inherent parallelism in typical present–day production systems does not allow speed–up factors of more than 20. Hence, the number of 32 transputers is no obstacle for performing significant run–time experiments.

We discuss in detail the mapping of the fine–grain algorithm onto the coarse–grain PRE architecture. Preliminary performance data of a few hand–coded examples show the efficiency of our algorithm in exploiting inherent parallelism. These experiences serve as a motivation for a full implementation of a parallelizing production system compiler, which is in the final stage of development.

2 Production Systems

A (forward chaining) production system (PS) consists of a production memory containing rules, and a working memory (WM) containing data (working memory elements, WMEs) representing the system state. Real–time production systems are able to communicate with the outside world, e.g. for sampling data or for sending messages to another system.

A rule resembles the well–known IF...THEN... statement. It consists of a left hand side (LHS, corresponding to the IF–part) and a right hand side (RHS, corresponding to the THEN–part). The PS execution breaks into a sequence of "recognize–act cycles" (RACs). A single RAC consists of the following steps:

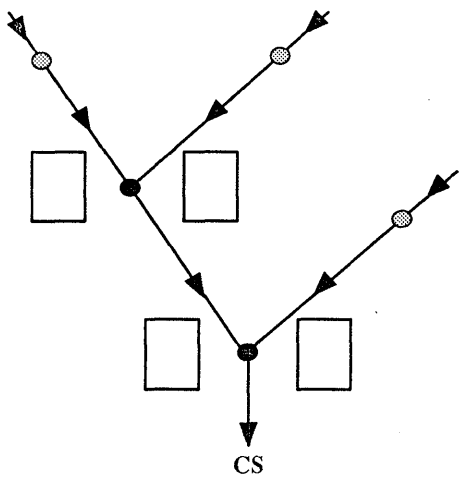
- During the "match phase", the LHSs satisfied by the WMEs are determined. For each valid rule a corresponding instantiation enters the "conflict set" (CS).
- During "conflict set resolution" (CSR) one of the rule instantiations in the CS is selected.
- During the "act phase" the RHS statements of the selected rule are executed. These statements usually change WM or initiate communication with the outside world.

1 This research is sponsored by the Austrian Innovations– und Technologiefonds as part of the InFACT project.

3 The Match Algorithm of Sequential Pamela

The RETE [Forgy 1979, Forgy 1982] and the TREAT [Miranker 1987] algorithm are the best known state saving algorithms, which avoid recomputations of comparisons done in previous RACs. Both algorithms map the patterns of the LHSs of the rules to nodes of a network. The inference engine of PAMELA uses a modified version [Barachini and Theuretzbacher 1988] of the RETE algorithm. Since we have chosen the RETE also for the implementation of our parallelization method, we sketch the basic mechanisms within a RETE network.

When a WME is added to or removed from the WM, a plus–token resp. a minus–token representing this action is passed to the RETE network. In one–input nodes (1INs) attributes of the incoming token are compared against constant values. Two–input nodes (2INs) have a token memory for each input. An incoming plus–token is stored in the token memory, a minus–token removes the corresponding 2IN token from the memory. In 2INs attributes of each incoming token will be compared against attributes of all tokens in the opposite token memory, according to the conditions in the LHS. On each (successful) match, a new token is generated and is sent to the successor node. If a token leaves the RETE network a rule instantiation enters the CS. Figure–1 shows a RETE network with three 1INs and two 2INs.



Figure–1: RETE network of a rule with three patterns

4 Parallelization of Production Systems

Before discussing parallelization, it seems appropriate first to distinguish several classes of parallel architectures. In the familiar Flynn taxonomy [Flynn 1972], SIMD (single instruction, multiple data), MISD (multiple instruction, single data), and MIMD (multiple instruction, multiple data) architectures constitute the variety of parallel architectures. Although there have been attempts to implement production systems on SIMD machines [Forgy 1980], MIMD architectures obviously better match the needs of production system algorithms. Parallel (distributed) systems of the MIMD class fall into two categories [Bhuyan 1987], multiprocessors (all processors share main memory) and multicomputers (each processor has its own local memory with its local address space, a processor cannot directly access another processor's local memory. Communication is accomplished via message–passing).

Two performance measures are of particular interest in evaluating parallel systems, speed–up (defined as the ratio of the execution times for one and for n processors) and efficiency (defined as speed–up divided by the number of processors) [Eager *et al.* 1989]. Efficiency depends on the ratio of communication and computation. Limiting factors are memory contention (with multiprocessors) and the communication overhead (with multicomputers), respectively.

Soon after the invention of the state saving algorithms, various investigations have been started on parallel architectures for production systems. There are several levels of parallelism inherent to production system algorithms like the one used in PAMELA. Apart from application parallelism (concurrent execution of loosely coupled production system tasks), there exists match parallelism on rule, inter–node, and intra–node level, act parallelism, and CSR parallelism [Gupta 1986]. The usefulness of exploiting a particular type of parallelism depends on the time spent for each phase. Typical numbers for RETE production systems are: match (up to) 90%, act 5%, and CSR 5% [Forgy 1979, Gupta 1986]. Most investigations therefore have concentrated on concurrent execution of the match phase. However, newer studies have shown[2] that some production systems spend considerably less than 90% in the match phase. With rule level parallelization, for the time of one RAC, each rule is assigned to a different processing element (PE). With inter–node level parallelization, each node of the RETE–network is assigned to a particular PE, whereas with intra–node level parallelization comparisons within a node are assigned to different PEs.

So far, none of the implementations of these ideas [Butler *et al.* 1988, Gupta 1986, Gupta and Tambe 1988, Kelly and Sevoria 1987, Miranker 1984, Oshisanwo and Dasiewicz 1985, Schreiner and Zimmermann 1987, Shaw 1987, Stolfo 1984, Tenorio 1984, Tien and Raghavendra 1987] has been

2 Private communication with Daniel Miranker

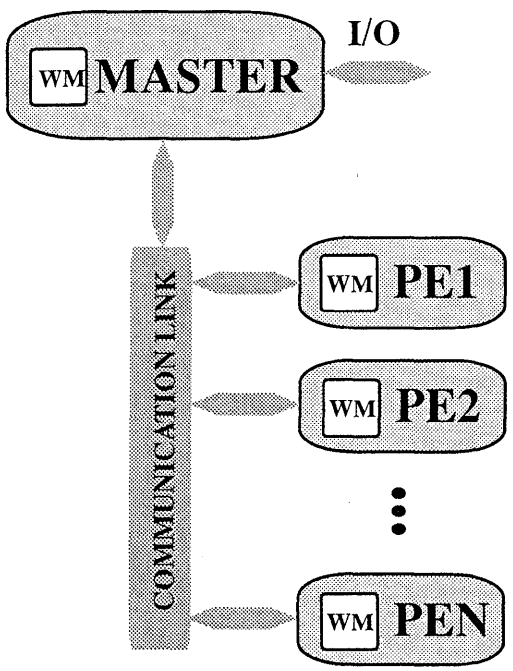
able to simultaneously cope with bottle–necks due to communication overhead or due to shared resources, and load balancing problems. The approach presented in this paper is placed among the intra node parallelizations, but avoids the above–mentioned problems. The algorithm also exploits parallelization of the CSR and is not restricted to RETE but can be applied to TREAT as well.

5 The Basic Idea of Independent Match Parallelization

Anticipating the very simple overall structure of the architecture (figure–2) we can sketch the steps of a RAC in Parallel PAMELA:

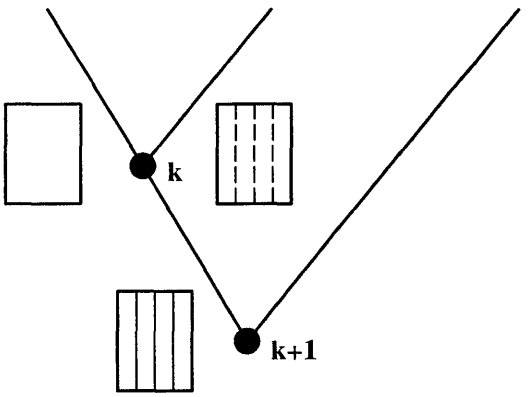
- During the match phase the comparisons are assigned to the PEs by a scheduling algorithm (without inter–PE communication).
- Each PE performs its local CSR (which means also a parallelization of the CSR) and sends its candidate rule instantiation to the "master" processor.
- The master selects one of these candidates (global CSR), executes the RHS of the corresponding rule, and sends the WME changes back to the PEs.

At the beginning of an RAC, each PE therefore must be able to decide independently which partition of the expected comparisons it intends to perform. This decision is made *dynamically during run–time* by a special scheduler running on each PE.



Figure–2: Hardware architecture

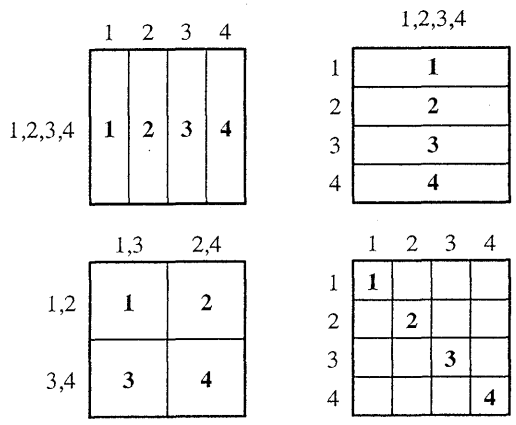
In order to illustrate the idea of the independent match parallelism, we consider a 2IN of a RETE–network (figure–3). It is assumed that both token memories of node *k* are known to all PEs. Each PE has physical copies of these memories and in this sense they are global. These memories have been independently generated from the WM, which is also global – ie. there is a copy of the WM on each PE. The task to be carried out is to compare each token of the left token memory with each token of the right memory, according to the comparison prescription of the 2IN.



Figure–3: Partitioning of comparisons

In order to partition the comparison among the PEs, either the left or the right memory is divided into a number of blocks, equal to the number of PEs (in our example we assume 4 PEs). This partitioning is only "virtual" in the sense that both memories are still global. This is indicated by the dashed lines in figure–3. The partition just means that during the match phase in node *k*, the *m*–th PE takes the tokens in the *m*–th block and compares them against all tokens in the opposite memory. In this way, all comparisons in node *k* are performed by the PEs $m = 1,...,N$. But the run–time is reduced by a factor $1/N$, provided that the partitioned memory contains enough tokens. Each PE generates its own tokens corresponding to its successful matches. This leads to disjoint parts of the left memory at node $k+1$. These parts are local to each PE, ie. part *m* is only known to PE *m* (indicated by the solid lines in figure–3). The matches in the subsequent nodes performed by PE *m* can only be done with its local data. One can easily see that the conjunction of all comparisons gives the whole set of comparisons of the uniprocessor version. This is a necessary condition for consistency.

We demonstrate the consistency of the partitioning algorithm by an example with four PEs, which should perform all matches in a two–input node. The input memories of this node are assumed to be global. The left memory is represented by the vertical axes of the squares in figure–4 and the right memory by the horizontal axes. Then we have three possibilities to distribute the matches between the PEs: (virtual) partitioning of either the left token memory (first square



Figure–4: Consistency considerations

in figure–4) or the right token memory into four parts (second square), or partitioning of both memories into two parts with a suitable assignment of tokens to PEs (third square in figure–4). The last square in figure–4 represents an assignment of the tokens violating the consistency since not the whole cross product of matches is performed. This restriction can be easily taken care of by the following procedure. A partitioning number p_k is assigned to the partitioning node k. p_k is the dyadic logarithm of the number of portions into which the matches in node k has been partitioned. Formally we can define $p_k = 0$ for unpartitioned matching in node k. Furthermore, a maximum partitioning number p_{max} is introduced. This number can be calculated by the dyadic logarithm of the number of PEs, which is always a power of two. Then the restriction takes the form $p_k \leq p_{max}$ In general, the left memory in a RETE–node need not be global due to previous partitionings in predecessor nodes. In this case the right memory must not be fully partitioned according to the consistency requirements mentioned above.

Since each PE holds the whole RETE–network the PEs can process the data, assigned by the partitioning algorithm, without communication with other PEs. The matches are performed by using data which is local or global in the logical sense but strictly local with respect to the physical PE. During the match phase all required global data items are *not accessed by communication* with other PEs but are generated from the global WM, which is located on each PE.

This somewhat simplified picture can be applied to all active nodes in the RETE–network and shows three major points of our approach:

- the approach relies on data parallelism in token memory rather than on program parallelism,
- a very fine grained *dynamic* distribution of matches among the PEs leads to good load balancing,
- no communication is necessary during the match phase, since no PE requires data from another PE.

Compared to a *static* assignment of partitioning nodes our method is much more flexible, which is crucial if data load varies over time. This is especially the case for real–time production systems communicating with the outside world.

In order to clarify some open questions, a few remarks should be made. So far, we have only considered comparisons in 2INs and have not included those in 1INs. In principle it is possible to split the token flow already in an 1IN. Since the 1IN matches consume less than 5% of the total match time of a typical production system, we decided to discard this possibility in favour of more flexibility in partitioning later RACs.

For simplicity it has been assumed that the token memories contain a sufficient number of tokens, so that partitioning leads to parts with nearly equal numbers of tokens. In real life this may be a bit too optimistic. Assuming we have 4 PEs and only 3 tokens in the memory of a node, a division into four parts excludes one PE from processing this node and all successor nodes for current path of token flow. Therefore, three is the maximum speed–up factor in this node. Since we need no synchronization point after the execution of a node, the free PE can process another node in the meantime. Nevertheless, it is advisable to partition large memories because this reduces the chance of idle PEs. A special scheduling algorithm is called on each PE at the beginning of a match phase, which estimates in advance the optimum nodes for partitioning.

In reality the matching procedure is more complex since several token packages may enter the RETE–network at different nodes each RAC. The interference between token packages can be easily handled by processing package by package. All matches in the course of the token package's flow through the network are performed before the next package is processed. Furthermore, a token package can be partitioned at several nodes. This is allowed as long as the generalized consistency requirement

$$\sum_{k=first}^{last} \pi_k \leq \pi_{max} \qquad (1)$$

is obeyed. Due to the fact that we can have nodes with partially partitioning, ie. $p_k < p_{max}$, left memories can hold data of different "degree of locality", generated during previous RACs. Such data is known by fixed subsets of PEs. Therefore, if an incoming token package has to be matched against a left memory, the package might branch into token packages of different locality degrees. In all subsequent nodes these token packages have to be processed separately.

When rule instances finally enter the CS it must be guaranteed that the local CSs are disjoint in order not to contain dublicated instances. This is achieved by enforcing condition (1) but with "=" instead of "≤". Since the PEs' conflict sets form disjoint sets most of the CSR is automatically parallelized. Only the CSR for the best candidates received from the PEs is done by the master. CSR parallelism is only

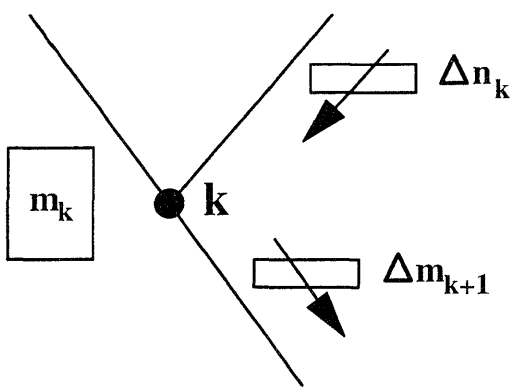
possible when the priority of rule instances do not depend on the existence or priority of other rule instances so that a global view of the CS is necessary. However, for LEX, MEA [Cooper and Wogrin 1988] and many other CSR strategies our algorithm exploits parallelism.

In contrast to rule–level parallelization our algorithm can take full advantage of RETE–network sharing. This is due to the fact that our algorithm relies on a kind of data flow splitting which is implicitely controlled by above mentioned consistency precautions.

6 The Scheduling Algorithm

At the beginning of each RAC, new tokens enter the RETE–network. They are counted and buffered into token packages. These incoming tokens have to be matched against the opposite memories in 2INs and emerging tokens are passed to the successor nodes. The task of the scheduling algorithm is to predict the match activity within the RETE–network for each token package. For this reason, the scheduler needs actual information about the number of entering tokens, size of token memories, and statistical data.

Since the scheduler works independently on each PE, it is only allowed to use globally known data. Otherwise, there would be no guarantee that the schedulers on the PEs arrive at the same results (e.g. decision on the partitioning nodes).



Figure–5: A typical two–input node

Figure–5 shows a typical situation in a 2IN. The number of matches in node k is just the product of the number of incoming tokens and the number of tokens in the opposite memory:

$$V_k = n_k \delta m_k$$

The number of emerging tokens (successful comparisons) can be estimated by

$$\delta m_{k+1} = p_k V_k$$

p_k is the probability that a particular match is successful. p_k itself is estimated by the ratio *(successful matches)/matches* of previously performed comparisons in node k and will be continuously updated.

Having calculated the number of expected matches for a certain entering token package in all relevant nodes of the RETE–network, the scheduler decides upon the optimum partitioning nodes for the considered token package. For this reason, the scheduler searches for the minimum of a special function μ representing a measure for the load balance among the PEs. The argument of this function is the number of the partitioning node *part*. For example, such a function could have the following form:

$$\mu(part) = \sum_{i=first}^{part-1} V_i + \sum_{i=part}^{i=last} \frac{V_i}{F_i}$$

From the entrance node *first* of the considered token package to the node *part–1*, no parallelization takes place. This contribution is represented by the first sum. The partitioning node *part* and all its successor nodes are parallelized. Hence, the number of comparisons per PE is only a fraction of the total number of comparisons in each node, leading to speed–up factors F_i in the second sum. These factors range between *1* and the number N of PEs. $F_i = 1$ means that one PE performs all matches in node i, $F_i = N$ refers to the most balanced parallelization. It can be easily shown that the speed–up factors cannot increase from one node to its successor, ie. $F_i \leq F_j$ for $i > j$. If, for instance, only one comparison has to be performed in node *first* then, obviously, $F_{first} = 1$ and $F_i = 1$ for all subsequent nodes. This kind of unbalanced distribution of comparisons has already been mentioned in the previous section. For the minimum function μ to work sufficiently well, the V_i's must represent the major portion of work to be performed during the matching. If it turns out that insertions into token memories take a significant amount of time, appropriate terms have to be added.

Of course, the scheduling scheme can be generalized to several partitioning nodes for each token package. This is achieved by iterative application of the minimum search, with updated V_ks for $k \geq part$ after each step.

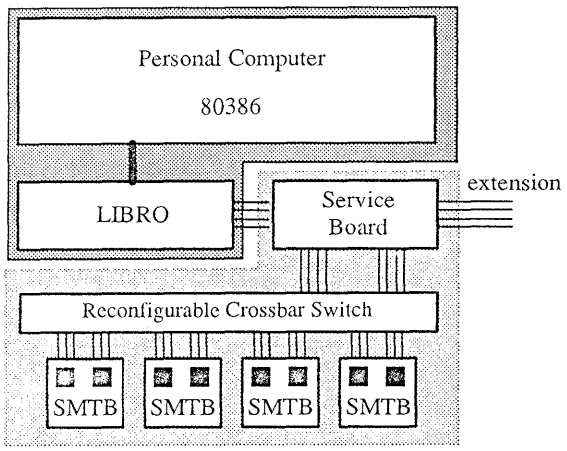
After having determined the partitioning node part, the token package actually enters the RETE–network.

7 The Parallel PAMELA Research Engine

It is a well–known experience that performance of (very expensive) shared–memory multiprocessors degrades at higher n (> 4) due to memory contention. The decision therefore has been made to construct a parallel architecture for PAMELA, offering the scalability of message passing machines as well as tightly coupled pairs of processors (figure–6). This tight coupling will facilitate future experiments using small

shared–memory subclusters constituting the otherwise loosely coupled architecture. The P²AMELA Research Engine (PRE) is a prototype serving for the evaluation of parallelized PAMELA (P²AMELA) expert systems.

In order to allow the PRE fit into a standard environment, an industry–standard 386–based PC was selected for the master processor. This allows the use of standard operating systems and tools (Unix, XWindows) as well as to interface to a variety of networks. The full PAMELA–C expert system shell therefore can be ported to the PRE.

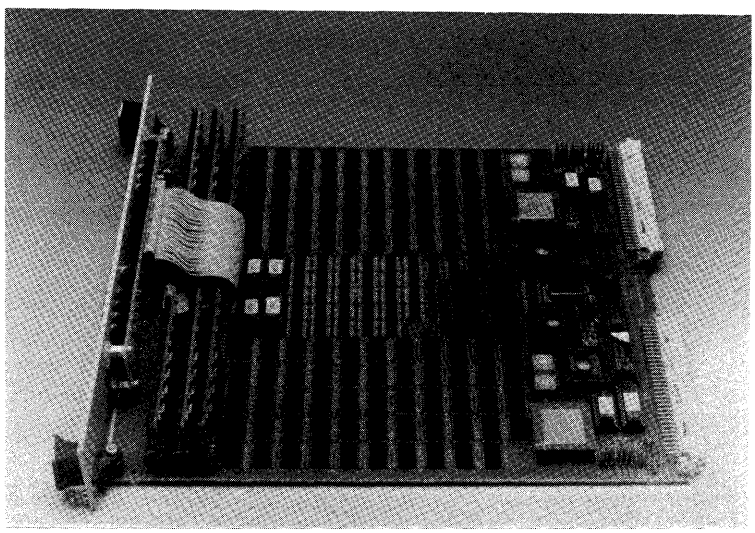


Figure–6: Basic architecture of the PRE

In implementing the PEs, the design is based on the Inmos transputer. In the course of the design, two particular problems were to be solved, namely (i) to effectively update working memory on the PEs, and, (ii) to exchange data between a transputer pair on one PE. Therefore, up to 16 double transputer boards can be served by the PC.

To solve the first problem indicated above, a 'Link Broadcast Interface' (LIBRO) board was developed. The first version of LIBRO was implemented on a PC add–on card. This PC LIBRO is a quadruple transputer link interface for personal computers; up to four boards per PC can be stacked and treated as a single device. The LIBRO solution allows WM contents and other global information to be broadcast through four to sixteen links under control of a master processor (PC). Each link channel is buffered in both directions, so fast access with string primitive instructions is possible.

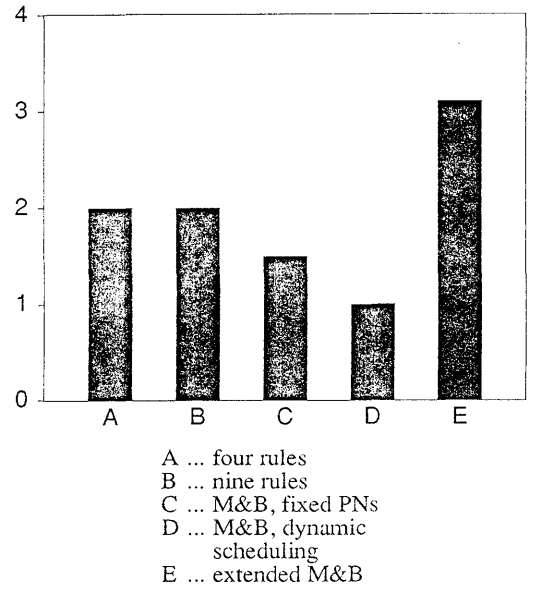
The core of the P²AMELA Research Engine consists of Swapable–Memory Transputer Board (SMTB) modules (figure–7) for the PEs. An SMTB incorporates two IMS–805 transputers at 25MHz with three free links each. The fourth link of each transputer is used to access a common memory swapping controller. The latter controls the access to four 1MByte blocks of memory. Each memory block is allocated to one of two transputers at a time. Control information supplied by the two transputers through a dedicated link is used to change the allocation status. Therefore, we have a kind of shared memory between the two transputers on an SMTB.



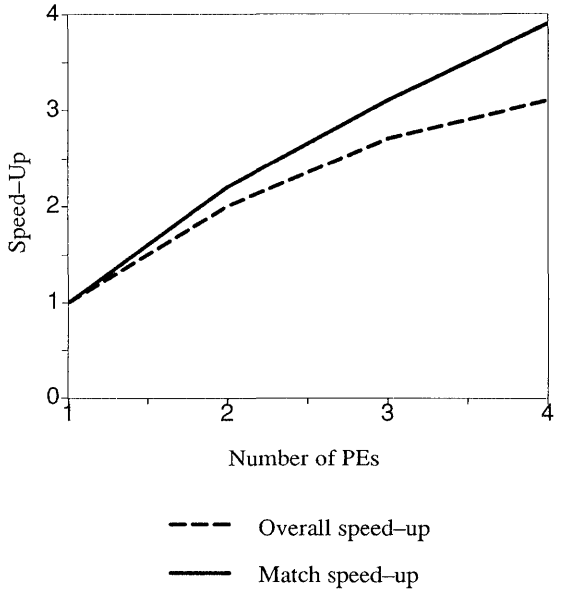
Figure–7: Swappable Memory Transputer Board

8 Preliminary run time measurements

In the absence of a production system compiler for the parallelization method described, we could only encode a few examples on a simple commercially available transputer based architecture. The results have been encouraging, although most of the examples exhibit rather low inherent parallelism. In addition, the scheduling algorithm has not yet been optimized and it therefore causes some overhead.



Figure–8: Example run–time measurements



Figure–9: Speed–up dependence for extended M&B

The examples A and B in figure–8 are simple production systems, characterized by four and nine rules, respectively. Examples C and D use implementations of "Monkeys and Bananas" with static and dynamic partitioning nodes, respectively. Unfortunately, the activity in the RETE–network for these examples is very low so that the low speed–up is not very surprising. Example E is an extended version of "Monkeys and Bananas" using more WMEs. It reveals the full power of the algorithm, yielding a speed–up factor of about 3.9 for the match phase (four PEs). Taking into account all overheads, the factor of 3.1 is still remarkable. Figure–8 shows the speed–up dependence on the number of PEs. Although these examples do not provide a representative set of production systems , they show the existence of expert systems with speed–ups ranging from minimal (one) to maximal (number of PEs) values. These results do not prove the efficiency of our parallelization algorithm but they serve as a motivation for further investigations.

9 Summary and Future Directions

We presented a new approach to parallel execution of production systems, exploiting data parallelism in token memory. The approach has the following advantages compared to other published parallelization methods that rely on program parallelism:

- high utilization of processing power,
- no need for locking mechanisms for the consistency of the RETE–network,
- small communication overhead, no bottle–neck on shared resources
- a scalable architecture

Possible disadvantages of the method presented may be:

- the memory per PE will be about the size of the mono–processor version,
- the scheduling algorithm causes additional computation overhead.

After the Parallel PAMELA–C system is fully implemented, measurements on a representative set of production systems will be performed in order to assess the quality of the parallelization method. Various strategies for scheduling on PRE and alternative parallel architectures will be investigated. In this respect, the adaption of the presented algorithm to a shared memory architecture is of particular interest. The usage of global data both simplifies the scheduling algorithm and increases its accuracy and flexibility. But in order to avoid memory and bus contention, the access to the global memory must either be infrequent or decoupled between the processing elements. Since the data of the RETE network are frequently accessed the contention problem does not allow a straightforward solution. Further investigations of this matter will be the subject of future research.

Acknowledgements

We owe thanks to J. Doppelbauer, H. Gräbner, F. Kasparec, and T. Mandl who constructed the multicomputer architecture we are going to use for the execution of production

systems. We are especially grateful to J. Doppelbauer for providing us with a photo of the Swapable Memory Transputer Board and with technical information about the hardware.

References

[Barachini and Theuretzbacher 1988] Barachini F., Theuretzbacher N.: "The Challenge of Real–Time Process Control for Production Systems", The Seventh National Conference on Artificial Intelligence (AAAI–88), St. Paul, Minnesota, Vol II, 1988.

[Barachini 1988] Barachini F. : "PAMELA: A Rule–Based AI Language for Process–Control Applications", Proceedings on the first International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, Vol 2, pp 860–867, Tennessee, 1988.

[Bhuyan 1987] Bhuyan L.N.: "Interconnection Networks for Parallel and Distributed Processing"; IEEE Computer, June 1987, pp. 9 ff.

[Bhuyan 1989] Bhuyan L.N., Yang. Q., Agrawal D.P.: "Performance of Multiprocessor Interconnection Networks", IEEE Computer, February 1989, pp. 25 – 37

[Butler *et al.* 1988] Butler P.L., Allen J.P., Bouldin D.W. : "Parallel Architecture for OPS5", The 15th Annual International Symposium on Computer Architecture, Honolulu, Proceedings pp 452–457, 1988.

[Cooper and Wogrin 1988] Cooper T.A., Wogrin N. : "Rule–based Programming with OPS5", Morgan Kaufmann Publishers, Inc., Palo Alto, USA, 1988.

[Eager *et al.* 1989] Eager D.L., Zahorian J., Lazowska E.: "Speedup Versus Efficiency in Parallel Systems", IEEE Transactions on Computers, Vol. 38, No. 3, March 1989, pp. 408 ff.

[Flynn 1972] Flynn M.J., Some Computer Organizations and Their Effectiveness, IEEE Trans. Computers Vol. 21, No. 9, Sept. 1972, pp. 948 – 960

[Forgy 1979] Forgy C.L. : "On the Efficient Implementation of Production Systems", Ph.D. Thesis, Carnegie–Mellon University, 1979.

[Forgy 1980] Forgy C.L.: "Note on Production Systems and ILLIAC IV", Technical Report CMU–CS–80–130, CMU, Pittsburgh, 1980

[Forgy 1982] Forgy C.L. : "RETE : A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem", Artificial Intelligence, Vol. 19, pp. 17–37, 1982.

[Gupta 1986] Gupta A.: "Parallelism in Production Systems"; CMU–CS–86–122, Ph.D. Thesis, Carnegie–Mellon University, March 1986

[Gupta 1987] Gupta A. et al.: "Results of Parallel Implementation of OPS5 on the Encore Multiprocessor"; CMU–CS–87–146, August 1987

[Gupta and Tambe 1988] Gupta A., Tambe M. : "Suitability of Message Passing Computers for Implementing Production Systems", Proceedings of AAAI–88, Vol 2, pp. 687–692, St.Paul, Minnesota, 1988.

[Kasparec *et al.* 1989] Kasparec F., Doppelbauer J., Gräbner H., Mandl T.: "Advanced Transputer Interconnection Techniques"; 1st International Conference on the Application of Transputers (SERC/DTI), Univ. of Liverpool, Aug. 1989

[Kelly and Sevoria 1987] Kelly M.A ., Seviora R.E.: "A Multiprocessor Architecture for Production System Matching"; Proceedings of the AAAI–87, Vol.1,pp. 36–41, 1987 1987

[Miranker 1984] Miranker, D.P.: "The performance Analysis of TREAT : A DADO Production System Algorithm", International Conference on Fifth Generation Computing, Tokyo 1984, revised article 1986

[Miranker 1987] Miranker D.P.: "TREAT: A New and Efficient Match Algorithm for AI Production Systems"; Ph.D.Thesis, Columbia Unversity 1987

[Oshisanwo and Dasiewicz 1985] Oshisanwo A.O., Dasiewicz P.P.: "A Parallel Model and Architecture for Production] Systems"; Proceedings of the 1987 International Conference on Parallel Processing, pp.147–153 May 1985

[Schreiner and Zimmermann 1987] Schreiner F., Zimmermann G.: "PESA 1 – A Parallel Architecture for Production Systems"; Proceedings of the 1987 International Conference on Parallel Processing, pp. 166–169

[Shaw 1987] Shaw D.E.: "NON–VON's Applicability to Three AI Task Areas", IJCAI 1987

[Stolfo 1984] Stolfo S.J.: "Five Parallel Algorithms for Production System Execution on the DADO Machine"; National Conference on Artificial Intelligence, AAAI–1984

[Tenorio 1984] Tenorio M.F.M., "Parallelism in Production Systems", Ph.D. Thesis, University of California, 1984.

[Tien and Raghavendra 1987] Tien S–B.R., Raghavendra C.S.: "A Parallel Algorithm for Execution of Production Systems on HMESH Architecture"; Fall Joint Computer Conference, 1987, pp.349–356

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Performance Evaluation of the Multiple Root Node Approach to the Rete Pattern Matcher for Production Systems†

Andrew Sohn
Department of Computer and Information Science
New Jersey Institute of Technology
Newark, NJ 07102, sohn@cis.njit.edu

Jean-Luc Gaudiot
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2563, gaudiot@usc.edu

Abstract- Much effort has been expanded on special architectures and algorithms dedicated to efficient processing of the pattern matching step of production systems. In this paper, we investigate the possible improvement on the Rete pattern matcher for production systems. Inefficiencies in the Rete match algorithm have been identified, based on which we introduce a pattern matcher with *multiple root nodes*. A complete implementation of the multiple root node-based production system interpreter is presented to investigate its relative *algorithmic* behavior over the Rete-based Ops5 production system interpreter. Benchmark production system programs are executed (*not* simulated) on a sequential machine Sun 4/490 by using both interpreters and various experimental results are presented. Our investigation indicates that the multiple root node-based production system interpreter would give a maximum of up to 6-fold improvement over the Lisp implementation of the Rete-based Ops5 for the match step.

1 Introduction

The importance of production systems in artificial intelligence (AI) has been repeatedly demonstrated by a large number of expert systems. As the number and size of expert systems grow, there has however been an emerging obstacle in the processing of such an important AI application: the large match time. In rule-based production systems, for example, it is often the case that the rules and the knowledge base needed to represent a particular production system would be on the order of hundreds to thousands. It is thus known that applying a simple matching algorithm to production systems would yield intolerable delays. The need for faster execution of production systems has spurred research in both the software [2,3,7,8] and hardware domains [6,11].

In the software domain, the Rete state-saving match algorithm has been developed for fast pattern matching in production systems [2]. The motivation behind developing the Rete algorithm was based on the observation, called *temporal redundancy*, which states that there is little change in database between cycles. By storing the previous match results and using them at later time, matching time can be reduced [1].

Inefficiencies in the state-saving Rete algorithm were identified, based on which the non-state-saving Treat match algorithm was developed [10]. The motivation behind developing the Treat algorithm was McDermott's conjecture, stating that the retesting cost will be less than the cost of maintaining the network of sufficient tests [9].

In this paper, we further identify the inefficiencies of the Rete algorithm, based on which we introduce a pattern matcher with *Multiple Root Nodes* (MRN). Section 2 gives a brief introduction to production systems and the Rete match algorithm. Section 3 explicates the inefficiencies of the Rete matcher. A Lisp implementation of the MRN-based production system interpreter is then presented along with the distinctive features of its implementation.

Section 4 presents benchmark production system programs and experimental results on both the Rete-based OPS5 interpreter and the MRN-based interpreter. Various statistics gathered both at compile time and runtime are presented as well. Performance evaluation on the two interpreters are made in Section 5 in terms of number of comparison operations and execution time. The last section concludes this paper.

2 Background

2.1 Production systems

A production system as shown in Figure 1 consists of a *production memory* (PM), a *working memory* (WM), and an *inference engine* (IE). PM (or rulebase) is composed entirely of conditional statements called productions (or rules). These productions perform some predefined actions when all the necessary conditions are satisfied. The left-hand side (LHS) is the condition part of a production rule, while the right-hand side (RHS) is the action part. LHS consists of one to many elements, called *condition elements* (CEs) while RHS consists of one to many actions.

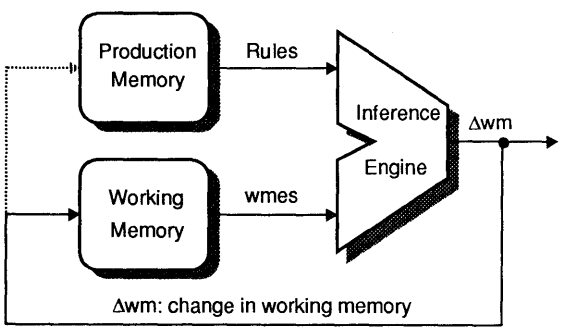


Figure 1: An architecture of production systems

† This work is supported in part by the NSF under grant No. CCR-9013965.

The productions operate on WM which is a database of assertions called *working memory elements* (wmes). Both condition elements and wmes have a list of elements, called *attribute-value pairs* (avps). The value to an attribute can be either *constant* or *variable* for CEs and can be constant only for wmes. A simple production system with one rule is shown in Figure 2. The inference engine executes an inference cycle which consists of the following three steps:

- *Pattern Matching*: The LHSs of all the production rules are matched against the current wmes to determine the set of satisfied productions.
- *Conflict Resolution*: If the set of satisfied productions is non-empty, one rule is selected. Otherwise, the execution cycle simply halts.
- *Rule Firing*: The actions specified in the RHS of the selected production are performed.

The above three steps are also known as Match-Recognize-Act, or MRA. The inference engine will halt the production system either when there are no satisfied productions or a user stops.

2.2 The Rete match algorithm

The Rete match algorithm is a highly efficient approach used in the matching of objects in production systems [2]. The simplest possible matching algorithm would consist in going through all the rules and wmes one by one to find match(es). The Rete algorithm, however, does not iterate over the wmes to match all the rules. Instead, it constructs a condition dependency network like shown in Figure 2, saves in the network results from previous cycles, and utilizes them at a later time.

Production Memory		Working Memory	
Rule1:		wme1:	[(p 1) (q 2) (r *)]
[(c X) (d Y)]	;CE1	wme2:	[(r =) (d +)]
[(b Y)]	;CE2	wme3:	[(c *) (d +)]
[(p 1) (q 2) (r X)]	;CE3	wme4:	[(b 3)]
→		wme5:	[(b +)]
[Remove (b Y)]	;Action 1	wme6:	[(p 1) (q 3) (r 7)]

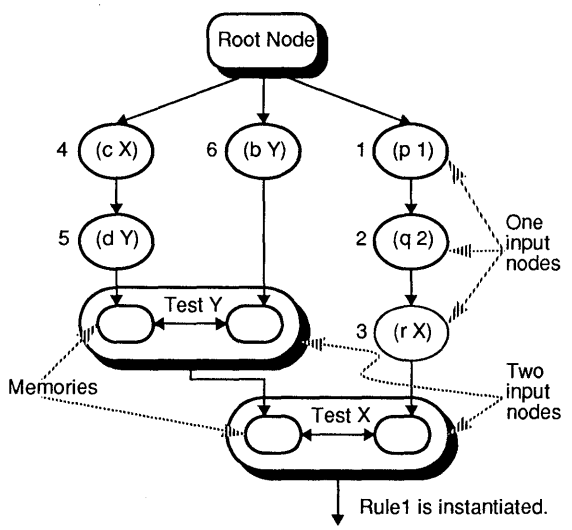


Figure 2: A Rete network for Rule 1.

Given a set of rules a network is built which contains information extracted from the LHSs of the rules. Figure 2 depicts a network for Rule 1, with the following nodes:

- *Root Node* (RN) distributes incoming tokens (or wmes) to sequences of children nodes, called one-input nodes.
- *One-Input Nodes* (OIN) test intra-element features contained in a condition element, i.e., compare the value of the incoming wmes to some preset values in the condition element. For example, CE1 of Rule1 contains 2 intra-element features and therefore 2 OINs are needed to test them. The test result of the one-input nodes are propagated to nodes, called two-input nodes.
- *Two-Input Nodes* (TIN) are designed to test inter-condition features contained in two or more condition elements. The variable *X*, which appeared in both CE1 and CE3, must be bound to the same value for rule instantiation. Attached to the TINs are left- and right memories in which wmes matched through OINs are saved. The result from two-input nodes, when successful, are passed to nodes, called terminal nodes.
- *Terminal Nodes* (TN) represent instantiations of rules Conflict resolution strategies are invoked to select and fire a rule.

There are other variations to the nodes listed above. Given the above network, the Rete algorithm performs pattern matching and we shall not go into detail. See [1,2,5] for more details.

3 The MRN Matcher and Its Implementation

The multiple root node based interpreter is presented along with its Lisp implementation.

3.1 The MRN Matcher

The Rete algorithm described earlier presents two apparent bottlenecks: one in the root node and the other in two-input nodes, as illustrated in Figure 3. Tokens coming into the root node will pile up on the input arc of the root node since there is one and only one root node which can distribute tokens one at a time to all CEs. For the network shown in

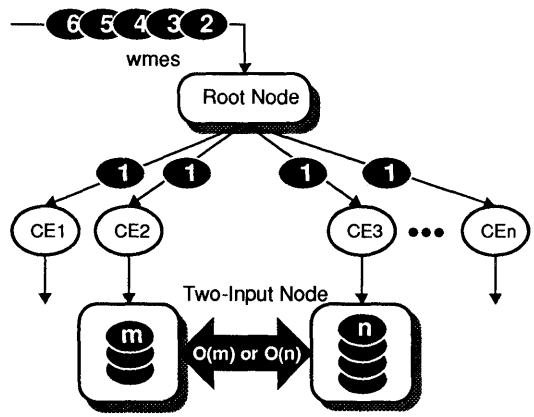


Figure 3: Two bottlenecks of the Rete. (1) piling up of wmes on an arc of the root node, resulting in a sequential distribution of wmes to all CEs one at a time. (2) $O(n)$ or $O(m)$ comparisons in TINs.

Figure 3 where there are n condition elements, the root node will have to make nx distributions to the network when x wmes are present on the input arc of the root node.

The second inefficiency can also be seen on Figure 3. Assuming that m tokens are stored in the left memory of the two-input node and a token is matched on the right input. The arrival of this last token will trigger the invocation of m comparisons with the wmes received and stored in the left memory. Should the situation have been reversed and n tokens be in the right memory, a token on the left side would provoke n comparisons. The internal workings of this two-input node are therefore purely sequential. In order to avoid wasting time in searching the entire memory, an effective allocation of two-input nodes and one-input nodes should be devised. In this paper, we will limit ourselves to the first bottleneck. Discussions on the second bottleneck can be found in[4,5].

The first bottleneck described above can be resolved by introducing *multiple* root nodes (MRN) in the network, as depicted in Figure 4. This introduction of multiple root node is based on the observation that a wme that has n AVPs never matches a CE that has m AVPs where $n<m$. For example, a wme, [(a 1) (b 2)], cannot match a CE, [(a X) (b Y) (c Z)], where X, Y, Z are variable, since the wme is missing the third AVP (c Z). However, a wme [(a 1) (b 2) (c 3) (d 4)] can match the CE. One should note here that the above observation is based on algorithmic behavior, not Ops5 syntactic behavior.

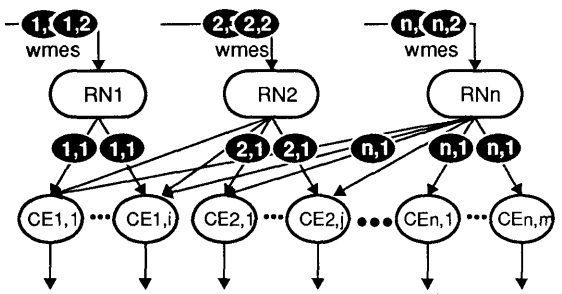


Figure 4: An MRN network. RNn distributes wmes to CEs under RN1 through RNn. A wme (i,j) refers to a wme with i AVPs, where j signifies its arrival order. The MRN network also demonstrates a parallel distribution of wmes, where n RNs can simultaneously distribute n different wmes to the network.

Constructing an MRN network is straightforward. All LHSs are split into condition elements (CEs). All CEs are grouped based on the number of AVPs in a CE, i.e., a CE with n AVPs belongs to a group n. Associated with each group is a root node which distributes a set of wmes to a particular group of CEs of the MRN network. For example, RN2 of Figure 4 distributes wmes with 2 AVPs to those CEs, where each CE has not more than 2 AVPs.

Suppose that the network has n groups, each of which has equally m CEs, i.e., the total number of CEs is nm. Assuming that the number of wmes generated in each production cycle is constant, i.e, k, then the original Rete network will need nmk distribution. Assuming that k wmes are equally distributed over the n groups, i.e., k/n wmes per each group, the MRN network will only need $(1+2+...+n)mk/n$ distributions. For an even distribution of wmes over groups, the MRN matcher is guaranteed to yield 2-fold improvement over the Rete network. Next section will substantiate our prediction. In the mean time, we shall present the Lisp implementation of the MRN-based matcher.

3.2 Characteristics of the MRN implementation

The MRN-based production system has been completely implemented in Common Lisp from scratch. A complete listing of Lisp codes can be found in [12]. Its functionality is 100% up to the Rete-based OPS5. The main features of the MRN implementation are:

- Free of global variables, except a single one which traces the number of wmes generated during the lifetime of a particular production system program,
- Over 90% of the functions written in tail-recursion, and
- A simple data structure using defstruct of Lisp.

A major reason to avoid using global variables is in that the program should be easily ported to various multiprocessor environments without having to change much of its source codes. By not using global variables, the potential communication and synchronization overhead between processes would be reduced when ported to a multiprocessor environment. Furthermore, encapsulating the scope of variables within a function would allow us to analyze the data dependency, if any, between functions, thereby resulting in easy program partitioning. The ultimate goal of parallel processing, extracting and exploiting more potential parallelism from given codes, would then become within a reachable distance. To substantiate this claim, the MRN-based production system interpreter has been implemented in a data-flow language SISAL (Streams and Iteration in a Single Assignment Language) and is currently being ported to multiprocessors, including shared memory multiprocessors such as Cray!

Much effort has been spent on writing the program in tail recursion. One reason to do so was also due partly to the portability to various multiprocessor environments. When functions are written in tail recursion, it can be much easier to understand its behavior since the program tracing is automatic. This easiness in understanding of the behavior of a program will directly translate into an easy conversion to iterations. Those vectorizing compilers or parallelizing compilers can be readily used to convert the Lisp programs into a language suitable for vector or multiprocessors.

The third feature, a simple data structure defstruct, would not necessarily be considered a good feature. The main reason employing defstruct is that it will simplify the implementation process due to its structuredness. This structured approach will shield the data dependency between data, i.e., dynamically changing memories in the network. However, this dynamic data structure consumes more memory space than other data structures such as lists. There is certainly a trade-off between the runtime memory

space and the easiness in programming and debugging. Due to the space constraints, we shall not illustrate implementation details. Complete implementation details can be found in [12].

4 Experimental Results

Benchmark production system programs are presented along with the surface characteristics measured at compile time. Both the Rete-based OPS5 and MRN-based interpreters were executed on Sun 4/490 to measure their *algorithmic* performance. Statistics collected at runtime are: the execution time of a match step, the number of comparison operations for one-input nodes, and the distribution of wmes. All the measurements are done against production cycle numbers.

4.1 Surface characteristics of benchmark programs

The five programs chosen for performance analysis are commonly used ones, as seen from Table 1. Note that the size of production systems is not central to its performance evaluation. Indeed, Gupta has commented that (1) we should not expect smaller production systems (in terms of number of productions) to run faster than larger ones, and (2) there is no reason to expect that larger production systems will necessarily exhibit more speedup from parallelism [5]. The programs used in this study are:

- *Brick Sorting*, to pick a brick from a pool and place them in ascending or descending order,
- *Monkey and Banana* (MAB), for a monkey to grab a banana hanging from the ceiling,
- *N Monkeys and M Bananas* (NMAB), the MAB with *n* monkeys and *m* bananas,
- *Waltz Labeling*, a labeling algorithm developed in computer vision, and
- *N-Queen*, a classical problem which places *n* queens on *nxn* board.

PS	a	b	c	d	e	f	g	h	i
Brick	7	16	15	2336	2	4	4	20	60
MAB	25	70	43	8409	59	5	14	16	58
5MAB	23	60	43	45195	39	5	12	113	278
Waltz	48	198	100	174891	90	5	40	245	297
8-Queen	19	68	71	151985	36	6	11	1044	3866

Table 1: Characteristics of benchmark production systems, where PS=production system program, a=No of rules, b=No of CEs, c=No of acts, d=OINs executed, e= No of TINs, f=No of groups, g=Avg CEs/group, h=Rule firings, i=WMEs generated.

The information collected from the above five production system programs characterize various aspects of the benchmark programs. Our purpose is to measure the relative performance of the MRN approach in terms of execution time along production cycles. What is important in our performance evaluation is the information on groups of a production system program. Indeed, we find that even a small size production system program such as Brick Sorting problem would suffice.

4.2 Measurements on grouping

Grouping the condition elements (CEs) based on the number of Attribute Value Pairs (AVPs) is central to the MRN approach. This would allow us to partition the production systems into many pieces each of which can be processed independent of the incoming newly generated wmes. Measuring the distribution of condition elements over groups at compile time, we can predict the potential parallelism in the given production systems. Figure 5 depicts the distribution curve, where the *x*-axis shows group numbers and the *y*-axis the percentage of each group in a particular production system program.

In the Brick Sorting problem, there are four different groups, where a group-*n* contains condition elements, each of which has *n* Attribute-value Pairs. For example, Group2 occupies slightly above 30% whereas Group 5 does 6% of the total number of CEs. Condition elements of Monkey and Banana are relatively equally distributed over four groups, compared to that of Waltz Labeling where one group is dominant over other groups. This dominance of a group over another is not desirable and does not yield a good performance. We shall come back to this analysis shortly.

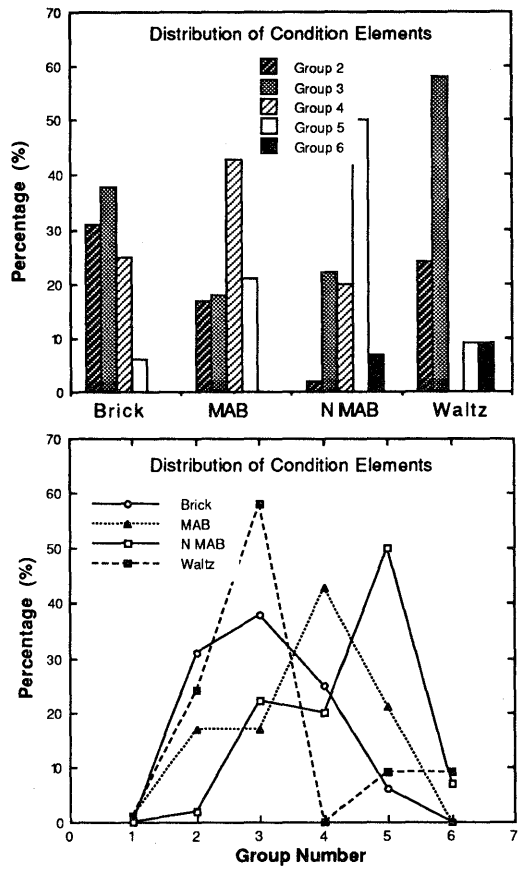


Figure 5: Distribution of CEs over groups measured at compile time.

4.3 Execution time on one-input nodes

Figure 6 shows the execution time of matching one-input nodes measured at each production cycle. There are several points at which execution time run off the boundary. Several points running off the boundary are unimportant since our purpose is to show the relative performance.

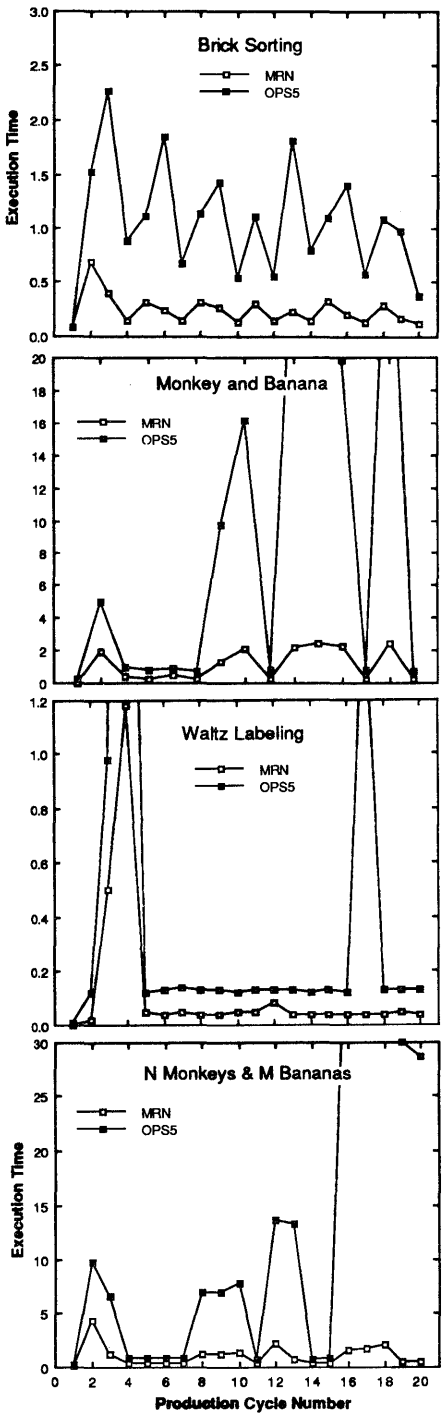


Figure 6: Execution time profile of matching one-input nodes.

For Brick Sorting and Waltz, it appears that both the MRN and OPS5 show a rather regular behavior while they maintain a reasonably constant distance between the two curves along the x-axis. For example, in the Waltz, the differences between two execution time curves for the cycle numbers 5 to 16 are relatively constant, except at the cycle numbers 3, 4, and 17. A similar behavior is also observed in Brick. This kind of proportional distance between two curves is important in predicting the possible outcome of the MRN approach.

The MAB and NMAB, however, exhibit a slightly different behavior compared to Brick and Waltz. For example, the MRN curve in MAB gives an amplification factor higher than the one for Brick or Waltz. This irregular behavior is due partly to the memory management policy, *garbage collection*, in Lisp which contributes to inaccurate performance measurements. We shall give a more accurate measurement shortly. Over all, it is obvious that the MRN outperforms the OPS5 in any of the four problems.

4.4 Number of comparison operations

Another criterion to measure statistics at runtime is counting the number of comparison operations. Consider the following simple Lisp function member:

```
(defun member (a l)
    (cond  ((null l) nil)
           ((equal a (car l)))
           (t (member a (cdr l)))))
```

Suppose that the function is called with (member 1 '(2 6 4 7 1)). It is clear that the function member will be called five times and therefore, the number of comparison operations will be five. Figure 7 shows the number of comparison operations for four programs. When we considered the execution time, we discussed that the behaviors of the four programs are rather irregular. The MRN curve of MAB in Figure 6 gave an amplification factor higher than the one for Brick or Waltz. However, that irregular behavior no longer persists in Figure 7. This consistent behavior is due mostly to the new criterion. Figure 7 again demonstrates that MRN outperforms OPS5 for all programs.

4.5 Distribution of groups

Figure 8 shows the runtime distribution of wmes and condition elements for four programs. Take the Brick Sorting, for example. At runtime, there is no wme generated for group2, group4, and group6. Those wmes generated for Brick at runtime fall into either group3 or group5. As we can observe from Figure 8, there is a considerable amount of discrepancy between the runtime wme distribution and the compile CE distribution.

For MAB, however, the situation becomes different. As we can observe from Figure 8, the discrepancy for MAB becomes much smaller compared to that for Brick. MAB and NMAB show a relatively low discrepancy whereas Brick and Waltz show a rather high discrepancy in terms of the wme distribution and the CE distribution.

Contrary to the compile time distribution of condition

elements, most of the wmes generated at runtime fall into a few distinctive groups. All the four problems have basically two groups actively working at runtime. However, these distribution curves are problem dependent and there is no single rule which can predict the behavior of the runtime distribution of wmes. A simple conclusion we could draw from these discrepancy plots would be that more the discrepancy there is, more the improvement there will be.

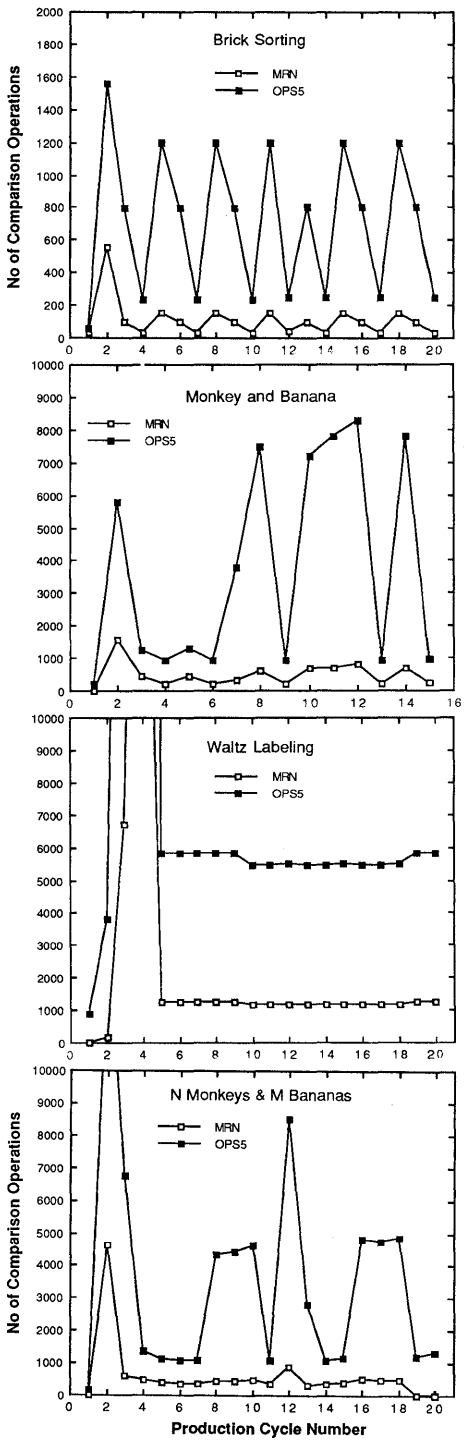


Figure 7: Number of comparison operations on one-input nodes.

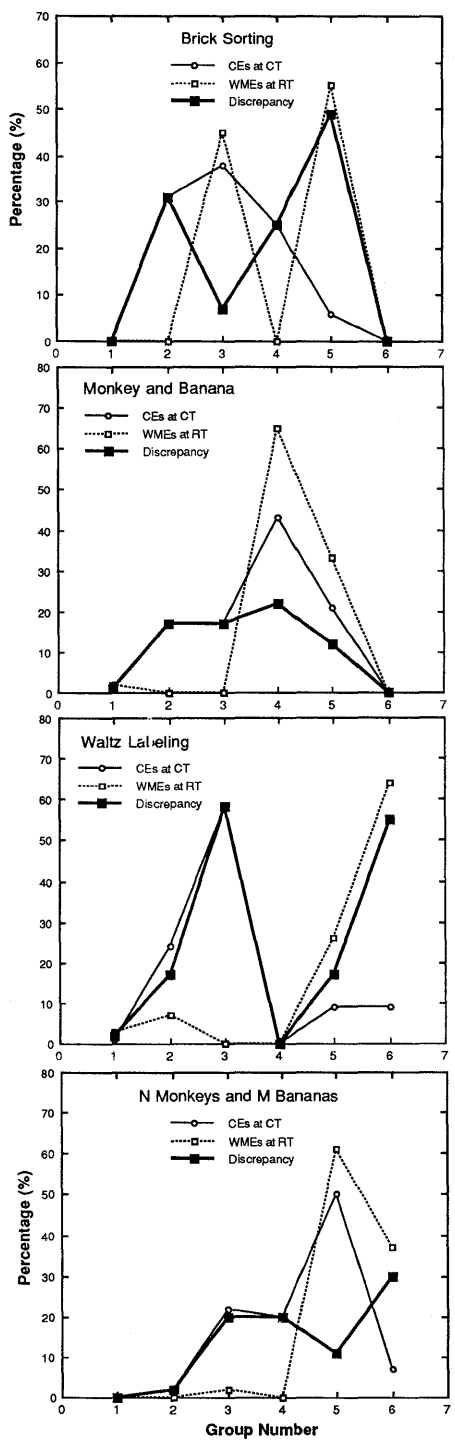


Figure 8: Runtime distribution of wmes.

5 Performance Evaluation

Based on the foregoing three different types of observations, i.e., one-input match time, number of comparison operations, and distribution of wmes, we analyze the performance of both interpreters.

5.1 Comparison of MRN and OPS5

Figure 9 shows the ratio of MRN to OPS5 on one-input match time for four different programs. Here, the ratio means simply the comparison of two match time units for one-input nodes. Again, x-axis is plotted against the production cycle numbers whereas y-axis indicates the ratio of two different approaches.

For Brick Sorting, for example, the one-input match time of OPS5 at production cycle number 13 is about 8 times more than that of MRN. For NMAB, the one-input match time of Ops5 at the cycle 13 is about 17 times more than that of MRN. It is clear that there is a substantial improvement, ranging from 2 to over 20, depending on the programs and the production cycle number.

Figure 10 gives a more accurate performance measure. It uses the number of comparison operations at each production cycle. The x-axis is plotted against the production cycle number while the y-axis is again the ratio of the MRN-based match to Rete-based match. To closely examine the improvement curve, consider again the production cycle number 13 as we did for Figure 9.

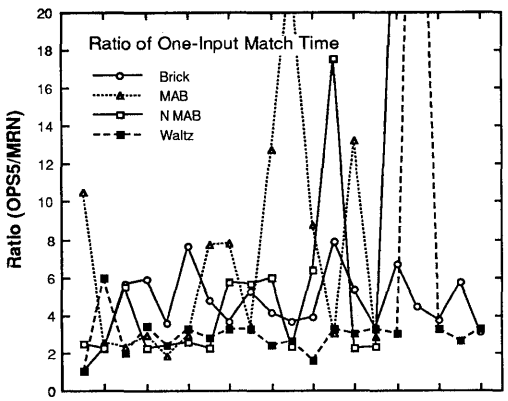


Figure 9: Ratio of one-input match time.

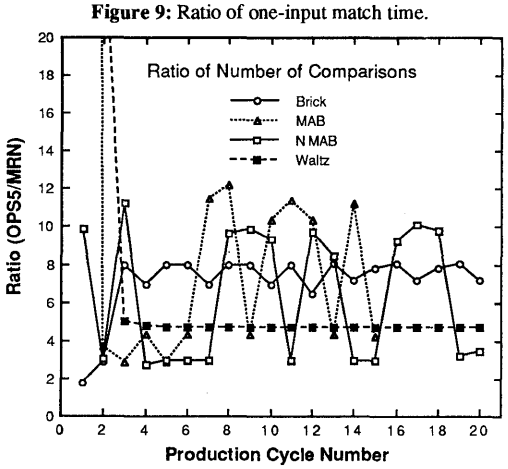


Figure 10: Ratio of number of comparison operations.

For Brick Sorting of Figure 10, the number of comparison operations performed by the Rete-based match at cycle 13 is about 8 times more than that by the MRN-based match. This improvement of 8 is exactly the same as the one we obtained from Figure 9. For NMAB, the improvement, however, becomes different from what we would expect. Closely examining the NMAB curve of Figure 10 at cycle 13, we find that the improvement is 8! We remember that the improvement we obtained from Figure 9 for NMAB at cycle 13 is 17. This is not surprising because measuring the real time can be affected by many factors which we iterated several times. Nevertheless, it is clear from the two improvement curves plotted in Figure 9 and Figure 10 that the MRN-based match algorithm outperforms the Rete-based match algorithm. Since the objective here is to compare the performance of the two match algorithms, the two figures would suffice the stated objective.

There are some other experimental results which are of particular interest but due to space constraints we shall have to be content with what we have presented thus far. When the two figures are summed and averaged along the production cycle number of x-axis, it gives an eventual improvement of *six*. The average improvement of the MRN approach over OPS5 on one-input match time would reach to *six* fold for the four production programs considered in this study.

5.2 Discrepancy in the distribution of wmes and CEs

It is interesting to observe the discrepancy between the compile time distribution of condition elements and the runtime distribution of wmes. By finding the discrepancy between them, we can more accurately locate the behavior of each production system, thereby identifying the potential improvement for a given production system program.

Figure 11 displays all the discrepancies for the four programs. Note the discrepancy curves in Figure 11 and the improvement curves of Figures 9 and 10. Among the four discrepancy curves, the Brick curve has a high and regular behavior, which can in turn translate to a high im-

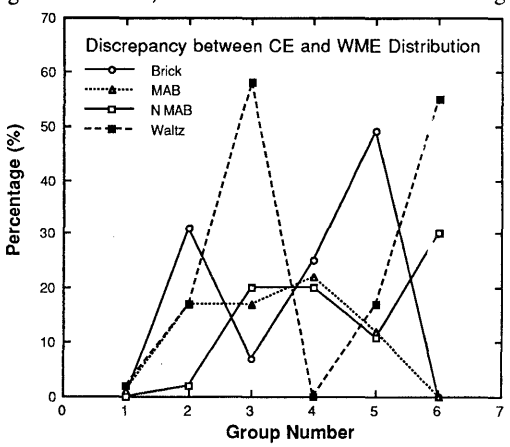


Figure 11: Discrepancy between CE and wme distribution.

provement. The curve for Brick in Figure 11 verifies this relation in which the improvement is high when the fluctuation is low.

However, the above statement on the relation between the discrepancy and the improvement would have to be further substantiated by more experimental results. Most problems are runtime dependent and a simple prediction rule would be problematic. Based on our observations, it could be concluded that if there is more discrepancy between the compile time distribution of CEs and the runtime distribution of wmes, the production system program would have more potential parallelism in match step.

A complete execution time of production cycles is illustrated in Figure 12 to help give an overall view of the two interpreters. Again, the x-axis is plotted in production cycle numbers but the y-axis at this time is plotted in the total production cycle time. The total matching time is dominantly high compared to the selection time or action time as Forgy has indicated [2]. In any case, the MRN-based production system interpreter outperformed the Rete-based OPS5.

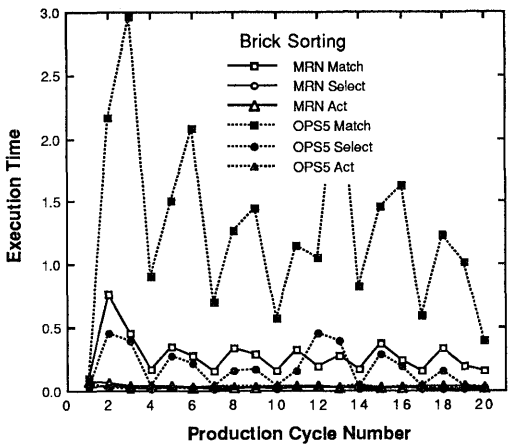


Figure 12: A complete execution time for two approaches.

6 Conclusions

The main purpose of this paper was to evaluate the performance of the multiple root node-based pattern matcher for production systems. A bottleneck was identified on the most efficient pattern Rete matcher. A solution to the bottleneck was proposed by introducing multiple root nodes to the Rete matcher. The MRN-based production system has been completely implemented in Lisp. To measure the relative algorithmic performance of the MRN-based matcher, benchmark production system programs were selected and executed on Sun 4/90 using both the MRN-based interpreter and Rete-based OPS5. Experimental results indicated that the MRN approach would give a multiplicative effect on the Rete-based production systems. The two criteria used in this study, one-input match time and number of comparison operations in a window of 20 production cycles, have shown that the MRN-based matcher can indeed give on the average a 6 fold improvement over the Lisp implementation of the Rete-based OPS5. Our experimental results suggest that production systems have more potential parallelism than what has been known. To further identify the complete source of parallelism in production systems, we have been implementing the MRN-based production system interpreter in SISAL, a pure functional language targeted to data-flow multiprocessors. The implementation is near completion and when complete we will be able to identify very fine-grain parallelism in production systems.

References

[1] Brownston, L., Farrell, R., Kant, E., and Martin, N., Programming Expert Systems in OPS5, Addison-Wesley Publishing Company, Reading, MA, 1985.

[2] Forgy, C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence* 19, September 1982, pp.17-37.

[3] Gaudiot, J-L., and Sohn, A., "Data-Driven Multiprocessor Implementation of the Rete Match Algorithm," in *Proc. of the Int'l Conf. on Parallel Processing*, St. Charles, IL, Aug. 1988, pp.256-260.

[4] Gaudiot, J-L., and Sohn, A., "Data-Driven Parallel Production Systems," *IEEE Transactions on Software Engineering*, March 1990, pp.281-293.

[5] Gupta, A., Parallelisms in Production Systems, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[6] Gupta, A., Forgy, C.L., Kalp, D., Newell, A., and Tambe, M., "Parallel OPS5 on the Encore Multimax," in *Proc. of the Int'l Conf. on Parallel Processing*, St. Charles, IL, August 1988, pp.271-280.

[7] Ishida, T., "Methods and Effectiveness of Parallel Rule Firing," in *Proc. of the IEEE Conference on AI Applications*, Santa Barbara, CA, March 1990.

[8] Kuo, S., and Moldovan, D. I., "Performance Comparison of Models for Multiple Rule Firing," in *Proc. of the 12th IJCAI*, Sydney, Australia, August 1991, pp.42-47.

[9] McDermott, J., and Forgy, C.L., "Production System Conflict Resolution Strategies," in Pattern Directed Inference Systems, D. Waterman and F. Hayes-Roth (Eds.), Academic Press, NY, NY, 1978.

[10] Miranker, D.P., Treat: A New and Efficient Match Algorithm for AI Production Systems, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[11] Sohn, A., and Gaudiot, J-L., "A Macro Actor/Token Implementation of Production Systems on a Data-flow Multiprocessor" in *Proc. of the 12th IJCAI*, Sydney, Australia, August 1991, pp.36-41.

[12] Sohn, A., "Parallel Processing of Production Systems on a Macro Data-flow Multiprocessor" *Technical Report*, EE-Systems Dept., University of Southern California, August 1991.

APPLICATIONS & SOCIAL IMPACTS

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Output in CLP($\mathcal{R}$)

JOXAN JAFFAR*
PETER J. STUCKEY†
MICHAEL J. MAHER*
ROLAND H.C. YAP*

Abstract

An important issue in Constraint Logic Programming (CLP) systems is how to output constraints in a usable form. Typically, only a small subset $\tilde{x}$ of the variables in constraints is of interest, and so an informal statement of the problem at hand is: given a conjunction $c(\tilde{x}, \tilde{y})$ of constraints, express $\exists \tilde{y}\ c(\tilde{x}, \tilde{y})$ in the simplest form. In this paper, we consider the constraints of the CLP($\mathcal{R}$) system and describe the essential features of its output module. In the main, we focus on the well-known problem of projection in *linear* arithmetic constraints. We start with a classical algorithm and augment it with a procedure for eliminating redundant constraints generated by the algorithm. The rest of the paper discusses the remaining kinds of constraints, equations over trees and nonlinear equations, and clarifies how they are output together with linear constraints.

1 Introduction

In its simplest description, the output of a constraint logic programming (CLP) [Jaffar and Lassez 1986] program is the collection of all constraints accumulated along a successful execution path. However such a collection is, in general, extremely complex because it is very large and contains many intermediate variables of no intrinsic interest to the user. Therefore, we can informally state that the problem at hand is: given a set $\tilde{x}$ of *target* variables and a conjunction $\mathcal{C}(\tilde{x}, \tilde{y})$ of constraints, express $\exists \tilde{y}\ \mathcal{C}(\tilde{x}, \tilde{y})$ in the most usable form. While we cannot define usability formally, it typically means both conciseness and readability. In this paper we consider the constraints of the CLP($\mathcal{R}$) system [Jaffar *et al.* 1990] and discuss the several issues and techniques that arose in implementing the output module for CLP($\mathcal{R}$).

*IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA.
†Dept. of Computer Science, Univ. of Melbourne, Parkville, Victoria 3052, Australia.

Consider some examples. Where $\{x, y\}$ are the target variables: (a) the constraints $x = f(z, z)$, $z = g(y, w)$ can be output as $x = f(g(y, \_1), g(y, \_1))$[1]; (b) the constraints $x = z + 1, y = 2 * z$ can be output as $x = 0.5 * y + 1$; (c) the constraints $x < z$, $z \leq y$, $z \leq y + 1$ can be output as $x < y$; (d) the constraints $x = \sin z * \sin z + \cos z * \cos z + y$ can be output as $x = 1 + y$.

We can classify the simplification of constraints in three directions:

(I) the elimination of auxiliary variables (as in (a), (b) and (c));
(II) the elimination of redundant constraints (as in (c)), and
(III) the replacement of expressions by simpler equivalent ones (as in (d)).

The problem (I) of expressing $\exists \tilde{y}\ \mathcal{C}(\tilde{x}, \tilde{y})$ as a formula involving only the variables $\tilde{x}$ is known variously as projection, variable elimination and quantifier elimination. Full variable elimination is not always possible in CLP($\mathcal{R}$), for example, in (a) above. However we note that it is theoretically possible to eliminate all auxiliary variables from purely *arithmetic* constraints [Collins 1982, Tarski 1951]. We will see later that additional requirements restrain us from always achieving this goal. Eliminating redundant constraints (II) is in general very difficult, often more so than the problem of determining constraint satisfiability. Discovering simpler equivalent expressions (III) is also difficult in general; in this paper, it affects us only in the nonlinear constraints.

A traditional approach to constraint simplification, is to use a notion of *canonical form* equipped with an efficient algorithm for its computation. Informally, such a form represents the information content of the original constraints in a minimal manner w.r.t. the target variables $\tilde{x}$. For example, in PROLOG (equations over trees), constraints can be represented by their mgu, and written in the form $\tilde{x} = t(\tilde{y})$ where $\tilde{y}$ are distinct from $\tilde{x}$ and

[1] Underscore notation is used to emphasize auxiliary variables.

t is a tuple of terms. For linear equations, a well known canonical form is called parametric form where equations are represented in the form $\tilde{x} = t(\tilde{y})$ where $\tilde{y}$ are distinct from $\tilde{x}$ and t is a tuple of linear expressions. If linear inequalities are also considered, there is still a natural notion of canonical form [Lassez and McAloon 1988]; however, it is not clear if there exists an efficient algorithm. Finally, if nonlinear equations (including functions like *sin*() and *abs*()) are also included, then it is not clear what a desired canonical form is, much less if there is an algorithm[2] at all.

In the context of CLP languages, the constraint simplification problem is compounded by other difficulties. One difficulty concerns backtracking: for efficiency the output module operates directly on the run-time structures representing the constraints, and consequently these operations need to be undone. Another difficulty is that constraints are represented in a form designed for testing satisfiability; this form is often unsuitable for computing output.

After a brief outline of the CLP($\mathcal{R}$) system, we focus on the classical problem of projection in *linear* arithmetic constraints. The core element here is a Fourier-based algorithm for eliminating non-target variables. The original Fourier algorithm [Fourier 1824] has the fundamental problem of generating too many redundant constraints, and the systematic removal of all such constraints is prohibitive. A major advance due to Černikov [Černikov 1963] made the Fourier algorithm plausibly practical by using an efficient, but partial, redundancy removal method. Combining the Černikov method with further redundancy removal is, unfortunately, unsound in general. The main technical result of this paper shows that augmenting the Černikov algorithm with strict redundancy removal is in fact sound.

The rest of the paper discusses the remaining kinds of constraints: equations over trees and nonlinear equations. Functor equations are straightforward. For nonlinear constraints, many possible simplifications are not performed because a nonlinear constraint solver is not employed. However, we do employ a general heuristic which is both efficient and effective. Finally, the various sub-algorithms are put together in a specific order, together with a substitution mechanism, to obtain the complete algorithm.

2 CLP($\mathcal{R}$)

Real constants and real variables are both *arithmetic terms*. If t, t_1 and t_2 are arithmetic terms, then so are $(t_1 + t_2)$, $(t_1 - t_2)$, $(t_1 * t_2)$, (t_1/t_2), $abs(t)$, $max(t_1, t_2)$, $min(t_1, t_2)$, $sin(t)$, $cos(t)$ and $pow(t_1, t_2)$. Uninterpreted constants and functors are like those in PROLOG. Uninterpreted constants and arithmetic terms are *terms*, and so is any variable. If f is an n-ary uninterpreted functor, $n \geq 0$, and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term. If t_1 and t_2 are arithmetic terms, then $t_1 = t_2$, $t_1 < t_2$ and $t_1 \leq t_2$ are all arithmetic *constraints*. If at least one of t_1 and t_2 is not an arithmetic term, then only the expression $t_1 = t_2$ is a constraint.

An *atom* is of the form $p(t_1, t_2, \ldots, t_n)$ where p is a predicate symbol distinct from $=$, $<$, and $\leq$, and $t_1, ..., t_n$ are terms. A CLP($\mathcal{R}$) program is defined to be a finite collection of *rules* of the form $A_0 :- \alpha_1, \alpha_2, \ldots, \alpha_k$ where each α_i, $0 \leq i \leq k$, is either a constraint or an atom. A CLP($\mathcal{R}$) *goal* consists of a set of *current constraints*, and a *goal body*. These constraints must be *linear consistent*, that is, it can be partitioned into a functor component, a linear component and a nonlinear component such that the conjunction of the first two components is satisfiable; the goal body is the same as a rule body. In an initial goal, the set of current constraints is empty.

Let the goal G be $\Psi ?- \alpha_1, \ldots, \alpha_k$ where Ψ denotes the current constraints. A *derivation step* from G is defined over two cases: if α_1 is a constraint, then G derives $\Psi \cup \alpha_1 ?- \alpha_2, \ldots, \alpha_k$ providing $\Psi \cup \alpha_1$ is linear consistent; if α_1 is an atom A and if there is a rule $A' :- \beta_1, \ldots, \beta_m$, then G derives $\Psi \cup A = A' ?- \beta_1, \ldots, \beta_m, \alpha_2, \ldots, \alpha_k$ providing $\Psi \cup A = A'$ is linear consistent. When no derivation step is possible, execution "backtracks" to a point where an alternate choice of matching rule is available.

A *derivation sequence* is a possibly infinite sequence of goals such that there is a derivation step to each goal from the preceding goal. A derivation sequence is *conditionally successful* if it is finite and the body in the last goal is empty. If, however, the current constraints in this last goal (sometimes called *answer constraints*) has an empty nonlinear component, we say that the derivation sequence is *successful*. Producing appropriate output from these constraints, given as target variables the variables appearing in the original goal, is the subject of this paper.

In the CLP($\mathcal{R}$) system the user can also call, anywhere in the computation, the predefined predicate $dump([x_1, \ldots, x_N])$; this invokes the output module on the current constraint set with target variables $x_1, \ldots, x_N$. For more details of the CLP($\mathcal{R}$) system and its implementation we refer the reader to [Jaffar *et al.* 1990].

3 Linear Constraints

Let $\mathcal{C}$ denote the collection of linear constraints at hand, let $x_1, x_2, \ldots, x_N$ (abbreviated $\tilde{x}$) denote the tar-

[2]The satisfiability problem here is in fact undecidable.

```
for (i = 1; i ≤ N; i = i + 1) {
    if (x_i is a parameter) continue;
    let E denote the equation x_i = r.h.s.(x_i) at hand;
    if (r.h.s.(x_i) contains a variable z of lower priority than x_i) {
        choose the z of lowest priority;
        rewrite the equation E into the form z = t;
        if (z is a target variable) mark the equation E as final;
        substitute t for z in the other linear equations and inequalities;
    } else mark the equation E as final;
}
return all final equations;
```

Figure 1: *Linear equations*

get variables within $\mathcal{C}$, and let $y_1, y_2, \ldots, y_M$ (abbreviated $\tilde{y}$) denote the remaining *auxiliary* variables. The linear solver in CLP($\mathcal{R}$) is partitioned into the equation solver and the inequality solver for efficiency reasons.

In this section, we describe an algorithm which outputs $\exists \tilde{y}\ \mathcal{C}(\tilde{x}, \tilde{y})$, where $\mathcal{C}$ is linear, in terms of target variables only, treating first the equations then the inequalities. The algorithm may produce an output not containing a particular target variable x which appears in $\mathcal{C}$ – for example, when eliminating y from $\exists y\ x = y + 2$ – or may produce an untyped equation – for example, producing $x = z$ from $\exists y\ x = y + 1 \wedge z = y + 1$. For such x, we add to the output the special constraint $real(x)$ restricting x to real number values.

3.1 Linear Equations

Equations are maintained in *parametric form*, that is, in the form $\tilde{u} = t(\tilde{v})$ where $\tilde{u}$, called the *object* variables are distinct from $\tilde{v}$, called the *parameters*. For each object variable z we write $r.h.s.(z)$ to denote the linear expression (of parameters) that z is equated to. Inequalities are always written in terms of parameters alone (in addition to other restrictions which do not concern us here).

The algorithm, essentially a form of Gaussian elimination, is described in Figure 1. It assumes there is a priority π among the variables, $\pi(x_1) > \cdots > \pi(x_N) > \pi(y_1) > \cdots > \pi(y_M)$, expressing the relative importance of each variable in the output[3]. The algorithm ensures that lower priority variables are represented in terms of higher priority variables. (We will see later how π is used in the context of functor and nonlinear equations to minimize the number of variables occurring in the output.)

A crucial point for efficiency is that the main loop in Figure 1 iterates N times, and $N \ll M$ in general, that is, the number of target variables is often far smaller than the total number of variables in the system.

Note that the order of variables in the predefined predicate $dump([x_1, \ldots, x_N])$ determines the priority relation over these variables. Hence the user can influence the output representation of the constraints.

3.2 Linear Inequalities

The constraint solver stores the linear inequalities in a Simplex tableau. (See [Jaffar *et al.* 1990] for details.) Each linear inequality is expressed internally as an equality by introducing a *slack* variable, one whose value is restricted to be either nonnegative or positive. Our first job, therefore, is the elimination of such slack variables. This is achieved by pivoting the inequality tableau to make all the slack variables basic so that each appears in exactly one equation. Hence each row can be viewed as $s = exp$ where $s \geq 0$ or $s > 0$, and this equation can now easily be rewritten into the appropriate inequality $exp \geq 0$ or $exp > 0$.

The remainder of this section deals with the problem of eliminating non-target variables which occur in these inequalities. We use a method based on Fourier's algorithm [Fourier 1824]. It is well-known that the direct application of this algorithm is impractical because it generates many redundant constraints. Attempting to eliminate all redundancy at every step is also impractical [Lassez *et al.* 1989]. Adaptations of Fourier's algorithm due to Černikov [Černikov 1963] substantially improve the performance. We show how to incorporate other redundancy elimination methods with those of Černikov to obtain a more practical algorithm for eliminating variables from linear inequalities.

In some circumstances, especially when constraints

[3]The priority among the y_i's is arbitrary.

when written as a matrix is dense, algorithms not based on Fourier such as [Huynh *et al.* 1990] can be more efficient; however, typical CLP($\mathcal{R}$) programs produce sparse matrices. In general, the size of projection can grow exponentially in the number of variables eliminated, even when all redundancy is eliminated. Fourier-based methods have the advantage over other methods that we can stop eliminating variables at any time, thus computing a partial projection.

3.2.1 Fourier-based Methods

We begin with some necessary definitions. A *labeled constraint* is a linear inequality labeled by a set of constraint names. We say that c is *label-subsumed* by c' if $label(c') \subseteq label(c)$. To simplify the explanation, we will not consider strict inequalities. We assume all constraints are written in the form $\sum_{i=1}^{m} \alpha_i x_i \leq \beta$.

We shall be using some algebraic manipulation of constraints. Let c_j be the constraint $\sum_{i=1}^{m} \alpha_{j,i} x_i \leq \beta_j$, for $j = 1, \ldots, n$. Then $\gamma * c_j$ (or γc_j) denotes the constraint $\sum_{i=1}^{m} \gamma\alpha_{j,i} x_i \leq \gamma\beta_j$ where γ is a real number, and $c_j + c_k$ denotes the constraint $\sum_{i=1}^{m} (\alpha_{j,i} + \alpha_{k,i}) x_i \leq \beta_j + \beta_k$. Similarly, $\sum_{j=1}^{n} c_j$ denotes an iterated sum of constraints. We consider constraints c and c' equal, $c = c'$, if $c \equiv \gamma * c'$ for some $\gamma > 0$.

Let $\mathcal{C}$ be a set of labeled constraints. Given a variable x_i, we divide $\mathcal{C}$ into three subsets: $\mathcal{C}^+_{x_i}$, those constraints in which x_i has a positive coefficient (i.e. c_j such that $\alpha_{j,i} > 0$); $\mathcal{C}^-_{x_i}$, those constraints in which x_i has a negative coefficient; and $\mathcal{C}^0_{x_i}$, those constraints in which the coefficient of x_i is zero. We omit the subscript when the given variable is clear from the context.

Let $c_k \in \mathcal{C}^+$ and $c_l \in \mathcal{C}^-$ and let $d = 1/\alpha_{k,i} * c_k + -1/\alpha_{l,i} * c_l$. Then, by construction, x_i does not occur in the constraint d. If $\mathcal{S}_k$ ($\mathcal{S}_l$) is the label of c_k (c_l) then d has label $\mathcal{S}_k \cup \mathcal{S}_l$. Let $\mathcal{D}$ be the collection of all such d. Then $\mathcal{D} \cup \mathcal{C}^0$ is the result of a *Fourier step eliminating* x_i. We write $fourier_i(\mathcal{C}) = \mathcal{D} \cup \mathcal{C}^0$. When both $\mathcal{C}^+$ and $\mathcal{C}^-$ are non-empty then $\mathcal{D}$ is non-empty, and the step is called an *active* variable elimination. After eliminating x_i the total number of constraints in $\mathcal{C}$ increases (possibly decreasing) by $measure(x_i, \mathcal{C}) = |\mathcal{C}^+| \times |\mathcal{C}^-| - |\mathcal{C}^+| - |\mathcal{C}^-|$.

Let $\mathcal{A}$ be a set of constraints where each constraint is labeled by its own name. Define $\mathcal{F}_0 = \mathcal{A}$ and $\mathcal{F}_{i+1} = fourier_{i+1}(\mathcal{F}_i)$. Then $\{\mathcal{F}_i\}_{i=0,1,\ldots}$ is the sequence of constraint sets obtained by Fourier's method, eliminating, in order, $y_1, y_2, \ldots$. We write $\mathbf{F}_i$ for $\{\mathcal{F}_j\}_{j=0,1,\ldots,i}$. It is straightforward (see [Lassez and Maher 1988], for example) that, if $m < n$, $\mathcal{F}_n \leftrightarrow \exists y_{m+1}, y_{m+2}, \ldots, y_n\ \mathcal{F}_m$. In particular, $\mathcal{F}_n \leftrightarrow \exists y_1, y_2, \ldots, y_n\ \mathcal{A}$. Thus Fourier's algorithm computes projections.

However Fourier's algorithm generates many redundant constraints and has doubly-exponential worst-case behavior. Černikov [Černikov 1963] (and later Kohler [Kohler 1967]; see also [Duffin 1974]) proposed modifications which allow some redundant constraints to be eliminated during a Fourier step, and address this problem. The first method eliminates all constraints generated at the n'th active step which have a label of cardinality $n + 2$ or greater, for every n. A second method retains, at each step, a set S of constraints such that every constraint generated at this step is label-subsumed by a constraint in S [4]. The first method eliminates a subset of the constraints eliminated by the second. These methods are *correct* in the following sense: If $\{\mathcal{C}_i\}_{i=0,1,\ldots}$ is the sequence generated by such a method, then $\mathcal{C}_i \leftrightarrow \exists y_1 \ldots y_i\ \mathcal{A}$, for every i.

Although it appears that the Černikov modifications to Fourier's algorithm could be augmented by deleting additional redundant constraints after each step, this is incorrect in general [Huynh *et al.* 1990]. The following example highlights this point by showing that the first Černikov algorithm, augmented by the simplest kind of redundancy removal, removal of *duplicate* constraints, is unsound.

3.2.2 An Example

Let $\mathcal{A}$ denote the following labeled constraints. Labels appear to the left of the constraints. It can be verified that $\mathcal{A}$ contains no redundancy.

$$
\begin{array}{lrrrrrrrrl}
\{1\} & & w & + & x & + & y & + & z & \leq 1 \\
\{2\} & & w & - & x & + & y & + & z & \leq 1 \\
\{3\} & & -w & + & x & + & y & + & z & \leq 1 \\
\{4\} & & -w & - & x & + & y & + & z & \leq 1 \\
\{5\} & v & & & & - & y & & & \leq 0 \\
\{6\} & -v & & & & & & & & \leq 0
\end{array}
$$

Upon eliminating v (by adding the last two constraints), we obtain in the first (Fourier) step:

$$
\begin{array}{lrrrrrrrl}
\{1\} & w & + & x & + & y & + & z & \leq 1 \\
\{2\} & w & - & x & + & y & + & z & \leq 1 \\
\{3\} & -w & + & x & + & y & + & z & \leq 1 \\
\{4\} & -w & - & x & + & y & + & z & \leq 1 \\
\{5,6\} & & & & & -y & & & \leq 0
\end{array}
$$

Next we eliminate w obtaining:

$$
\begin{array}{lrrrrrrl}
\{1,3\} & x & + & y & + & z & \leq 1 \\
\{1,4\} & & & y & + & z & \leq 1 \\
\{2,3\} & & & y & + & z & \leq 1 \\
\{2,4\} & -x & + & y & + & z & \leq 1 \\
\{5,6\} & & & -y & & & \leq 0
\end{array}
$$

[4] In the English translation of [Černikov 1963], this is misstated.

Observe that Černikov's criterion does not allow us to delete any constraints. Since the second and third constraints are duplicates, we could delete one. However, we choose not to in this step. Next x is eliminated to obtain:

$$\begin{array}{ll} \{1,2,3,4\} & y + z \leq 1 \\ \{1,4\} & y + z \leq 1 \\ \{2,3\} & y + z \leq 1 \\ \{5,6\} & -y \leq 0 \end{array}$$

The first three constraints are identical, and now we choose to delete the second and third, obtaining:

$$\begin{array}{ll} \{1,2,3,4\} & y + z \leq 1 \\ \{5,6\} & -y \leq 0 \end{array}$$

In the final Fourier step, we eliminate y to obtain:

$$\{1,2,3,4,5,6\} \quad z \leq 1$$

Černikov's criterion allows us to delete this constraint, and so we finally obtain an empty set of constraints. This outcome is incorrect since it implies $\exists v, w, x, y\ \mathcal{A}$ is true for all values of z, and it is straightforward to verify that, in fact, $\exists v, w, x, y\ \mathcal{A} \leftrightarrow (z \leq 1)$. Observe that we could have achieved the same incorrect outcome if, after eliminating w, one of the duplicate constraints was deleted.

3.2.3 Combining Fourier-based Methods with Strict Redundancy Elimination

Given a set $\mathcal{C}$ of constraints, $c \in \mathcal{C}$ is *redundant* in C if $\mathcal{C} \leftrightarrow \mathcal{C} - \{c\}$. A subset $\mathcal{R}$ of $\mathcal{C}$ is redundant if $\mathcal{C} \leftrightarrow \mathcal{C} - \mathcal{R}$. We define $\mathcal{C} \gg c$ iff, for some constraint c', $\mathcal{C} \rightarrow c'$ and $c' \rightarrow c$ but $c \not\rightarrow c'$. Equivalently, (if we are dealing with only non-strict inequalities) $\mathcal{C} \gg c$ means $\mathcal{C} \rightarrow c'$ where $c = c' + (0 \leq \epsilon)$ for some constraint c' and some $\epsilon > 0$. (Recall that $c' + (0 \leq \epsilon)$ denotes the sum of the constraint c' and the constraint $(0 \leq \epsilon)$.) If also $c \in \mathcal{C}$ then c is said to be *strictly redundant* in $\mathcal{C}$. Geometrically, a strictly redundant constraint c determines a hyperplane which does not intersect the volume defined by $\mathcal{C}$. We write $\mathcal{C} \gg \mathcal{C}'$ if $\mathcal{C} \gg c$ for every $c \in \mathcal{C}'$. A constraint $c \in \mathcal{C}$ is said to be *quasi-syntactic redundant* [Lassez *et al.* 1989] if, for some $c' \in \mathcal{C}$ and some $\epsilon > 0$, $c = c' + (0 \leq \epsilon)$. Clearly quasi-syntactic redundancy is one kind of strict redundancy.

We capture Černikov's modifications of Fourier's algorithm and others in the following definition. Let r be a constraint deletion procedure which, at step i, determines a redundant subset of $\mathcal{F}_i$ as a function of the sequence $\mathbf{F}_i$. Let a *Fourier-based* algorithm be one which generates a sequence of constraint sets $\{\mathcal{C}_i\}_{i=0,1,\ldots}$ where $\mathcal{C}_0 = \mathcal{A}$ and $\mathcal{C}_{i+1} = fourier_{i+1}(\mathcal{C}_i) - r(\mathbf{F}_{i+1})$. It is important to note that, in general, it is not necessary for a Fourier-based method to compute the sequence $\{\mathcal{F}_j\}_{j=0,1,\ldots}$. Indeed, these methods are valuable to the extent that they *do not* compute $\mathbf{F}_i$. All that is required is that $\mathcal{C}_i$ can be viewed as being computed using a function of this sequence.

Let r be the function associated with a Fourier-based method, and let s map every constraint set to a subset obtained by deleting some strictly redundant constraints. The sequence of constraint sets $\{\mathcal{K}_i\}_{i=0,1,\ldots}$ where $\mathcal{K}_0 = \mathcal{A}$ and $\mathcal{K}_{i+1} = s(fourier_{i+1}(\mathcal{K}_i) - r(\mathbf{F}_{i+1}))$ is the result of augmenting the Fourier-based method with the deletion of (some) strictly redundant constraints.

We let $\mathcal{D}_i$ denote the set of constraints deleted by s at step i, that is, $\mathcal{D}_i = (fourier_i(\mathcal{K}_{i-1}) - r(\mathbf{F}_i)) - s(fourier_i(\mathcal{K}_{i-1}) - r(\mathbf{F}_i))$. A *removed constraint* in $\mathcal{C}_i$ is defined to be a constraint in $\mathcal{C}_i$ which uses some constraint in $\mathcal{D}_j, j \leq i$ during generation. That is, for some $j \leq i$, if we view generation of $\mathcal{C}_i$ as starting from $\mathcal{C}_j$ (instead of $\mathcal{C}_0$) then c is generated using constraints from $\mathcal{D}_j$. $\mathcal{R}_i$ denotes the set of all removed constraints in $\mathcal{C}_i$. Clearly $\mathcal{F}_i \supseteq \mathcal{C}_i \supseteq \mathcal{K}_i$, $\mathcal{K}_i = \mathcal{C}_i - \mathcal{R}_i$, and $\mathcal{D}_i \subseteq \mathcal{R}_i$.

Numbers denoted by λ's, μ's, ν's and ϵ's are non-negative throughout. Thus $(0 \leq \epsilon)$ is a tautologous constraint. The notation $var(c)$ denotes the set of variables with non-zero coefficient in constraint c.

The following theorem from the folklore underlies all the work below.

Theorem 1 *Let $\mathcal{C} = \{c_0, c_1, \ldots, c_k\}$ be a consistent set of constraints and let c be a constraint. $\mathcal{C} \rightarrow c$ iff $c = \sum_{i=0}^{k} \lambda_i c_i + (0 \leq \epsilon)$ where the λ's and ϵ are non-negative.* □

The next lemma shows that all strictly redundant constraints can be deleted simultaneously from a consistent set of constraints. Consequently it is meaningful to speak of a strictly redundant subset of $\mathcal{C}$. It also shows that a set of redundant constraints can be deleted simultaneously with a set of strictly redundant constraints. The corresponding results for the class of all redundant constraints do not hold.

Lemma 2 *Let $\mathcal{C}$ be a consistent set of constraints.*

1. *If $\mathcal{D} \subseteq \mathcal{C}$ and each $c \in \mathcal{D}$ is strictly redundant in $\mathcal{C}$ then $\mathcal{C} \leftrightarrow \mathcal{C} - \mathcal{D}$.*
2. *If $\mathcal{S}$ is strictly redundant in $\mathcal{C}$ and $\mathcal{R}$ is redundant in $\mathcal{C}$ then $\mathcal{S} \cup \mathcal{R}$ is redundant in $\mathcal{C}$.* □

We now prove that elimination of strict redundancy does not affect correctness of Fourier-based methods.

```
C = initial set of inequalities (after linear substitutions);
label(c) = {c} for each c ∈ C;
n = 0;
while (there exists an auxiliary variable x in C) {
    choose a variable x with minimal measure(x, C);
    D = C_x^0;
    if (|C_x^+| > 0 and |C_x^-| > 0 ) {
        n = n + 1; /* count active eliminations */
        for (each pair c_k ∈ C_x^+, c_l ∈ C_x^-)
            if (|label(c_k) ∪ label(c_l)| ≥ n + 2) continue; /* first Černikov method */
            d is the constraint obtained from c_k and c_l eliminating x;
            label(d) = label(c_k) ∪ label(c_l);
            if (d is not quasi-syntactic redundant wrt D) {
                E = quasi-syntactic redundant constraints in D wrt d;
                D = D ∪ {d} − E;
/**                             second Černikov method
                if (d is label-subsumed in D)
                    D = D − {d};
                else {
                    F = constraints in D label-subsumed by d;
                    D = D − F;
                }
***/
            }
        }
        C = D;
    }
    return C;
```

Figure 2: *Linear inequalities*

Theorem 3 *Suppose $\mathcal{A}$ is consistent and $\{\mathcal{C}_i\}$ is correct. Then $\{\mathcal{K}_i\}$ is correct.*

Proof: Note that, since $\mathcal{A}$ is consistent, $\mathcal{F}_n$ is consistent for every n, and consequently so are $\mathcal{C}_n$ and $\mathcal{K}_n$. Suppose $c \in \mathcal{R}_n$ and c depends on constraints in $\mathcal{D}_m$, $m \leq n$; say $c = \sum_i \lambda_i c_i + \sum_j \mu_j d_j$ where $d_j \in \mathcal{D}_m$ for each j and $c_i \in \mathcal{C}_m - \mathcal{D}_m$ for each i, and for some j, $\mu_j > 0$. Now for each j, since $\mathcal{C}_m \twoheadrightarrow \mathcal{D}_m$ (Lemma 2), $d_j = \sum_i \nu_{ji} c_i + (0 \leq \epsilon_j)$ where $\epsilon_j > 0$, and $c_i \in \mathcal{C}_m - \mathcal{D}_m$ for each i. Hence $c = \sum_i (\lambda_i + \sum_j \mu_j \nu_{ji}) c_i + (0 \leq \epsilon')$ where $\epsilon' = \sum_i \mu_j \epsilon_j$ and $\epsilon' > 0$. Let $c' = \sum_i (\lambda_i + \sum_j \mu_j \nu_{ji}) c_i$ so that $c = c' + (0 \leq \epsilon')$.

Now $\mathcal{F}_m \rightarrow c'$ since every $c_i \in \mathcal{F}_m$. Furthermore $\mathcal{F}_n \rightarrow c'$ since $var(c') = var(c) \subset \{y_{n+1}, y_{n+2}, \ldots\}$ (since $c \in \mathcal{R}_n$). Since $\{\mathcal{C}_i\}$ is correct, $\mathcal{C}_n \rightarrow c'$, that is, $\mathcal{C}_n \twoheadrightarrow c$.

By applying this argument for every c depending on $\mathcal{D}_m$ and every $m \leq n$, $\mathcal{C}_n \twoheadrightarrow c$ for every $c \in \mathcal{R}_n$. By Lemma 2, $\mathcal{C}_n \leftrightarrow \mathcal{C}_n - \mathcal{R}_n$. But $\mathcal{C}_n - \mathcal{R}_n = \mathcal{K}_n$. Hence $\mathcal{K}_n \leftrightarrow \mathcal{C}_n$ and, since $\{\mathcal{C}_i\}$ is correct, $\mathcal{K}_n \leftrightarrow \mathcal{F}_n$. □

This result extends to sets of constraints containing both strict and non-strict inequalities. Fourier's algorithm and Černikov's modification extend straightforwardly. The definition of $\twoheadrightarrow$ stands, but it is no longer equivalent to $\mathcal{C} \rightarrow c'$ and $c = c' + (0 \leq \epsilon)$, for some $\epsilon > 0$.

Before discussing our algorithm, we briefly outline the costs of various redundancy elimination procedures. Let $\mathcal{C}$ be a set of m inequalities involving n variables, obtained in a Fourier-based method from m_0 original inequalities. Full redundancy elimination using the simplex algorithm has exponential worst-case complexity, although in the average case it is $O(m^3 n)$. Strict redundancy elimination has essentially the same cost as full redundancy elimination. Quasi-syntactic redundancy elimination on the constraints $\mathcal{C}$ has worst-case complexity $O(m^2 n)$. The cost of eliminating redundancy in $\mathcal{C}$ using the first Černikov method has worst-case complexity $O(m m_0)$, and it has the important advantage that a constraint can be deleted *before* the (relatively expen-

sive) process of explicitly constructing it. Application of the second Černikov method has worst-case complexity $O(m^2 m_0)$.

In [Černikov 1963] it is recommended that the first, and then the second Černikov elimination method be applied at each step. The variation in which the second method is applied only intermittently is suggested in [Kohler 1967]. If we want to incorporate the elimination of strict redundancy, the above complexity analysis suggests that quasi-syntactic redundancy elimination may be most cost-effective. The analysis also suggests that this elimination should be performed between the first and second Černikov methods.

Our tests tended to support this reasoning. Using the first Černikov method followed by quasi-syntactic redundancy elimination produced significant improvement over the first method alone. However further processing in accord with the second Černikov method only marginally reduced the number of constraints eliminated and led to an overall increase in computation time. Full redundancy elimination after each Fourier step, which is incompatible with the Černikov methods, slows computation by an order of magnitude. Full strict redundancy elimination added to the Černikov method is also unprofitable.

The algorithm is shown in Figure 2. It uses a heuristic (from [Duffin 1974]) attempting to minimize the number of new constraints generated. There remains the matter of verifying the correctness of the algorithm. It is easy to see that a step i is active in $\{\mathcal{F}_i\}$ iff it is active in $\{\mathcal{K}_i\}$ iff it is active in $\{\mathcal{C}_i\}$. Thus the first Černikov method is Fourier-based, and the corresponding part of the algorithm implements this method, and so is correct. The second part of the algorithm deletes some remaining quasi-syntactic redundancies and, by the previous theorem, is correct. If the third part, which is commented out in Figure 2, is included in the algorithm then theorem 3 does not apply directly. However it is not difficult to show that this algorithm is equivalent to eliminating some of the constraints eliminable by applying the second Černikov method *en bloc* and then eliminating some strictly redundant (not necessarily quasi-syntactic redundant) constraints. Thus the theorem applies and the algorithm is correct.

4 Constraints over Trees

The constraints at hand are equations involving uninterpreted functors, the functor equations. As in PROLOG systems a straightforward way of printing these equations is to print an equation between each target variable and its value.

Consider equivalence classes of variables obtained as the reflexive, symmetric and transitive closure of the relation: $\{(x, y) : x \text{ is bound to } y\}$; write $rep(x)$ to denote the variable of highest priority equivalent to x. Now define the printable value of a variable as:

$$value(f(t_1, \cdots, t_n)) = f(value(t_1), \cdots, value(t_n));$$

$$value(x) = \begin{cases} value(t), & \text{if } x \text{ is bound to a term } t; \\ rep(x), & \text{if } x \text{ is unbound} \end{cases}$$

The output is a set of equations of the form $x = value(x)$ for each target variable x, excepting those variables x for which $value(x)$ is x itself. We remark that most PROLOG systems do not use equivalence classes as above, and thus for example, the binding structure $x \mapsto \_1, y \mapsto \_1$ is generally not printed as $x = y$.

One well-known drawback of the above output method is that the output can be exponentially larger than the original terms involved. For example, the output of $x_1 = f(x_2, x_2), x_2 = f(x_3, x_3), \ldots x_{n-1} = f(x_n, x_n), x_n = a$, where $x_1, \ldots, x_n$ are target variables, is such that the binding of x_1 is a term of size $O(2^n)$. This exponential blowup can be avoided by other methods [Paterson and Wegman 1978], but in practice it occurs rarely. Hence the binding method is adopted in the CLP($\mathcal{R}$) system.

The output of functor equations in the context of other (arithmetic) constraints raises another issue. Recall that is not always possible to eliminate non-target variables appearing in functor equations (e.g. eliminating z in $x = f(z)$). Consequently, arithmetic constraints which affect these unavoidable non-target variables must also be output. We resolve this issue by augmenting the problem description sent to the linear constraint output module (c.f. Section 3) as follows: the target variables now consist of the original target variables and the unavoidable non-target variables, with the latter having priority intermediate between the original target variables and remaining variables.

These secondary target variables are given lower priority than the original target variables in order to minimize their occurrence in the output. The lower priority ensures that such variables appear on the left hand side of arithmetic equations as much as possible. We can then substitute the right hand side of the equation for the variable and omit the equation, thus eliminating the variable. For example, if x and y are the target variables in $x = f(z), y = z + 2$ the output is $x = f(y - 2)$. We discuss this further in section 6.

5 Nonlinear Constraints

In general all nonlinear constraints need to be printed, regardless of the target variables, because omitting them may result in an output which is satisfiable when the original set of constraints is not. For example, given the constraints $x < 0, y * y = -2$ and target variable x, we cannot simply output $x < 0$. This problem arises since we have no guarantee that the nonlinear constraints are satisfiable. When the only nonlinear constraints are caused by multiplication the auxiliary variables in the nonlinear constraints can, in theory, be eliminated. However this approach is not practical with current algorithms [Collins 1982] and not possible once trigonometric functions are introduced. Thus, as with functor equations, the nonlinear equations contribute additional target variables. These are simply all the variables which remain in the nonlinear constraints, and we give them priority lower than the target variables but higher than the variables added from functor equations.

However, there is one observation which can significantly reduce the number of nonlinear equations printed and the number of additional target variables: Suppose a non-target variable y occurs exactly once in the constraints, say in the constraint c, and $p(\tilde{x})$ implies $\exists y\ c(\tilde{x}, y) \leftrightarrow c'(\tilde{x})$, for some constraint c' and some condition p, then c can be replaced by c', provided the remaining constraints imply that $p(\tilde{x})$ holds. Some specific applications of this observation follow.

If y occurs in the form $y = \mathbf{f}(\tilde{x})$ then this constraint can be eliminated provided $\mathbf{f}$ is a total function on the real numbers (this excludes functions such as exponentiation and division[5]). If y occurs as $y = x^z$ then we can delete the constraint, provided we know that $x > 0$ or z is an integer other than 0. Similarly, we can delete $x = z^y$ provided that $x > 0$, $z > 0$ and $z \neq 1$, and delete $x = y^z$ provided $x > 0$ and $z \neq 0$. A constraint $x = |y|$ can be replaced by $x \geq 0$; $x = \sin y$ (and $x = \cos y$) can be replaced by $-1 \leq x \leq 1$; $x = min(y, z)$ can be replaced by $x \leq z$ (and similarly for max). A constraint $x = y * z$ (equivalently $y = x/z$) can be eliminated, provided it is known that $z \neq 0$. In this latter case, which can be expected to occur more often than most of the other delayed constraints, we can use linear programming techniques on the linear constraints to test whether z is constrained to be non-zero. Specifically, we add the constraint $z = 0$ to the linear constraint solver and if the solver finds that the resulting set of constraints is inconsistent then we delete $x = y * z$. We undo the effects of the additional constraint using the same mechanism as used for backtracking during execution of a goal.

[5]Strictly speaking, division is not a function, since $y = x/z$ is defined to be equivalent to $x = y * z$ and so $0/0$ can take any value.

There is a significant complication due to the linear constraints which are generated as a result of simplifying nonlinear constraints. As each such linear constraint is generated, it is passed to the linear constraint solver so that a consistency check can be performed[6]. If the resulting constraint system is not consistent then the simplifications are undone and the system backtracks to the nearest choice-point as it normally does after executing a failure.

6 Summary of the Output Module

We now present the output algorithm in its entirety, a collation of the various sub-algorithms described above corresponding to the different kinds of constraints. Note that the order in which the sub-algorithms are invoked is important; essentially, the processing of functor and nonlinear equations must be done first in order to determine the set of secondary target variables. Then the linear constraints are processed in such a way as to maximize the number of secondary target variables that can be eliminated. Step V below, not previously described, performs this elimination. It suffers the same drawback as processing functor equations - potentially the size of output is exponential in the size of the original equations.

Step I

Process the functor equations, in order to obtain the secondary target variables. These are essentially the non-target variables appearing in the bindings of the primary target variables. Obtain a (possibly empty) collection of functor equations.

Step II

Simplify the nonlinear equations, and expand the set of secondary target variables to include all the variables in the simplified collection. Obtain a collection of nonlinear equations. This step might also produce additional linear equations.

Step III

Process the linear equations (Figure 1) with respect to the primary and secondary target variables, using some priority such that the primary variables are higher priority than the secondary variables and the auxiliary variables are of the lowest priority. Obtain a collection of final linear equations involving only target variables.

Step IV

Process the linear inequalities (Figure 2), and note that these may have been modified as a result of

[6]Thus the output module implements a more powerful constraint solver than that used during run-time.

Step III above, using the primary and secondary target variables. Obtain a collection of linear inequalities involving only target variables.

Step V

For each secondary target variable y appearing in a linear equation of the form $y = t$, substitute t for y everywhere, and remove the equation. For each secondary target variable y appearing in a nonlinear equation of the form $y = t$, where y appears elsewhere but not in t, substitute t for y everywhere, and remove the equation.

Step VI

Output all the remaining constraints.

7 Conclusion

The output module of CLP($\mathcal{R}$) has been described. While a large part of the problem coincides with the classical problem of projection in linear constraints, dealing with functor and nonlinear equations, and working in the context of a CLP runtime structure, significantly increase the problem difficulty.

The core element of our algorithm deals with projecting linear constraints; it extends the Fourier/Černikov algorithm with strict redundancy removal. The rest of the paper deals with functor and nonlinear equations and how they are output together with the linear constraints. What is finally obtained is an output module for CLP($\mathcal{R}$) which has proved to be both practical and effective.

We finally remark that the introduction of meta-level facilities [Heintze *et al.* 1989] in a future version of CLP($\mathcal{R}$) significantly complicates the output problem, since the constraint domain is expanded to include representations/codings of constraints.

References

[Černikov 1963] S.N. Černikov. Contraction of Finite Systems of Linear Inequalities (In Russian). *Doklady Akademiia Nauk SSSR*, Vol. 152, No. 5 (1963), pp. 1075 – 1078. (English translation in *Soviet Mathematics Doklady*, Vol. 4, No. 5 (1963), pp. 1520–1524.)

[Collins 1982] G.E. Collins. Quantifier Elimination for Real Closed Fields: a Guide to the Literature. In *Computer Algebra: Symbolic and Algebraic Computation*, Computing Supplement #4, B. Buchberger, R. Loos and G.E. Collins (Eds), Springer-Verlag, 1982, pp. 79–81.

[Duffin 1974] R.J. Duffin. On Fourier's Analysis of Linear Inequality Systems. *Mathematical Programming Study*, Vol. 1 (1974), pp. 71–95.

[Fourier 1824] J-B.J. Fourier. Reported in: Analyse des travaux de l'Acadamie Royale des Sciences, pendant l'annee 1824, Partie mathematique, *Histoire de l'Academie Royale des Sciences de l'Institut de France*, Vol. 7 (1827), pp. xlvii-lv. (Partial English translation in: D.A. Kohler. Translation of a Report by Fourier on his work on Linear Inequalities. *Opsearch*, Vol. 10 (1973), pp. 38-42.)

[Heintze *et al.* 1989] N.C. Heintze, S. Michaylov, P.J. Stuckey and R. Yap. Meta-programming in CLP($\mathcal{R}$). In *Proc. North American Conf. on Logic Programming*, Cleveland, 1989. pp. 1-19.

[Huynh *et al.* 1990] T. Huynh, C. Lassez and J-L. Lassez. Practical Issues on the Projection of Polyhedral Sets. *Annals of Mathematics and Artificial Intelligence*, to appear. (Also: IBM Research Report RC 15872, IBM T.J. Watson Research Center, 1990.)

[Jaffar *et al.* 1990] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap. The CLP($\mathcal{R}$) Language and System, *ACM Transactions on Programming Languages*, to appear. (Also: IBM Research Report RC 16292, IBM T.J. Watson Research Center, 1990.)

[Jaffar and Lassez 1986] J. Jaffar and J-L. Lassez. Constraint Logic Programming. Technical Report 86/73, Dept. of Computer Science, Monash University (June 1986). (An abstract appears in: *Proc.* 14^{th} *Principles of Programming Languages*, Munich, 1987, pp. 111–119.)

[Kohler 1967] D.A. Kohler. Projections of Polyhedral Sets. Ph.D. Thesis, Technical report ORC-67-29, Operations Research Center, University of California at Berkeley (August 1967).

[Lassez *et al.* 1989] J-L. Lassez, T. Huynh and K. McAloon. Simplification and Elimination of Redundant Linear Arithmetic Constraints. In *Proc. North American Conference on Logic Programming*, Cleveland, 1989. pp. 35–51.

[Lassez and McAloon 1988] J-L. Lassez and K. McAloon. Generalized Canonical Forms for Linear Constraints and Applications. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988. pp. 703–710.

[Lassez and Maher 1988] J-L. Lassez and M. Maher. On Fourier's Algorithm for Linear Arithmetic Constraints. *Journal of Automated Reasoning*, to appear.

[Paterson and Wegman 1978] M.S. Paterson and M.N. Wegman. Linear Unification. *Journal of Computer and System Sciences*, Vol. 16, No. 2 (1978), pp. 158–167.

[Tarski 1951] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, USA, 1951.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Adapting CLP($\mathcal{R}$) To Floating-Point Arithmetic

J.H.M. Lee and M.H. van Emden

Logic Programming Laboratory, Department of Computer Science
University of Victoria, Victoria, B.C., Canada V8W 3P6
{jlee,vanemden}@csr.uvic.ca

Abstract

As a logic programming language, Prolog has shortcomings. One of the most serious of these is in arithmetic. CLP($\mathcal{R}$) though a vast improvement, assumes perfect arithmetic on reals, an unrealistic requirement for computers, where there is strong pressure to use floating-point arithmetic. We present an adaptation of CLP($\mathcal{R}$) where the errors due to floating-point computation are absorbed by the use of intervals in such a way that the logical status of answers is not jeopardized. This system is based on Cleary's "squeezing" of floating-point intervals, modified to fit into Mackworth's general framework of the Constraint-Satisfaction Problem. Our partial implementation consists of a meta-interpreter executed by an existing CLP($\mathcal{R}$) system. All that stands in the way of correct answers involving real numbers is the planned addition of outward rounding to the current prototype.

1 Introduction

Mainstream computing holds that programming should be improved by gradual steps, as exemplified by the methods of structured programming and languages such as Pascal and Ada. Revolutionaries such as Patrick Hayes and Robert Kowalski advocated radical change, as embodied in Hayes's motto: "Computation is deduction." [Hayes 1973] According to this approach, programs are definitions in a declarative language and every computation step is a valid inference, so that results are logical consequences of program and data. Logic programming, Prolog and the CLP scheme are examples of this radical alternative in programming languages and method. Where Prolog coincides with logic programming, *certainty* of knowledge is obtained.

In numerical analysis there is a similar tension between mainstream thinking and the radicals. The first is satisfied without rigorous control of errors. When successive approximations differ by a small amount, it is assumed that a result has been obtained with an error of approximately that amount. Of course sophisticated error analyses can be made to suggest more certain knowledge. But such analyses are typically valid only asymptotically. In practice one does not know whether one is close enough to the true value for the asymptotic analysis to be applicable at all.

The radical alternative in numerical computation is represented by interval methods, where the ideal is to be sure that the true value is contained in an interval. It is then the purpose of iteration to shrink such an interval till it is no greater than an acceptable width. Here again the goal is *certainty* of knowledge.

In the research reported in this paper, we bring these two radical streams together. Both streams are, in their present form, deficient. Logic programming lacks in control of numerical errors. Interval methods rely on conventional algorithmic languages and hence lack computation as deduction. We show that the two can be combined in such a way that rigorously justified claims can be made about the error in numerical computation even if conventional floating-point arithmetic is used.

Problem statement. Logic programming, exemplified by Prolog, is the most successful realization of Hayes's motto. In certain application areas, Prolog can be used to program efficient computations that are also logical deductions. However, Prolog arithmetic primitives, which are functional in nature, are incompatible with the relational paradigm of logic programming. The advent of CLP($\Re$), an instance of the CLP scheme [Jaffar and Lassez 1987], takes us closer to relational arithmetic but its implementation CLP($\mathcal{R}$)[1] [Jaffar *et al.* 1990] is obtained by substituting each real number by a single floating-point approximation. As a result, round-off errors destroy soundness and disqualify CLP($\mathcal{R}$) computation as deduction.

Solution. Our solution consists of three parts. First, we tackle the round-off error problem with interval arithmetic introduced in [Moore 1966]. Instead of operating on individual floating-point numbers, interval arithmetic

[1] In this paper, we use CLP($\Re$) to denote the CLP instance with $\Re$ being the algebraic structure of finite trees of reals; and CLP($\mathcal{R}$) is the name of a CLP($\Re$) implementation.

manipulates intervals. The guaranteed inclusion property of interval arithmetic ensures the soundness of computation.

Second, traditional interval arithmetic is functional and has been embedded in functional or imperative languages. To develop the required relational version, we use an interval narrowing operation based on work in [Cleary 1987] and similar to the one used in [Sidebottom and Havens 1992].

Finally, we make a modification to the CLP scheme by including an operation that reduces goal to normal form and show that interval narrowing is such an operation. By modifying a meta-interpreter for CLP($\mathcal{R}$) accordingly, we obtain a prototype implementation.

The paper is organized as follows. Section 2 discusses related work. Relational interval arithmetic, which consists of interval narrowing and a relaxation algorithm, is presented in section 3. In section 4, we describe the semantics of ICLP($\mathcal{R}$), which is CLP($\mathcal{R}$) extended with relational interval arithmetic. We summarize and conclude in section 5.

2 Related Work

Interval arithmetic. While it is important to derive new and more efficient interval arithmetic algorithms and ensure delivery of practical interval bounds, recent development in interval arithmetic [Moore 1988] emphasizes: (1) automatic verification of computed answers, and (2) clear mathematical description of the problem. Users of numerical programs are usually only interested in the solution of a problem. They do not want to take the burden (a) to understand how the problem is solved, (b) to validate the correctness of the answers, and (c) to calculate error bounds. Logic programming shares these goals.

Constraint interval arithmetic. Constraint interval arithmetic stems from constraint propagation techniques. It is a form of "label inference," where the labels are intervals [Davis 1987]. ENVISION [de Kleer and Brown 1984] performs qualitative reasoning about the behaviour of physical systems over time. TMM [Dean 1985] is a temporal constraint system that records and reasons with changes to the world over time. SPAM [McDermott and Davis 1984] performs spatial reasoning. These systems are based on consistency techniques [Mackworth 1977], which handle only static constraint networks. To be able to generate constraints, the described systems are equipped with programming languages tailored to the application.

Constraint logic programming. Cleary incorporates a relational version of interval arithmetic, which he calls *Logical Arithmetic* [Cleary 1987], into Prolog. He introduces a new term "interval", which requires an extension of the unification algorithm. Cleary presents several "squeezing" algorithms that reduce arithmetic constraints over intervals. A constraint relaxation cycle coordinates the execution of the squeezing algorithms. However, there is a semantic problem in this approach. Variables bound to intervals, which are terms in the Herbrand universe, can be re-bound to smaller intervals. This is not part of resolution, where only a variable can be bound. It is not clear in what other, if any, sense this may be a logical inference. BNR Prolog [Older and Vellino 1990] has a partial implementation of logical arithmetic, which only handles closed intervals. The Echidna constraint reasoning system also supports relational interval arithmetic [Sidebottom and Havens 1992]. It is based on hierarchical consistency techniques [Mackworth *et al.* 1985]. Echidna is close to CHIP [Dincbas *et al.* 1988]; whereas we remain within the CLP framework.

3 Relational Interval Arithmetic

Cleary describes several algorithms to reduce constraints on intervals [Cleary 1987]. These algorithms work under a basic principle: they narrow intervals associated with a constraint by removing values that do not satisfy the constraint. We study the set-theoretic aspect of the algorithms and generalize them for narrowing intervals constrained by a relation p on $\mathbb{R}^n$. We then discuss interval narrowing for several common arithmetic relations. Interval narrowing is designed for the reduction of a single constraint. Typically, several constraints interact with one another by sharing intervals, resulting in a constraint *network*. We present an algorithm that coordinates the applications of interval narrowing to constraints in the network.

3.1 Basics of Interval Arithmetic

We use $\mathbb{R}$ to denote the set of real numbers and $\mathbb{F}$ a set of floating-point numbers. We also distinguish between *real intervals* and *floating-point intervals*. The set of real intervals, $I(\mathbb{R})$, is defined by

$$\begin{aligned} I(\mathbb{R}) \;=\; & \{(a,b] \mid a \in \mathbb{R} \cup \{-\infty\}, b \in \mathbb{R}\} \cup \\ & \{[a,b) \mid a \in \mathbb{R}, b \in \mathbb{R} \cup \{+\infty\}\} \cup \\ & \{[a,b] \mid a,b \in \mathbb{R}\} \cup \\ & \{(a,b) \mid a \in \mathbb{R} \cup \{-\infty\}, b \in \mathbb{R} \cup \{+\infty\}\}. \end{aligned}$$

Replacing $\mathbb{R}$ by $\mathbb{F}$ in the definition of $I(\mathbb{R})$, we obtain the definition of floating-point intervals. The symbols $-\infty$ and $+\infty$ are used to represent intervals without lower and upper bounds respectively. Every interval has the usual set denotation. For example, $[e,\pi) = \{x \mid e \leq x < \pi\}$, $(-\infty, 4.5] = \{x \mid x \leq 4.5\}$, and $(-\infty, +\infty) = \mathbb{R}$. We impose a partial ordering on real intervals; an interval I_1

is *smaller than or equal to* an interval I_2 if and only if $I_1 \subseteq I_2$. Given a set of intervals $\mathcal{I}$. $I \in \mathcal{I}$ is the *smallest* interval in $\mathcal{I}$ if I is smaller than or equal to I' for all $I' \in \mathcal{I}$.

Conventionally, real numbers are approximated by floating-point numbers by means of rounding or truncation. We approximate real intervals by floating-point intervals using the *outward rounding* function, $\xi : I(\mathbb{R}) \rightarrow I(\mathbb{F})$; if I is a real interval, $\xi(I)$ is the smallest floating-point interval containing I. It follows that $\xi(I) = I$ for each floating-point interval I. The IEEE floating-point standard [IEEE 1987] provides three user-selectable *directed rounding modes*: round toward $+\infty$, round toward $-\infty$, and round toward 0. The first two modes are essential and sufficient to implement outward rounding: we round toward $+\infty$ at the upper bound and toward $-\infty$ at the lower bound. In most hardware that conforms to the IEEE standard, performing directed rounding amounts to setting a hardware flag before performing the arithmetic operation.

We state without proof the following properties of the outward rounding function.

Lemma 1: If $A \in I(\mathbb{R})$ and $a' \in \xi(A)$, then there exists $a \in A$ such that $a' \in \xi([a,a])$. ∎

Lemma 2: If $A \in I(\mathbb{F})$, $a' \in A$, and $a \in \xi([a',a'])$, then $a \in A$. ∎

3.2 Interval Narrowing

An *i-constraint* is of the form $(p, \vec{I})$, where p is a relation on $\mathbb{R}^n$ and $\vec{I} = \langle I_1, \ldots, I_n \rangle$ is a tuple of floating-point intervals. Note that the number of intervals in the tuple $\vec{I}$ is equal to the arity of p. For any relation p of arity n, we can associate n set-valued functions with p:

$$\begin{aligned} & F_i(p)(S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_n) \\ = \; & \{s_i \mid (s_1, \ldots, s_n) \in \mathcal{S}(p)\} \\ = \; & \pi_i(\mathcal{S}(p)), \end{aligned}$$

where $i = 1, \ldots, n$, the S_i's are sets, π_i is the projection function defined by $\pi_i(p) = \{s_i \mid (s_1, \ldots, s_n) \in p\}$, and $\mathcal{S}(p) = (S_1 \times \cdots \times S_{i-1} \times \pi_i(p) \times S_{i+1} \times \cdots \times S_n) \bigcap p$.

To ensure that the result of narrowing is an interval, we consider only relations p on $\mathbb{R}^n$, such that each $F_i(p)$ maps intervals to intervals. We now specify *interval narrowing* as an input-output pair.

Input: $\vec{I} = \langle I_1, \ldots, I_n \rangle$, where
I_i is a floating-point interval $(1 \leq i \leq n)$.
Output: $\vec{I'} = \langle I'_1, \ldots, I'_n \rangle$, where
$I'_i = I_i \bigcap \xi(F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n))$.

The application of ξ in the formula ensures that the output intervals are floating-point intervals. If one or more I'_i is empty, then interval narrowing fails and the i-constraint $(p, \vec{I})$ is *i-inconsistent*. Otherwise it succeeds with $I'_1, \ldots, I'_n$ as output. Note that the output interval I'_i is a subset of the corresponding input interval I_i.

For example, the $F_i(\texttt{add})$'s of the relation $\texttt{add} = \{(x,y,z) \mid x,y,z \in \mathbb{R}, x+y=z\}$ are

$$F_1(\texttt{add})(I_2, I_3) = I_3 \ominus I_2, \quad F_2(\texttt{add})(I_1, I_3) = I_3 \ominus I_1, \quad F_3(\texttt{add})(I_1, I_2) = I_1 \oplus I_2,$$

where $A \oplus B = \{a + b \mid a \in A, b \in B\}$ and $A \ominus B = \{a - b \mid a \in A, b \in B\}$.

The following theorem shows the soundness and completeness of interval narrowing.

Theorem 3: Let C be $(p, \langle I_1, \ldots, I_n \rangle)$. If $(x_1, \ldots, x_n) \in p$ and $\langle I'_1, \ldots, I'_n \rangle$ are the output intervals obtained from interval narrowing of C, then $(x_1, \ldots, x_n) \in I_1 \times \cdots \times I_n$ if and only if $(x_1, \ldots, x_n) \in I'_1 \times \cdots \times I'_n$.

Proof: Since $I'_1 \times \cdots \times I'_n \subseteq I_1 \times \cdots \times I_n$, the if-part of the lemma is true. In the following, we prove the only-if-part of the lemma.

Suppose $(x_1, \ldots, x_n) \in I_1 \times \cdots \times I_n \bigcap p$. We have $x_i \in I_i$ for $i = 1, \ldots, n$ and $x_i \in F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)$ by definition. Therefore $x_i \in I'_i$ and $(x_1, \ldots, x_n) \in I'_1 \times \cdots \times I'_n$. ∎

The next lemma assists in expressing interval narrowing in terms of relational operations.

Lemma 4: For $A \in I(\mathbb{F})$ and $B \in I(\mathbb{R})$, $A \bigcap \xi(B) = \xi(A \bigcap B)$. ∎

We rewrite the output of interval narrowing as follows:

$$\begin{aligned} I'_i &= I_i \bigcap \xi(F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)) \\ &= \xi(I_i \bigcap F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)) \text{ by lemma 4} \\ &= \xi(I_i \bigcap \pi_i(\mathcal{I}(p))) \\ &= \xi(I_i \bigcap \{x_i \mid (x_1, \ldots, x_n) \in \mathcal{I}(p)\}) \\ &= \xi(\{x_i \in I_i \mid (x_1, \ldots, x_n) \in \mathcal{I}(p)\}) \\ &= \xi(\{x_i \mid (x_1, \ldots, x_n) \in ((I_1 \times \cdots \times I_n) \bigcap p)\}) \\ &= \xi(\pi_i((I_1 \times \cdots \times I_n) \bigcap p)), \end{aligned}$$

where $\mathcal{I}(p) = (I_1 \times \cdots \times I_{i-1} \times \pi_i(p) \times I_{i+1} \times \cdots \times I_n) \bigcap p$. In essence, interval narrowing computes the intersection of $I_1 \times \cdots \times I_n$ and p, and outward-rounds each projection of the resulting relation. We show in [Lee and van Emden 1991b] that interval narrowing is an instance of the LAIR rule [Van Hentenryck 1989], which is based on the arc consistency techniques [Mackworth 1977]. Figure 1 illustrates the interval narrowing of the constraint $(\texttt{le}, \langle I_1, I_2 \rangle)$, where $\texttt{le} = \{(x,y) \mid x,y \in \mathbb{R}, x \leq y\}$. In the diagram, the initial floating-point intervals are I_1 and I_2. The dotted region denotes the relation le; the region for $I_1 \times I_2$ is shaded with a straight-line pattern. Interval narrowing returns I'_1 and I'_2 by taking the projections of the intersection of the two regions. There is no need to perform outward-rounding in this example since the bounds of I'_1 and I'_2 share those of I_1 and I_2.

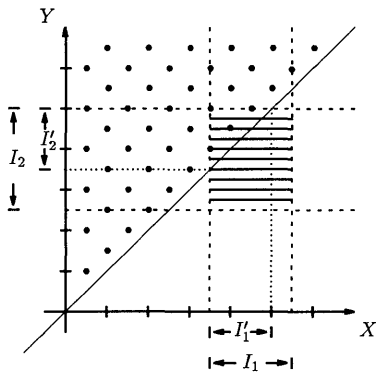


Figure 1: Pictorial illustration of interval narrowing.

3.3 Arithmetic Primitives

A useful relational interval arithmetic system should support some primitive arithmetic constraints, such as addition and multiplication. More complex constraints can then be built from these primitives. To ensure that a relation p is suitable for interval narrowing, we need to check that each $F_i(p)$ maps from real intervals to real intervals. If p is one of

$$\begin{aligned} \texttt{eq} &= \{(x,x) \mid x \in I\!R\}, \\ \texttt{add} &= \{(x,y,z) \mid x,y,z \in I\!R, x+y=z\}, \\ \texttt{le} &= \{(x,y) \mid x,y \in I\!R, x \leq y\}, \\ \texttt{lt} &= \{(x,y) \mid x,y \in I\!R, x < y\}, \end{aligned}$$

we can verify easily that the $F_i(p)$'s satisfy the criterion.

The case for the multiplication relation $\texttt{multiply} = \{(x,y,z) \mid x,y,z \in I\!R, xy = z\}$, requires further explanation. Consider

$$F_1(\texttt{multiply})(I_2, I_3) = I_3 \oslash I_2 \text{ and}$$
$$F_2(\texttt{multiply})(I_1, I_3) = I_3 \oslash I_1,$$

where $A \oslash B = \{a/b \mid a \in A, b \in B, b \neq 0\}$. Note that $A \oslash B$ is not an interval in general. For example, $[1,1] \oslash [-2,3] = (-\infty, -1/2] \cup [1/3, +\infty)$ is a union of two disjoint intervals. The `multiply` relation does not satisfy the criterion for interval narrowing.

As suggested in [Cleary 1987], we can circumvent the problem by partitioning `multiply` into $\texttt{multiply}^+$ and $\texttt{multiply}^-$, where

$$\begin{aligned} \texttt{multiply}^+ &= \{(x,y,z) \mid x,y,z \in I\!R, x \geq 0, xy = z\}, \\ \texttt{multiply}^- &= \{(x,y,z) \mid x,y,z \in I\!R, x < 0, xy = z\}. \end{aligned}$$

By restricting interval narrowing to one partition or the other, we can guarantee that the result of interval division is an interval. When a `multiply` constraint is encountered, we choose one of the partitions and perform narrowing; the other partition is visited upon automatic backtracking or under user control.

An advantage of relational interval arithmetic is that we do not have the division-by-zero problem. For example, the i-constraint $(\texttt{multiply}^+, \langle (4, +\infty), 0, [-3, 5) \rangle)$ is reduced to $(\texttt{multiply}^+, \langle (4, +\infty), 0, 0 \rangle)$.

Relations induced from transcendental functions and the disequality relation, such as $\texttt{sin} = \{(x,y) \mid x,y \in I\!R, y = \sin(x)\}$ and $\texttt{dif} = \{(x,y) \mid x,y \in I\!R, x \neq y\}$, also suffer from the same problem as the `multiply` relation. Similarly, we can solve the problem by appropriate partitioning of the relations.

3.4 Constraint Networks

The interval narrowing discussed so far reduces individual i-constraints. In practice, we have more than one constraint in a problem. These constraints may depend on one another by sharing intervals. By naming an interval by a variable and by having a variable occur in more than one constraint, we indicate that constraints share intervals. Note that the material in this section is not related to logic programming but is in conventional notation with destructive assignment[2]. We define an *i-network* to be a set of i-constraints. Consider the quadratic equation $x^2 - x - 6 = 0$, which can be rewritten to $x(x-1) = 6$. Suppose our initial guess for the positive root of the equation is $[1, 100]$. We can express the equation by the following i-network:

$$\{(\texttt{add}, \langle V_1, [1,1], V \rangle), (\texttt{multiply}^+, \langle V, V_1, [6,6] \rangle)\},$$

where the variables V and V_1 are intervals $[1, 100]$ and $(-\infty, +\infty)$ respectively.

Our goal is to use interval narrowing to reduce i-networks. Note the following two observations. First, the reduction of an i-constraint C in the i-network affects other i-constraints that share variables with C. Second, interval narrowing is idempotent as shown in the following lemma.

Lemma 5: Let $\vec{I} = \langle I_1, \ldots, I_n \rangle$, $\vec{I}' = \langle I_1', \ldots, I_n' \rangle$, and $\vec{I}'' = \langle I_1'', \ldots, I_n'' \rangle$ be tuples of floating-point intervals and p a relation on $I\!R^n$. If, by interval narrowing, $\vec{I}'$ is obtained from $\vec{I}$ and $\vec{I}''$ is obtained from $\vec{I}'$, then $\vec{I}' = \vec{I}''$.

Proof: To prove the equality of $\vec{I}'$ and $\vec{I}''$, we prove $I_i' = I_i''$ for $i = 1, \ldots, n$. By the definition of interval narrowing and lemma 4, we have

$$\begin{aligned} I_i' &= \xi(I_i \bigcap F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)), \\ I_i'' &= \xi(I_i' \bigcap F_i(p)(I_1', \ldots, I_{i-1}', I_{i+1}', \ldots, I_n')). \end{aligned}$$

It is obvious that $I_i'' \subseteq I_i'$. Next we prove $I_i' \subseteq I_i''$.

Suppose $a_i' \in I_i'$. There exists

$$a_i \in (I_i \bigcap F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n))$$

[2] In section 4, we show how we use logical variables to replace the conventional variables.

such that $a_i' \in \xi([a_i, a_i])$ by lemma 1. By the definition of $F_i(p)$, for $j = 1, \ldots, i-1, i+1, \ldots, n$, there exists $a_j \in I_j$ such that $(a_1, \ldots, a_n) \in p$. Since $a_i \in I_i$, we have $a_j \in I_j'$ for each j. Thus, $a_i \in F_i(p)(I_1', \ldots, I_{i-1}', I_{i+1}', \ldots, I_n')$. This implies that $a_i \in I_i''$. By lemma 2, $a_i' \in I_i''$. ∎

An i-constraint $(p, \vec{I})$ is *stable* if applying interval narrowing on $\vec{I}$ results in $\vec{I}$. An i-network is *stable* if every i-constraint in the i-network is stable. The reduction of an i-network amounts to transforming it into a stable one.

A naive approach for the reduction of an i-network is to reduce each i-constraint in the i-network in turn until every i-constraint becomes stable. As suggested by lemma 5, this method is inefficient since much computation is wasted in reducing stable i-constraints. Algorithm 1, which is based on the constraint relaxation algorithm described in [Cleary 1987], is the pseudocode of a more efficient procedure. The algorithm tries to avoid reductions of stable i-constraints and, in this respect, it is similar to AC-3 [Mackworth 1977] and the Waltz algorithm [Waltz 1975]. Without loss of generality, we assume that every i-constraint in the i-network is of the form $(p, \langle V_1, \ldots, V_n \rangle)$, where V_i's are interval-valued variables.

```
initialize list A to hold all i-constraints in the i-network
initialize P to the empty list
while A is not empty
   remove the first i-constraint, (p, V⃗), from A
   apply interval narrowing on V⃗ to obtain V⃗'
   if interval narrowing fails then
      exit with failure
   else if V⃗ ≠ V⃗' then
      V⃗ ← V⃗'
      foreach i-constraint (q, Y⃗) in P
         if V⃗ and Y⃗ share narrowed variable(s) then
            remove (q, Y⃗) from P and append it to A
         endif
      endforeach
   endif
   append (p, V⃗) to the end of P
endwhile
```

Algorithm 1: A Relaxation Algorithm.

Algorithm 1 resembles a classical iterative numerical-approximation technique called "relaxation" [Southwell 1946], which was first adopted in a constraint system in [Sutherland 1963]. Numerical relaxation may have numerical stability problems; the procedure may fail to converge or terminate even when the constraints have a solution. Algorithm 1 does not suffer from this problem as shown in the following theorem.

Theorem 6: Algorithm 1 terminates. The resulting i-network is either i-inconsistent or stable. ∎

The validity of theorem 6 is easy to check since the precision of a floating-point system is finite and thus interval narrowing cannot occur indefinitely due to the use of outward rounding.

In the following, we show how algorithm 1 finds the positive root of the equation $x^2 - x - 6 = 0$ with initial guess in $[1, 100]$. Initially, the passive list, P, is empty and the active list of i-constraints, A, is $[C_1, C_2]$, where

$$\begin{aligned} C_1 &= (\texttt{add}, \langle V_1, [1,1], V \rangle) \text{ and} \\ C_2 &= (\texttt{multiply}^+, \langle V, V_1, [6,6] \rangle). \end{aligned}$$

We remove the first i-constraint C_1 from A and reduce it as shown in figure 2.

The updated values of V and V_1 are $[1, 100]$ and $[0, 99]$ respectively. Similar narrowing is performed on the $\texttt{multiply}^+$ i-constraint. The process repeats until the precision of the underlying floating-point system is reached and no more narrowing takes place. The history of the values of A, P, V, and V_1, with four significant figures, after each narrowing is summarized in table 1.

Table 1: Traces of A, P, V, and V_1.

A	P	V	V_1
$[C_1, C_2]$	[]	$[1, 100]$	$(-\infty, +\infty)$
$[C_2]$	$[C_1]$	$[1, 100]$	$[0, 99]$
$[C_1]$	$[C_2]$	$[1, 100]$	$[0.06, 6]$
$[C_2]$	$[C_1]$	$[1.06, 7]$	$[0.06, 6]$
$[C_1]$	$[C_2]$	$[1.06, 7]$	$(0.8571, 5.661)$
$[C_2]$	$[C_1]$	$(1.857, 6.661)$	$(0.8571, 5.661)$
$[C_1]$	$[C_2]$	$(1.857, 6.661)$	$(0.9009, 3.231)$
$[C_2]$	$[C_1]$	$(1.900, 4.231)$	$(0.9009, 3.231)$
$[C_1]$	$[C_2]$	$(1.900, 4.231)$	$(1.418, 3.157)$
⋮	⋮	⋮	⋮
[]	$[C_1, C_2]$	$(2.999, 3.001)$	$(1.999, 2.001)$

It is well-known that arc-consistency techniques are "incomplete" [Mackworth 1977]: a network can be stable but neither a solution nor inconsistency is found. In the finite domain case, enumeration, instantiation, and backtracking can be used to find a particular solution after the constraint network becomes stable. This method is infeasible for interval domains, which are infinite sets. We use domain splitting [Van Hentenryck 1989] in place of enumeration and instantiation. When an i-network becomes stable, we split an interval into two partitions, choose one partition and visit the other upon automatic backtracking or user control.

Input Intervals = $\langle I_1, I_2, I_3 \rangle$					$(-\infty, +\infty)$	$[1,1]$	$[1,100]$
F_1(add)	=	$[1,100]$	$\ominus$	$[1,1]$	$[0,99]$		
F_2(add)	=	$[1,100]$	$\ominus$	$(-\infty,+\infty)$		$(-\infty,+\infty)$	
F_3(add)	=	$(-\infty,+\infty)$	$\oplus$	$[1,1]$			$(-\infty,+\infty)$
Output Intervals = $\langle I_1', I_2', I_3' \rangle$					$[0,99]$	$[1,1]$	$[1,100]$

Figure 2: Interval narrowing of an add constraint.

4 ICLP($\mathcal{R}$)

So far, we have explained how a network of constraints in terms of floating-point intervals can be made stable. We have not considered how such networks can be specified. One language is CLP($\mathcal{R}$). An i-constraint $(p, \langle I_1, \ldots, I_n \rangle)$ can be expressed in CLP($\mathcal{R}$) as

$$X_1 \in I_1, \ldots, X_n \in I_n, p(X_1, \ldots, X_n),$$

where $X_i \in I_i$ stands for an appropriate set of inequalities. In this representation, we don't need conventional variables. Constraints share intervals when they share logical variables.

When a query to a logic program is answered according to the CLP scheme at each step, a network of constraints is solved. A special case of such a constraint is a variable's membership of a domain. According to the CLP scheme, it is possible at any stage that the domain inclusions of the current set of constraints is inconsistent in Mackworth's sense of the Constraint-Satisfaction Problem. We modify the basic CLP scheme by inserting a constraint simplification step and show that interval narrowing is a constraint simplification operation.

In principle, any of the constraint-satisfaction algorithms by Mackworth can be used. In this paper we are concerned with real-valued variables. As we argued, the only known way of obtaining correct answers involving real numbers on a machine with floating-point numbers is to use interval arithmetic. Accordingly, we explain the theory of ICLP($\mathcal{R}$), where the sub-network consisting of i-constraints is narrowed according to the method described above.

An example. The i-constraint

$$(\text{add}, \langle [0,2], [1,3], [4,6] \rangle)$$

is represented by

$$X \geq 0, X \leq 2, Y \geq 1, Y \leq 3, Z \geq 4, Z \leq 6, X + Y = Z.$$

However, if we submit the above example as a query to CLP($\mathcal{R}$) Version 2.02, we get the answer constraint

$$X = -Y + Z, 2 + Y \geq Z, Y \geq 1, 3 \geq Y, Z \geq 4, \\ 6 \geq Z, X \geq 0.$$

Examining the answer more carefully, we note that

$$X = -Y + Z \equiv X + Y = Z \text{ and } 2 + Y \geq Z \equiv X \leq 2.$$

Thus the answer constraint is the original query disguised in a slightly different form. CLP($\mathcal{R}$) only checks the solvability of the constraint but does not remove undesirable values from the intervals. A more useful answer constraint is

$$X \geq 1, X \leq 2, Y \geq 2, Y \leq 3, Z \geq 4, Z \leq 5, X + Y = Z.$$

The modified CLP scheme ICLP($\mathcal{R}$) is CLP($\mathcal{R}$) enhanced with interval narrowing and algorithm 1. The operational semantics of ICLP($\mathcal{R}$) is based on a generalization of $\mathcal{M}_\mathcal{X}$-derivations [Jaffar and Lassez 1986]. Let P be a CLP($\mathcal{X}$) program, where $\mathcal{X}$ is a structure with model $\mathcal{M}_\mathcal{X}$, and $\leftarrow G_i$ be a goal. $\leftarrow G_{i+1}$ is *$\mathcal{M}'_\mathcal{X}$-derived* from $\leftarrow G_i$ if

1. $\leftarrow G'$ is $\mathcal{M}_\mathcal{X}$-derived from $\leftarrow G_i$, and
2. $\leftarrow G_{i+1} = \nu(\leftarrow G')$, where ν is a *normal-form function* that maps from goal to goal such that $P \models_{\mathcal{M}_\mathcal{X}} \exists(G') \Leftrightarrow P \models_{\mathcal{M}_\mathcal{X}} \exists(G_{i+1})$.

An *$\mathcal{M}'_\mathcal{X}$-derivation* is a, possibly infinite, sequence of goals $G = G_0, G_1, G_2, \ldots$ such that G_{i+1} is $\mathcal{M}'_\mathcal{X}$-derived from G_i. A $\mathcal{M}'_\mathcal{X}$-derivation is *successful* if it is finite and the last goal contains no atoms. The soundness and completeness of $\mathcal{M}'_\mathcal{X}$-derivations follow directly from the soundness and completeness of $\mathcal{M}_\mathcal{X}$-derivations and the definition of the normal-form function. A $\mathcal{M}'_\mathcal{X}$-derivation is *finitely-failed* if it is finite, the last goal has one or more atoms, and condition 1 does not hold.

The $\mathcal{M}'_\mathcal{X}$-derivation step is not new. In fact, it has been implemented in other CLP systems as a constraint simplification step. The $\mathcal{M}_\mathcal{X}$-derivation step only checks the solvability of the constraint accumulated so far. Therefore, the answer constraint of a successful $\mathcal{M}_\mathcal{X}$-derivation is usually complex and difficult to interpret. A useful system should simplify the constraint to a more "readable" form. For example, CLP($\mathcal{R}$) simplifies the constraint $\{X + Y = 4, X - Y = 1\}$ to $\{X = 2.5, Y = 1.5\}$. Suppose the goal $\leftarrow c', \vec{A}'$ is $\mathcal{M}_\mathcal{X}$-derived from $\leftarrow c, \vec{A}$. CLP($\mathcal{R}$) simplifies c' to c'' such that $\models_{\mathcal{M}_\mathcal{X}} \exists(c') \Leftrightarrow \models_{\mathcal{M}_\mathcal{X}} \exists(c'')$ and thus

$$P \models_{\mathcal{M}_\mathcal{X}} \exists(c', \vec{A}') \Leftrightarrow P \models_{\mathcal{M}_\mathcal{X}} \exists(c'', \vec{A}');$$

CLP($\mathcal{R}$) is based on $\mathcal{M}'_{\mathcal{X}}$-derivation.

Theorem 7: If

$$\begin{aligned} \mathcal{C} &= \{X_1 \in I_1, \ldots, X_n \in I_n, p(X_1, \ldots, X_n)\} \text{ and} \\ \mathcal{C}' &= \{X_1 \in I'_1, \ldots, X_n \in I'_n, p(X_1, \ldots, X_n)\}, \end{aligned}$$

where I'_i is obtained from I_i by interval narrowing for $i = 1, \ldots, n$, then $\models_{\mathcal{M}_{\mathcal{X}}} \exists(\mathcal{C}) \Leftrightarrow \models_{\mathcal{M}_{\mathcal{X}}} \exists(\mathcal{C}')$.

Proof: The theorem follows directly from theorem 3. ■

Theorem 7 guarantees that interval narrowing transforms a constraint into a stable constraint with the same solution space. Algorithm 1, which performs narrowing repeatedly on i-constraints in a network, is thus a normal-form function.

Partitioning of relations can also be expressed compactly in ICLP($\mathcal{R}$). For example, the `multiply` relation can be defined by

```
multiply(X,Y,Z) :- X ≥ 0,multiply+(X,Y,Z).
multiply(X,Y,Z) :- X < 0,multiply-(X,Y,Z).
```

A meta-interpreter ICLP($\mathcal{R}$) written in CLP($\mathcal{R}$) is described in [Lee and van Emden 1991a]. We have not yet included outward rounding in the current implementation. Table 1 is derived from a trace produced by our prototype, except that the outward rounding has been added manually.

5 Concluding Remarks

We have developed the essential components of a relational interval arithmetic system. Interval narrowing establishes (1) the criterion that an arithmetic relation has to satisfy to be used as arithmetic constraint in relational interval arithmetic, and (2) the reduction of arithmetic constraint using the interval functions induced from the constraint. Algorithm 1 then coordinates the applications of narrowing to transform a constraint network into its stable form.

The incorporation of relational interval arithmetic in CLP($\mathcal{R}$) makes it possible to describe programs, constraints, queries, intervals, answers, and variables in a coherent and semantically precise language—*logic*. The semantics of ICLP($\mathcal{R}$) is based on $\mathcal{M}'_{\mathcal{X}}$-derivation, which is a logical deduction. Consequently, *numerical computation is deduction* in ICLP($\mathcal{R}$), which is a general-purpose programming language allowing compact description and dynamic growth of constraint networks. One advantage of ICLP($\mathcal{R}$) over CLP($\mathcal{R}$) is the ability to handle nonlinear constraints, which are delayed in CLP($\mathcal{R}$). It is important to note that ICLP($\mathcal{R}$) is not another instance of the CLP scheme. It is a correct implementation of CLP($\Re$).

The ICLP($\mathcal{R}$) meta-interpreter shows the feasibility of our approach. Future work includes extending CLP($\mathcal{R}$) at the source level, to ICLP($\mathcal{R}$) to improve efficiency. We also plan to investigate applications in such areas as finite element analysis, and spatial and temporal reasoning.

Acknowledgements

We thank Mantis Cheng, Bill Havens, Alan Mackworth, Clifford Walinsky, and the anonymous referees for helpful suggestions. Paul Strooper gave constructive comments to early drafts of this paper. Generous support was provided by the British Columbia Advanced Systems Institute, the Canada Natural Science and Engineering Research Council, the Canadian Institute for Advanced Research, the Institute of Robotics and Intelligent Systems, and the Laboratory for Automation, Communication and Information Systems Research.

References

[Cleary 1987] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[Davis 1987] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[de Kleer and Brown 1984] J. de Kleer and J.S. Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[Dean 1985] T. Dean. Temporal imagery: An approach to reasoning about time for planning and problem solving. Technical Report 433, Yale University, New Haven, CT, USA, 1985.

[Dincbas *et al.* 1988] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, December 1988.

[Hayes 1973] P.J. Hayes. Computation and deduction. In *Proceedings of the Second MFCS Symposium*, pages 105–118. Czechoslovak Academy of Sciences, 1973.

[Jaffar and Lassez 1986] J. Jaffar and J-L. Lassez. Constraint logic programming. Technical Report 72, Department of Computer Science, Monash University, Clayton, Victoria, Australia, 1986.

[Jaffar and Lassez 1987] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM POPL Conference*, pages 111–119, Munich, January 1987.

[Jaffar *et al.* 1990] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP($\mathcal{R}$) language and system. Technical Report CMU-CS-90-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[Lee and van Emden 1991a] J.H.M. Lee and M.H. van Emden. Adapting CLP($\mathcal{R}$) to floating-point arithmetic. Technical Report LP-18 (DCS-183-IR), Department of Computer Science, University of Victoria, Victoria, B.C., Canada, December 1991.

[Lee and van Emden 1991b] J.H.M. Lee and M.H. van Emden. Numerical computation can be deduction in CHIP. Technical Report LP-19 (DCS-184-IR), Department of Computer Science, University of Victoria, Victoria, B.C., Canada, December 1991. (submitted for publication).

[Mackworth 1977] A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.

[Mackworth *et al.* 1985] A.K. Mackworth, J.A. Mulder, and W.S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

[McDermott and Davis 1984] D.V. McDermott and E. Davis. Planning routes through uncertain territory. *Artificial Intelligence*, 22:107–156, 1984.

[Moore 1966] R.E. Moore. *Interval Analysis.* Prentice-Hall, 1966.

[Moore 1988] R.E. Moore, editor. *Reliability in Computing—The Role of Interval Methods in Scientific Computing.* Academic Press, 1988.

[IEEE 1987] Members of the Radix-Independent Floating-point Arithmetic Working Group. IEEE standard for radix-independent floating-point arithmetic. Technical Report ANSI/IEEE Std 854-1987, The Institute of Electrical and Electronics Engineers, New York, USA, 1987.

[Older and Vellino 1990] W. Older and A. Vellino. Extending Prolog with constraint arithmetics on real intervals. In *Proceedings of the Canadian Conference on Computer & Electrical Engineering*, Ottawa, Canada, 1990.

[Sidebottom and Havens 1992] G. Sidebottom and W.S. Havens. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8(2), May 1992.

[Southwell 1946] R.V. Southwell. *Relaxation Methods in Theoretical Physics.* Oxford University Press, 1946.

[Sutherland 1963] I.E. Sutherland. *SKETCHPAD: a Man-Machine Graphical Communication System.* PhD thesis, MIT Lincoln Labs, Cambridge, MA, 1963.

[Van Hentenryck 1989] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[Waltz 1975] D. Waltz. Understanding line drawings of scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision.* McGraw-Hill, 1975.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Domain Independent Propagation

Thierry Le Provost, Mark Wallace
ECRC, Arabellastr 17
8000 München 81, Germany
{thierry | mark}@ecrc.de

Abstract

Recent years have seen the emergence of two main approaches to integrating constraints into logic programming. The *CLP Scheme* introduces constraints as basic statements over built-in computation domains. On the other hand, systems such as CHIP have introduced new inference rules, which enable certain predicates to be used for propagation thereby pruning the search tree. Unfortunately these two complementary approaches were up to now incompatible, since propagation techniques appeared intimately tied to the notion of finite domains. This paper introduces a generalisation of propagation that is applicable to any CLP computation domain, thereby restoring orthogonality and bridging the gap between two important constraint logic programming paradigms. The practical interest of this new notion of "domain independent" propagation is demonstrated by applying a prototype system for solving some hard search problems.

1 Introduction

There are two main approaches for integrating constraints into logic programming. The first approach, formalised as $CLP(X)$ [Jaffar and Lassez, 1987], is to replace the usual domain of computation with a new domain X. The computation domain X specifies a universe of values; a set of predefined functions and relations on this universe; and a class of *basic constraints*, which are formulae built from predefined predicate and function symbols, and logical connectives. The CLP scheme requires that an effective procedure decide on the satisfiability of the basic constraints. The facility to define new predicates as facts or rules, possibly involving the built-in's, is carried over from logic programming. The evaluation of queries involving such user-defined predicates is performed using an extension of resolution, where syntactic unification is replaced with deciding the satisfiability of basic constraints (constraint solving). As with standard logic programming the default search method for evaluating program-defined predicates is depth-first, based on the ordering of program clauses and goals.

The second main approach to integrating constraints in logic programming uses the standard, syntactic, domain of computation, except that the variables may be restricted to explicitly range over finite subsets of the universe of values (*finite domain variables*) [Van Hentenryck and Dincbas, 1986]. In this approach, inaugurated by *CHIP* [Dincbas *et al.*, 1988], it is the proof system that is extended. The new type of controlled inference is termed *constraint propagation* or consistency techniques [Van Hentenryck, 1989]. These techniques combine solution-preserving simplification rules and tree search, and were originally introduced for solving constraint satisfaction problems [Montanari, 1974; Mackworth, 1977].

Informally constraint propagation aims at exploiting program-defined predicates as constraints. It operates by looking ahead at yet unsolved goals to see what locally consistent valuations there remain for individual problem variables. Such constraint techniques can have a dramatic effect in cutting down the size of the search space [Dincbas *et al.*, 1990].

To date the technique of propagation has only been defined for search involving finite domain variables. Each such variable can only take a finite number of values, and looking ahead is a way of deterministically ruling out certain locally inconsistent values and thus reducing the domains. This restriction has prevented the application of propagation to new computation domains introduced by the $CLP(X)$ approach. In addition propagation, as currently defined, cannot reason on com-

pound terms, thereby enforcing an unnatural and potentially inefficient encoding of structured data as collections of constants.

This has meant that the two approaches to integrating constraints into logic programming have had to remain quite separate. Even in the CHIP system which utilises both types of integration, propagation is excluded from those parts of the programs involving new computation domains, such as Boolean algebra or linear rational arithmetic.

This paper proposes a generalisation of propagation, which enables it to be applied on arbitrary computation domains. Generalised propagation can be applied in $CLP(X)$ programs, whatever the domain X. Furthermore its basic concepts, theoretical foundations, and abstract operational semantics can be defined independently of the computation domain. This allows programmers to reason about the efficiency of CLP programs involving propagation in an intuitive and uniform way. This generality carries over to the implementation, where algorithms for executing generalised propagation apply across a large range of basic constraint theories. Last but not least, the declarative semantics of CLP programs is preserved.

The main idea behind generalised propagation is to use whatever basic constraints are available in a $CLP(X)$ language to express restrictions on problem variables. Goals designated as propagation constraints are repeatedly approximated to the finest basic constraint preserving their solutions. When no further refinement of the current resolvent's basic constraint is feasible, a resolution step is performed and propagation starts again.

The practical relevance of generalised propagation has been tested by implementing it in the computation domain of Prolog. Programs are just sets of Prolog rules with annotations identifying the goals to be used for propagation. The language has enabled us to write programs which are simple, yet efficient, without the need to resort to constructs without a clear declarative semantics such as demons. The performance results have been very encouraging.

In the next section we recall the interest of integrating propagation over finite domains into logic programming. We then present a logical basis for propagation that will provide the basis for generalisation. The following section introduces generalised propagation, and sketches its theoretical basis. The fourth section introduces our prototype system on top of Prolog, and discusses some of the examples that we tackled with it. In conclusion we identify the directions that this work is now taking.

2 Propagation over Finite Domains

2.1 Propagation in Constraint Satisfaction Problems

The study of constraint satisfaction problems has a long history, and we mention here just a few important references. The concept of arc consistency was introduced in [Mackworth, 1977]; its combination with backtrack search was described in [Haralick and Elliot, 1980]; the notion of value propagation is due to [Sussman and Steele, 1980]; the application of constraint methods to real arithmetic was surveyed in [Davis, 1987]; finally [Van Hentenryck, 1989] extensively motivates and describes in detail the integration of finite-domain propagation methods into logic programming.

A constraint satisfaction problem (CSP) can be represented as

- a set of variables, $\{X1, \ldots, Xn\}$, each Xi ranging over a finite domain Di;
- a set of constraints $C1, \ldots, Cm$ on these variables, where each constraint Ci is an atomic goal $p_i(Xi_1, \ldots Xi_k)$ defined by a k-ary predicate p_i.

A solution to the problem is an assignment of values from the domains to the variables (a *labelling*) such that all the constraints are satisfied. We now briefly recall the main approaches to solving CSP's in a logic programming setting, using the following toy example. The problem has four variables $X1, X2, X3, X4$, each with domain $\{a, b, c\}$. There are four constraints, each involving the same binary predicate p:
$p(X3, X1) \wedge p(X2, X3) \wedge p(X2, X4) \wedge p(X3, X4)$
The relation denoted by p has three tuples: $< a, b >, < a, c >, < b, c >$.

Generate and Test This approach enumerates labellings in a systematic way until one is found that satisfies all the constraints. It is hopelessly inefficient for all but the smallest problem instances. In our example the system will go through all 27 labellings which begin with an a, before discovering that $X1$ cannot take this value due to the first constraint $p(X3, X1)$. In general reordering the constraint goals may only bring minor improvements. Analysing the cause for the failure of goals so as to avoid irrelevant backtrack steps (*selective backtracking*) makes the runtime structures more complex and is insufficient for complex problems (see for instance [Wolfram, 1989]).[1]

[1]Selective, or intelligent, backtracking [Codognet and Sola, 1990] addresses the symptom of too many choice points. Propagation addresses the cause, by reducing the number of choice points in advance.

Backtrack Search A first improvement on pure generate-and-test is to check each constraint goal as soon as all its variables have received values [Golomb and Baumert, 1965]. Backtrack search thus performs an implicit enumeration over the space of possible labellings, discarding partial labellings as soon as they can be proved *locally inconsistent* with respect to some constraint goal. Backtrack search demonstrates considerable gains over generate-and-test (the inconsistent assignment $X1 = a$ is detected at once). However this procedure still suffers from "maladies" [Mackworth, 1977], the worst being its repeated discovery of local inconsistencies. For instance it is obvious from $p(X3, X1) \wedge p(X2, X3)$ alone that $X1$ cannot take the value b. Backtrack search will nonetheless consider all 9 combinations of values for $X2$ and $X3$ before rescinding $X1 = b$.

Local Propagation The idea behind local propagation methods for CSP's is to work on each constraint independently, and deterministically to extract information about locally consistent assignments. This has lead to various consistency algorithms for networks of constraints, the most widely applicable of these being arc-consistency [Montanari, 1974]. Consistency can be applied as a preliminary to the search steps or interleaved with them [Haralick and Elliot, 1980]. The application of these techniques in the constraint logic programming language CHIP was accomplished through two complementary extensions [Van Hentenryck and Dincbas, 1986; Van Hentenryck, 1989]

- explicit *finite domains* of values to allow the expression of range restrictions, together with the corresponding extension of unification (FD-resolution)
- new *lookahead inference rules* to reduce finite domains in a deterministic way

The effect of applying lookahead on a goal is to reduce the domains associated with the variables in the goal, so that the resulting domains approximate *as closely as possible* the set of remaining goal solutions. The solutions can be determined by simply calling the goal repeatedly. Application of the lookahead rule is repeated on all constraint goals until no more domain reductions are possible, forming a *propagation sequence*. Constraint goals that are satisfied by any combination of values in the domains of their arguments can now be dropped.

Our example problem can be encoded in a CHIP-like syntax as follows:

```
csp(X1,X2,X3,X4) :-
    lookahead p(X3,X1),  /* [1] */
    lookahead p(X2,X3),  /* [2] */
    lookahead p(X2,X4),  /* [3] */
    lookahead p(X3,X4),  /* [4] */
    dom(X1),dom(X2),dom(X3),dom(X4).
```

The `lookahead` annotations identify goals that must be treated by the new inference rule. Annotations can be ignored for a declarative reading.

For our example problem, the initial propagation sequence is sufficient to produce the only solution; domain goals merely check each of the variable bindings already produced. A possible computation sequence is as follows (though the ordering is immaterial for the final result):

```
lookahead on:          produces:
p(X3,X1)      [1]  X3::{a,b}, X1::{b,c}
p(X2,X3::{a,b}) [2]      X2=a, X3=b
p(a,X4)         [3]      X4::{b,c}
p(b,X1::{b,c})  [4]      X4=c
p(b,X1::{b,c})  [1]      X1=c
p(a,b)          [2]  succeeds
p(a,c)          [3]  succeeds
p(b,c)          [4]  succeeds
```

Note that the constraint [1] takes part in two propagation steps before it is solved. In general constraints may be involved in any number (> 0) of propagation steps.

From this brief summary of consistency techniques for CSP's and their integration into logic programming, it may appear that finite domain variables form the cornerstone of propagation. The purpose of this paper is to show that this is not the case, and that propagation has a very general, natural and useful counterpart in constraint logic programming languages that do not feature finite domains.

2.2 A Logical Basis for Propagation

The effect of (finite domain) propagation on a constraint is to reduce the domains associated with the variables appearing in the constraint. The resulting domains capture as precisely as possible the meaning of the constraint. The aim of this section is to say in what sense the meaning of a constraint is captured by a set of domains, and to give a formal characterisation of the qualification "as precisely as possible".

A constraint $C(X_1, \ldots, X_n)$ is to be understood as a logical formula with free variables $X_1, \ldots, X_n$.
A *constraint formula* has the syntactic form:
$(X_1 = a_{11} \wedge \ldots \wedge X_n = a_{1n}) \vee \ldots \vee (X_1 = a_{k1} \wedge \ldots \wedge X_n = a_{kn})$.
A *domain formula* $Dom(X)$ is a disjunction of equalities involving a single variable X:
$X = a_1 \vee X = a_2 \vee \ldots \vee X = a_n$.

Generally many variables are involved in a problem, and we therefore introduce a syntactic class of formulae representing the conjunction of their domains. These are the *basic formulae*. Thus a basic formula $D(X_1, \ldots, X_n)$ has the form:
$Dom_1(X_1) \wedge \ldots \wedge Dom_n(X_n)$.

The reduced domains, resulting from propagation on a constraint, approximate the constraint formula as closely as is possible using only a basic formula. Propagation is "precise" if this basic formula is logically equivalent to the constraint formula. The problem is that basic formulae have a limited expressive power, and it is not in general possible to find one logically equivalent to a given constraint formula.

For example the constraint formula $C(X_1, X_2)$, $(X_1 = a \wedge X_2 = b) \vee (X_1 = a \wedge X_2 = c) \vee (X_1 = b \wedge X_2 = c)$,
is best approximated by the basic formula
$(X_1 = a \vee X_1 = b) \wedge (X_2 = b \vee X_2 = c)$.
However there is no basic formula logically equivalent to $C(X_1, X_2)$.

Definition 1 *A propagation step takes a constraint formula C and a basic formula D and yields a "least" basic formula D' which satisfies $(C \wedge D) \rightarrow D'$. D' is the least such formula in the sense that for any other basic formula D'' satisfying $(C \wedge D) \rightarrow D''$ it is also true that $D' \rightarrow D''$.*

This definition will be illustrated using the constraint $C(X_1, X_2)$:
$(X_1 = a \wedge X_2 = b) \vee (X_1 = b \wedge X_2 = c) \vee (X_1 = c \wedge X_2 = a)$.
The input basic formula $D(X_1, X_2)$ is:
$(X_1 = a \vee X_1 = b) \wedge (X_2 = a \vee X_2 = b \vee X_2 = c)$.

Propagation on a constraint involves two steps: the *simplification* of the constraint and the *reduction* of domains associated with its variables.

The simplification of the constraint $C(X_1, X_2)$, with respect to the basic constraint $D(X_1, X_2)$ is just the calculation of a simplified constraint logically equivalent to $C(X_1, X_2) \wedge D(X_1, X_2)$. The result of simplifying is $C'(X_1, X_2) \equiv (C(X_1, X_2) \wedge D(X_1, X_2)) \equiv$
$(X_1 = a \wedge X_2 = b) \vee (X_1 = b \wedge X_2 = c)$.

The reduction of the domains is the calculation of a new basic formula which approximates as closely as possible the simplified constraint. The result of reducing is $D'(X_1, X_2) \equiv$
$(X_1 = a \vee X_1 = b) \wedge (X_2 = b \vee X_2 = c)$.
For this example there is no basic constraint logically equivalent to $C'(X_1, X_2)$. However $D'(X_1, X_2)$ is the least basic formula implied by $C'(X_1, X_2)$ since the domain of X_1 must include at least a and b, and the domain of X_2 must include at least b and c.

Definition 2 *Propagation is the result of applying a propagation sequence, which is the repeated application of propagation steps on every constraint until no more domain reductions are possible.*

This definition does not mention the order in which propagation steps are done. In fact the result of performing propagation on a set of constraints is independent of the order. We prove this as follows.

Lemma 1 *If basic formulae are ordered by logical entailment, propagation steps are increasing and monotonic on basic formulae.*

This is easily deduced from the definition of a propagation step.

Lemma 2 *Each (ordered) propagation sequence yields a fixpoint.*

This follows from the fact that there are only finitely many basic formulae greater than a given basic formula under the logical entailment ordering, and propagation steps are increasing.

Theorem 1 *The result of a propagation sequence is independent of the order of the steps.*

Suppose $fix1$ and $fix2$ were distinct fixpoints of a propagation sequence, resulting from an initial basic formula $s0$. Since propagation is increasing, $fix1 \geqslant s0$. $fix2$ results from applying a particular ordered sequence of propagations on $s0$. By monotonicity this same sequence applied to $fix1$ yields a result $fix3 \geqslant fix2$. However since $fix1$ is a fixpoint of the propagation sequence, $fix3 = fix1$. We conclude that $fix1 \geqslant fix2$. Symmetrically we can conclude that $fix2 \geqslant fix1$, and therefore $fix1 = fix2$.

It is also possible to show that propagation can be performed in parallel, and still yield the same fixpoint. These and other results fall out very naturally when lattice theory is used to describe the constraints. The lattice theoretic formalisation of generalised propagation is described in another paper [Le Provost and Wallace, 1992].

3 Generalised Propagation

For finite domain propagation, the basic formulae express domains associated with the problem variables, and the constraint formulae express membership of tuples in relations. Each class of formulae has a certain limited expressive power. However the definition of a propagation step and a propagation sequence do not depend on the particular syntactic classes chosen for basic formulae or constraint formulae. In this section we will explore

the consequences of admitting different classes of formulae. We shall propose a notion of *generalised propagation* parameterised on the classes of formulae.

In the CLP(X) approach a class of basic constraints is identified for each domain X. Generalised propagation on a domain X is the result of admitting the *basic constraints* on X as *basic formulae* as described in the last section. The class of constraint formulae is the class of goals expressible in CLP(X).

The basic formulae used for finite domain propagation involve only the equality predicate and no function symbols. For generalised propagation over a domain X the basic formulae may include other predicates, such as $<$ and $>$, and function symbols such as $+$ and $*$. However the purpose and effects of propagation remain the same. To detect inconsistencies early and to extract as much information as possible from a set of goals deterministically before making any choices. The information extracted is expressed as a basic formula, which is added to the current constraint set, either yielding inconsistency immediately, or else helping to prune the remaining search.

As a simple example of generalised propagation, consider $CLP(\mathcal{Q})$ with atomic constraints $Var \geq num$ and $Var \leq num$, where num is any rational number. Let us define a predicate p on which we shall perform generalised propagation.

```
p(X) :- X >= 3.4, X =< 4.6
p(X) :- X >= 2.8, X =< 3.9
```

Assume the current constraints include $X \leq 4.0$, and $p(X)$ is a goal. The CLP(X) approach requires us to treat user-defined predicates such as p a la Prolog. One clause in the definition of p is selected, and if that yields an inconsistency the other is tried on backtracking.

Generalised propagation on the predicate p, treated this time as a constraint, deterministically derives the tightest basic constraint $C(X)$ satisfying $(p(X) \wedge X \leq 4.0) \rightarrow C(X)$, and adds $C(X)$ to its current set of constraints. In this case $C(X) \equiv (X \geq 2.8 \wedge X \leq 4.0)$, which can be used to prune the remaining search tree.

The case of finite domains can be viewed as an instance, CLP(FD), of the constraint logic programming scheme, where the basic constraints are the basic formulae as defined in section 2.2. Propagation on finite domains can now be seen as an instance of generalised propagation, just as CLP(FD) is an instance of CLP(X). Notice that the expressive power of CLP(FD) is weaker than that of standard logic programming, since it is impossible using domains to state that two variables are equal, until their domains are reduced to one value. This is indeed a weakness of propagation over finite domains, and in the next section we shall present an implementation of generalised propagation that overcomes it.

Unfortunately it is not the case that generalised propagation can be automatically derived for any computation domain X. There is a practical requirement to constructively define a propagation step. Specifically, for propagating on a goal the system requires an efficient way to extract a basic formula which generalises all the answers to the goal.

More fundamentally a theoretical problem arises when we move from finite domain constraints to arbitrary basic constraints. There are only finitely many finite domain constraints tighter than a given constraint. This fact ensures that propagation is bound to reach a fixpoint. However for many sets of basic constraints, such as inequalities over the rationals as exampled above, there is no similar guarantee of termination. This problem has been addressed by introducing a notion of *approximate generalised propagation* in [Le Provost and Wallace, 1992].

4 Propia: An Implementation of Generalised Propagation

4.1 An Overview of the Implementation

The behaviour of generalised propagation in practise has proved to be more than satisfactory. An implementation of generalised propagation has been completed based on ECRC's Sepia prolog system. We call it *Propia*. The underlying domain is the Herbrand domain of standard logic programming. The built-in relation on this domain is '=', and basic constraints are conjunctions of equalities (or equivalently substitutions).

A simple example of generalised propagation over this domain, is propagation on the predicate p defined as follows:

```
p(g(1),a,b).
p(f(a),a,a).
p(g(2),b,a).
p(f(b),b,b).
```

The result of generalised propagation on the goal `p(A,X,X)`, is the deterministic addition of a new equation, $A = f(X)$. Although there are two different possible values for A, they both have the form $f(X)$, where X is the *same* variable occurring as the second and third arguments in the goal. Using finite domains (even if structured terms were admitted) it would only be possible to infer that the domain of X was $\{a, b\}$ and the domain of A

was $\{f(a), f(b)\}$, but not that $A = f(X)$. This is the weakness of finite domain pointed out on page 5 above.

Implementationally constraint simplification with respect to this goal amounts to selecting those clauses in the definition which unify with the goal, as done by Prolog. The reduction step, given a set of answers, finds the set of equations which best approximates them. The best approximation is, in fact, their most specific generalisation.

Computations interleave the making of choices and propagation. When a propagation sequence terminates, goals are called a la Prolog until a new binding, or set of bindings, occur thereby conjoining new equations $X = T$ to the current basic constraint. At this point propagation restarts. When a fixpoint is reached, the propagation sequence is complete and further goals are called a la Prolog.

It would be prohibitively expensive to attempt propagation on all the constraints at each choice. In practise the system determines on which variables new equalities have been added and only propagates on constraints involving those variables. When further equalities are added during a propagation sequence, then propagation is also attempted on constraints involving these variables.

The purpose of propagation is to extract as much information as possible deterministically before making any choices. The *Andorra principle* [Warren, 1988] has a similar intent: it states that deterministic goals should be executed before other goals. The goal $p(A, X, X)$ in the previous example is clearly not deterministic, yet deterministic information can be extracted from it. Lee Naish coined the term *data determinacy* for the determinism detected and used by generalised propagation, as opposed to Andorra's weaker *control determinacy*.

4.2 An Example of Propagation

The behaviour of generalised propagation in the syntactic equality theory can be illustrated using a simple example. We shall investigate what propagation is possible for various calls on the 'and' predicate defined as follows:

```
and(true,true,true).
and(true,false,false).
and(false,true,false).
and(false,false,false).
```

We treat the goal as a propagation constraint by making the call `?- propagate and(_,_,_)`. Note that finite domain variables are not part of our chosen propagation language.

For "most specific generalisation" we shall use the abbreviation *msg*. First if the call is fully uninstantiated `?- propagate and(X,Y,Z)` the system finds the first two answers and forms the msg $and(true, Y, Z)$. After the third answer the msg becomes $and(X, Y, Z)$, which is as little instantiated as the query, and propagation stops.

Second if the call has its first argument instantiated to false `?- propagate and(false,Y,Z)` there are two answers whose msg is $and(false, \_, false)$. Thus the equality $Z = false$ is returned.

Third if the call has its first argument instantiated to true `?- propagate and(true,Y,Z)` there are again two answers, $and(true, false, false)$ and $and(true, true, true)$. Our generalisation procedure is able to derive the equality of the last two arguments and the final msg is $and(true, Y, Y)$. Thus the equality $Y = Z$ is returned.

We note that the behaviour is very similar to that obtained by encoding and using "cut guards" in Andorra, GHC rules, or "demons" in CHIP. For example in CHIP we would write:

```
?- demon and/3.
and(false,Y,Z) :- Z=false.
and(true,Y,Z)  :- Z=Y.
and(X,false,Z) :- Z=false.
and(X,true,Z)  :- Z=X.
and(X,Y,true)  :- X=true, Y = true.
and(X,X,Z)     :- Z=X
```

The difference is that the use of `propagate` enables us to separate the specification of the predicate, from its control. When using guards or demons we are forced to mix them together. Indeed generalised propagation allows declarative specifications to be directly used as constraints!

We used Propia for a benchmark set of propositional satisfiability problems distributed by the *FAW* research institute [Mitterreiter and Radermacher, 1991]. Its behaviour was in general quite comparable to that of CHIP's demons or built-in constraints.

Another application we examined was that of crossword puzzle compilation. The problem is to fill up an empty crossword grid using words from a given (possibly large) lexicon. The propagation constraints enforce membership of words in the given lexicon. Intersections are expressed through shared variables.

The statement of the problem is as follows:

```
/* some lexicon of available words */
word(a).
word(a,b,a,c,k).
...

prog :-
propagate word(A,B,C,D),
/* Note the shared letter B  */
```

```
propagate word(E,F,B,H),
...,
```

The program just comprises a set of propagation constraints. (There is no need for a labelling since Propia itself selects a propagation constraint for resolution when the propagation sequence terminates.) Immediately certain letters are instantiated by the original propagation. Subsequently, each time some letters are instantiated after selecting a *word* goal for resolution, the affected propagation constraints are re-executed in the hope of instantiating further letters.

The crossword compilation problem has also been addressed using CLP by Van Hentenryck [Van Hentenryck, 1989]. Generalised propagation yields a performance improvement of about 15 times on Van Hentenryck's example. However much more significant is the power of generalised propagation for solving large problems. Van Hentenryck's example uses a lexicon which contained precisely the 150 words needed to compile the crossword. With generalised propagation it is possible to compile crosswords from a 25000 word lexicon. It is interesting to note that generalised propagation automatically yields a similar algorithm for generating crosswords as that developed for specialised crossword puzzle generating programs [Berghel, 1987].

A further way to control the evaluation of the crossword puzzle example is to divide the *word* goals into clusters, reflecting connected subareas of the crossword grid. A predicate *cluster* can be defined which combines all the words in a cluster:

```
cluster(A,B,C,D,E,F) :-
    word(A,B,C),word(A,D,F),word(C,E,F).
```

Generalised propagation can then be applied to the whole cluster:

```
propagate cluster(A,B,C,D,E,F)
```

In general propagation on *cluster* yields strictly more information than propagation on each of the *word* goals individually. However the amount of computing required to perform the propagation on *cluster* is also likely to be greater than propagating on the *word* goals individually.

If propagation is applied to larger subproblems, then we term it more "global". Global propagation is more expensive than local propagation but the amount of pruning of the search tree that results can be very significant.

4.3 Topological Branch and Bound

Generalised propagation is based on the idea of finding all answers to a query and eliciting the most specific generalisation. However it much more efficient to alternate the finding of answers and calculating the most specific generalisation. We call this "topological branch and bound".

For example after finding two words which satisfy a *word* goal in the crossword example, the system immediately attempts to "generalise" by finding common letters within and between words. If there are no common letters, the propagation process ceases immediately. Only if there are common letters does the system now search for a third word. As a result, the system very rarely needs to find more than a few answers to any *word* goal during propagation. This is the reason that the program has such an excellent runtime, even with a dictionary of 25000 words compiling real crosswords in a minute. It also accounts for Propia's good performance on the propositional satisfiability benchmarks despite its recalculating at runtime propagation information which in the CHIP program was hard coded by the programmer using demons.

Further optimisations can be applied if the predicate being used for propagation is defined by rules instead of facts. The exploration of a new branch in the search tree incrementally builds a new set of equalities. If, when exploring a branch, the partial set of equalities becomes larger than the current most specific generalisation, then the search on this branch can be stopped. This means that propagation can terminate even when the actual search tree is infinite. For example given the definition

```
p(s(0)).
p(s(X)) :- p(X).
```

propagation on $p(X)$ terminates after finding two solutions yielding the constraint $X = s(\_)$.

5 Conclusion

Constraint logic programming systems offer a range of tools for writing simple and efficient programs over various computation domains. Unfortunately it is not always possible to use different tools together. For example classical propagation cannot be used in programs working on domains such as Prolog III's trees.

A second drawback is that the logic of the program, when efficiency considerations are taken into account, has to be transformed extensively, or parts of it replaced altogether with rules expressed in some reactive language such as demons. The result for non-toy programs is a loss of clarity and, possibly, efficiency. If the programmer is not extremely competent these problems compound themselves, too often yielding a result which is not only inefficient but incorrect.

Generalised propagation makes a contribution to both problems. Firstly propagation can be used for arbitrary domains of computation, thereby improving orthogonality. Secondly the propagation annotations keep the control very simple and quite separate from the program logic, thereby preserving clarity and correctness.

Current experiments show generalised propagation to be a powerful and flexible tool for expressing control. More global propagation is more costly but it can bring a drastic reduction of the search tree. Local propagation is a cheap solution which is much easier to program and debug than guarded clauses or demons.

We are continuing to investigate the effectiveness of generalised propagation on a range of applications, studying its practical applicability to other computation domains, and following up the study of its lattice theoretic basis.

6 Acknowledgements

This work was funded by the Esprit 2 project, no.5291 *CHIC*. Thanks also to Bull, ICL and Siemens for providing a wonderful working environment at ECRC.

References

[Berghel, 1987] H. Berghel. Crossword compilation with Horn clauses. *The Computer Journal*, 30(2):183–188, 1987.

[Codognet and Sola, 1990] P. Codognet and T. Sola. Extending the WAM for intelligent backtracking. In *Proc. 8th International Conference on Logic Programming*. MIT Pres, 1990.

[Davis, 1987] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[Dincbas *et al.*, 1988] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, November 1988.

[Dincbas *et al.*, 1990] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:74–94, 1990.

[Golomb and Baumert, 1965] S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 1965.

[Haralick and Elliot, 1980] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, October 1980.

[Jaffar and Lassez, 1987] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages (POPL'87)*, Munich, FRG, January 1987.

[Le Provost and Wallace, 1992] T. Le Provost and M. Wallace. Generalised propagation. Technical Report ECRC-92-1, ECRC, Munich, 1992.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Mitterreiter and Radermacher, 1991] I. Mitterreiter and F. J. Radermacher. Experiments on the running time behaviour of some algorithms solving propositional calculus problems. Technical Report Draft, FAW, Ulm, 1991.

[Montanari, 1974] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[Sussman and Steele, 1980] G.J. Sussman and G.L. Steele. CONSTRAINTS: A language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, January 1980.

[Van Hentenryck and Dincbas, 1986] P. Van Hentenryck and M. Dincbas. Domains in logic programming. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86)*, Philadelphia, PA, August 1986.

[Van Hentenryck, 1989] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. The MIT Press, 1989.

[Warren, 1988] D.H.D. Warren. The andorra model. Presented at the Gigalips Workshop, Univ. of Manchester, 1988.

[Wolfram, 1989] D.A. Wolfram. Forward checking and intelligent backtracking. *Information Processing Letters*, 32(2):85–87, July 1989.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Feature-based Constraint System for Logic Programming with Entailment

Hassan Aït-Kaci* Andreas Podelski* Gert Smolka†

Abstract

This paper presents the constraint system FT, which we feel is an intriguing alternative to Herbrand both theoretically and practically. As does Herbrand, FT provides a universal data structure based on trees. However, the trees of FT (called feature trees) are more general than the trees of Herbrand (called constructor trees), and the constraints of FT are finer grained and of different expressivity. The basic notion of FT are functional attributes called features, which provide for record-like descriptions of data avoiding the overspecification intrinsic in Herbrand's constructor-based descriptions. The feature tree structure fixes an algebraic semantics for FT. We will also establish a logical semantics, which is given by three axiom schemes fixing the first-order theory FT.

FT is a constraint system for logic programming, providing a test for unsatisfiability, and a test for entailment between constraints, which is needed for advanced control mechanisms.

The two major technical contributions of this paper are (1) an incremental entailment simplification system that is proved to be sound and complete, and (2) a proof showing that FT satisfies the so-called "independence of negative constraints".

1 Introduction

An important structural property of many logic programming systems is the fact that they factorize into a constraint system and an extension facility. Colmerauer's Prolog II [8] is an early language design making explicit use of this property. CLP (Constraint Logic Programming [10]), ALPS [16], CCP (Concurrent Constraint Programming [21]), and KAP (Kernel Andorra Prolog [9]) are recent logic programming frameworks that exploit this property to its full extent by being parameterized with respect to an abstract class of constraint systems. The basic operation these frameworks require of a constraint system is a test for unsatisfiability. ALPS, CCP, and KAP in addition require a test for entailment between constraints, which is needed for advanced control mechanisms such as delaying, coroutining, synchronisation, committed choice, and deep constraint propagation. Given this situation, constraint systems are a central issue in research on logic programming.

The constraint systems of most existing logic programming languages are variations and extensions of Herbrand [14], the constraint system underlying Prolog. The individuals of Herbrand are trees corresponding to ground terms, and the atomic constraints are equations between terms. Seen from the perspective of programming, Herbrand provides a universal data structure as a logical system.

This paper presents a constraint system FT, which we feel is an intriguing alternative to Herbrand both theoretically and practically. As does Herbrand, FT provides a universal data structure based on trees. However, the trees of FT (called feature trees) are more general than the trees of Herbrand (called constructor trees), and the constraints of FT are finer grained and of different expressivity. The basic notion of FT are functional attributes called features, which provide for record-like descriptions of data avoiding the overspecification intrinsic in Herbrand's constructor-based descriptions. For the special case of constructor trees, features amount to argument selectors for constructors.

Suppose we want to say that x is a wine whose grape is riesling and whose color is white. To do this in Herbrand, one may write the equation

$$x = \mathsf{wine}(\mathsf{riesling}, \mathsf{white}, y_1, \ldots, y_n)$$

with the implicit assumption that the first argument of the constructor wine carries the "feature" grape, the second argument carries the "feature" color, and the remaining arguments $y_1, \ldots, y_n$ carry the remaining "features" of the chosen representation of wines. The obvious difficulty with this description is that it says more than we want to say, namely, that the constructor wine has $n+2$ arguments and that the "features" grape and color are represented as the first and the second argument.

The constraint system FT avoids this overspecification by allowing the description

$$x\colon \mathsf{wine}[\mathsf{grape} \Rightarrow \mathsf{riesling},\ \mathsf{color} \Rightarrow \mathsf{white}] \qquad (1)$$

*Digital Equipment Corporation, Paris Research Laboratory (PRL), 85 avenue Victor Hugo, 92500 Rueil-Malmaison, France (email: {hak,podelski}@prl.dec.com).

†German Research Center for Artificial Intelligence (DFKI) and Universität des Saarlandes, Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany (email: smolka@dfki.uni-sb.de). Supported in part by the Bundesminister für Forschung und Technologie under contract ITW 9105.

saying that x has sort wine, its feature grape is riesling, and its feature color is white. Nothing is said about other features of x, which may or may not exist.

The individuals of FT are so-called feature trees, examples of which are shown in Figure 1. A feature tree is a possibly infinite tree whose nodes are labeled with symbols called sorts, and whose edges are labeled with symbols called features. The labeling with features is deterministic in that all edges departing from a node must be labeled with distinct features. Thus, every direct subtree of a feature tree can be identified by the feature labeling the edge leading to it. The constructor trees of Herbrand can be represented as feature trees whose edges are labeled with natural numbers indicating the corresponding argument positions.

All but the second and third feature tree in Figure 1 satisfy the description (1).

The constraints of FT are ordinary first-order formulae taken over a signature that accommodates sorts as unary and features as binary predicates. Thus the description (1) is actually syntactic sugar for the formula

$$\mathsf{wine(x)} \wedge \exists y(grape(x,y) \wedge riesling(y)) \wedge \exists y(color(x,y) \wedge white(y)).$$

The set of all rational feature trees is made into a corresponding logical structure $\mathcal{T}$ by letting $A(x)$ hold iff the root of x is labeled with the sort A, and letting $f(x,y)$ hold iff x has y as direct subtree via the feature f. The feature tree structure $\mathcal{T}$ fixes an algebraic semantics for FT.

We will also establish a logical semantics, which is given by three axiom schemes fixing a first-order theory FT. Backofen and Smolka [6] show that $\mathcal{T}$ is a model of FT and that FT is in fact a complete theory, which means that FT is exactly the theory induced by $\mathcal{T}$. However, we will not use the completeness result in the present paper, but show explicitly that entailment with respect to $\mathcal{T}$ is the same as entailment with respect to FT.

The two major technical contributions of this paper are (1) an incremental entailment simplification system that is proved to be sound and complete, and (2) a proof showing that FT satisfies the so-called "independence of negative constraints" [7, 14, 15]. The incremental entailment simplification system is the prerequisite for FT's use with either of the constraint programming frameworks ALPS, CCP or KAP mentioned at the beginning of this section. The indepence property means among other things that negative constraints can essentially be handled through entailment simplification.

One origin of FT is Aït-Kaci's ψ-term calculus [1], which is at the heart of the programming language LOGIN [3] and further extended in the language LIFE [5] with functions over feature structures thanks to a generalization of the concept of residuation of Le Fun [4]. Other precursors of FT are the feature descriptions found in so-called unification grammars [13, 12] developed for natural language processing, and also the formalisms of Mukai [17, 18]. These early feature structure formalism were presented in a nonlogical form. Major steps in the process of their understanding and logical reformulation are the articles [20, 23, 11, 22]. Feature trees, the feature tree structure $\mathcal{T}$, and the axiomatization of $\mathcal{T}$ were first given in [6].

The paper is organized as follows. Section 2 defines the basic notions and discusses the differences in expressivity between Herbrand and FT. Section 3 gives a basic simplification system that decides satisfiability of positive constraints. Section 4 is not committed to FT but discusses the notion of incremental entailment checking and its connection with the indepence property and negation. Section 5 gives the entailment simplification system, proves it sound, complete and terminating, and also proves that FT satisfies the independence property.

2 Feature Trees and Constraints

To give a rigorous formalization of feature trees, we first fix two disjoint alphabets $\mathcal{S}$ and $\mathcal{F}$, whose symbols are called *sorts* and *features*, respectively. The letters A, B, C will always denote sorts, and the letters f, g, h will always denote features. Words over $\mathcal{F}$ are called paths. The concatenation of two paths v and w results in the path vw. The symbol ε denotes the empty path, $v\varepsilon = \varepsilon v = v$, and $\mathcal{F}^\star$ denotes the set of all paths.

A *tree domain* is a nonempty set $D \subseteq \mathcal{F}^\star$ that is prefix-closed, that is, if $vw \in D$, then $v \in D$. Thus, it always contains the empty path.

A *feature tree* is a mapping $t : D \to \mathcal{S}$ from a tree domain D into the set of sorts. The paths in the domain of a feature tree represent the nodes of the tree; the empty path represents its root. The letters s and t are used denote feature trees.

If convenient, we consider a feature tree t as a relation, *i.e.*, $t \subseteq \mathcal{F}^\star \times \mathcal{S}$, and write $(w, A) \in t$ instead of $t(w) = A$. As relations, *i.e.*, as subsets of $\mathcal{F}^\star \times \mathcal{S}$, feature trees are partially ordered by set inclusion. We say that s is *smaller than* t if $s \subseteq t$.

The *subtree* wt of a feature tree t at one of its nodes w is the feature tree defined by (as a relation):

$$wt := \{(v, A) \mid (wv, A) \in t\}.$$

If D is the domain of t, then the domain of wt is the set $w^{-1}D = \{v \mid wv \in D\}$. Thus, wt is given as the mapping $wt : w^{-1}D \to \mathcal{S}$ defined on its domain by $wt(v) = t(wv)$. A feature tree s is called a *subtree* of a feature tree t if it is a subtree $s = wt$ at one of its nodes w, and a direct subtree if $w \in \mathcal{F}$.

A feature tree t with domain D is called *rational* if (1) t has only finitely many subtrees and (2) t is finitely branching, which is: for every $w \in D$, $w\mathcal{F} \cap D = \{wf \in D \mid f \in \mathcal{F}\}$ is finite. Assuming (1), (2) is equivalent to

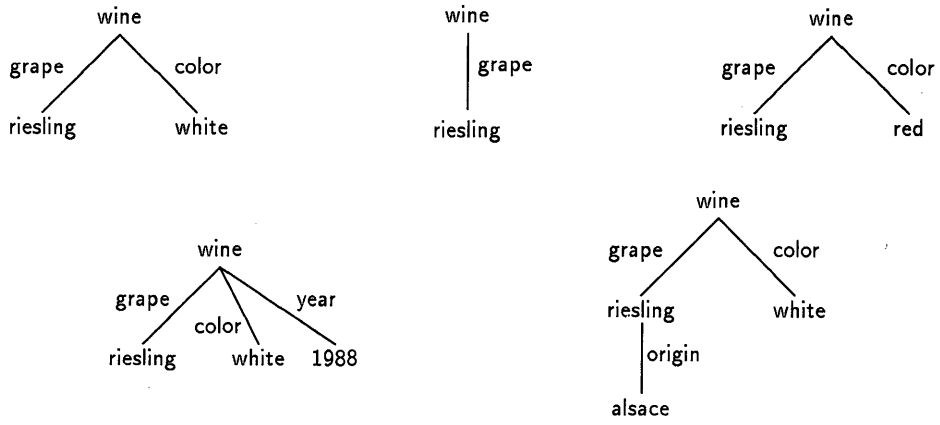


Figure 1: Examples of Feature Trees.

saying that there exist finitely many features $f_1, \ldots, f_n$ such that $D \subseteq \{f_1, \ldots, f_n\}^\star$.

Constraints over feature trees will be defined as first-order formulae. We first fix a first-order signature $\mathcal{S} \uplus \mathcal{F}$ by taking sorts as unary and features as binary relation symbols. Moreover, we fix an infinite alphabet of *variables* and adopt the convention that x, y, z always denote variables. Under this signature, every term is a variable and an *atomic formula* is either a feature constraint xfy ($f(x,y)$ in standard notation), a sort constraints Ax ($A(x)$ in standard notation), an equation $x \doteq y$, $\bot$ ("false"), or $\top$ ("true"). Compound formulae are obtained as usual by the connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$ and the quantifiers $\exists$ and $\forall$. We use $\tilde{\exists}\phi$ and $\tilde{\forall}\phi$ to denote the existential and universal closure of a formula ϕ, respectively. Moreover, $\mathcal{V}(\phi)$ is taken to denote the set of all variables that occur free in a formula ϕ. The letters ϕ and ψ will always denote formulae. In the following we won't make a distinction between formulae and constraints, that is, a *constraint* is a formula as defined above.

$\mathcal{S} \uplus \mathcal{F}$-structures and *validity* of formulae in $\mathcal{S} \uplus \mathcal{F}$-structures are defined as usual. Since we consider only $\mathcal{S} \uplus \mathcal{F}$-structures in the following, we will simply speak of structures. A *theory* is a set of closed formulae. A *model* of a theory is a structure that satisfies every formulae of the theory. A formula ϕ is a *consequence* of a theory T ($T \models \phi$) if $\tilde{\forall}\phi$ is valid in every model of T. A formula ϕ is *satisfiable* in a structure $\mathcal{A}$ if $\tilde{\exists}\phi$ is valid in $\mathcal{A}$. Two formulae ϕ, ψ are *equivalent* in a structure $\mathcal{A}$ if $\tilde{\forall}(\phi \leftrightarrow \psi)$ is valid in $\mathcal{A}$. We say that a formula ϕ *entails* a formula ψ in a structure $\mathcal{A}$ [theory T] and write $\phi \models_{\mathcal{A}} \psi$ [$\phi \models_T \psi$] if $\tilde{\forall}(\phi \rightarrow \psi)$ is valid in $\mathcal{A}$ [is a consequence of T]. A theory T is *complete* if for every closed formula ϕ either ϕ or $\neg\phi$ is a consequence of T.

The *feature tree structure* $\mathcal{T}$ is the $\mathcal{S} \uplus \mathcal{F}$-structure defined as follows:

- the domain of $\mathcal{T}$ is the set of all rational feature trees;
- $t \in A^{\mathcal{T}}$ iff $t(\varepsilon) = A$ (t's root is labeled with A);
- $(s,t) \in f^{\mathcal{T}}$ iff $f \in D_s$ and $t = fs$ (t is the subtree of s at f).

Next we discuss the expressivity of our constraints with respect to feature trees (that is, with respect to the feature tree structure $\mathcal{T}$) by means of examples. The constraint

$$\neg\exists y(xfy)$$

says that x has no subtree at f, that is, that there is no edge departing from x's root that is labeled with f. To say that x has subtree y at path $f_1 \cdots f_n$, we can use the constraint

$$\exists z_1 \cdots \exists z_{n-1}(xf_1z_1 \wedge z_1f_2z_2 \wedge \ldots \wedge z_{n-1}f_ny).$$

Now let's look at statements we cannot express (more precisely, statements of whom the authors believe they cannot be expressed). One simple unexpressible statement is "y is a subtree of x" (that is, "$\exists w$: $y = wx$"). Moreover, we cannot express that x is smaller than y. Finally, if we assume that the alphabet $\mathcal{F}$ of features is infinite, we cannot say that x has subtrees at features $f_1, \ldots, f_n$ but no subtree at any other feature. In particular, we then cannot say that x is a primitive feature tree, that is, has no proper subtree.

The theory FT_0 is given by the following two axiom schemes:

(Ax1) $\forall x\, \forall y\, \forall z\, (xfy \wedge xfz \rightarrow y \doteq z)$
(for every feature f)

(Ax2) $\forall x\, (Ax \wedge Bx \rightarrow \bot)$
(for every two distinct sorts A and B).

The first axiom scheme says that features are functional and the second scheme says that sorts are mutually disjoint. Clearly, $\mathcal{T}$ is a model of FT_0. Moreover, FT_0 is

incomplete (for instance, $\exists x(Ax)$ is valid in $\mathcal{T}$ but invalid in other models of FT_0). We will see in the next section that FT_0 plays an important role with respect to basic constraint simplification.

Next we introduce some additional notation needed in the rest of the paper. This notation will also allow us to state a third axiom scheme that, as shown in [6], extends FT_0 to a complete axiomatization of $\mathcal{T}$.

Throughout the paper we assume that the conjunction of formulae is an associative and commutative operator that has $\top$ as neutral element. This means that we identify $\phi \wedge (\psi \wedge \theta)$ with $\theta \wedge (\psi \wedge \phi)$, and $\phi \wedge \top$ with ϕ (but not, for example, $xfy \wedge xfy$ with xfy). A conjunction of atomic formulae can thus be seen as the finite multiset of these formulae, where conjunction is multiset union, and $\top$ (the "empty conjunction") is the empty multiset. We will write $\psi \subseteq \phi$ (or $\psi \in \phi$, if ψ is an atomic formula) if there exists a formula ψ' such that $\psi \wedge \psi' = \phi$.

We will use an additional atomic formula $xf\uparrow$ ("f undefined on x") that is taken to be equivalent to $\neg\exists y\,(xfy)$, for some variable y (other than x).

Only for the formulation of the third axiom we introduce the notion of a *solved-clause*, which is either $\top$ or a conjunction ϕ of atomic formulae of the form xfy, Ax or $xf\uparrow$ such that the following conditions are satisfied:

1. if $Ax \in \phi$ and $Bx \in \phi$, then $A = B$;
2. if $xfy \in \phi$ and $xfz \in \phi$, then $y = z$;
3. if $xfy \in \phi$, then $xf\uparrow \notin \phi$.

Given a solved-clause ϕ, we say that a variable x is *dependent* in ϕ if ϕ contains a constraint of the form Ax, xfy or $xf\uparrow$, and use $\mathcal{DV}(\phi)$ to denote the set of all variables that are dependent in ϕ.

The theory FT is obtained from FT_0 by adding the axiom scheme:

(Ax3) $\tilde{\forall}\exists X\phi$
(for every solved-clause ϕ and $X = \mathcal{DV}(\phi)$).

Theorem 2.1 *The feature tree structure $\mathcal{T}$ is a model of the theory FT.*

Proof. We will only show that FT is a model of the third axiom. Let X be the set of dependent variables of the solved-clause ϕ, $X = \mathcal{DV}(\phi)$. Let α be any $\mathcal{T}$-valuation defined on $\mathcal{V}(\phi) - X$; we write the tree $\alpha(y)$ as t_y. We will extend α on X such that $\mathcal{T}, \alpha \models \phi$.

Given $x \in X$, we define the "punctual" tree $t_x = \{(\varepsilon, A)\}$, where $A \in \mathcal{S}$ is the sort such that $Ax \in \phi$, if it exists, and arbitrary, otherwise. Now we are going to use the notion of *tree sum* of Nivat [19], where $w^{-1}t = \{(wv, A) \mid (v, A) \in t\}$ ("the tree t translated by w"), and we define:

$$\alpha(x) = \biguplus\{w^{-1}t_y \mid x \overset{w}{\rightsquigarrow} y \text{ for some } y \in \mathcal{V}(\phi),\ w \in \mathcal{F}^\star\}.$$

Here the "leads-to" relation $\overset{w}{\rightsquigarrow}$ is given by: $x \overset{\varepsilon}{\rightsquigarrow} x$, and $x \overset{wf}{\rightsquigarrow} y$ if $x \overset{w}{\rightsquigarrow} y'$ and $y'fy \in \phi$, for some $y' \in \mathcal{V}(\phi)$ and some $f \in \mathcal{F}$. Since

$$\alpha(x) = \bigcup\{w^{-1}\alpha(y) \mid \dots\}$$

and, for a node w of $\alpha(x)$, $w\alpha(x) = \alpha(y)$, it follows that $\alpha(x)$ is a rational tree and that $\mathcal{T}, \alpha \models \phi$.

□

3 Basic Simplification

A *basic constraint* is either $\bot$ or a possibly empty conjunction of atomic formulae of the form Ax, xfy, and $x \doteq y$. The following five *basic simplification* rules constitute a simplification system for basic constraints, which, as we will see, decides whether a basic constraint is satisfiable in $\mathcal{T}$.

1. $\dfrac{xfy \wedge xfz \wedge \phi}{xfz \wedge y \doteq z \wedge \phi}$

2. $\dfrac{Ax \wedge Bx \wedge \phi}{\bot}$ $\quad A \neq B$

3. $\dfrac{Ax \wedge Ax \wedge \phi}{Ax \wedge \phi}$

4. $\dfrac{x \doteq y \wedge \phi}{x \doteq y \wedge \phi[x \leftarrow y]}$ $\quad x \in \mathcal{V}(\phi)$ and $x \neq y$

5. $\dfrac{x \doteq x \wedge \phi}{\phi}$

The notation $\phi[x \leftarrow y]$ is used to denote the formula that is obtained from ϕ by replacing every occurrence of x with y. We say that a constraint ϕ *simplifies to* a constraint ψ by a simplification rule ρ if $\frac{\phi}{\psi}$ is an instance of ρ. We say that a constraint ϕ *simplifies to* a constraint ψ if either $\phi = \psi$ or ϕ simplifies to ψ in finitely many steps each licensed by one of the five simplification rules given above.

Example 3.1 We have the following basic simplification chain, leading to a solved constraint:

$$\begin{aligned}
& xfu \wedge yfv \wedge Au \wedge Av \wedge z \doteq x \wedge y \doteq z \\
\Rightarrow\ & xfu \wedge yfv \wedge Au \wedge Av \wedge z \doteq x \wedge y \doteq x \\
\Rightarrow\ & xfu \wedge xfv \wedge Au \wedge Av \wedge\ z \doteq x \wedge y \doteq x \\
\Rightarrow\ & xfv \wedge Au \wedge Av \wedge\ u \doteq v \wedge z \doteq x \wedge y \doteq x \\
\Rightarrow\ & xfv \wedge Av \wedge Av \wedge\ u \doteq v \wedge z \doteq x \wedge y \doteq x \\
\Rightarrow\ & xfv \wedge Av \wedge\ u \doteq v \wedge z \doteq x \wedge y \doteq x
\end{aligned}$$

Using the same steps up to the last one, the constraint $xfu \wedge yfv \wedge Au \wedge Bv \wedge z \doteq x \wedge y \doteq z$ simplifies to $\bot$ (in the last step, Rule 2 instead of Rule 3 is applied). □

Proposition 3.2 *If the basic constraint ϕ simplifies to ψ, then $FT_0 \models \phi \leftrightarrow \psi$.*

Proof. The rules 3, 4 and 5 perform equivalence transformations with respect to every structure. The rules 1 and 2 correspond exactly to the two axiom schemes of FT_0 and perform equivalence transformations with respect to every model of FT_0. □

We say that a basic constraint ϕ *binds* a variable x to y if $x \doteq y \in \phi$ and x occurs only once in ϕ. At this point it is important to note that we consider equations as ordered, that is, assume that $x \doteq y$ is different from $y \doteq x$ if $x \neq y$. We say that a variable x is *eliminated*, or *bound by* ϕ, if ϕ binds x to some variable y.

Proposition 3.3 *The basic simplification rules are terminating.*

Proof. First observe that the simplification rules don't add new variables and preserve eliminated variables. Furthermore, rule 4 increases the number of eliminated variables by one. Hence we know that if an infinite simplification chain exists, we can assume without loss of generality that it only employs the rules 1, 3 and 5. Since rule 1 decreases the number of feature constraints "xfy", which is not increased by rules 3 and 5, we know that if an infinite simplification chain exists, we can assume without loss of generality that it only employs the rules 3 and 5. Since this is clearly impossible, an infinite simplification chain cannot exist. □

A basic constraint is called *normal* if none of the five simplification rules applies to it. A constraint ψ is called a *normal form* of a basic constraint ϕ if ϕ can be simplified to ψ and ψ is normal. A *solved constraint* is a normal constraint that is different from $\bot$.

So far we know that we can compute for any basic constraint ϕ a normal form ψ by applying the simplification rules as long as they are applicable. Although the normal form ψ may not be unique for ϕ, we know that ϕ and ψ are equivalent in every model of FT_0. It remains to show that every solved constraint is satisfiable in $\mathcal{T}$.

Every basic constraint ϕ has a unique decomposition $\phi = \phi_N \wedge \phi_G$ such that ϕ_N is a possibly empty conjunction of equations "$x \doteq y$" and and ϕ_G is a possibly empty conjunction of feature constraints "xfy" and sort constraints "Ax". We call ϕ_N the *normalizer* and and ϕ_G the *graph* of ϕ.

Proposition 3.4 *A basic constraint $\phi \neq \bot$ is solved iff the following conditions hold:*

1. *an equation $x \doteq y$ appears in ϕ only if x is eliminated in ϕ;*
2. *the graph of ϕ is a solved clause;*
3. *no primitive constraint appears more than once in ϕ.*

Proposition 3.5 *Every solved constraint is satisfiable in every model of FT.*

Proof. Let ϕ be a solved constraint and $\mathcal{A}$ be a model of FT. Then we know by axiom scheme *Ax3* that the graph ϕ_G of a solved constraint ϕ is satisfiable in an FT-model $\mathcal{A}$. A variable valuation α into $\mathcal{A}$ such that $\mathcal{A}, \alpha \models \phi_G$ can be extended on all eliminated variables simply by $\alpha(x) = \alpha(y)$ if $x \doteq y \in \phi$, such that $\mathcal{A}, \alpha \models \phi$. □

Theorem 3.6 *Let ψ be a normal form of a basic constraint ϕ. Then ϕ is satisfiable in $\mathcal{T}$ if and only if $\psi \neq \bot$.*

Proof. Since ϕ and ψ are equivalent in every model of FT_0 and $\mathcal{T}$ is a model of FT_0, it suffices to show that ψ is satisfiable in $\mathcal{T}$ if and only if $\psi \neq \bot$. To show the nontrivial direction, suppose $\psi \neq \bot$. Then ψ is solved and we know by the preceding proposition that ψ is satisfiable in every model of FT. Since $\mathcal{T}$ is a model of FT, we know that ψ is satisfiable in $\mathcal{T}$. □

Theorem 3.7 *For every basic constraint ϕ the following statements are equivalent:*

$$\mathcal{T} \models \tilde{\exists}\phi \;\Leftrightarrow\; \exists \text{ model } \mathcal{A} \text{ of } FT_0\colon \mathcal{A} \models \tilde{\exists}\phi \;\Leftrightarrow\; FT \models \tilde{\exists}\phi.$$

Proof. The implication $1 \Rightarrow 2$ holds since $\mathcal{T}$ is a model of FT_0. The implication $3 \Rightarrow 1$ follows from the fact that $\mathcal{T}$ is a model of FT. It remains to show that $2 \Rightarrow 3$.

Let ϕ be satisfiable in some model of FT_0. Then we can apply the simplification rules to ϕ and compute a normal form ψ such that ϕ and ψ are equivalent in every model of FT_0. Hence ψ is satisfiable in some model of FT_0. Thus $\psi \neq \bot$, which means that ψ is solved. Hence we know by the preceding proposition that ψ is satisfiable in every model of FT. Since ϕ and ψ are equivalent in every model of $FT_0 \subseteq FT$, we have that ϕ is satisfiable in every model of FT. □

4 Entailment, Independence and Negation

In this section we discuss some general properties of constraint entailment. This prepares the ground for the next section, which is concerned with entailment simplification in the feature tree constraint system.

Throughout this section we assume that $\mathcal{A}$ is a structure, γ and ϕ are formulae that can be interpreted in $\mathcal{A}$, and that X is a finite set of variables.

We say that γ *disentails* ϕ in $\mathcal{A}$ if γ entails $\neg\phi$ in $\mathcal{A}$. If γ is satisfiable in $\mathcal{A}$, then γ cannot both entail and disentail $\exists X\phi$ in $\mathcal{A}$. We say that γ *determines* ϕ in $\mathcal{A}$ if γ either entails or disentails ϕ in $\mathcal{A}$.

Given γ, ϕ and X, we want to determine in an *incremental* manner whether γ entails or disentails $\exists X\phi$. Typically, γ will *not* determine $\exists X\phi$ when $\exists X\phi$ is considered first, but this may change when γ is strengthened to $\gamma \wedge \gamma'$. The basic idea leading to an incremental entailment checker is to simplify ϕ with respect to the *context* γ and the *local variables* X. Given γ, X and ϕ, simplification must yield a formula ψ such that

$$\gamma \models_{\mathcal{A}} \exists X\phi \leftrightarrow \exists X\psi.$$

The following facts provide some evidence that this is the right invariant for entailment simplification.

Proposition 4.1 *Let* $\gamma \models_{\mathcal{A}} \exists X\phi \leftrightarrow \exists X\psi$. *Then:*

1. $\gamma \models_{\mathcal{A}} \exists X\phi$ *iff* $\gamma \models_{\mathcal{A}} \exists X\psi$;
2. $\gamma \models_{\mathcal{A}} \neg\exists X\phi$ *iff* $\gamma \models_{\mathcal{A}} \neg\exists X\psi$;
3. *if* $\psi = \bot$, *then* $\gamma \models_{\mathcal{A}} \neg\exists X\phi$;
4. *if* $\exists X\psi$ *is valid in* $\mathcal{A}$, *then* $\gamma \models_{\mathcal{A}} \exists X\phi$.

Statements 1 and 2 say that it doesn't matter whether entailment and disentailment are decided for ϕ or ψ. Statement 3 gives a local condition for disentailment, and Statement 4 gives a local condition for entailment. The entailment simplification system for feature trees given in the next section will in fact decide entailment and disentailment by simplifying such that the condition of Statement 4 is met in the case of entailment, and that the condition of Statement 3 is met in the case of disentailment.

In practice, one can ensure by variable renaming that no variable of X occurs in γ. The next fact says that then it suffices if entailment simplification respects the more convenient invariant

$$\mathcal{A} \models \gamma \wedge \phi \leftrightarrow \gamma \wedge \psi.$$

This is the invariant respected by our system (*cf.* Proposition 5.4).

Proposition 4.2 *Let* $X \cap \mathcal{V}(\gamma) = \emptyset$. *Then:*

1. *if* $\mathcal{A} \models \gamma \wedge \phi \leftrightarrow \gamma \wedge \psi$, *then* $\gamma \models_{\mathcal{A}} \exists X\phi \leftrightarrow \exists X\psi$;
2. $\gamma \models_{\mathcal{A}} \neg\exists X\phi$ *iff* $\gamma \wedge \phi$ *is unsatisfiable in* $\mathcal{A}$.

That is, the conjunction $\gamma \wedge \phi$ is satisfiable if and only if γ either entails $\exists X\phi$, or it does not determine $\exists X\phi$.

The so-called independence of negative constraints [7, 14, 15] is an important property of constraint systems. If it holds, simplification of conjunctions of positive and negative constraints can be reduced to entailment simplification of conjunctions of positive constraints.

To define the independence property, we assume that a constraint system is a pair consisting of a structure $\mathcal{A}$ and a set of so-called basic constraints. From basic constraints one can build more complex constraints using the connectives and quantifiers of predicate logic. We say that a constraint system satisfies the *independence property* if

$$\gamma \models_{\mathcal{A}} \exists X_1\phi_1 \vee \ldots \vee \exists X_n\phi_n \quad \textit{iff} \quad \exists i\colon \gamma \models_{\mathcal{A}} \exists X_i\phi_i$$

for all basic constraints γ, $\phi_1, \ldots, \phi_n$ and all finite sets of variables $X_1, \ldots, X_n$.

Proposition 4.3 *If a constraint system satisfies the independence property, then the following statements hold (γ, ϕ and $\phi_1, \ldots, \phi_n$ are basic constraints):*

1. $\gamma \wedge \neg\exists X_1\phi_1 \wedge \ldots \wedge \neg\exists X_n\phi_n$ *unsatisfiable in* $\mathcal{A}$ *iff* $\exists i\colon \gamma \models_{\mathcal{A}} \exists X_i\phi_i$;
2. *if* $\gamma \wedge \neg\exists X_1\phi_1 \wedge \ldots \wedge \neg\exists X_n\phi_n$ *is satisfiable in* $\mathcal{A}$, *then* $\gamma \wedge \neg\exists X_1\phi_1 \wedge \ldots \wedge \neg\exists X_n\phi_n \models_{\mathcal{A}} \exists X\phi$ *iff* $\gamma \models_{\mathcal{A}} \exists X\phi$.

5 Entailment Simplification

We now return to the feature tree constraint system. Throughout this section we assume that γ is a solved constraint and X is a finite set of variables not occurring in γ. We will call γ the *context*, the variables in X *local*, and all other variables *global*.

If T is a theory and ϕ and ψ are possibly open formulae, we write $\phi \models_T \psi$ (read: ϕ entails ψ in T) if $\bar{\forall}(\phi \rightarrow \psi)$ is valid in T.

Theorem 5.1 *For every basic constraint ϕ, the following equivalences hold:*

$$\gamma \models_{\mathcal{T}} \neg\exists X\phi \quad \textit{iff} \quad \gamma \models_{FT_0} \neg\exists X\phi \quad \textit{iff} \quad \gamma \models_{FT} \neg\exists X\phi.$$

Proof. Implication "2 $\Rightarrow$ 3" holds since $FT_0 \subseteq FT$. Implication "3 $\Rightarrow$ 1" holds since $\mathcal{T}$ is a model of FT. To show implication "1 $\Rightarrow$ 2", suppose $\gamma \models_{\mathcal{T}} \neg\exists X\phi$. Then we know by Proposition 4.2 that $\gamma \wedge \phi$ is unsatisfiable in $\mathcal{T}$. Thus we know by Theorem 3.7 that $\gamma \wedge \phi$ is unsatisfiable in every model of FT_0. Hence we know by Proposition 4.2 that $\gamma \models_{FT_0} \neg\exists X\phi$. □

For every basic constraint ϕ and every variable x we define

$$\phi x := \begin{cases} y & \text{if } x \doteq y \in \phi \text{ and } x \text{ is eliminated;} \\ x & \text{otherwise.} \end{cases}$$

A basic constraint ϕ is *X-oriented* if $x \doteq y \in \phi$ always implies $x \in X$ or $y \notin X$. A basic constraint ϕ is *pivoted* if $x \doteq y \in \phi$ implies that x is eliminated in ϕ (and then y is a "pivot").

The following *entailment simplification rules* simplify basic constraints to basic constraints with respect to a context γ and local variables X.

1. $\dfrac{xfu \wedge \phi}{u \doteq v \wedge \phi}$ $\quad yfv \in \gamma \wedge \phi,\ \ \phi y = x$

2. $\dfrac{\phi}{\phi u \doteq \phi v \wedge \phi}$ $\quad \begin{cases} xfu \wedge yfv \subseteq \gamma, \\ \phi x = \phi y,\ \ \phi u \neq \phi v, \\ \phi\ X\text{-oriented and pivoted} \end{cases}$

3. $\dfrac{\phi}{\bot}$ $\quad Ax \wedge By \subseteq \gamma \wedge \phi,\ \ \phi x = \phi y,\ \ A \neq B$

4. $\dfrac{Ax \wedge \phi}{\phi}$ $\quad Ay \in \gamma \wedge \phi,\ \ \phi y = x$

5. $\dfrac{x \doteq y \wedge \phi}{x \doteq y \wedge \phi[x \leftarrow y]}$ $\quad \begin{cases} x \neq y,\ \ x \in \mathcal{V}(\phi), \\ (x \in X \text{ or } y \notin X) \end{cases}$

6. $\dfrac{x \doteq y \wedge \phi}{y \doteq x \wedge \phi}$ $\quad x \notin X,\ \ y \in X$

7. $\dfrac{\phi}{\phi[x \leftarrow y]}$ $\quad x \doteq y \in \gamma,\ \ x \in \mathcal{V}(\phi)$

8. $\dfrac{x \doteq x \wedge \phi}{\phi}$

We say that a basic constraint ϕ *simplifies* to a constraint ϕ *with respect to* γ *and* X if $\phi = \psi$ or ϕ simplifies to ψ in finitely many steps each licensed by one of the eight simplification rules given above. The notions of *normal* and *normal form with respect to* γ are defined accordingly.

Example 5.2 Let $\gamma = xfu \wedge yfv \wedge Au \wedge Bv$ and $X = \{z\}$. Then we have the following simplification chain with respect to γ and X:

$$\begin{array}{lll} & x \doteq z \wedge y \doteq z & \\ \Rightarrow_{\gamma,X} & z \doteq x \wedge y \doteq z & \text{by Rule E6} \\ \Rightarrow_{\gamma,X} & z \doteq x \wedge y \doteq x & \text{by Rule E5} \\ \Rightarrow_{\gamma,X} & u \doteq v \wedge z \doteq x \wedge y \doteq x & \text{by Rule E2} \\ \Rightarrow_{\gamma,X} & \bot & \text{by Rule E3.} \end{array}$$

Let us now take as context $\tilde{\gamma} = xfu \wedge yfv \wedge Au$. Then $\tilde{\phi} = u \doteq v \wedge z \doteq x \wedge y \doteq x$ is normal with respect to $\tilde{\gamma}$ and X. We shall see that this normal form tells us that $\tilde{\gamma}$ does not determine $\tilde{\phi}$. If $\tilde{\gamma}$ gets strengthened either to $\tilde{\gamma} \wedge Bv$ (as above), or to $\tilde{\gamma} \wedge x \doteq y$, then the strengthened context does determine: it disentails in the first and entails in the second case. The basic normal form of $\tilde{\gamma} \wedge x \doteq y$ is $yfu \wedge Au \wedge v \doteq u \wedge x \doteq y$; with respect to this context $\tilde{\phi}$ simplifies to $z \doteq y$. □

In the previous example, $\phi = z \doteq x \wedge y \doteq x$ simplifies to $\phi_1 = u \doteq v \wedge z \doteq x \wedge y \doteq x$ with respect to $\gamma = xfu \wedge yfv \wedge Au \wedge Bv$ and $X = \{z\}$. This corresponds to a basic simplification as follows:

$$\begin{array}{rrll} & \gamma & \wedge \phi & = \\ & xfu \wedge yfv \wedge Au \wedge Bv & \wedge z \doteq x \wedge y \doteq x & \\ \Rightarrow & xfu \wedge xfv \wedge Au \wedge Bv & \wedge z \doteq x \wedge y \doteq x & \\ \Rightarrow & xfv \wedge Au \wedge Bv & \wedge u \doteq v \wedge z \doteq x \wedge y \doteq x & \\ & = \quad \gamma' & \wedge \phi_1' & \end{array}$$

We observe that $\gamma \wedge \phi_1$ is equal to $\gamma' \wedge \phi_1'$, modulo renaming y by $\phi_1 y = x$ and u by $\phi_1 u = v$, and modulo the repetition of xfv.

Lemma 5.3 *Let ϕ simplify to ϕ_1 with respect to γ and X, not using Rule E6 (in an entailment simplification step). Then $\gamma \wedge \phi$ simplifies to some $\gamma' \wedge \phi_1'$ which is equal to $\gamma \wedge \phi_1$ up to variable renaming and repetition of conjuncts.*

Proof. Clearly, each entailment simplification rule, except for E6, corresponds directly to a basic simplification rule (namely, E1 and E2 to B1, E3 to B2, E4 to B3, E5 and E7 to B4, and E8 to B5).

If the application of the entailment simplification rule to ϕ relies on a condition of the form $\phi x = y$ or $\phi x = \phi y$ where $x \neq \phi x$ or $y \neq \phi y$, then $x \doteq \phi x \in \phi$ or $y \doteq \phi y \in \phi$, and Rule B4 is first applied to $\gamma \wedge \phi$, eliminating x by ϕx (y by ϕy).

When comparing $\gamma \wedge \phi_1$ and $\gamma' \wedge \phi_1'$, renamings take account of these variable eliminations. Note that, if the rule applied to ϕ is E2, then γ' has one feature constraint xfv less than γ — which, after renaming, has a repetition of exactly this constraint. □

Proposition 5.4 *If ϕ simplifies to ψ with respect to γ and X, then $\gamma \wedge \phi$ and $\gamma \wedge \psi$ are equivalent in every model of FT_0.*

Proof. Follows from Lemma 5.3 and Proposition 3.2. □

Proposition 5.5 *The entailment simplification rules are terminating, provided γ and X are fixed.*

Proof. First we strengthen the statement by weakening the applicability conditions $\phi y = x$ in Rules E1 and E4 to $\phi y = \phi x$. Then from Lemma 5.3 follows: (*) Each entailment simplification rule applies to ϕ_1 with respect to γ and X if and only if it applies to ϕ_1' with respect to γ' and X — except possibly for E5, when the corresponding variable has already been eliminated in an "extra" basic simplification step.

If γ' has one conjunct of the form xfu less than γ, then (*) still holds; regarding a new application of E2 this is ensured by its (therefore so complicated...) applicability condition.

With condition (*), it is possible to prove by induction on n: For every entailment simplification chain

$\phi, \phi_1, \ldots, \phi_n$ with respect to γ and X, there exists a 'basic plus Rule E6' simplification chain $\gamma \wedge \phi,\ \gamma_1 \wedge \phi'_1, \ldots, \gamma_{n+k} \wedge \phi'_{n+k}$, where $k \geq 0$ is the number of "extra" variable elimination steps. Since, according to Proposition 3.3, basic simplification chains are finite, so are entailment simplification chains. □

So far we know that we can compute for any basic constraint ϕ a normal form ψ with respect to γ and X by applying the simplification rules as long as they are applicable. Although the normal form ψ may not be unique, we know that $\gamma \wedge \phi$ and $\gamma \wedge \psi$ are equivalent in every model of FT_0.

Proposition 5.6 *For every basic constraint ϕ one can compute a normal form ψ with respect to γ and X. Every such normal form ψ satisfies: $\gamma \models_{\mathcal{T}} \exists X\phi$ iff $\gamma \models_{\mathcal{T}} \exists X\psi$, and $\gamma \models_{FT} \exists X\phi$ iff $\gamma \models_{FT} \exists X\psi$.*

Proof. Follows from Propositions 5.4, 5.5, 4.2 and 4.1. □

In the following we will show that from the entailment normal form ψ of ϕ with respect to γ it is easy to tell whether we have entailment, disentailment or neither. Moreover, the basic normal form of $\gamma \wedge \phi$ is exactly $\gamma \wedge \psi$ in the first case (and in the second, where $\gamma \wedge \bot = \bot$), and "almost" in the third case (*cf.* Lemma 5.3).

Proposition 5.7 *A basic constraint $\phi \neq \bot$ is normal with respect to γ and X if and only if the following conditions are satisfied:*

1. *ϕ is solved, X-oriented, and contains no variable that is bound by γ;*
2. *if $\phi x = y$ and $xfu \in \gamma$, then $yfv \notin \phi$ for every v;*
3. *if $\phi x = \phi y$ and $xfu \in \gamma$ and $yfv \in \gamma$, then $\phi u = \phi v$;*
4. *if $\phi x = y$ and $Ax \in \gamma$, then $By \notin \phi$ for every B;*
5. *if $\phi x = \phi y$ and $Ax \in \gamma$ and $By \in \gamma$, then $A = B$.*

Lemma 5.8 *If $\phi \neq \bot$ is normal with respect to γ and X, then $\gamma \wedge \phi$ is satisfiable in every model of FT.*

Proof. Let $\phi \neq \bot$ be normal with respect to γ and X. Furthermore, let $\gamma = \gamma_N \wedge \gamma_G$ and $\phi = \phi_N \wedge \phi_G$ be the unique decompositions in normalizer and graph. Since the variables bound by γ_N occur neither in γ_G nor in ϕ, it suffices to show that $\gamma_G \wedge \phi_N \wedge \phi_G$ is satisfiable in every model of FT.

Let $\phi_N(\gamma_G)$ be the basic constraint that is obtained from γ_G by applying all bindings of ϕ_N. Then $\gamma_G \wedge \phi_N \wedge \phi_G$ is equivalent to $\phi_N \wedge \phi_N(\gamma_G) \wedge \phi_G$ and no variable bound by ϕ_N occurs in $\phi_N(\gamma_G) \wedge \phi_G$. Hence it suffices to show that $\phi_N(\gamma_G) \wedge \phi_G$ is satisfiable in every model of FT. With the conditions 2–5 of the preceding proposition it is easy to see that $\phi_N(\gamma_G) \wedge \phi_G$ is a solved clause. Hence we know by axiom scheme *Ax3* that $\phi_N(\gamma_G) \wedge \phi_G$ is satisfiable in every model of FT. □

Theorem 5.9 (Disentailment) *Let ψ be a normal form of ϕ with respect to γ and X. Then $\gamma \models_{\mathcal{T}} \neg\exists X\phi$ iff $\psi = \bot$.*

Proof. Suppose $\psi = \bot$. Then $\gamma \models_{\mathcal{T}} \neg\exists X\psi$ and hence $\gamma \models_{\mathcal{T}} \neg\exists X\phi$ by Proposition 5.6.

To show the other direction, suppose $\gamma \models_{\mathcal{T}} \neg\exists X\phi$. Then $\gamma \models_{\mathcal{T}} \neg\exists X\psi$ by Proposition 5.6 and hence $\gamma \wedge \psi$ unsatisfiable in $\mathcal{T}$ by Proposition 4.2. Since $\mathcal{T}$ is a model of FT (Theorem 2.1), we know by the preceding lemma that $\psi = \bot$ (since ψ is assumed to be normal). □

We say that a variable x is *dependent* in a solved constraint ϕ if ϕ contains a constraint of the form Ax, xfy or $x \doteq y$. (Recall that equations are ordered; thus y is not dependent in the constraint $x \doteq y$.) We use $\mathcal{DV}(\phi)$ to denote the set of all variables that are dependent in a solved constraint ϕ.

In the following we will assume that the underlying signature $\mathcal{S} \uplus \mathcal{F}$ has at least one sort and at least one feature that does not occur in the constraints under consideration. This assumption is certainly satisfied if the signature has infinitely many sorts and infinitely many features.

Lemma 5.10 (Spiting) *Let $\phi_1, \ldots, \phi_n$ be basic constraints different from $\bot$, and $X_1, \ldots, X_n$ be finite sets of variables disjoint from $\mathcal{V}(\gamma)$. Moreover, for every $i = 1, \ldots, n$, let ϕ_i be normal with respect to γ and X_i, and let ϕ_i have a dependent variable that is not in X_i. Then $\gamma \wedge \neg\exists X_1\phi_1 \wedge \ldots \wedge \neg\exists X_n\phi_n$ is satisfiable in every model of FT.*

Proof. Let $\gamma = \gamma_N \wedge \gamma_G$ be the unique decomposition of γ into normalizer and graph. Since the variables bound by γ_N occur neither in γ_G nor in any ϕ_i, it suffices to show that $\gamma_G \wedge \neg\exists X_1\phi_1 \wedge \ldots \wedge \neg\exists X_n\phi_n$ is satisfiable in every model of FT. Thus it suffices to exhibit a solved clause δ such that $\gamma_G \subseteq \delta$ and, for every $i = 1, \ldots, n$, $\mathcal{V}(\delta)$ is disjoint with X_i and $\delta \wedge \phi_i$ is unsatisfiable in every model of FT.

Without loss of generality we can assume that every X_i is disjoint with $\mathcal{V}(\gamma)$ and $\mathcal{V}(\phi_j) - X_j$ for all j. Hence we can pick in every ϕ_i a dependent variable x_i such that $x_i \notin X_j$ for any j.

Let $z_1, \ldots, z_k$ be all variables that occur on either side of equation $x_i \doteq y \in \phi_i$, $i = 1, \ldots, n$ (recall that x_i is fixed for i). None of these variables occurs in any X_j since every ϕ_i is X_i-oriented. Next we fix a feature g and a sort B such that neither occurs in γ or any ϕ_i.

Now δ is obtained from γ by adding constraints as follows: if $Ax_i \in \phi_i$, then add Bx_i; if $x_ify \in \phi_i$, then

add $x_i f\uparrow$; to enforce that the variables $z_1, \ldots, z_k$ are pairwise distinct, add

$$z_k g z_{k-1} \wedge \ldots \wedge z_2 g z_1 \wedge z_1 g\uparrow .$$

It is straightforward to verify that these additions to γ yield a solved clause δ as required. □

Proposition 5.11 *If ϕ is solved and $\mathcal{DV}(\phi) \subseteq X$, then $FT \models \tilde{\forall}\exists X\phi$.*

Proof. Let $\phi = \phi_N \wedge \phi_G$ be the decomposition of ϕ in normalizer and graph. Since every variable bound by ϕ is in X, it suffices to show that $\tilde{\forall}\exists X\phi_G$ is a consequence of FT. This follows immediately from axiom scheme $Ax3$ since ϕ_G is a solved clause. □

Theorem 5.12 (Entailment) *Let ψ be a normal form of ϕ with respect to γ and X. Then $\gamma \models_{\mathcal{T}} \exists X\phi$ iff $\psi \neq \bot$ and $\mathcal{DV}(\psi) \subseteq X$.*

Proof. Suppose $\gamma \models_{\mathcal{T}} \exists X\phi$. Then we know $\gamma \models_{\mathcal{T}} \exists X\psi$ by Proposition 5.6, and thus $\gamma \wedge \neg\exists X\psi$ is unsatisfiable in $\mathcal{T}$. Since γ is solved, we know that γ is satisfiable in $\mathcal{T}$ and hence that $\gamma \wedge \exists X\psi$ is satisfiable in $\mathcal{T}$. Thus $\psi \neq \bot$. Since $\gamma \wedge \neg\exists X\psi$ is unsatisfiable in $\mathcal{T}$ and $\mathcal{T}$ is a model of FT, we know by Lemma 5.10 that $\mathcal{DV}(\psi) \subseteq X$.

To show the other direction, suppose $\psi \neq \bot$ and $\mathcal{DV}(\psi) \subseteq X$. Then $FT \models \tilde{\forall}\exists X\psi$ by Proposition 5.11, and hence $\mathcal{T} \models \tilde{\forall}\exists X\psi$. Thus $\gamma \models_{\mathcal{T}} \exists X\psi$, and hence $\gamma \models_{\mathcal{T}} \exists X\phi$ by Proposition 5.6. □

Theorem 5.13 *Let ϕ be a basic constraint. Then $\gamma \models_{\mathcal{T}} \exists X\phi$ iff $\gamma \models_{FT} \exists X\phi$.*

Proof. One direction holds since $\mathcal{T}$ is a model of FT. To show the other direction, suppose $\gamma \models_{\mathcal{T}} \exists X\phi$. Without loss of generality we can assume that ϕ is normal with respect to γ and X. Hence we know by Theorem 5.12 that $\phi \neq \bot$ and $\mathcal{DV}(\psi) \subseteq X$. Thus $FT \models \tilde{\forall}\exists X\phi$ by Proposition 5.11 and hence $\gamma \models_{FT} \exists X\phi$. □

Theorem 5.14 (Independence) *Let $\phi_1, \ldots, \phi_n$ be basic constraints, and $X_1, \ldots, X_n$ be finite sets of variables. Then*

$$\gamma \models_{\mathcal{T}} \exists X_1\phi_1 \vee \ldots \vee \exists X_n\phi_n \text{ iff } \exists i\colon \gamma \models_{\mathcal{T}} \exists X_i\phi_i.$$

Proof. To show the nontrivial direction, suppose $\gamma \models_{\mathcal{T}} \exists X_1\phi_1 \vee \ldots \vee \exists X_n\phi_n$. Without loss of generality we can assume that, for all $i = 1, \ldots, n$, X_i is disjoint from $\mathcal{V}(\gamma)$, ϕ_i is normal with respect to γ and X_1, and $\phi_i \neq \bot$. Since $\gamma \wedge \neg\exists X_1\phi_1 \wedge \ldots \wedge \neg\exists X_n\phi_n$ is unsatisfiable in $\mathcal{T}$ and $\mathcal{T}$ is a model of FT, we know by Lemma 5.10 that $\mathcal{DV}(\phi_k) \subseteq X_k$ for some k. Hence $\gamma \models_{\mathcal{T}} \exists X_k\phi_k$ by Theorem 5.12. □

6 Conclusion

We have presented a constraint system FT for logic programming providing a universal data structure based on rational feature trees. FT accommodates record-like descriptions, which we think are superior to the constructor-based descriptions of Herbrand.

The declarative semantics of FT is specified both algebraicly (the feature tree structure $\mathcal{T}$) and logically (the first-order theory FT given by three axiom schemes).

The operational semantics for FT is given by an incremental constraint simplification system, which can check satisfiability of and entailment between constraints. Since FT satisfies the independence property, the simplification system can also check satisfiability of conjunctions of positive and negative constraints.

We see four directions for further research.

First, FT should be strengthened such that it subsumes the expressivity of rational constructor trees [7, 8]. As is, FT cannot express that x is a tree having direct subtrees at exactly the features $f_1, \ldots, f_n$. It turns out that the system CFT [24] obtained from FT by adding the primitive constraint

$$x\{f_1, \ldots, f_n\}$$

(x has direct subtrees at exactly the features $f_1, \ldots, f_n$) has the same nice properties as FT. In contrast to FT, CFT can express constructor constraints; for instance, the constructor constraint $x \doteq A(y, z)$ can be expressed equivalently as $Ax \wedge x\{1, 2\} \wedge x1y \wedge x2z$, if we assume that A is a sort and the numbers $1, 2$ are features.

Second, it seems attractive to extend FT such that it can accommodate a sort lattice as used in [1, 3, 4, 5, 23]. One possibility to do this is to assume a partial order $\leq$ on sorts and replace sort constraints Ax with *quasi-sort constraints* $[A]x$ whose declarative semantics is given as

$$[A]x \equiv \bigvee_{B \leq A} Bx.$$

Given the assumption that the sort ordering $\leq$ has greatest lower bounds if lower bounds exist, it seems that the results and the simplification system given for FT carry over with minor changes.

Third, the worst-case complexity of entailment checking in FT should be established. We conjecture it to be quasi-linear in the size of γ and ϕ, provided the available features are fixed a priory.

Fourth, implementation techniques for FT at the level of the Warren abstract machine [2] need to be developed.

References

[1] H. Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.

[2] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, MA, 1991.

[3] H. Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *The Journal of Logic Programming*, 3:185–215, 1986.

[4] H. Aït-Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[5] H. Aït-Kaci and A. Podelski. Towards a Meaning of LIFE. *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (Passau, Germany)*, J. Maluszyński and M. Wirsing, editors. LNCS 528, pages 255–274, Springer-Verlag, 1991.

[6] R. Backofen and G. Smolka. A complete and decidable feature theory. Draft, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany, 1991. To appear.

[7] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[8] A. Colmerauer, H. Kanoui, and M. V. Caneghem. Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2(4):255–292, 1983.

[9] S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In D. Warren and P. Szeredi, editors, *Logic Programming, Proceedings of the 7th International Conference*, pages 31–48, Cambridge, MA, June 1990. The MIT Press.

[10] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, Jan. 1987.

[11] M. Johnson. *Attribute-Value Logic and the Theory of Grammar.* CSLI Lecture Notes 16. Center for the Study of Language and Information, Stanford University, CA, 1988.

[12] R. M. Kaplan and J. Bresnan. Lexical-Functional Grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–381. The MIT Press, Cambridge, MA, 1982.

[13] M. Kay. Functional grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, Berkeley, CA, 1979. Berkeley Linguistics Society.

[14] J.-L. Lassez, M. Maher, and K. Marriot. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, Los Altos, CA, 1988.

[15] J. L. Lassez and K. McAloon. A constraint sequent calculus. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 52–61, June 1990.

[16] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, Cambridge, MA, 1987. The MIT Press.

[17] K. Mukai. Partially specified terms in logic programming for linguistic analysis. In *Proceedings of the 6th International Conference on Fifth Generation Computer Systems*, 1988.

[18] K. Mukai. *Constraint Logic Programming and the Unification of Information.* PhD thesis, Tokyo Institute of Technology, Tokyo, Japan, 1991.

[19] M. Nivat. Elements of a theory of tree codes. In M. Nivat, A. Podelski, editors, *Tree Automata (Advances and Open Problems)*, Amsterdam, NE, 1992. Elsevier Science Publishers.

[20] W. C. Rounds and R. T. Kasper. A complete logical calculus for record structures representing linguistic information. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, pages 38–43, Boston, MA, 1986.

[21] V. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, CA, January 1990.

[22] G. Smolka. Feature constraint logics for unification grammars. *The Journal of Logic Programming*, 12:51–87, 1992.

[23] G. Smolka and H. Aït-Kaci. Inheritance hierarchies: Semantics and unification. *Journal of Symbolic Computation*, 7:343–370, 1989.

[24] G. Smolka and R. Treinen. Relative simplification for and independence of CFT. Draft, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany, 1992. To appear.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Range Determination of Design Parameters by Qualitative Reasoning and its Application to Electronic Circuits

Masaru Ohki, Eiji Oohira, Hiroshi Shinjo, and Masahiro Abe

Central Research Laboratory, Hitachi, Ltd.
Higashi-Koigakubo, Kokubunji, Tokyo 185, Japan
ohki@crl.hitachi.co.jp

Abstract

There are numerous applications of qualitative reasoning to diverse fields of engineering. The main application has been to diagnosis, but there are a few applications to design. We show a new application to design, suggesting valid ranges for design parameters; this application follows the step of structure determination. The application does not provide more innovative design, but it is one of the important steps of design. To implement it, we use an envisioning mechanism, which determines all possible behaviors of a system through qualitative reasoning. Our method: (1) performs envisioning with design parameters whose values are initially undefined, (2) selects preferable behaviors from all possible behaviors found by the envisioning process, and (3) calculates the ranges of those design parameters that give the preferable behaviors.

We built a design-support system Desq (Design support system based on qualitative reasoning) by improving an earlier qualitative reasoning system Qupras (Qualitative physical reasoning system). We added three new features: envisioning, calculating the undefined parameters, and propagating new constraints on constant parameters. The Desq system can deal with quantities qualitatively and quantitatively, like Qupras. Accordingly, we may someday be able to determine the quantitative ranges, if the parameters can be expressed quantitatively. Quantitative ranges are preferable to qualitative values, to support the determination of design parameters.

1 Introduction

Recently, many expert systems have been used in the diverse fields of engineering. However, several problems still exist. One is the difficulty of building knowledge bases from the experience of human experts. The other is that these expert systems cannot deal with unimaginable situations [Mizoguchi 87]. Reasoning methods using deep knowledge, which is the fundamental knowledge of a domain, are expected to solve these problems. One reasoning method is qualitative reasoning [Bobrow 84]. Qualitative reasoning determines dynamic behaviors, which are the states of a dynamic system and its state changes, using deep knowledge of the dynamic system. Another feature of qualitative reasoning is that it can deal with quantities qualitatively. So far, there have been many applications of qualitative reasoning to engineering [Nishida 88a, Nishida 88b, Nishida 91]. The main application has been to diagnosis [Yamaguchi 87, Ohwada 88], but recently there have also been applications to design [Murthy 87, Williams 90].

In this paper, we show a new application to design that supports decisions by suggesting valid ranges for design parameters; it follows the step of structure determination. This application is not considered to be more innovative than the previous applications to design [Murthy 87, Williams 90], but it is one of the important steps of design [Chandrasekaran 90].

The key to design support is applying an envisioning mechanism, which predicts the behaviors of the dynamic system, to those design parameters whose values are undefined. If the envisioning is performed on condition that the design parameters whose values a designer wants to determine are undefined, all possible behaviors under the undefined design parameters can be predicted by the envisioning process. Some hypotheses are made to obtain each behavior. The main reason why hypotheses are made is that conditions written in the definitions of objects and physical rules cannot be evaluated because the design parameters are undefined. Among the obtained possible behaviors, more than one behavior desired by the designer is expected to exist. The designer can select the behaviors which he/she exactly prefers. Although the designer may not know the values of the design parameters, he/she knows the desired behavior. The values of the undefined parameters can be calculated from the hypotheses made to obtain the desired behavior.

To sum up, the method of determining valid ranges for design parameters offers the following:

(1) Performs envisioning for design parameters whose values are initially undefined,

(2) Selects preferable behaviors from possible behaviors found by the envisioning process, and

(3) Calculates the ranges of those design parameters that give the preferable behaviors.

We used a qualitative reasoning system Qupras (Qualitative physical reasoning system) [Ohki 86, Ohki 88, Ohki 92] to construct a decision support system Desq (Design support system based on qualitative reasoning) that suggests valid ranges for design parameters.

Qupras, using knowledge about physical rules and objects after being given an initial state, determines the followings:

(1) Relations between objects that are components of physical systems.

(2) The subsequent states of the system following a transition.

We extended Qupras to construct Desq as follows:

(1) Envisioning

In Qupras, if a condition of a physical rule or an object cannot be evaluated, Qupras asks the user to specify the condition. We extended Qupras to allow it to continue assuming an unevaluated condition.

(2) Calculating the undefined parameters

After envisioning all possible behaviors, Desq calculates the ranges of the undefined design parameters that give the behavior specified by the designer.

(3) Propagation of new constraints on constants

In the envisioning process, constraints related to some constant parameters become stronger because conditions in the definitions of physical rules and objects are hypothesized. The constraints propagate to the subsequent states.

(4) Parallel constraint solving

Qupras uses a combined constraint solver consisting of three basic constraint solvers: a Supinf method constraint solver, an Interval method constraint solver, and a Groebner base method constraint solver, all written in ESP. The processing load of the combined constraint solver was heavy, so we converted it to KL1 to speed up processing.

Desq can deal with quantities qualitatively and quantitatively like Qupras. Accordingly, we may someday be able to get quantitative ranges, if the parameters can be given as quantitative values. Quantitative ranges may be preferable for decision support. The usual qualitative reasoning like [Kuipers 84] gives qualitative ranges.

Section 2 shows how Desq suggests ranges for design parameters, Section 3 describes the system organization of Desq, Section 4 shows an example of Desq suggesting the value of a resistor in a DTL circuit, Section 5 describes related works and Section 6 summarizes the paper.

2 Method of determining design parameters

In design, there are many cases in which a designer does not directly design a new device, but changes or improves an old device. Sometimes designers only change parameters of components in a device to satisfy the requirements. The designer, in such cases, knows the structure of the device, and needs only to determine the new values of the components. This is common for electronic circuits. We apply qualitative reasoning to the design decisions.

The key process used to determine design parameters is envisioning. Our method is as described in Section 1:

(1) All possible behaviors of a device are found by envisioning, with design parameters whose values are initially undefined.

(2) Designers select preferable behaviors from these possible behaviors.

(3) The ranges of the design parameters that give the preferable behaviors are calculated using a parallel constraint solver.

If a condition in the definitions of a physical rule or an object cannot be evaluated, Desq hypothesizes one case where the condition is valid and another where it is not valid, and separately searches each case to find all possible behaviors. This method is called envisioning, and is the same as [Kuipers 84]. If a contradiction is detected, the reasoning is abandoned. If no contradiction is detected, the reasoning is valid. Finally, Desq finds several possible behaviors of a device.

The characteristics of this approach are as follows:

(1) Only deep knowledge is used to determine design parameters.

(2) All possible behaviors with regard to undefined design parameters are found. Such information may be used in safety design or danger estimation.

(3) Ranges of design parameters giving preferable behaviors are found. If a designer uses numerical CAD systems, for example,

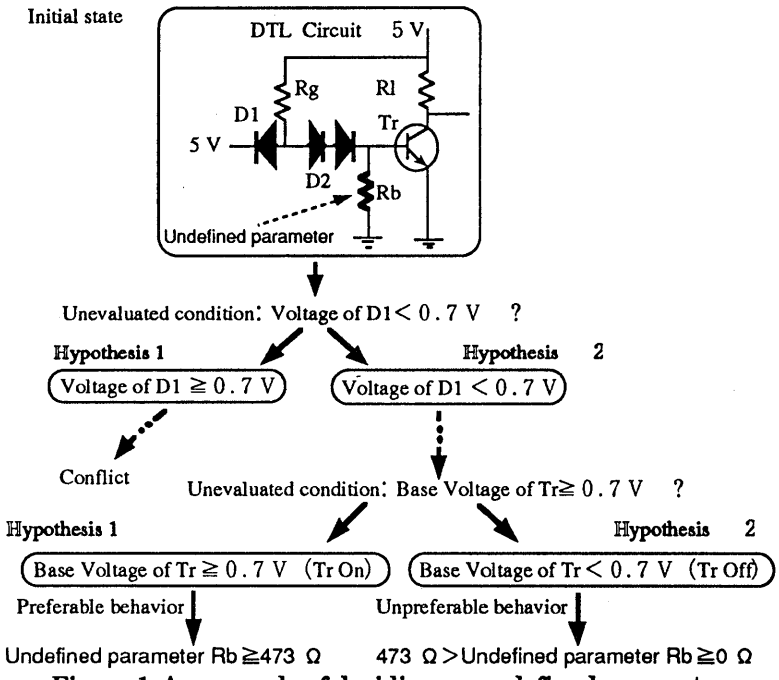


Figure 1 An example of deciding an undefined parameter

SPICE, he/she need not simulate values outside the ranges.

Figure 1 shows an example of suggesting ranges for a design parameter. This example illustrates the determination of a resistance value in a DTL circuit. The designer inputs the DTL structure and the parameters of the components except for the resistance Rb.

Desq checks the conditions in the definitions of physical rules and objects. If they are satisfied, the equations in their consequences are sent to the parallel constraint solvers. But, it is not known what state the diode D1 is in, because the resistance Rb is undefined. The first condition is whether the voltage of D1 is lower than 0.7 volts. Desq hypothesizes two cases; in the first the condition is not satisfied, and in the second it is. The first hypothesis is abandoned because the parallel constraint solver detects a conflict with the other equations. In the second hypothesis, no conflict is detected. After some more hypotheses are made, another state is detected where it is not known whether or not the condition giving the state of the transistor Tr is satisfied. Desq similarly hypothesizes this condition. Finally, Desq finds two possible behaviors for the initial data. Then, Desq calculates the resistance Rb. The resistance must be larger than 473 ohms to give the desired behavior, where the circuit acts as a NOT circuit because the transistor is "on". If the resistance is smaller than 473 ohms, the circuit shows another behavior which is not preferable. Thus, the resistance Rb must be larger than 473 ohms. This proves that Desq can deal with quantities qualitatively and quantitatively.

3 System organization

This section describes the system organization of Desq. Figure 2 shows that Desq mainly consists of three subsystems:

(1) Behavior reasoner

This subsystem is based on Qupras. It determines all possible behaviors.

(2) Design parameter calculator

This subsystem calculates ranges of design parameters.

(3) Parallel constraint solver

This subsystem is written in KL1, and is executed on PIM, Multi-PSI, or Pseudo Multi-PSI.

When the designer specifies initial data, the behavior reasoner builds its model corresponding to the initial state, by evaluating conditions of physical rules and objects. The physical rules and objects are stored in the knowledge base. The model in Desq uses simultaneous inequalities in

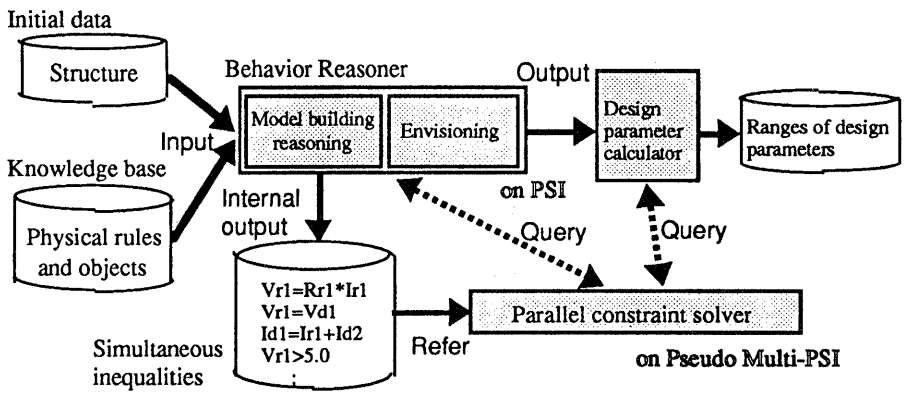


Figure 2 System organization

the same way as that in Qupras. Simultaneous inequalities are passed to the parallel constraint solver to check the consistency and store them. If an inconsistency is detected, the reasoning process is abandoned. Conditions in the definitions of physical rules and objects are checked by the parallel constraint solver. If the conditions are satisfied, the inequalities in the consequences of the physical rules and objects are added to the model in the parallel constraint solver. If a condition cannot be evaluated by the parallel constraint solver, envisioning is performed. Finally, when all possible behaviors are found, the design parameter calculator deduces the ranges of design parameters that give preferable behaviors.

3.1 Behavior reasoner

3.1.1 Qupras Outline

Qupras is a qualitative reasoning system that uses knowledge from physics and engineering textbooks. Qupras has the following characteristics:

(1) Qupras has three primitive representations: physical rules (laws of physics), objects and events.

(2) Qupras determines the dynamic behaviors of a system by building all equations for the system using knowledge of physical rules, objects and events. The user need not enter all the equations of the system.

(3) Qupras deals with equations that describe basic laws of physics qualitatively and quantitatively.

(4) Qupras does not require quantity spaces to be given in advance. It finds the quantity spaces for itself during reasoning.

(5) Objects in Qupras can inherit definitions from their super objects. Thus, physical rules can be defined generally by specifying the definitions of object classes with super objects.

Qupras is similar to QPT [Forbus 84], but does not use influence. The representations describing relations of values in Qupras are only equations. Qupras aims to represent laws of physics given in physics textbooks and engineering textbooks. Laws of physics are generally described not by using influences in the textbooks, but by using equations. Therefore, Qupras uses only equations.

The representation of objects mainly consists of existential conditions and relations. Existential conditions correspond to conditions needed for the objects to exist. Objects satisfying these conditions are called active objects. The relations are expressed as relative equations which include physical variables (hereafter physical quantities are referred to as physical variables). If existential conditions are satisfied, their relations become known as relative equations that hold for physical variables of the objects specified in the physical rule definition.

The representation of physical rules mainly consists of objects, applied conditions and relations. The objects are those necessary to apply a physical rule. The representations of applied conditions and relations are similar to those of objects. Applied conditions are those required to activate a physical rule, and relations correspond to the laws of physics. Physical rules whose necessary objects are activated and whose conditions are satisfied are called active physical rules. If a given physical rule is active, its relations become known as in the case of objects.

Qualitative reasoning in Qupras involves two forms of reasoning: propagation reasoning and prediction reasoning. Propagation reasoning determines the state of the physical system at a given moment (or during a given time interval). Prediction reasoning determines the physical variables that change with time, and predicts their values at the next given point in time. The propagation reasoning also determines the subsequent states of the physical system using the results from the prediction reasoning.

3.1.2 Behavior Reasoner

The behavior reasoner is not much different from that of Qupras. The two features below are additions to that of Qupras.

(1) Envisioning

In Qupras, if conditions of physical rules and objects cannot be evaluated, Qupras asks the user to specify the conditions. It is possible for Desq to continue to reason in such situations by assuming unevaluated conditions.

(2) Propagation of new constraints on constants

There are two types of parameters (quantities): constant and variable. In envisioning, the constraints related to some constant parameters become stronger by hypothesizing some conditions in the definitions of physical rules and objects. The constraints propagate to the subsequent states.

Before the reasoning, all initial relations of the objects defined in the initial state are set as known relations, which are used to evaluate the conditions of objects and physical rules. Initial relations are mainly used to set the initial values of the physical variables. If there is no explicit change to an initial relation, the initial relation is held. An example of an explicit change is the prediction of the next value in the prediction reasoning.

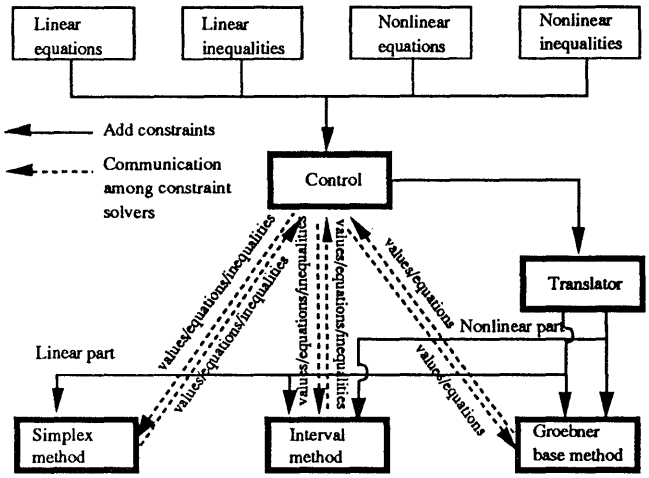


Figure 3 Combined constraint solver

Propagation reasoning finds active objects and physical rules whose conditions are satisfied by the known relations. If a contradiction is detected, the propagation reasoning is stopped. If no condition of physical rules and objects can be evaluated, the reasoning process is split by the envisioning mechanism into two process: one process hypothesizing that the condition is satisfied and other hypothesizing that it is not.

Prediction reasoning first finds the physical variables changing with time from the known relations that result from the propagation reasoning. Then, it searches for the new values or the new intervals of the changing variables at the next specified time or during the next time interval. Desq updates the variables according to the sought values or intervals in the same way as Qupras. The updated values are used as the initial relations at the beginning of the next propagation reasoning.

3.2 Design parameter calculator

The method of calculating the design parameters is simple. After finding all possible behaviors, the designer specifies which design parameters to calculate. Then, the upper and lower values of the specified parameters are calculated by the parallel constraint solver.

3.3 Parallel constraint solver

The parallel constraint solver tests whether the conditions written in the definitions of physical rules and objects are proven by the known relations obtained from active objects and active physical rules, and from initial relations.

We want to solve nonlinear simultaneous inequalities to test the conditions in the definitions of objects, physical rules and events. More than one algorithm is used to build the combined constraint solver, because we do not know of any single efficient algorithm for nonlinear simultaneous inequalities. We connected the three solvers as shown in Figure 3. The combined constraint solver consists of the following three parts:

(1) Nonlinear inequality solver based on the interval method [Simmons 86],

(2) Linear inequality solver based on the Simplex method [Konno 87], and

(3) Nonlinear simultaneous equation solver based on the Groebner base method [Aiba 88].

If any one of the three constraint solvers finds new results, the results are passed on to the other constraint solvers by the control parts. This combined constraint solver can solve broader equations than each individual solver can. However, its results are not always valid, because it cannot solve all nonlinear simultaneous inequalities.

The reason why we can get quantitative ranges is that the combined constraint solver can process quantities quantitatively as well as qualitatively.

4 Example

4.1 Description of Model

We show another example of the operator. We use a DTL circuit identical to that the same as in Figure 1. In this example, however, the input voltage and the resistance Rb are undefined.

```
initial_state dtl
 objects -
  Rl-resistor ;
  Rg-resistor ;
  Rb-resistor ;
  Tr-transistor ;
  D1-diode ;
  D2-diode2 ;
 initial_relations
  connect(t1!Rl,t1!Rg) ;
  connect(t2!Rg,t1!D1,t1!D2) ;
  connect(t2!D3,t1!Rb,tb!Tr) ;
  connect(t2!Rl,tc!Tr) ;
  resistance@Rl=6000.0 ;
  resistance@Rg=2000.0 ;
  resistance@Rb >= 0.0;
  v@t1!Rl = 5.0 ;
  v@t2!D1 >= 0.0 ;
  v@t2!D1 =< 10.0 ;
  v@te!Tr = 0.0 ;
  v@t2!Rb = 0.0 ;
end.
```

Figure 4 Initial state for DTL circuit

The initial data is shown in Figure 4. The "objects" field specifies components and their classes in the DTL circuit. The "initial_relations" field specifies the relations holding in the initial state. For example, "connect(t2!Rg, t1!D1, t1!D2)" specifies that the terminal t2 of the resistor Rg, the terminal t1 of the diode D1, and the terminal t1 of the diode D2 are connected. The "!" is a symbol specifying a part. The "t2!Rg" expresses the terminal t2 which is one part of Rg. Rb is specified as a resistor in the "objects" definition. The "@" indicates a parameter. The "resistance@Rl" represents the resistance value of Rl. The "resistance@Rl = 6000.0" specifies that Rl is 6000.0 ohms. The resistance Rb is constrained to be positive, and the input voltage, which is the voltage of the terminal t2 in the diode D1, is constrained to be between 0.0 and 10.0 volts. Both values are undefined, and Rb is a design parameter.

Figure 5 shows the definition of a diode. Its super object is a two_terminal_device, so the diode inherits the properties of the two_terminal_device, i.e., it has two parts, both of which are terminals. Each terminal has two attributes "v" for voltage and "i" for current. The diode has an initial

```
object terminal:Terminal
 attributes
  v ;
  i ;
end.

object two_terminal_device:TTD
 parts_of
  t1-terminal ;
  t2-terminal ;
end.

object diode:Di
 supers
  two_terminal_device;
 attributes
   v ;
   i ;
   resistance-constant ;
 initial_relations
    v@Di=v@t1!Di-v@t2!Di ;
 state on
  conditions
    v@Di >= 0.7 ;
  relations
    v@Di= 0.7 ;
    i@Di >= 0.0 ;
 state off
  condition
    v@Di < 0.7 ;
  relations
    resistance@Di=100000.0 ;
    v@Di=resistance@Di*i@Di ;
end.
```

Figure 5 Definition of diode

```
physics three_connect_1
 objects
  TTD1 - two_terminal_device ;
  TTD2 - two_terminal_device ;
  TTD3 - two_terminal_device ;
  T1-terminal partname t1 part_of TTD1 ;
  T2-terminal partname t1 part_of TTD2 ;
  T3-terminal partname t1 part_of TTD3 ;
 conditions
  connect(T1,T2,T3);
 relations
  v@T1 = v@T2 ;
  v@T2 = v@T3 ;
  i@T1 + i@T2 + i@T3 = 0 ;
end.
```

Figure 6 Definition of physics

relation, which specifies the voltage difference between its terminals. The diode also has two states: one is the "on" state where the voltage difference is greater than 0.7, and the other is the "off" state where the voltage difference is less than 0.7. If the diode is in the "on" state, it behaves like a conductor. In the "off" state, it behaves like a resistor. A transistor is defined like a diode, but it has three states, "off", "on" and "saturated" (In the example of Figure 1, we used a transistor model with two states, "off" and "on").

Figure 6 shows the definition of a physical rule. The rule shows Kirchhoff's law when the terminals t1 of three two_terminal_devices are connected. It is assumed that the current into t1 of a two_terminal_device flows to the terminal t2. In fact, three two_terminal_devices can be connected in eight ways depending on how the terminals are connected.

Table 1 All behaviors of DTL circuit

State	Range of input	Range of resistance value	Range of output
1 ON-ON-SAT	1.40081 ~ 1.5381	486.16 ~ infinity	0.2
2 ON-ON-ON	1.4 ~ 1.40081	482.75 ~ infinity	0.2~5.0
3 ON-ON-OFF	0.7 ~ 1.4	0 ~ 233,567	4.94
4 ON-OFF-ON	0 ~ 1.4007	100,000 ~ infinity	0.842~5.0
5 ON-OFF-OFF	0 ~ 1.4	0 ~ 233,567	4.94
6 OFF-ON-SAT	1.40081 ~ 10.0	460.9 ~ infinity	0.2
7 OFF-ON-ON	1.4 ~ 10.0	457.8 ~ 488.53	0.2~5.0
8 OFF-ON-OFF	0.7 ~ 10.0	0 ~ 484.1	4.94
9 OFF-OFF-*	Conflict		

4.2 Results

Table 1 shows all behaviors of the DTL circuit obtained by envisioning. The state column indicates the states of the diode, the diode2 and the transistor. The following columns show the range of the input voltage (volts), the range of the resistance Rb (ohms), and the range of the output voltage (volts). As is shown, the envisioning found nine states. Because the input voltage and the resistance Rb were undefined, the conditions of the two diodes and the transistor could not be

evaluated. So, Desq was used to hypothesize both cases, and to search all paths. Figure 7 shows the relationship between the resistance and the input voltage. The reason why the ranges in Table 1 overlap is because the models of the diodes and the transistor are approximate models.

A designer can decide, by looking at Figure 7, the resistance Rb for the DTL circuit to behave as a NOT circuit. It is desired for Rb to be greater than about 0.5 k ohms, and less than about 100 k ohms, so that the DTL circuit can output a low voltage (nearly 0 volts) when the input is greater than 1.5 volts, or can output a high voltage (nearly 5 volts) when the input is less than about 1.5 volts. The range is shown by the area enclosed by the dotted lines in Figure 7.

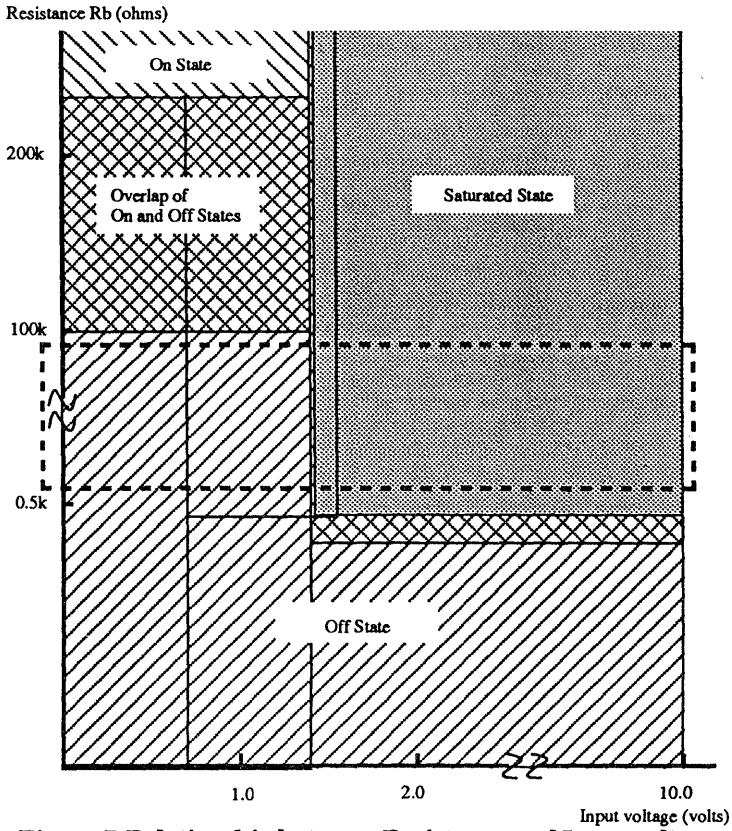


Figure 7 Relationship between Resistance and Input voltage

5. Related Works

Desq does not suggest structures of devices like the methods of [Murthy 87] and [Williams 90]. Rather, it suggests the ranges of design parameters for preferable behaviors. The suggestion is also useful, because determining values of design parameters is one of the important steps of design [Chandrasekaran 90].

This approach may be regarded as one application of constraint satisfaction problem solving. There are several papers that deal with electronic circuits as examples, using constraint satisfaction problem solving [Sussman 80, Heintze 86, Mozetic 91]. Sussman and Steele's system cannot suggest ranges for design parameters, because their system uses only equations. Heintze, Michaylov and Stuckey's work using CLP(R) to design electronic circuits is the most similar to Desq, but Desq is different from Heintze's work for the following points:

(1) Knowledge on objects and laws of physics is more declarative for Desq.

(2) Desq can design ranges of design parameters (of devices) that change with time.

(3) Desq can deal with nonlinear inequalities, and Desq can solve nonlinear inequalities in some cases.

In Mozetic and Holzbaur's work, numerical and qualitative models are used. In their view, our approach uses numerical models rather than qualitative models. But, if a constraint solver is used to solve inequalities, it is possible to use both numerical and qualitative calculations.

6. Conclusion

We have described a method of suggesting ranges for design parameters using qualitative reasoning, and implemented the method in Desq. The ranges obtained are quantitative, because our system deals with quantities quantitatively as well as qualitatively. In an example utilizing the DTL circuit, Desq suggested that the range of a resistance (Rb in Figure 1) should be greater than about 0.5 k ohms and less than about 100 k ohms to work the DTL circuit as a NOT circuit. If the designer wishes for a more detailed design, for example, to minimize the response time by performing numerical calculation, he/she need not calculate outside the range, and thus can save on the calculation cost, which is greater for direct numerical calculation (outside range).

However, there are some possibilities that Desq cannot suggest valid ranges or the best ranges for design parameters. This is because of the following:

(1) The ability to solve nonlinear inequalities in Consort is short

Desq may suggest invalid or weak ranges because Consort cannot perfectly solve nonlinear inequalities. But, almost all results can be obtained by performing more

detailed analysis using numerical analysis systems, for example, SPICE.

(2) Inexact definitions are used

It may be difficult to describe the definitions of physical rules and objects. This is because from inexact definitions, inexact results may be obtained.

(3) The ability to analyze circuits is short

The current Desq cannot analyze positive feedback. If there are any positive feedbacks in a circuit, Desq may return wrong results.

The example in this paper does not change with time. We are currently working on how to determine ranges of design parameters (of circuits) that change with time, for example, a Schmidt trigger circuit. In such a case, we need to propagate new constraints on constant parameters. Moreover, we are investigating the load balancing of the parallel constraint solver to speed it up.

7. Acknowledgements

This research was supported by the ICOT (Institute of New Generation Computer Technology). We wish to express our thanks to Dr. Fuchi, Director of the ICOT Research Center, who provided us with the opportunity of performing this research in the Fifth Generation Computer Systems Project. We also wish to thank Dr. Nitta, Mr. Ichiyoshi and Mr. Sakane (Current address: Nippon Steel Corporation) in the seventh research laboratory for their many comments and discussions regarding this study, and Dr. Aiba, Mr. Kawagishi, Mr. Menjyu and Mr. Terasaki in the fourth research laboratory of the ICOT for allowing us to use their GDCC and Simplex programs, and helping us to implement our parallel constraint solver. And we wish to thank Prof. Nishida of Kyoto University for his discussions, Mr. Kawaguti, Miss Toki and Miss Isokawa of Hitachi Information Systems for their help in the implementation of our system, and Mr. Masuda and Mr. Yokomizo of Hitachi Central Research Laboratory for their suggestions on designing electric circuits using qualitative reasoning.

References

[Aiba 88] Aiba, A., Sakai, K., Sato Y., and Hawley, D. J.: Constraint Logic Programming Language CAL, pp. 263-276, Proc. of FGCS, ICOT, Tokyo, 1988.

[Bobrow 84] Bobrow, D. G.: Special Volume on Qualitative Reasoning about Physical Systems, Artificial Intelligence, 24, 1984.

[Chandrasekaran 90] Chandrasekaran, B.: Design Problem Solving: A Task Analysis, AI Magazine, pp. 59-71, 1990.

[Forbus 84] Forbus, K. D.: Qualitative Process Theory, Artificial Intelligence, 24, pp. 85-168, 1984.

[Hawley 91] Hawley, D. J.: The Concurrent Constraint Language GDCC and Its Parallel Constraint Solver, Proc. of KL1 Programming Workshop '91, pp. 155-165, ICOT, Tokyo, 1991.

[Heintze 86] Heintze, N., Michaylov, S., and Stuckey P.: CLP(R) and some Electrical Engineering Problems, Proc. of the Fourth International Conference of Logical Programming, pp.675-703, 1986.

[Konno 87] Konno, H.: Linear Programming, Nikka-Girren, 1987 (in Japanese).

[Kuipers 84] Kuipers, B.: Commonsense Reasoning about Causality: Deriving Behavior from Structure, Artificial Intelligence, 24, pp. 169-203, 1984.

[Mozetic 91] Mozetic, I. and Holzbaur, C. : Integrating Numerical and Qualitative Models within Constraint Logic Programming, Proc. of the 1991 International Symposium on Logic Programming, pp. 678-693, 1991.

[Mizoguchi 87] Mizoguchi, R.: Foundation of expert systems, Expert system - theory and application, Nikkei-McGraw-Hill, pp. 15, 1987 (in Japanese).

[Murthy 87] Murthy, S. and Addanki, S.: PROMPT : An Innovative Design Tool, Proc. of AAAI-87, pp.637-642, 1987.

[Nishida 88a] Nishida, T.: Recent Trend of Studies with Respect to Qualitative Reasoning (I) Progress of Fundamental Technology, pp. 1009-1022, 1988 (in Japanese).

[Nishida 88b] Nishida, T.: Recent Trend of Studies with Respect to Qualitative Reasoning (II) New Research Area and Application, pp. 1322-1333, 1988 (in Japanese).

[Nishida 91] Nishida, T.: Qualitative Reasoning and its Application to Intelligent Problem Solving, pp. 105-117, 1991 (in Japanese).

[Ohki 86] Ohki, M. and Furukawa, K.: Toward Qualitative Reasoning, Proc. of Symposium of Japan Recognition Soc., 1986, or ICOT-TR 221, 1986.

[Ohki 88] Ohki, M., Fujii, Y., and Furukawa, K.: Qualitative Reasoning based on Physical Laws, Trans. Inf. Proc. Soc. Japan, 29, pp. 694-702, 1988 (in Japanese).

[Ohki 92] Ohki, M., Sakane, J., Sawamoto, K., and Fujii, Y.: Enhanced Qualitative Physical Reasoning System: Qupras, New Generation Computing, 10, 1992 (to appear).

[Ohwada 88] Ohwada, H., Mizoguchi, F., and Kitazawa, Y.: A Method for Developing Diagnostic Systems based on Qualitative Simulation, J. of Japanese Soc. for Artif. Intel., 3, pp. 617-626, 1988 (in Japanese).

[Simmons 86] Simmons, S.: Commonsense Arithmetic Reasoning, Proc. of AAAI-86, pp. 118-128, 1986.

[Sussman 80] Sussman, G. and Steele, G. : CONSTRAINTS - A Language for Expressing Almost-Hierachical Descriptions, Artificial Intelligence, 14, pp. 1-39, 1980.

[Yamaguchi 87] Yamaguchi, T., Mizoguchi, R., Taoka, N., Kodaka, H., Nomura, Y., and Kakusho, O: Basic Design of Knowledge Compiler Based on Deep Knowledge, J. of Japanese Soc. for Artif. Intel., 2, pp. 333-340, 1987 (in Japanese).

[Williams 90] Williams B. C.: Interaction-based Invention: Designing Novel Devices from First Principles, Proc. of AAAI-90, pp.349-356, 1990.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Logical Implementation of Dynamical Models

Yoshiteru Ishida
Division of Applied Systems Science
Kyoto University, Kyoto 606 Japan
ishida@kuamp.kyoto-u.ac.jp

Abstract

In this paper, we explore the logical system which reflect the dynamical model. First, we define the "causality" which requires "time reference". Then, we map the causation to the specific type of logical implications which requires the time fragment $dt > 0$ at each step when causal changes are made. We also propose a set of axioms, which reflect the feature of state-space and the relation between time and state-space. With these axioms and logical implications mapped from the dynamical systems, the dynamical state transition can be deduced logically. We also discussed an alternative way of deducing the dynamical state change using time operators and state-space operators.

1 Introduction

Although the dynamical systems and logical systems are considered to be completely different systems, there are several elements in common. We mapped from dynamical systems to logical systems to investigate the following questions:

(1) How the fundamental concepts in dynamical systems such as observability, stability can be related to those in logical systems such as completeness, soundness ? (2) In order to attain the dynamical simulation on the mapped logical systems, what are necessary ? (3) Can the qualitative simulation be carried out by deducing the future state from the current state and some axioms characterizing time, state-space and their relations ?

We consider it is crucial to discriminate (physical) causality explicitly from logical deducibility. We studied a causality characterized by "the time reference" other than event dependency for the discussion of physical causality. The physical causality (or equivalently "change" through physical time) is intrinsically embedded in a dynamical model which states the causal relation between what is changed and what makes the change.

In this paper, we treat the physical causality as specific type of deduction which always requires the fact of the time fragment $dt > 0$ at each step. By mapping the dynamical model as well as some meta-rules which reflects that the state-space of dynamical systems is continuous to logical rules, the qualitative reasoning on dynamica systems can be done by logical deductions.

Section 2 discusses the causality on the dynamica model. The causality is defined in terms of physica time. Then the causation is mapped to the logical implication which requires time fragment ($dt > 0$) at each step. Cause-effect sequence is obtained by the deduction where the new fact $dt > 0$ is required at each step. Section 3 discusses the relation between some concepts on dynamical models and those on logical systems. Section 4 presents a set of axioms from which state transitions are deduced logically. Section 5 discusses an alternative formalization of logical systems for deducing the dynamical changes.

2 Mapping Causality in Dynamical Models to Logical Implications

2.1 Causality referring to time

The causality has the following two requirements, which seem intuitively sound for a causality for the discussion of dynamical change. When we say "the event A caused the event B", we must admit

(1) Time Reference : The event A occurred "before" the event B, (2) Event Dependency: The occurrence o the event B must be "dependent on" the occurrence o the event A.

The "time reference" plays a crucial role to make clear distinction between "the causality" and logical deduction. In the original dynamical model of the form: $dY/dt = X$

contains the "built-in causal" direction from the right hand side to the left hand side. We restrict ourselves to interpret the form $dY/dt = X$ as follows: $X > 0$ caused $dt > 0$) or is capable of causing the event of Y increase $dY > 0$). The requirement of the new fact $dt > 0$ should be claimed to verify the "built-in causality". Thus the

form will be mapped to the logical form:

2.2 Language for dynamics

In order to logically describe the constraint of dynamical model, we use the following *First Order Predicate Calculus*. We use the 4-place predicates p(x,i), n(x,i), z(x,i) which should be interpreted as positive, negative, zero of the variable x at certain moment i. p(x,i), for example is interpreted as follows:

$$p(x,i) = \begin{cases} true, & \text{if x (at time i)} > 0; \\ false, & \text{otherwise.} \end{cases}$$

Since the state must be unique at any moment, these predicates must satisfy the following uniqueness axioms U.

U-(1) $\forall x \forall i(p(x,i) \rightarrow ((\sim n(x,i)) \wedge (\sim z(x,i))))$
U-(2) $\forall x \forall i(n(x,i) \rightarrow ((\sim p(x,i)) \wedge (\sim z(x,i))))$
U-(3) $\forall x \forall i(z(x,i) \rightarrow ((\sim n(x,i)) \wedge (\sim p(x,i))))$

We also use the 2-place predicate of inequality $>(x,y)$. Other than these three predicates, we also use functions such as d/dt(time derivative), +(addition), −(substraction), ·(multiplication), /(division) defined on the time varying function x(t) in our language.

With these predicates, the causality defined from X to Y can be written by:

$$p(X(t),i) \wedge p(dt,i) \rightarrow p(dY/dt,i)$$

$$n(X(t),i) \wedge p(dt,i) \rightarrow n(dY/dt,i)$$

$$z(X(t),i) \vee z(dt,i) \rightarrow z(dY/dt,i)$$

2.3 Causality in dynamical models

We formalize the "causality" by the propagation of sign in the dynamical model. In the propagation, time reference is included, since $p(dt,i)$ is always needed to conclude the causation.

Example 2.1.

In order to compare the simulation results with those done by other qualitative simulation [de Kleer and Brown 1984], we use the same example of pressure regulator as shown in Fig. 1.

We can identify the causality in the feedback path. The flow also is caused by a driving force and by the available area for the flow. Further, the pressure at a point is caused by the flow through the point. Reflecting these causal path, the following model is obtained.

$$d\delta Xs/dt = -a \cdot \delta Po - d \cdot \delta Xs$$
$$d\delta Q/dt = b \cdot (\delta Pi - \delta Po - c \cdot (2Xs \cdot Q\delta Q - Q^2 \delta Xs)/Xs^2$$
$$d\delta Po/dt = e \cdot (2Q\delta Q - f \cdot \delta Po)$$

where a, b, c, d, e, and f are appropriately chosen positive constants. δ x denotes the variance from the equilibrium point of x.

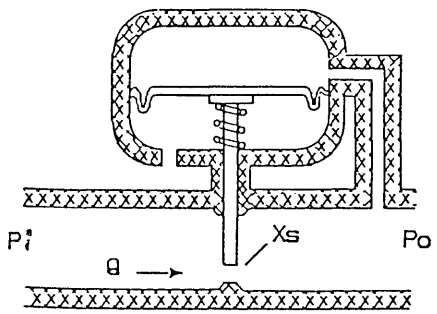


Fig.1 A Schematic Diagram of Pressure Regulator

The first equation of the model, for example, is mapped to the logical formulae:

$$n(\delta po(t),i) \wedge p(dt,i) \rightarrow p(d\delta xs/dt,i)$$

$$p(\delta po(t),i) \wedge p(dt,i) \rightarrow n(d\delta xs/dt,i)$$

$$z(\delta po(t),i) \vee z(dt,i) \rightarrow z(d\delta xs/dt,i)$$

With the set of logical formulae, which are mapped from the dynamical equations and the following axioms, we can obtain a cause-effect sequence by the causal deduction on this model.

I-(1)

$$\forall x \forall i \forall j(z(x,i) \wedge p(dx/dt,i) \wedge (j > i) \rightarrow$$

$$\exists k((j > k) \wedge (k > i) \wedge p(x,j,k)))$$

I-(2)

$$\forall x \forall i \forall j(z(x,i) \wedge n(dx/dt,i) \wedge (j > i) \rightarrow$$

$$\exists k((j > k) \wedge (k > i) \wedge n(x,j,k)))$$

These are the instant change rules [de Kleer and Bobrow 1984], which state that $z(x,i)$ is a point with measure zero.

Suppose Pi is disturbed $p(\delta Pi, 0)$ when the system is in a stationary state (all the derivatives are zeros

then the initial sign vector is $(\delta Pi, \delta Po, \delta Q, \delta Xs) = (+, 0, 0, 0)$. We will use this state-state vector notation when needed instead of an awkward notation of $p(\delta Pi, 0), z(\delta Po, 0), z(\delta Q, 0), z(\delta Xs, 0)$.

By the causal deduction, $p(\delta Q, N1)$ is first obtained(first step). Including this new state as the fact, we can then obtain $p(\delta Po, N2)$ by the causal deduction again(2nd step). Including this state as the new results and using third time fragment $dt > 0$, we obtain $n(\delta Xs, N3)$ by the causal deduction (3rd step).

3 Logical System and Dynamical System

In the previous section, we regarded the causality built-in the dynamical model as logical implication. Then, the dynamical state change can be carried out in a similar manner to deduce the new fact from the logical formulae corresponding to the dynamical model and the time fragment $p(dt, i)$. In order to use the causal relation in the dynamical model, the dynamical model must be original one. That is, the original dynamical model must reflect causal path between two physical entities.

In this section, we consider some correspondence between the important concepts in dynamical systems and those in logical systems.

Theorem 3.1(observability and deducibility)

The dynamical system is qualitative observable from a observer y iff the non-zero of the observer y can be deduced in the mapped logical system when the fact that some variables (corresponding to the dynamical system)are non-zero is given.

This result can be used to save some deduction processes when some variables are known to be observable or not. Further, this result can also used to investigate the qualitative stability which can be known by the observability of the system [Ishida 1989].

Definition 3.2 (completeness and soundness)

The mapped logical system is called complete (sound) if all the state which can (not) be attained by the corresponding dynamical system in the finite time can (not) be deduced in the finite number of steps.

Conjecture 3.3

The mapped logical system is always complete but not always sound.

This fact is often stated in qualitative reasoning, but not formally proved yet. Most formal discussion may be found in [Kuipers 1985,Kuipers 1986], stating that

> Each actual behavior of the system is necessarily among those produced by the simulation.

But,

> There are behaviors predicted by qualitative simulation which do not correspond to the behaviorof any system sarisfying the qualitative structure description.

We will see the example showing the lack of soundness of the mapped logical system in the next section. The lack of soundness is due to the following fact.

Proposition 3.4

Two equivalent dynamical systems may be mapped to the different logical systems.

That is, two dynamical systems which can be transformed to each other, may be mapped to the different logical systems. In fact, a dynamical system is usually mapped to a part of the exact logical system. Therefore, in order to make the mapped logical system close to the dynamical model, we must map from the multiple dynamical models which are equivalent as a dynamica model, and combine these mapped logical systems. We have not yet known what kinds of equivalent dynamica models suffice to make the mapped logical system exact.

4 Reasoning about State by Deduction

The causal deduction stated in the previous section cannot say anything as to changes when some time interva passed. That is, when many variables approaching to zero, which one reaches zero first. In order to determine this, meta-rules which are implicit in dynamical models must be explicitly introduced. The following axioms reflect the fact that the state-space of the dynamical models are continuous. The lack of continuous and dense space in the logical system is the fundamental points which discriminate logical systems from dynamical systems.

T-(1)

$$\forall x \forall i (p(x, i) \wedge n(dx/dt, i) \rightarrow \exists j ((j > i) \wedge z(x, j)))$$

T-(2)

$$\forall x \forall i (n(x, i) \wedge p(dx/dt, i) \rightarrow \exists j ((j > i) \wedge p(x, j)))$$

These axioms T-(1),(2) comes from value continuity rule stated in [de Kleer and Bobrow 1984]. This axiom T does not correctly reflect the world of dynamica model. Even if $x > 0$ and $dx/dt < 0$, x does not necessarily become zero in the finite or infinite time.

M-(1)

$$\forall x \forall j1 \forall j2(p(x,j1) \wedge n(x,j2) \wedge (j2 > j1)) \rightarrow$$

$$\exists j3(z(x,j3) \wedge (j3 > j1) \wedge (j2 > j3)))$$

M-(2)

$$\forall x \forall j1 \forall j2(n(x,j1) \wedge p(x,j2) \wedge (j2 > j1)) \rightarrow$$

$$\exists j3(z(x,j3) \wedge (j3 > j1) \wedge (j2 > j3)))$$

These axioms M-(1),(2) corresponds to the well-known *intermediate value theorem*, which reflects the continuity of the function x. Axioms T and M states the continuity of the state-space and that of the function from time to state-space. Other than axioms U,I,M,T, we need the following assumptions. That is, the state remains to be the same as the nearest past state unless otherwise deduced. We call this *no change assumption*. We could not formalize this assumption by a logical formula of our language so far. This seems to be a common problem to any formalization for reasoning about such dynamic concepts as state change, actions, and event. The situation calculus [McCarthy and Hayes 1969], for example, uses the *Frame Axioms*[1] to avoid this problem.

Example 4.1.

Let us consider the mass-spring system with friction [de Kleer and Bobrow 1984] (Fig. 2) whose model is of the form:

(4-1) $dx/dt = v$

(4-2) $dv/dt = -kx - fv$ where k and f are positive constants.

(4-2) is the original form containing the built-in causality whereas (4-1) is the definition of v.

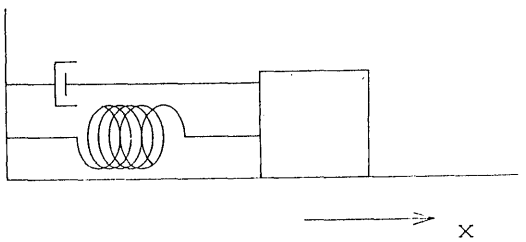


Fig.2 A Schematic Diagram of Mass-Spring System with Friction

As for the initial sign patterns of $(x, v, dv/dt)$, we consider only three cases; $(+,-,-),(+,+,-),(+,-,+)$. Let G_{dm} denote the set of logical formulae corresponding to the dynamical model, and G_{ch} those corresponding to the axioms U,I,T,M. The sign pattern $(+,+,+)$ and its opposite pattern $(-,-,-)$ are are inconsistent, since $(p(x,0) \wedge p(v,0)) \cup G_{dm} \rightarrow n(dv/dt,0)$. This result $n(dv/dt,0)$ is inconsistent with the initial pattern $p(dv/dt,0)$ under the uniqueness axiom U.

We do not consider the initial sign pattern which contains zero for any variables, since the pattern will change immediately to the sign pattern with only non zero patterns by axiom I. Thus these three patterns cover all the possible sign combinations.

We only show the deduction for the simulation of the case 1 when $p(x,0)$, $n(v,0)$, $n(dv/dt,0)$ are given as the initial pattern. Other cases can be deduced in a similar manner to this case 1 from the initial sign pattern, the set of logical formulae G_{dm} and G_{ch}. By the axiom T,

$$p(x,0) \wedge n(v,0) \rightarrow \exists N1((N1 > 0) \wedge z(x,N1)$$

By the *no change assumption*, other variables are assumed to remain the nearest past signs; that is

$$n(v,N1), n(dv/dt,N1).$$

However,

$$(n(v,N1) \wedge z(x,N1)) \cup G_{dm} \rightarrow p(x,N1).$$

Thus, we have

$$z(x,N1),\ n(v,N1),\ p(dv/dt,N1).$$

Then by the axiom M,

$$(N1 > 0) \wedge n(dv/dt,0) \wedge p(dv/dt,N1) \rightarrow \exists N2((N2 > 0) \wedge (N1 > N2) \wedge (z(dv/dt,N2))$$

By the *no change assumption*, other variables at time N2 are assumed to remain the nearest past signs; that is $p(x,N2)$, $n(v,N2)$. Since $n(v,N2) \wedge z(dv/dt,N2) \cup G_{dm} \rightarrow p(d^2v/dt^2,N2)$
and by the axiom I,

$$\mathrm{p}(\mathrm{d}^2v/dt^2,N2) \wedge z(dv/dt,N2) \wedge (N1 > N2) \rightarrow \exists N3((N1 > N3) \wedge (N3 > N2) \wedge p(dv/dt,N3)).$$

Again by the *no change assumption*,

$$p(x,N3), n(v,N3).$$

By applying the axiom I to the state at N1,

$$n(v,N1) \wedge z(x,N1) \rightarrow \exists N4((N4 > N1) \wedge p(x,N4)).$$

$n(v,N4)$ and $p(dv/dt,N4)$ are obtained by the *no change assumption*. By applying the axiom T to the

[1]Frame axioms are collection of statements that do not change when an action is performed.

state N4,

$n(v, N4) \wedge p(dv/dt, N4) \rightarrow \exists N5((N5 > N4) \wedge (z(v, N5))$.

Again signs of other variables at N5 remain to be the same as those at N4. By applying the axiom I to the state N5, we have

$z(v, N5) \wedge p(dv/dt, N5) \rightarrow \exists N6((N6 > N5) \wedge p(v, N6))$

In summary we have deduced, the set of the state at different time $p(x, N2)$, $n(v, N2)$, $z(dv/dt, N2)$, $p(x, N3)$, $n(v, N3)$, $p(dv/dt, N3)$, $z(x, N1)$, $n(v, N1)$, $p(dv/dt, N1)$,
$n(x, N4)$, $n(v, N4)$, $p(dv/dt, N4)$, $n(x, N5)$, $z(v, N5)$, $p(dv/dt, N5)$, $n(x, N6)$, $n(v, N6)$, $p(dv/dt, N6)$ and the order of time $(0 < N2 < N3 < N1 < N4 < N5 < N6)$. Tables 1 show the state transitions starting the initial patterns case 1, case2 and case3.

Tables 1 State Transition by Logical Deduction

case 1

t	x	dx/dt	d^2x/dt^2
0	+	-	-
1	+	-	0
2	+	-	+
3	0	-	+
4	-	-	+
5	-	0	+
6	-	+	+

At step 6, the opposite pattern of the initial pattern comes.

case 2

t	x	dx/dt	d^2x/dt^2
0	+	+	-
1	+	0	-
2	+	-	-

At step 2, the same pattern as the initial pattern of case 1 comes.

case 3

t	x	dx/dt	d^2x/dt^2
0	+	-	+
1	0	-	+
2	-	-	+

At step 2, the opposite pattern of the initial pattern of case 2 comes.

In the logical system mapper from the dynamical model (4-1) and (4-2), it is impossible to deduce the state which corresponds to the convergence to the point $(x, dx/dt, d^2x/dt^2) = (0, 0, 0)$ which is attained when infinite time passed in the dynamical model. In fact, we only have periodic states as shown in Tables 1. However, the infinite sequences of deduction similar to this convergence can be found. When the initial sign pattern is $(x, dx/dt, d^2x/dt^2, ...) = (+, -, +, ...)$, apply the axiom T to $n(dx/dt, 0)$ then we have

$\exists N1((N1 > 0) \wedge z(dx/dt, N1))$.

Then applying the axiom M to this result, we will have

$\exists N2((N1 > N2) \wedge z(d^2x/dt^2, N2))$.

This application of the axiom M progressively to any higer order time derivative of x. That is we have

$\exists Ni + 1((Ni > Ni + 1) \wedge z(d^{i+1}x/dt^{i+1}, Ni + 1))$.

This is an interesting corresponding between the dynamical model and the mapped logical systems. It may suggest to introduce some operations in the logical system (other than deduction) which corresponds to the operation $\lim_{t \rightarrow \infty} x(t)$.

We will show this convergence can be deduced even in the finite step using the logical implications mapped from a different (but equivalent) dynamical model. The dynamical model (4-1), (4-2) is equivalent to the dynamical model:

(4-3) $E = x^2 + 1/k * (dx/dt)^2$
(4-4) $dE/dt = -f(dx/dt)^2$

This states that E and hence x will eventually become zero as long as $f > 0$. Table 2 shows the state transition of the mapped logical system. This convergence of the dynamical system is attained in the infinite time, and hence need not be deduced in the mapped logical system. Since the current logical system does not have the concepts of convergence and infinite step, these concepts are out of scope of the mapped logical systems.

The results show that the logical system mapped from the dynamical model (4-3)(4-4) is quite differet from that mapped from the dynamical model (4-1)(4-2), althogh these dynamical models are equivalent. Therefore, this example shows the correctness of Proposition 3.3. This point is also fundamental difference between dynamica systems and logical systems.

Table 2 State Transition of Mass-Spring System(Energy Model)

t	x	dx/dt	E	dE/dt
0	*	*	+	-
1	0	0	0	0

* denotes any sign +, - 0.

5 Discussions

We first discuss the *temporal logic* with the temporal operators F,P [Rescher and Urquhart 1971], where FA (PA) means A will(was) be true at some(past) future time. With the axiom schemata, the feature of these temporal operators, and even the features of time (e.g. whether it is transitive, dense, continuity) can be characterized. However, since the logic does not tell anything about the feature of the state-space and the relation between the state-space and time, it is not possible to infer the change in the state-space. In fact, the axioms I, T, M given at section ? characterize the feature of the state-space. An alternative to our approach is to define the space operators similar to the time operators. One way of defining space operator follows:

F_x, P_x where $F_xA(P_xA)$ means that A is true at some point where x is larger(smaller) than the current value. With this definition, the previous time operator can be written as F_t, P_t.

With these space operators, the axioms I,T,M may be written as:

I-(1) z(x) $\rightarrow G_x(p(x))$
I-(2) z(x) $\rightarrow H_x(n(x))$
T-(1) p(x) $\rightarrow P_x(z(x))$
T-(2) n(x) $\rightarrow F_x(z(x))$
M-(1) p(x) $\wedge F_x(n(x)) \rightarrow F_x(F_x(z(x)))$
M-(2) n(x) $\wedge F_x(p(x)) \rightarrow F_x(F_x(z(x)))$

Since these axiom schemata I,T,M characterize the feature of only state-space itself, we need the following axioms TS which characterize the monotonic relation between time and state-space.

TS-(1) p(dx/dt) $\rightarrow ((FA \leftrightarrow F_xA) \wedge (PA \leftrightarrow P_xA))$
TS-(2) n(dx/dt) $\rightarrow ((PA \leftrightarrow F_xA) \wedge (FA \leftrightarrow P_xA))$

where $G_xA \leftrightarrow\sim F_x \sim A, H_xA \leftrightarrow\sim P_x \sim A$.

Here, the time operators are used instead of the time index for the sign predicates p,n,z. The good point of this space operator approach is that it can be discussed as a natural extension of temporal logic with temporal operators. However, its critical point is that although these space/time operators can tell the temporal precedence of the event but it cannot describe that the different event A, B occurred at the same time. In the approach taken in section 4, it is described by putting the same time tags.

When compared with the qualitative reasoning [de Kleer and Bobrow 1984], our way of qualitative reasoning is different from theirs in the following two points: (1) In reasoning; we defined another causality which refers to time strictly. Causal reasoning is carried out by mapping causality in dynamical models to the deduction under the condition of $dt > 0$. Time independent relations are mapped to only deductions. Then causa reasoning is done by requiring the facts $dt > 0$ in every step. This logical reasoning can be implemented on the the logical reasoning system such as prolog by providing axioms so far proposed and the mapped dynamical models. (2) In modeling; since we use the causality built in the dynamical model, we skip qualitative modeling process. That is, we use the dynamical model as qualitative model. However, the dynamical models must be carefully selected to insure the causal path in the dynamica models can be reflected on the mapped logical systems.

6 Conclusion

We discussed a mapping from dynamical systems to logical systems to see the correspondence of the fundamental concepts in these two domains, to implement the causal reasoning system on a logical deduction system. To clearly separate the physical causality from the usua deduction, we defined causality in physical system by making time explicit.

Many fundamental problems remains such as; whether or not the complete and sound logical system for a dynamical system exists ? If yes, how the complete and sound logical system can be attained?

References

[de Kleer and Brown 1984] de Kleer, J. and Brown, J.S. A qualitative physics based on confluences. *Artificia Intelligence* , 24, 7-83, 1984.

[Forbus 1984] Forbus, K. D. Qualitative process theory. *Artificial Intelligence* 24, 85-168, 1984.

[de Kleer and Bobrow 1984] de Kleer, J. and Bobrow, D. G. Qualitative Reasoning with Higher-Order Derivatives. *Proc. of AAAI 84* , 86-91, 1984.

[Kuipers 1986] Kuipers, B. Qualitative simulation. *Artificial Intelligence* 29, 289-337, 1986.

[Kuipers 1985] Kuipers, B. The Limits of Qualitative Simulation. *IJCAI 85*, pp.128-136, 1985.

[Ishida 1989] Ishida Y. Using Global Properties for Qualitative Reasoning: A Qualitative System Theory. *Proc. of IJCAI 89*,Detroit,pp.1174-1179, 1989.

[Struss 1988] Struss, P. Global filters for qualitative behaviors *Proc. of AAAI 88*, 301-306, 1988.

[Rescher and Urquhart 1971] Rescher, J. and Urquhart, A. *Temporal Logic* , Springer Verlag 1971.

[McCarthy and Hayes 1969] McCarthy, J. and Hayes, P.J. Some Philosophical Problems from the Standpoint of Artificial Intelligence, *Machine Intelligence 4*, Edinburgh University Press, 1969.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

The CLASSIC Knowledge Representation System or, KL-ONE: The Next Generation

Ronald J. Brachman Alexander Borgida* Deborah L. McGuinness
Peter F. Patel-Schneider Lori Alperin Resnick†
AT&T Bell Laboratories, 600 Mountain Ave.,
Murray Hill, NJ 07974-0636, U. S. A.

Abstract

CLASSIC is a recently developed knowledge representation (KR) system, based on a view of frames as structured descriptions, with several important inferable relationships, including description classification. While much about CLASSIC is novel and important in its own right, it is especially interesting to consider the system in light of its unusual (for Artificial Intelligence) intellectual history: it is the result of over a decade of research and evolution in representation systems that trace their origins back to work on KL-ONE, arguably one of the most long-lived and influential approaches to KR in the history of AI. We outline some of the novel contributions of CLASSIC, but pay special attention to its roots, illustrating the maturation of some of the original features of KL-ONE, and the decline and fall of others. A number of key ideas are analyzed—including the interpretation of frames as descriptions, the classification inference, and the role of a knowledge representation system in a knowledge-based application. The rare traceable relationship between CLASSIC and its ancestor gives us an opportunity to assess progress in a generation of knowledge representation research.

1 Introduction

An unfortunately large fraction of work in Artificial Intelligence is ephemeral, accompanied by much sound and fury, but, in the end, signifying virtually nothing. Work on systems with significant longevity to the basic ideas, such as STRIPS, appears to be the exception rather than the rule in AI.

In the area of knowledge representation (KR), there are ideas that have lived on for years, but very few systems or approaches have seen more than a minimal number of users for a minimal number of years.[1] The KL-ONE system [7, 11] is different: it was "born" over a dozen years ago, and has had continuous evolution and influence ever since. Its offspring now number at least twenty significant projects worldwide, all based directly on its key ideas of classification and structured inheritance. With well more than a decade behind us, this rich history bears closer examination, especially with the advent of the CLASSIC Knowledge Representation System, a recent development that clarifies and amplifies many of the central ideas that were more crudely approximated in the KL-ONE of 1978. CLASSIC goes substantially beyond KL-ONE in its treatment of individuals and rules, its clarification of subsumption and classification, its integration with its host language, and its concrete stand on the role of a KR system as a limited deductive database management system.

While a description of the CLASSIC system would be interesting in its own right, its motivation and contribution are more easily understood by placing it in the proper context. Thus, rather than describe the system in isolation, we here briefly explore some of its key properties in light of their intellectual debt to KL-ONE and its children. Besides making the case for CLASSIC, this will also provide us an opportunity to assess in retrospect the impact of some of the original ideas introduced by KL-ONE. This is a chance to see how far we have come in a "generation" of knowledge representation research.

2 KL-ONE: The First Generation

KL-ONE was the first implementation (*ca.* 1978) of a representation system developed in Brachman's thesis [7]. It was influenced in part by the contemporary Zeitgeist of "frames" (e.g., see [20]), with emphasis on structured objects and complex inheritance relationships. But KL-ONE's roots were really in semantic networks, and it had a network notation of labeled nodes and links.

Despite its appearance, in some key respects KL-ONE was quite different from both the semantic network systems that preceded it, and the frame systems that grew up as its contemporaries. Following papers by Woods [33] and Brachman [6], KL-ONE rejected the prevailing idea of an open-ended variety of (domain-specific) link- and node-names, and instead embraced a small, fixed set of (non-domain-specific) "epistemological primitives" [8] for constructing complex structured objects. These constructs—which represented basic general relationships like "defines-an-attribute-of" and "is-a-specialization-of," rather than domain-specific relationships like "owns" or "has-employee"—were considered to be at a higher level of representation than the data-structuring primitives used to implement them. They could be used as a foundation for building application-dependent conceptual models in a semantically meaningful way (rather than in the ad hoc fashion typical of semantic nets).

*Also with the Department of Computer Science, Rutgers University, New Brunswick, NJ.

†Electronic mail addresses: rjb@research.att.com, borgida@cs.rutgers.edu, dlm@research.att.com, pfps@research.att.com, resnick@research.att.com.

[1] SNePS and Conceptual Graphs are among the few exceptions.

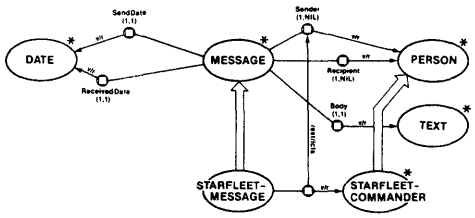


Figure 1: A KL-ONE Concept.

In addition to its clear stand on the semantics of semantic networks, the original KL-ONE introduced a number of important ideas, including these:

- rather than manipulating "slots"—which are in reality low-level data structures—KL-ONE looked at relationships as *roles* to be played; roles get their meanings from their interrelations—just like the roles in a drama—and they are not just meaningless labeled fields of records or indistinguishable empty bins into which values are dropped;
- a *role taxonomy*, which allowed roles to be subdivided into more specific roles; e.g., if `child` is a more specific role than `relative`, then being a child entails something more constrained than being a relative, but includes everything that being a relative in general does;
- *structural descriptions*, which served to define the relationships between role players; e.g., the difference between a `buyer` and a `seller` in a PURCHASE event would be specified by reference to other concepts that specified in which direction money and goods would flow. These concepts would give substance to the roles, rather than leaving their meanings open and subject only to human interpretation of strings like "buyer."
- *structured inheritance*, which reflected the fact that concepts (KL-ONE's name for frames/classes) were complex structured constructs and their parts were not independent items to be manipulated arbitrarily.

The KL-ONE language showed its semantic-network heritage rather directly, in that KL-ONE structures were drawn in diagrams, with different link-types being indicated with different pictorial realizations. For example, Figure 1 illustrates a typical KL-ONE concept: the "`STARFLEET-MESSAGE`" concept uses its parent, "`MESSAGE`," to create the description corresponding to "a `MESSAGE` whose `Sender` is a `STARFLEET-COMMANDER`." In general, a user built a KL-ONE net like this by calling rather low-level LISP functions, whose actions might be to "create a role node" or "add a superconcept link."

After a number of years of use and reimplementation, it gradually became clear that KL-ONE's approach to structured objects was substantially different than that of virtually all of its contemporary systems. The primary realization was that those objects had previously been used for (at least) two purposes [6, 9]: (1) to represent *statements*, usually of some typical properties (e.g., "elephants are gray"), and (2) to act as *structured descriptions*, somewhat like complex mathematical types (e.g., "a black telephone," rather than "all telephones are black"). In the KL-ONE community, the structured-description aspect came to be emphasized over the assertional one.

Viewing frames as descriptional, rather than assertional, emphasized the *intensional* aspects of knowledge representation. This had one primary benefit: it yielded the idea that the central inference to be drawn was *subsumption*—whether or not one description is necessarily more general than another. Subsumption in turn led to the idea of description *classification*—taking a description and finding its proper place in a partial order of other descriptions, by finding all subsuming (more general) descriptions and all subsumed (more specific) descriptions. KL-ONE-based classification systems were subsequently used in a number of interesting applications, including natural language understanding [11], information retrieval [27], expert systems [22], and more. Because of this view of frames, the research foci in the KL-ONE family gradually diverged somewhat from those of other frame projects, which continued to emphasize typicality and defaults.

Another key issue in the KL-ONE community has been the tension between the need for expressiveness in the language and the desire to keep implementations computationally reasonable. Two somewhat different approaches can be seen: NIKL [17], and subsequently LOOM [19], added expressive power to the original KL-ONE language, and admitted the possibility of *incomplete* classification. KRYPTON [12], and subsequently KANDOR [26], on the other hand, emphasized computational tractability and completeness. While neither of these approaches is right for every situation, they provide an interesting contrast and highlight a significant current issue in knowledge representation. This topic is still under active exploration (see Sections 4.5–4.6).

Over the last decade, systems based on the ideas in KL-ONE have proliferated in the United States and Europe (with significant ESPRIT funding), with at least twenty related efforts currently underway (see [34]). The work has also inspired seven workshops, two recently being held in Germany (in 1991) and one coming soon in the US (1992). These workshops have attracted both theoretical and practical scientists from several countries, and made it clear that the class of "KL-ONE-like" representation systems has both important theoretical substance and practical impact.

3 The CLASSIC System

The CLASSIC Knowledge Representation System[2] represents a new generation of KL-ONE-like systems, emphasizing simplicity of the description language, a formal approach, and tractability of its inference algorithms. In this regard, it is most like KANDOR (and also BACK [32]), which, while setting important directions for limited subsumption-based reasoning, had a number of inadequacies. However, the CLASSIC system goes significantly

[2]CLASSIC stands for "CLASSification of Individuals and Concepts." It has a complete, fully documented implementation in Common Lisp, and runs on SUN workstations, Apple Macintoshes, Symbolics Machines, etc. It has been distributed to numerous (> 40) universities for research use.

beyond previous description-based KR systems in many important respects, including its language, integration with the host system, treatment of individuals, and clarity on the role of a KR system.

In CLASSIC's language, there are three types of objects:

- *concepts*, which are descriptions with potentially complex structure, formed by composing a limited set of *description-forming constructors*; concepts correspond to one-place predicates;
- *roles*, which are simple formal terms for properties; roles correspond to two-place predicates; within this class, CLASSIC distinguishes *attributes*, which are functional, from *multi-roles*, which can have multiple fillers;
- *individuals*, which are simple formal constructs intended to directly represent objects in the domain of interest; individuals are given properties by asserting that they are described by concepts (e.g., "`Chardonnay` is a `GRAPE`") and that their roles are filled by other individuals (e.g., "`Bell-Labs' parent-company` is `AT&T`").

The CLASSIC description language is uniform and compositional—the meaning of a complex description is a simple combination of the meanings of its parts.[3] The complete description language grammar in Figure 2 illustrates its simplicity. Besides the description language, the interface to CLASSIC has a small number of operators on knowledge bases for the creation of new concepts (and the assignment of names to them), which include *defined concepts*, with full necessary and sufficient conditions; *primitive concepts*, which have only necessary conditions (see [9]); and *disjoint* (*primitive*) *concepts*, which cannot share instances (e.g., `MALE` and `FEMALE`). There is also an operator to explicitly "close" a role; this makes the assertion that there can be no more fillers for the role (see below).

It is important to emphasize that the description constructors and knowledge base operators were chosen only after careful study and extensive experience with numerous KR systems. For example, virtually every object-centered representation system has a way to restrict the type of an attribute; this yields our **ALL** constructor. All KR languages need to assert that a role is filled by an object; this corresponds to **FILLS**. CLASSIC's set captures the central core of virtually all KL-ONE-like systems in an elegant way: the constructors are minimal, in that one can not be reduced to a combination of others; and they have a uniform, prefix notation syntax, which allows them to be composed in a simple and powerful way. Rules (see Sec. 4.4), procedural tests, numeric ranges (**MAX**, **MIN**) and host language values expand the scope of KL-ONE-like concepts; these were included after clear user need was demonstrated. Certain more complex operators were excluded because they would have clearly made inference intractable or undecidable. Thus, CLASSIC's language is arguably the cleanest structured description language that tempers expressiveness of descriptions with tractability of inference (but see Section 4.5), elegantly balancing representational needs and inferential constraints in a uniform, simple, compositional framework.

[3] CLASSIC has a formal semantics, but we will not be able to elaborate on it here. See [4].

CLASSIC has many novel features, and improves on its predecessors in a number of ways, one of the most telling of which is its treatment of individuals. Anything that can be said about a concept can be said about an individual; thus, *partial knowledge* about individuals is maintained and used for inference. For example, we can assert that a person has at least three children (`(`**`AT-LEAST`** `3 child)`) without identifying them, or that all of the children—whoever they are—are female (`(`**`ALL`** `child FEMALE)`). Individuals from the host language (e.g., LISP), such as strings and numbers, can be freely used where CLASSIC-supported individuals can, with consistent treatment. When any individual is added or augmented, or when a new concept is defined, complete propagation of properties is carried out, so that all individuals are continuously classified properly, and monotonic updates are treated completely. The role-fillers of an individual are not considered under the usual closed-world assumption; this better supports the accumulation of partial knowledge about individuals. Roles can be "closed" explicitly when all of their fillers are known. Most crucially, an individual cannot be proven to satisfy an **ALL** restriction or an **AT-MOST** restriction by looking at its fillers for the role unless all of those fillers are known. Previous systems either treated this aspect of assertions incompletely or incorrectly.

Rather than delve further into CLASSIC's individual features, we will attempt to better articulate its more general contributions by examining its relation to the issues that started this whole line of thinking over a decade ago. In that respect we can not only appreciate gains made in CLASSIC, but understand the strengths and weaknesses of the original KL-ONE proposals.

4 Key Intellectual Developments

CLASSIC is innovative in a number of ways, and bears little surface resemblance to KL-ONE. But it is also very much a descendant of that system, which introduced a number of key ideas to the knowledge representation scene. While we will not have an opportunity here to delve into all of these ideas, we will examine a few of the more important issues raised by the original system and its successors.

4.1 Subsumption as a Central Inference

In KL-ONE, as in all semantic networks that preceded it (and most systems to follow), the backbone of a domain representation was an "IS-A" hierarchy. The IS-A ("superc" in KL-ONE) link served to establish that one concept was a subconcept of another, and thus deserved to inherit all of the features of the superconcept. Virtually all of these systems forced the user to state directly that such a link should be placed between two explicitly named concepts. This type of user responsibility is still common in virtually all frame-based systems and expert system shells.

In the early 1980's we discovered that in a classification-based system this was the wrong way around. In the KL-ONE-descendant languages of KRYPTON and KANDOR, where the meaning of a concept could be determined simply and directly from its structure (be-

```
<concept-expression> ::=  THING | CLASSIC-THING | HOST-THING | <concept-name> |        built-in names
                          (AND <concept-expression>+) |                                 conjunction
                          (ALL <role-expression><concept-expression>) |                 universal value restriction
                          (AT-LEAST <positive-integer><role-expression>) |              minimum cardinality
                          (AT-MOST <non-negative-integer><role-expression>) |           maximum cardinality
                          (FILLS <role-expression> <individual-name>+) |                role-filling
                          (SAME-AS <attribute-path><attribute-path>) |                  role-filler equality
                          (TEST-C <fn><argument>*) |                                    test (CLASSIC concept)
                          (TEST-H <fn><argument>*) |                                    test (HOST concept)
                          (ONE-OF <individual-name>+) |                                 set of individuals
                          (MIN <number>) | (MAX <number>)                               numeric range limits
<individual-expression> ::= <concept-expression> | <individual-name>
<concept-name> ::=        <symbol>
<individual-name> ::=     <symbol> | <string> | <number> | '<COMMONLISP-expression>
<role-expression> ::=     <multi-role-name> | <attribute-name>
<attribute-path> ::=      (<attribute-name>+)
<fn> ::= a function in the host language (COMMON LISP) with three-valued logical return type
```

Figure 2: The CLASSIC Description Language (comments in italics).

cause the logic had a compositional semantics and necessary and sufficient definitions), it became clear that IS-A relations were *purely derivative* from the structure of the concepts. In other words, the subsumption relation[4] between two descriptions was determined without any need for a complete explicit hierarchy of IS-A connections.

Of course, it might make a difference to the efficiency of the system if all subsumption relationships that had been calculated were cached in some kind of structure that obviated the need to compute them a second time, and this is now common practice. But in a system like CLASSIC, it is clear that this is strictly an efficiency issue.

In essence, systems that force a user to think only in terms of direct IS-A links place the entire burden of knowledge structuring on that user. Since every IS-A assertion is taken at its word, the system can provide no feedback that the correct relationship has been represented; all responsibility is the user's. On the other hand, the CLASSIC system (and others like it) can reliably decide under which concepts a new concept or individual must fit, since it has a compositional interpretation of the parts of any concept. This provides valuable help to the user in structuring large knowledge bases, because it is all too easy for us to assume that just because *we* know something that a term (e.g., a complex concept, like RED-WINE) implies, the system will know it as well. This advantage has been documented in the LASSIE system [14], which uses classification to support a software information system. Systems that do not do classification do not have defined concepts, and therefore treat everything as primitive [9]. Thus we can be falsely lulled into assuming that when we assert that a particular WINE has color = Red, the system will know that it is a RED-WINE; but a non-classification system will *not* make that inference.[5]

4.2 From LISP Functions to Languages

The realization that the structure of a concept is the only source of its meaning, and that any IS-A hierarchy is *induced* by such structures, leads to another significant point of departure for the CLASSIC system. CLASSIC has a true knowledge representation *language*—a grammar of expressions. KL-ONE and even many of its successors treated a knowledge base as a set of data structures to be more or less directly manipulated by a user, and thus the user interface was strictly in terms of node- and link-managing functions. Instead (following KRYPTON) CLASSIC is really based on a formal *logic*, with a formal syntax, rules of inference, and a formal interpretation of the syntax (see [4]).

Of all of the KL-ONE-like systems, the CLASSIC system has the cleanest language. As shown in Figure 2, the language is simple, uniform, and compositional. Figure 3 illustrates the difference in style between KL-ONE structures and the lexical language of the CLASSIC system.[6] The advantages of a true logic over a set of data-structure-manipulating programs should be obvious: one can write parsers and syntax checkers for the language, formal semantics can be specified, inference mechanisms can be verified to adhere to the semantics, etc.

4.3 Attached Procedures

One of the more popular features of the early frame systems was the ability to "attach" programs to pieces of the data structures. The ultimate incarnation of this idea was probably KRL [3], which had an elaborate process framework, including "servants," "demons," "traps," and "triggers." The program fragments could be invoked at various times, and cause arbitrary computations to occur. KL-ONE had its own elaborate procedure attachment and invocation framework. However, arbitrary access to LISP meant that KR systems with this feature ceded control completely to the user—an at-

[4]Subsumption is defined formally in [18] and [4]. Concept a subsumes concept b iff instances of b are instances of a in all possible interpretations.

[5]Note that CLASSIC and its cousins all do normal inheritance of properties. Most of these systems are strictly monotonic for simplicity, but LOOM [19] has a default component.

[6]The symbols $\sqsubseteq$ and $\doteq$ indicate a *primitive concept specification* and a *defined concept specification*, respectively. The KL-ONE community has developed an algebraic notation that includes operators like these for all constructs in CLASSIC and related languages.

```
MESSAGE ⊑ (AND (AT-LEAST 1 sender)
               (ALL sender PERSON)
               (AT-LEAST 1 recipient)
               (ALL recipient PERSON)
               (AT-LEAST 1 body)
               (AT-MOST 1 body)
               (ALL body TEXT))
PRIVATE-MESSAGE ≐ (AND MESSAGE
                       (AT-MOST 1 recipient))
```

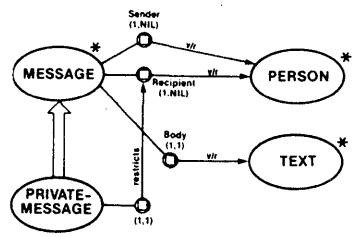


Figure 3: CLASSIC Expressions and KL-ONE Diagrams (adapted from [11]).

tached procedure could alter any data structure in any way at any time. The semantics of KL-ONE networks and other frame systems thus became very hazy once attached procedures were utilized.

In CLASSIC, we have invented an important way to control the use of such "escape hatches." Through the notion of the **TEST-C** and **TEST-H** constructors, we have isolated the use of procedures in the host language to testing predicates. As one can see from the grammar, such concepts are treated syntactically uniformly with other concepts. The procedure simply provides a *primitive sufficiency condition* for the concept—it will be invoked only when trying to recognize an instance. These test functions are particularly useful when trying to relate individuals from the host language, such as when two roles are filled with numbers, and one should be a multiple of another. In their use, the user agrees to avoid side-effects and to use only monotonic procedures (i.e., those whose value never changes from true to false or *vice versa* in the presence of purely monotonic updates). While under arbitrary circumstances, resorting to program code for tests renders the semantics of the language useless, in CLASSIC, if the user abides by this "contract," the semantics of concepts with tests is manageable, and the inferences that the system draws are still guaranteed to be sound. Indeed, tests work just like other restrictions on concepts as far as classification of individuals goes, but since the procedures are inscrutable they have the flavor of primitive concepts. While primitive concepts allow primitive necessary conditions, tests give us primitive sufficient conditions.

Another innovation in CLASSIC is the requirement that the test functions must be 3-valued. If a system like CLASSIC says that an individual does not satisfy a concept, then that means only that it cannot be currently *proven* to do so. A complementary question can still be asked—whether it can be proven that the individual could *never* satisfy the description (i.e., that it is disjoint from the concept). For example, if `Fred` has exactly one child (i.e., **(AND (AT-LEAST** `1 child`**) (AT-MOST** `1 child`**))**), but nothing is known about it yet, then he cannot be proven to satisfy the description **(ALL** `child FEMALE`**)**. But it is possible that at a later time he could be, if he were stated to have a known female child. On the other hand, if it were asserted that his child was `Barney`, who was known to be a `MALE`, and `MALE` and `FEMALE` were disjoint concepts, then it would be provable that `Fred` could never satisfy the description. Thus, in order to fit into the classification framework, procedural tests must provide the same facility—to differentiate between a guarantee never to satisfy a description and lack of ability to prove it given the current knowledge base.

4.4 Definitions, Assertions, Individuals

As mentioned, KL-ONE ultimately distinguished itself from other frame languages by its emphasis on structured descriptions and their relationships, rather than on contingent and typical facts. At one point in its development, the system was in a strange state: there were facilities for building complex concepts, but none for actually using them to describe individual objects in the domain. "Individual concepts" were KL-ONE's initial attempt to distinguish between generic class descriptions and descriptions that could apply only to single individuals. As it turned out, these were typically misused: an individual concept with two parent concepts could only really mean a conjunctive description. One example that was used often was the conjunction of `DRIVING-IN-MASSACHUSETTS` and `HAZARDOUS-ACTIVITY`, intended to express the fact that driving in Massachusetts is hazardous. However, in truth the concept including them both was just a compound concept with no assertional force at all.

While KL-ONE initially correctly distinguished between the import of different links between concepts, it failed to distinguish between those and a link that would make a contingent assertion about some individual. Eventually an alternative mechanism was proposed—the "nexus," to stand for an individual—but this was never really used. In the end, it took the work on KRYPTON to get this right. In KRYPTON, it was proposed that *terminological* knowledge (knowledge about the structure of descriptions) and *assertional* knowledge (facts) are two complementary aspects of knowledge representation competence, and that they should be maintained by distinct components, with an appropriate logical connection between them. From this distinction arose the terms "TBox" and "ABox," which are used extensively in the KL-ONE community to refer to the two components.

But KRYPTON went too far in another direction, integrating an entire first-order logic theorem-prover as its assertional component. The CLASSIC system makes what we think is a better compromise: it has a limited object-centered logic that properly relates descriptions and individuals. As is apparent from the grammar, CLASSIC treats assertions about individuals in a parallel and uniform manner with its treatment of the formation of subconcepts; but it also carefully distinguishes the logical meaning of the different relationships. Thus, for example, while individuals can be used in concept value restrictions (i.e., in a **ONE-OF** expression, e.g., **(ALL** `wine-color` **(ONE-OF** `Red White Blush`**))**), no contingent property of an individual can be used in determining subsumption between two concepts (e.g., if `White` just happens to be my favorite color for a wine, that fact cannot be used in any subsumption inference).

As mentioned, CLASSIC also supports the propagation of information between individuals. If we assert that some individual is described by a complex description (e.g., that Rebecca is a PERSON whose mother is a DOCTOR), then that may imply some new properties about other related individuals (e.g., we should assert that Rebecca's mother, if known, is a DOCTOR). Such propagated properties can in turn cause other properties to propagate (e.g., that Rebecca's mother's office is a DOCTOR'S-OFFICE).[7] This type of inference was never handled in KL-ONE, and only partially handled in some of its successors. Note that as soon as a property propagates from one individual to another, the latter individual might now fall under some new descriptions. CLASSIC takes care of this *re-classification* inference as well (as well as any further propagations that result, etc.).

The CLASSIC system has two other features along these lines that distinguish it from its predecessors. First, the previously mentioned apparatus does not allow the expression of general contingent rules about individuals. Thus, given only what is in the CLASSIC concept grammar, while we could form the concept of, for example, a LATE-HARVEST-WINE, we could not assert that all LATE-HARVEST-WINEs are SWEET-WINEs. The sweetness is a derivative property—it is not part of the *meaning* of LATE-HARVEST-WINE, but rather a simple contingent property of such wines. In CLASSIC, one can also express general rules of a simple form. A rule has a named concept as the left-hand side, with an applicability condition (filter) that limits the rule's firing to the desired subcases (i.e., if x is a $<concept_a>$ with property $<filter>$, then x is a $<concept_b>$). These rules are used only in reasoning about individuals, and do not affect subsumption relationships.[8]

Most KL-ONE-like systems were unclear about the status of individuals that could easily be expressed in the host implementation language (i.e., numbers and strings in LISP). CLASSIC integrates such individuals in a simple and uniform way, and makes it virtually transparent whether an individual is implemented directly in the host language, or in the normal complex structure for CLASSIC individuals. This aspect of CLASSIC has proven critical in applications that deal with real data (for example, from a database), as in [29].

4.5 KR and Computational Complexity

Once it was apparent that the clearly defined logical relationship of subsumption was central to the KL-ONE family, a new factor could be introduced to the analysis of frame-based knowledge representation systems. In 1984, Brachman and Levesque gave a formal analysis of the complexity of computing subsumption in some frame languages [10]. That analysis showed that the apparent simplicity of some frame languages could be deceptive, and that the crucial subsumption inference was co-NP-hard. The original paper initiated a sequence of results on the complexity of computing in the KL-ONE family, culminating most recently in two that show that the original language is in fact *undecidable* [24, 28].

This line of analysis has caused some major rethinking of the knowledge representation enterprise. No longer can we view language features as simply providing more expressiveness (which was the common view in the early years of knowledge representation). Rather, as in other areas of computer science, we must consider how expensive it will be to add a feature to a language. The addition of new features may demand the excision of some others in order to maintain computational manageability, or the system must be clear on where it is incomplete. In CLASSIC, subsumption is complete and tractable, but with respect to a slightly non-standard semantics; that is, it is clear what CLASSIC computes, and how fast it can compute it, but it does not compute all the standard logical consequences of a knowledge base. In this regard, we have opted for a less conservative approach than in KANDOR, but a more limited and disciplined approach than in LOOM. The consequences of this are explored briefly in the next section. We should point out that the viability of our approach has been proven in practice: CLASSIC is the first KL-ONE-derived system to be deployed in a fielded (AT&T proprietary) product, used every day in critical business operations. It was expressive enough to do the job.

4.6 The Role of a KR System

The above developments in the KL-ONE saga give rise to an important general question that usually goes unasked in AI: what role is a knowledge representation system expected to play? There are clearly different approaches here. On one extreme we have the large commercial systems, or expert system shells, which include substantial knowledge representation apparatus. The philosophy of those systems seems to be that a KR system should provide whatever apparatus is necessary to support virtually *any* AI application. In that regard, such systems are like very powerful programming languages, with complex data-structuring facilities.

But this is definitely not the only approach, and in many respects its requirements are overly demanding. Given the kind of complexity results mentioned above, users of such powerful systems must be very careful in "programming" their KR tools: predicting when a computation will return is difficult or impossible in a very expressive logic.

In many contexts (but not all, of course), it may be appropriate for a knowledge representation system to act in a more constrained fashion, rather like the database component of an application system. This is the point of view explicitly espoused in CLASSIC. Users cannot expect to program arbitrary computations in CLASSIC, but in return they get predictable response time and clear semantics. The burden of programming an application, such as a medical diagnostician, must be placed on some other component of the overall system. Since most KR systems attempt to be application-independent, it is ap-

[7] In order to keep the complexity down, CLASSIC only propagates properties to known individuals. Thus, if Rebecca's mother were unknown, the system would not attempt to create an individual about which to assert the DOCTOR description. If it did, it would then have to do very complex reasoning about existentials.

[8] Some of the newer KL-ONE-derivatives, such as LOOM, have developed similar rule mechanisms.

propriate for them not to be asked to provide general diagnostic, planning, or natural language-specific support. What is gained in return for certain limitations (and this in part accounts for the appeal of databases) is a system that is both complete with respect to an intuitive and simple semantic model and efficient to use.

Failure to acknowledge this general issue has been a source of difficulty with knowledge representation systems in AI. KL-ONE, uniformly with its contemporary KR systems (and subsequently NIKL), never really took a stand as to the role it should play. This has resulted, for example, in a pair of recent critiques of NIKL [15, 30], for failing to live up to a promise it perhaps was never intended to make. With CLASSIC, on the other hand, we expect to provide a powerful database service, but with limited deductive and programming support. This is a unique kind of database service, as it is both deductive and object-oriented (see [5]). But nevertheless it is firmly limited. To use the CLASSIC system in the context of an expert system, for example, it would be appropriate to use it as a substitute for working memory in a rule-based programming system like OPS5, not for all computation to be done by the overall system. Several recent applications ([14], [29], [23], and others) have shown convincingly that this approach, while not satisfying all needs for all applications, is quite successful in important cases.

5 Perspective

While CLASSIC is a "KL-ONE-like" system, it differs in so many ways from the original that it must be treated in its own right. While KL-ONE began the thinking on numerous key issues, it has taken us until CLASSIC to begin to truly understand many of them. Among its virtues, the CLASSIC Knowledge Representation System

- isolates an important set of language constructs, distilled from many years of use of frame representations, and knits them together in an elegant, straightforward language with a compositional interpretation; novel language features include enumerated sets of individuals treated in a uniform manner with other concepts (**ONE-OF**), and limited generic equalities between role fillers (**SAME-AS**);
- treats individuals in a more complete way than its predecessors, supporting propagation of facts and reclassification of individuals;
- allows contingent universal rules that are automatically applied, with the affected individuals being reclassified and any derived facts being propagated;
- offers tight, uniform integration of individuals from the host language, including numeric range concepts (**MAX**, **MIN**);
- offers a facility for writing procedural 3-valued tests as primitive sufficiency conditions, and integrates such tests into the language and semantics in a clean way.[9]

[9] CLASSIC also allows retraction of any asserted fact, with full dependency maintenance, but we have not had room to discuss this here.

CLASSIC offers these facilities in the context of complete computation of subsumption, while remaining computationally tractable. The CLASSIC system can be thought of as a limited, deductive, object-oriented database management system as well as a knowledge representation system, and has been used to support several real-world applications.[10]

In this discussion, we have limited ourselves to considering the KL-ONE family and its contributions. Related work involving manipulation of types and their relations can be found in programming language research, in some semantic data modeling work, and in feature logics in support of (among other things) natural language processing. We do not have room to draw comparisons with this other work, but in general it is clear that the bulk of that work does not include classification and description-processing of the sort found so prevalently in KL-ONE-like systems. Recent work in some of these areas does bear a strong relationship to ours, but not by accident: work on KL-ONE and its descendants has had direct influence, for example, on LOGIN [1] (a programming language), CANDIDE [2] (a DBMS), and feature logics [21].

There are still, of course, many open questions yet to challenge CLASSIC and its relatives. Technically, the notion of a "structural description," introduced by KL-ONE, has still not been treated adequately (although the **SAME-AS** construct provides a limited form of relationship between roles). And there are important computational questions to be answered so that CLASSIC can handle significant-sized databases, involving persistence of KB's, automatic loading of data from conventional DBMS's, and complex query processing.

But perhaps chief among the remaining research questions is how exactly to cope with the tradeoff we are forced to make between expressive power and computational tractability. Is it even possible to provide the kind of knowledge representation and inference services demanded by AI applications in a computationally manageable way? The CLASSIC Knowledge Representation System has provided convincing evidence that this is possible at least for a limited set of applications, but it is but one point in a large space of possibilities that we are still mapping out, after more than a dozen years of research inspired by KL-ONE.

References

[1] Aït-Kaci, H., and Nasr, R. "LOGIN: A Logic Programming Language with Built-in Inheritance," *Journal of Logic Programming*, **3**:187–215, 1986.

[2] Beck, H. W., Gala, S. K., and Navathe, S. B. "Classification as a Query Processing Technique in the CANDIDE Data Model," *Proc. Fifth Intl. Conf. on Data Engineering*, Los Angeles, 1989, pp. 572–581.

[3] Bobrow, D. G., and Winograd, T. A., "KRL, A Knowledge Representation Language," *Cognitive Science* **1**(1), 1977, pp. 3–46.

[10] One testimony to the success of CLASSIC's clean and simple approach is the fact that a group from the University of Calgary has simply picked up a written description of the system and quickly implemented their own version as a C++ library to support their work in knowledge acquisition [16].

[4] Borgida, A., and Patel-Schneider, P. F., "A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic," unpublished manuscript, AT&T Bell Laboratories, Murray Hill, NJ, 1992. Submitted for publication.

[5] Borgida, A., Brachman, R. J., McGuinness, D. L., and Resnick, L. A., "CLASSIC: A Structural Data Model for Objects," *Proc. 1989 ACM SIGMOD Intl. Conf. on Management of Data*, Portland, Oregon, June, 1989, pp. 59–67.

[6] Brachman, R. J., "What's in a Concept: Structural Foundations for Semantic Networks," *Intl. Journal of Man-Machine Studies*, **9**(2), 1977, pp. 127–152.

[7] Brachman, R. J., "A Structural Paradigm for Representing Knowledge," Ph.D. Thesis, Harvard University, Division of Engineering and Applied Physics, 1977. Revised as *BBN Report No. 3605*, Bolt Beranek and Newman, Inc., Cambridge, MA, May, 1978.

[8] Brachman, R. J., "On the Epistemological Status of Semantic Networks." In *Associative Networks: Representation and Use of Knowledge by Computers.* N. V. Findler (*ed.*). New York: Academic Press, 1979, pp. 3–50.

[9] Brachman, R. J., " 'I Lied about the Trees,' or, Defaults and Definitions in Knowledge Representation," *AI Magazine*, Vol. 6, No. 3, Fall, 1985.

[10] Brachman, R. J., and Levesque, H. J., "The Tractability of Subsumption in Frame-Based Description Languages," *Proc. AAAI-84*, Austin, TX, August, 1984, pp. 34–37.

[11] Brachman, R. J., and Schmolze, J. G., "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, **9**(2), April–June, 1985, pp. 171–216.

[12] Brachman, R. J., Fikes, R. E., and Levesque, H. J., "Krypton: A Functional Approach to Knowledge Representation," *IEEE Computer*, Vol. 16, No. 10, October, 1983, pp. 67–73.

[13] Brachman, R. J., McGuinness, D. L., Patel-Schneider, P. F., Resnick, L. Alperin, and Borgida, A. "Living with CLASSIC: How and When to Use a KL-ONE-like Language." In *Principles of Semantic Networks.* J. Sowa (*ed.*). San Mateo, CA: Morgan Kaufmann, 1991, pp. 401–456.

[14] Devanbu, P., Brachman, R. J., Selfridge, P. G. and Ballard, B. W., "LaSSIE: A Knowledge-Based Software Information System," *CACM*, Vol. 34, No. 5, May, 1991, pp. 34–49.

[15] Doyle, J., and Patil, R. S., "Two Theses of Knowledge Representation: Language Restrictions, Taxonomic Classification, and the Utility of Representation Services," *Artificial Intelligence*, Vol. 48, No. 3, April, 1991, pp. 261–297.

[16] Gaines, B. R., "Empirical Investigation of Knowledge Representation Servers: Design Issues and Applications Experience with KRS," *SIGART Bulletin*, Vol. 2, No. 3, pp. 45–56.

[17] Kaczmarek, T. S., Bates, R., and Robins, G., "Recent Developments in NIKL," *Proc. AAAI-86*, Philadelphia, PA, 1986, pp. 978–985.

[18] Levesque, H. J., and Brachman, R. J., "Expressiveness and Tractability in Knowledge Representation and Reasoning," *Computational Intelligence*, Vol. 3, No. 2, Spring, 1987, pp. 78–93.

[19] MacGregor, R. M., "A Deductive Pattern Matcher," *Proc. AAAI-87*, St. Paul, MN, pp. 403–408.

[20] Minsky, M., "A Framework for Representing Knowledge." In *The Psychology of Computer Vision.* P. H. Winston (*ed.*). New York: McGraw-Hill Book Company, 1975, pp. 211–277.

[21] Nebel, B., and Smolka, G., "Attributive Description Formalisms ... and the Rest of the World." In *Text Understanding in LILOG.* O. Herzog and C.-R. Rollinger (*eds.*). Berlin: Springer-Verlag, 1991, pp. 439–452.

[22] Neches, R., Swartout, W. R., and Moore, J., "Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development," *Proc. IEEE Workshop on Principles of Knowledge-Based Systems*, Denver, CO, 1984, pp. 173–183.

[23] Nonnenmann, U., and Eddy, J. K., "Knowledge-Based Functional Testing for Large Software Systems," *Proc. FGCS-92, Intl. Conf. on Fifth Generation Computer Systems*, Tokyo, June, 1992.

[24] Patel-Schneider, P. F., "Undecidability of Subsumption in NIKL," *Artificial Intelligence*, Vol. 39, No. 2, June, 1989, pp. 263–272.

[25] Patel-Schneider, P. F., "A Four-Valued Semantics for Terminological Logics," *Artificial Intelligence*, Vol. 38, No. 3, April, 1989, pp. 319–351.

[26] Patel-Schneider, P. F., "Small can be Beautiful in Knowledge Representation," *Proc. IEEE Workshop on Principles of Knowledge-Based Systems*, Denver, CO, December, 1984, pp. 11–16.

[27] Patel-Schneider, P. F., Brachman, R. J., and Levesque, H. J., "ARGON: Knowledge Representation meets Information Retrieval," *Proc. First Conf. on Artificial Intelligence Applications*, Denver, CO, December, 1984, pp. 280–286.

[28] Schmidt-Schauss, M., "Subsumption in KL-ONE is Undecidable," *Proc. KR '89: The First Intl. Conf. on Principles of Knowledge Representation and Reasoning*, Toronto, May, 1989, pp. 421–431.

[29] Selfridge, P. G., "Knowledge Representation Support for a Software Information System," *Proc. Seventh IEEE Conf. on Artificial Intelligence Applications*, Miami Beach, FL, February, 1991, pp. 134–140.

[30] Smoliar, S. W., and Swartout, W., "A Report from the Frontiers of Knowledge Representation," Technical Report, USC Information Sciences Institute, Marina del Rey, CA, 1988.

[31] Vilain, M., "The Restricted Language Architecture of a Hybrid Representation System," *Proc. IJCAI-85*, Los Angeles, 1985, pp. 547–551.

[32] von Luck, K., Nebel, B., Peltason, C., and Schmiedel, A., "The Anatomy of the BACK System," *KIT Report 41*, Technical University of Berlin, January, 1987.

[33] Woods, W. A., "What's in a Link: Foundations for Semantic Networks." In *Representation and Understanding: Studies in Cognitive Science.* D. G. Bobrow and A. M. Collins (*eds.*). New York: Academic Press, 1975, pp. 35–82.

[34] Woods, W. A., and Schmolze, J. G., "The KL-ONE Family," to appear in *Computers and Mathematics with Applications*, Special Issue on Semantic Networks in Artificial Intelligence.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Morphe: A Constraint-Based Object-Oriented Language Supporting Situated Knowledge

Shigeru Watari, Yasuaki Honda, and Mario Tokoro*
e-mail: {watari, honda, mario}@csl.sony.co.jp
Sony Computer Science Laboratory Inc.
3-14-13 Higashi-Gotanda, Shinagawa-ku, Tokyo 141, Japan

Abstract

This article introduces Morphe, a programming language aimed to support construction of open systems. In open systems, the programmer cannot completely anticipate the future use of his programs as components of new environments. When independently developed systems are integrated into an open system, we eventually have inconsistent representations of the same object. This is because knowledge about the world is partial and relative to a perspective. We show how Morphe treats relative (and eventually inconsistent) knowledge by incorporating the notions of *situations* and *perspectives*.

1 Introduction

In modeling complex systems, one is often required to work with multiple representations of some aspects of reality. The notion of *situation* has been studied in computer science [Barwise 83][Barwise 89][Cooper 90] as an important concept in capturing the relative representation of knowledge about the world. The importance of such a notion stems from the epistemological assumption that any representation of the world is *partial* and *relative* to some perspective—that of the observer. In the cognitive process, the observer *abstracts* from reality only those aspects that he finds relevant; irrelevant portions are discarded. Sometimes this limited, abstracted representation is sufficient to allow one to perform certain tasks. In such cases we do not need to think about relative perspectives, and we can work as though our knowledge were an absolute and unique mapping of the real world. However, there are plenty of examples that show this is not true. In order to understand what is happening in the target world, we are forced to assume that the representation we are working with is relative, and furthermore, that we must eventually change perspectives in order to capture the real properties of the system we are representing. This is often the case when we have ambiguous representations and we are not able to resolve this ambiguity until we have some further information at hand.

Typically ambiguity arises when we try to combine information from different sources. For example, in dialogue understanding the knowledge of the one must be combined with the knowledge of the other to capture the exact meaning of an utterance [Numaoka 90]. Whenever there is some inconsistent information, the speakers must exchange further information in order to resolve the inconsistency. Other examples can be seen in multi-agent systems [Bond 88][Osawa 91]—where we have different agents with different knowledge bases that must be partially shared—and versioning systems as used in software development tools and engineering databases [Katz 90]—where we have different versions of the same object. A ground for extensive use of the notion of *situation* is in *open systems* [Hewitt 84], because in open systems the designer of a program cannot know *a priori* the nature of the environments in which their pieces of knowledge (called objects henceforth) will be used in the future. Along with its continuous evolution, an open system must be capable of integrating pieces of knowledge from different sources, and eventually these new pieces will conflict with existing ones.

In this paper we formalize the notion of *situation* as embedded in **Morphe**, a knowledge base and programming system which supports construction of open systems. *Situation* in Morphe is associated with a general notion of environment of interpretation. It represents a consistent set of properties (described by formulas) in a multi-version knowledge base. Rather than being a mere name for a part of the absolute real world, a *situation* has its own representation in Morphe, namely a routed,

*Also with Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223 JAPAN e-mail: mario@keio.ac.jp

directed, acyclic, and colored graph.

The notion of *situation* provides for two novel concepts: *compositional adaptation* and *situated polymorphic objects*. With *compositional adaptation* component objects are grouped within composite objects so that a component object is made to adapt to the requirements of the environment represented by the composite object. *Situated polymorphic objects* are objects that have multiple representations which depend on the situation they are used in. Situation is used to disambiguate the ambivalent interpretation of situated polymorphic objects.

The remainder of this paper is organized as follows: Section 2 gives an overview of Morphe's features through some examples. Morphe's formal syntax and semantics are sketched in Section 3 and Section 4, respectively. In this work we concentrate on the data modeling aspect of Morphe. Some important features (such as set-valued attributes, distinction between local and sharable attributes, user-defined constraints, and dynamic generation of new situations at update transactions) were not treated in the presentation for the sake of brevity and clarity. In Section 4 we give emphasis in showing how the domain of colored dags fits well to representing different perspectives to a shared object. In the last section we conclude this work.

2 Overview of Morphe

Morphe is a programming language which integrates object-oriented programming, constraint-based logic programming, and situated programming. It features:

- Querying capability for knowledge bases,
- Incremental construction of systems with inheritance and adaptive reuse of existent software,
- Multiple representations,
- Treatment of inconsistent knowledge through the notion of situation.

The basic aim of Morphe is to provide a system that supports easy construction of open information systems. There are two areas of support that are essential:

1. Easy integration of new pieces of knowledge, and
2. Treatment of shared inconsistent knowledge.

The Morphe system is a multi-version knowledge base with multi-versioned objects. We use the term *multi-version knowledge base* following the notion of *multi-version databases* as introduced by Cellary and Jomier in [Cellary 90]. Our approach differs from Cellary-Jomier's in that in Morphe even in a single knowledge base version we can have different object versions. The programmer can chose a particular version of the knowledge base through *situation descriptors*—formulas that index terms—which can be used within programs or in queries. In the development phase of a system, Morphe keeps track of transaction updates and creates consistent versions of the knowledge base.[1]

2.1 Example: Mario Joins Sony CSL

We will represent Sony CSL, a computer science laboratory, where Mario works as a director. We know that a representation of Mario already exists in the system and we want to share that representation. The existing representation is of Mario as a professor at an university.

```
1. person : [
       name : string;
       age : integer;
       sex : {male, female};
       age ≥ 0];
2. laboratory : [
       name : string;
       director : person;
       researcher :: person];
3. mario : person * [
       name : "Mario";
       age : 44];
4. scsl : laboratory * [
       name : "Sony CSL";
       director : person * [
              machine : "NEWS"]];
5. scsl.director = mario.
```

The first two expressions define the types for `person` and `laboratory`, and expressions 3 and 4 define `mario` and `scsl` as "instances" of `person` and `laboratory`, respectively. Expression 5 makes `mario` join `scsl` as its `director`.

Objects in Morphe are typed. For example, the expressions `name : string` and `age : integer` specify that the `name` of a `person` has type `string` and the `age` of a person has type `integer`. `String` and `integer` are primitive types provided in Morphe. The colon in those expressions represents a built-in predicate that specifies the type of the term on its left-hand side. Another built-in predicate is the one represented by the equal sign, as in `director = mario`, which specifies that `director`

[1]The operational aspects of manipulating situations are not emphasized in this work. Instead we will emphasize the declarative (or modeling) aspects of objects and situations.

and `mario` should have the same type. Expressions comprising these built-in predicates are called *formulas* or *constraints.*[2]

We can also construct complex types from primitive ones through *object descriptors.* An object descriptor is a set of formulas enclosed in brackets ("[]"). In the example, the expression `person` : [...] introduces a new type named `person` defined by the object descriptor on the right hand side of the colon.

As in unification grammar formalisms [Shieber 86] and some logic based programming languages [Kifer 89] [Yokota 92], Morphe does not make a distinction between classes and instances. Strictly speaking, every expression in Morphe is a type expression, and the execution of a Morphe program consists of finding the appropriate types for the variables, or in other words, solving the set of type constraints. Morphe provides domain specific constraint solvers and allows users to define predicates for new domains, as the predicate $\geq$ in the expression `age` ≥ 0. In this article we concentrate on showing how Morphe treats the notions of *situations* and *polymorphic objects*, leaving the discussion of other forms of constraints for another paper. Expressions using the colon predicate resemble attribute-value pairs of feature structure grammars and hence we sometimes refer, though improperly, to terms on the left-hand side of the colon operator as *attributes* and those on the right-hand side as *values.*

Besides object descriptors, there is another type of constructor: *braces* ("{}"). While object descriptors construct types intensionally, from formulas, *braces* construct types extensionally, from *terms.* For example, the expression `sex` : {`male`, `female`} specifies that the attribute `sex` of a `person` has type `male` or `female`. Stated in another way, the same expression defines a new type `person.sex` as a set of two constant types {`male`, `female`}.

A type can be made more and more specific as we add more restrictive constraints (formulas) into the associated object descriptor, and it becomes an "instance" when all the attributes are assigned constant types. In the code above, `scsl` is an instance of `laboratory` because the formulas in the object descriptor of the former are more restrictive than those in the object descriptor of the latter. Because all terms are types, even `scsl`, which is an "instance", can be made more specific by adding more formulas into its object descriptor. The way to do so is by *composing* object descriptors through `''*''`, the *composition* operator. The code which defines `mario` composes the type `person` with the object descriptor [`name` : `mario` ; `age` : 44]. The resulting object descriptor contains all the formulas of both operand types. The constraint solver then evaluates the most specific set of formulas in the resulting object descriptor, yielding [`name` : "`Mario`"; `age` : 44] as the type of `mario`. Determining the most specific sets of formulas is the same as determining the greatest lower bound of a set of terms. The associated procedure for determining the greatest lower bound is called *unification,* following the terminology of feature-structure grammar formalisms [Shieber 86].

2.2 Compositional Adaptation

With composition we can refine a type by giving more specific "values" for the attributes—as in `mario` above—or we can add new properties to an existing type. The type `laboratory.director` in the example is defined as a `person` plus an additional attribute: `machine`. Morphe allows for creating new types in a very particular way. The type `director` is defined in a specific context: `scsl`. This is an essential aspect of what we call *compositional adaptation*[Honda 92].

With compositional adaptation we make an object "adapt" to a new environment by transforming the object so that it obeys the type constraints specified in the environment. This process takes place when the predicate `''='' ` is evaluated. When the expression `director = mario` is evaluated, it either succeeds or fails. If it succeeds, the object denoted by `scsl.director` is *unified* with the object denoted by `mario`, and the result of the unification can be accessed from both `scsl.director` and `mario`.[3] The object enters a new environment "acquiring" new properties and constraints. In the example, `mario` acquires the additional attribute `machine` as specified in the environment `scsl`, and `scsl.director` acquires all the original properties of `mario`.

2.3 Situated Polymorphic Objects

In programming languages, the term polymorphism has been traditionally associated with the capability of giving different things the same name. Morphe's notion of polymorphism follows in the same vein. In Morphe the

[2]The term "constraint" used here follows the terminology of constraint logic programming framework as formalized by Jaffar and Lassez in [Jaffar 87].

[3]The full version of Morphe allows programmers to specify which components of the type are private (i.e., local) and which are public (i.e., sharable). The public part of two objects must be compatible for the unification to succeed, while the private part is not affected in the unification.

same object can have different versions, eventually incompatible with each other. Incompatible versions of an object are called *morphes*, and objects that have multiple *morphes* are called *polymorphic objects*.

By incompatibility of morphes we mean incompatibility of their types.[4] Different morphes of the same (polymorphic) object may fundamentally mean two things: 1) different states due to updates, or 2) different representations due to different perspectives. Each *morphe* of a polymorphic object is situated. The evaluation of a polymorphic object is the evaluation of a *morphe*, the selection of which is subordinated to the selection of a *situation* where the object participates.

Each *morphe* is a consistent set of constraints that describe the behavior of the object in a given *situation*. For instance, a person may exhibit different and eventually contradictory behavior depending on the situation in which he acts. Inconsistent sets of constraints yield different values to be assigned to the same attribute. For example, suppose that the definition of `mario`, instead of that given in expression 3, had been: *mario* : *person* * `[name : ''Mario''; birthyear` : 1947; *sex* : *male*; `machine : ''Mac''];` After `mario` joins `scsl`, the attribute `machine` of `mario` is assigned the value `''News''` when he plays his role as `scsl.director` and a different value—`''Mac''`— in other situations.

2.4 Specifying a Situation

Morphe's notion of *situation* is tied to the notion of environment of interpretation. In the domain of interpretation, a *situation* is a graph representing the program being interpreted. *Situations* are used to disambiguate inconsistencies in the knowledge base. When an object participates in different environments (eventually created by independent programs) and is subject to independent transformations, it is often the case that the object must behave differently in each of them. Once the programmer wants a different view (or representation) for the object, the system creates a new version of the object in such a way that the situation is kept consistent.

When evaluating an expression within a situation, the system keeps track of the path through which the object containing that expression is being accessed. Access to an object from different *perspectives* is realized as different paths to the object. A *path* is a sequence of labels that allows one to navigate through the entire system, along the arcs in the graph. For example, if we want to refer to Mario when he plays his role of a director at SCSL we use the path `scsl.director`. *Paths* can be combined with formulas which filters the morphes of an object referred from the same path. For example, if we had several versions of Mario distinguished according to his age, we could access the representation of Mario at Sony CSL when he was at the age of 40 by using the expression: `scsl.director@[age = 40]`. We can also change the *perspective* by switching the path in the navigation. For example, we can switch the view from `mario` to `scsl.director` with the path `mario` ↑ `scsl.director`, which gives us the representation of `mario` from `scsl.director`'s perspective.

3 Syntax

The alphabet of Morphe consists of: 1) A: a set of *atoms*, 2) L: a set of *labels*, 3) X: an infinite set of *variables*, 4) the distinguished *predicate symbols*: ":" (*colon*) and "=" (*equal*), 5) the *composition* operator "*", 6) the logical connective ";", 7) the path constructors: ".", "↑", and "@"; 8) the auxiliary symbols "( )", "[ ]", "{ }", ",", and ".".

Atoms denote primitive indivisible objects. Example atoms are: `integer`, `string`, 3, and `''Mary''`. *Labels* are the names of the objects. The distinguished label `Home` denotes the topmost object in a particular situation.[5] In the semantic domain, the label names an arc which allows access to the objects down the (directed) graph.

3.1 Terms ($\mathcal{T}$)

Objects are denoted by *terms*. Terms are defined by:

$$\tau ::= x \mid a \mid p \mid [f] \mid \tau * \tau$$

where x are variables, a are atoms, p are paths, f are formulas, and $\tau * \tau$ are compositions.

The terms of the form $[f]$ are called *object descriptors*. Object descriptors construct complex objects through formulas, which are defined by:

$$f ::= p : \tau \mid \tau = \tau \mid f; f$$

A *colon* predicate is a typing constraint. An expression $e : t$, where e is a path and t is a term, specifies that the

[4] Informally, incompatible types means that the values of a type cannot be the values of the other. We give a formal definition of type incompatibility in the next section.

[5] Typically, the object denoted by `Home` represents the user's "home object", which is the user's entry-point into the Morphe system.

type of the object denoted by e has *at least* the properties defined by t. For example, the formula `mario : person` specifies that the `mario` has at least the properties specified by `person`.

The *equal* predicate specifies object sharing. Given $e_1 : t_1$ and $e_2 : t_2$, where e_1 and e_2 are paths, the expression $e_1 = e_2$ states that e_1 and e_2 denote the same object, and hence they have equal types. The shared object is "viewed" from different perspectives: any change to the object performed from a perspective must be reflected into other perspectives.

Because the atomic predicates *colon* (":") and *equal* ("=") impose a structure on the objects in the domain of interpretation (i.e., graphs), they are called *structural predicates*, in contrast to other domain predicates and user defined predicates. In this article we discuss only the structural predicates and hence we call them simply predicates.

A *path* names an object through a sequence of labels. Paths are defined by:

$$p ::= l \mid l.p \mid p \uparrow p \mid p@[f]$$

where l are labels. When an object is polymorphic due to different access paths, we select a morphe by the associated path. For example, in the subsystem:

$$\mathtt{a} : [\mathtt{b} : [\mathtt{c} : \mathtt{x}]; \mathtt{d} : [\mathtt{c} : \mathtt{y}]; \mathtt{a.b} = \mathtt{a.d}]$$

the polymorphic value of c can be disambiguated through the appropriate path: `a.b.c : x`, and `a.d.c : y`.

A path of the form $p_1 \uparrow p_2$ is a *path switch*. It allows one to view the same object from a different perspective. For example, the value of `a.b` $\uparrow$ `d.c` is `y`, instead of `x`.

A path of the form $p@[f]$ is called a *conditional path*. The formula enclosed in brackets on the right hand side of the @ sign is called a *situation descriptor*, because it specifies a family of situations which entail f. A conditional path has a meaning only in the situations where the formula enclosed in the brackets is entailed. For notational convenience we write $\mathtt{l} : \{\mathtt{t}_1@[\mathtt{f}_1], \mathtt{t}_2@[\mathtt{f}_2]\}$ instead of $\mathtt{l}@[\mathtt{f}_1] : \mathtt{t}_1; \mathtt{l}@[\mathtt{f}_2] : \mathtt{t}_2$. Conditional paths are used to select version morphes of polymorphic objects. For example, given

$$\mathtt{a} : [\mathtt{b} : \{\mathtt{x}, \mathtt{y}\}; \mathtt{c} : \{\mathtt{w}@[\mathtt{b} : \mathtt{x}], \mathtt{v}@[\mathtt{b} : \mathtt{y}]\}]$$

where `a`, `b`, and `c` are labels and `x`, `y`, `w`, and `v` are atoms, there are two possible values of `a.c`, which depend on the possible values of `b`. The formulas `b : x; c : w`) and `b : y; c : v` determine two distinct situations of `a`. The value of `a.c` can be disambiguated by providing an appropriate conditional path: `a.b.c@[b : x] : w`, and `a.b.c@[b : y] : v`.

Composition is a binary operation $\mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$ which composes two terms to produce a new term. Given two terms t_1 and t_2, their composition $t_1 * t_2$ is the union of the formulas contained in both terms. For example, [`name : "John"; age : "integer"`] * [`age : 23`] $\equiv$ [`name : "John"; age : integer; age : 23`].

3.2 Ordering on Terms

We have seen that terms denote objects in the intended domain, and formulas associate terms in order to represent complex structures in that domain. The *colon* operator specifies the structure of the object denoted by a given path. We can now amplify its use as a binary predicate over two terms to construct a partial ordering in the set of terms. We start with atoms. We assume that the atoms in A are partially ordered according to a binary relation represented by "$\leq_A$". For example: "*Mary*" $\leq_A$ *string*, and $3 \leq_A$ *integer*.

If $x \leq_A y$ and $y \leq_A x$ we say that x and y are *congruent*, and write $x \cong_A y$. The *greatest lower bound* of a set of elements $B \subset A$, denoted by $\downarrow B$ is defined as usual: $\downarrow B = inf \in A$ such that $\forall x \in B.\ inf \leq_A x$.

For notational convenience, we will denote the greatest lower bound of two atoms x and y by $x \downarrow y$. The greatest lower bound does not always exist. The elements c of A such that $x : c$ implies $x \cong_A c$ are called the *constants* of A.

We extend the partial ordering to the set of terms with the binary relation ":", defined by the rules below. In these rules, Γ is a set of formulas which defines a *situation*.

$$\Gamma \vdash x : y \quad (\text{ if } x, y \in A \text{ and } x \leq_A y)$$

$$\Gamma \vdash t : []$$

$$\Gamma, (e : t) \vdash e : t$$

$$\frac{\Gamma \vdash e@[\phi] : t}{\Gamma, \phi \vdash e : t}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma, (e_1 = e_2) \vdash e_1 : t_3} \quad (t_3 = t_1 \sqcup t_2)$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma, (e_1 = e_2) \vdash e_2 : t_3} \quad (t_3 = t_1 \sqcup t_2)$$

$$\frac{\Gamma \vdash t_1 : t'_1 \ldots \Gamma \vdash t_n : t'_n}{\Gamma \vdash [l_1 : t_1; \ldots; l_n : t_n; \ldots; l_m : t_m] : [l_1 : t'_1; \ldots; l_n : t'_n]}$$

$$\Gamma \vdash t : t$$

$$\frac{\Gamma \vdash t_1 : t_2 \; \Gamma \vdash t_2 : t_3}{\Gamma \vdash t_1 : t_3}$$

The congruence relation on the set of terms is defined by: $x \cong y$ iff $x : y$ and $y : x$. The operation $\downarrow$ that gives the greatest lower bound of a set of atoms is also extended to terms. The rules below describe $\sqcup$, the greatest lower bound of two terms, defined so that $t_1 \sqcup t_2 : t_1$ and $t_1 \sqcup t_2 : t_2$.

$[] \sqcup t \cong t$

$x \sqcup y \cong x \downarrow y$

$[l : t] \sqcup [l : t'] \cong [l : (t \sqcup t')]$

$[l_1 : t_1] \sqcup [l_2 : t_2] \cong [l_1 : t_1; l_2 : t_2]$

$[l_1 : t_1; ...; l_n : t_n; l_1^i : t_1^i; ...; l_m^i : t_m^i] \sqcup [l_1 : t_1'; ...; l_n : t_n'; l_1^j : t_1^j; ...; l_k^j : t_k^j] \cong [l_1 : t_1 \sqcup t_1'; ...; l_n : t_n \sqcup t_n'; l_1^i : t_1^i; ...; l_m^i : t_m^i; l_1 : t_1'; ...; l_n : t_n'; l_1^j : t_1^j; ...; l_k^j : t_k^j]$

$t_1 \sqcup t_2 \cong t_2 \sqcup t_1$

$t_1 \sqcup (t_2 \sqcup t_3) \cong (t_1 \sqcup t_2) \sqcup t_3$

$t \sqcup t \cong t$

Two terms t_1 and t_2 are *incompatible* iff $t_1 \sqcup t_2$ does not exist.

4 Semantics

The formal semantics of Morphe is based on the algebraic approach to graph grammars as described in [Ehrig 86] and [Ehrig 90]. The domain of interpretation of Morphe is a set of colored, rooted, directed, and acyclic graphs. Following [ParisiPresicce 86][6], we impose a structure in the coloring alphabet in order to represent unification in that domain.

4.1 Definition: Colored Graphs

Let X be an infinite set of variables, A the set of atoms, L the set of labels (as introduced in Section 3), and O a set of identifiers. Let $C = (C_N, C_A)$ be a pair of alphabets where $C_N = O \cup A \cup X$ and $C_A = L$. The partial-order in A, $\leq_A$, is extended on C_N (and denoted $\leq_N$) such that $x \leq_N y$ iff $x \leq_A y$ or $y \in X$. A *C-colored graph* (or C-dag, for short) is a graph g over C defined as a tuple

$$< N_g, A_g, color_g^N, color_g^A, src_g, tgt_g, root_g >$$

where: N_g is the set of nodes; A_g is the set of arcs; $color_g^N : N_g \rightarrow C_N$ associates a color to each node; $color_g^A : A_g \rightarrow C_A$ associates a color to each arc; $src_g : A_g \rightarrow N_g$ associates with each arc a unique source node; $tgt_g : A_g \rightarrow N_g$ associates with each arc a unique target node; $root_g$ is a distinguished node called the *root* of the graph. It satisfies: $tgt^{-1}(root_g) = \emptyset$.

In what follows we refer to C-dags as graphs. A graph g is a *subgraph* of g' (written $g \subseteq_{\mathcal{G}} g'$) iff $N_g \subseteq N_{g'}$, $A_g \subseteq A_{g'}$, and the functions $color_g^N$, $color_g^A$, src_g, and tgt_g are the restrictions of the corresponding mappings of g'.

4.2 Definition: Graph Morphism

A *graph morphism* $f : g \rightarrow g'$ is a pair of functions $f_N : N_g \rightarrow N_{g'}$ and $f_A : A_g \rightarrow A_{g'}$ such that:

1. f_N and f_A preserve the incidence relations: $src(f_A(a)) = f_N(src(a))$ and $tgt(f_A(a)) = f_N(tgt(a))$,
2. f_A preserve the arc colors: $\forall a \in A_g.\ color_{g'}^A(f_A(a)) = color_g^A(a)$, and
3. $\forall x \in N_g.\ color_{g'}^N(f_N(x)) \sqsubseteq color_g^N(x)$.

A graph morphism indicates the occurrence of a graph within another graph. A graph morphism $f = (f_N, f_A)$ is called *injective* if both f_N and f_A are injective mappings, and it is called *surjective* if both f_N and f_A are surjective. If $f : g \rightarrow g'$ is injective and surjective it is called an *isomorphism*, and there is also an inverse isomorphism $f^{-1} : g' \rightarrow g$. In this case we say that g and g' are *congruent* and write $g \cong_{\mathcal{G}} g'$.

4.3 Subsumption

Subsumption is an ordering on graphs which corresponds to the relative specifity of their structures. A graph g *subsumes* h ($h \sqsubseteq_{\mathcal{G}} g$) iff there exists a graph-morphism $f : g \rightarrow h$ such that $f(root_g) = root_h$.

The semantic counterpart of the greatest upper bound of a set of terms (ref. Section 3.2) is the join of two graphs, which is their "most general unifier". The *join* of graphs g_1 and g_2 (notated $g_1 \sqcup_G g_2$) is a graph h such that $h \sqsubseteq_{\mathcal{G}} g_1$ and $h \sqsubseteq_{\mathcal{G}} g_2$.

4.4 Semantic Structure

The semantic structure of Morphe is a tuple

$$\mathcal{A} =< \mathcal{G}^*, \sqsubseteq_{\mathcal{G}}, \sqcup_{\mathcal{G}}, \top >$$

where:

1. $\mathcal{G}^*$, the domain of interpretation, is the set of all variable-free (i.e., ground) C-dags.

[6] F. Parisi-Presicce, H. Ehrig, and U. Montanari allowed variables in graphs (and productions) so that they could represent composition of graphs using relative unification. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, and M. Löwe [Corradini 90] further extended that work to represent general logic programs with hypergraphs and graph productions.

2. The relation $\sqsubseteq_{\mathcal{G}}$ and the operation $\sqcup_{\mathcal{G}}$ are as defined above.
3. *Top* ($\top$) is the distinguished element of $\mathcal{G}^*$ defined by: $\forall g \in \mathcal{G}^*.\ g \sqsubseteq_{\mathcal{G}} \top$.

4.5 Interpretation

A consistent set of formulas is represented with a C-dag with variables. The C-dag representation of a set of formulas is called a *situation*. A Morphe program is mapped by the interpreter into a set of situations which are ordered according to the subsumption relation. The evaluation of a query is a mapping from the C-dag representing the query to the set of *situations* in the hierarchy. If no situation is specified, the interpreter evaluates in a default situation. While parsing its input, the interpreter keeps track of this situation in order to resolve eventual ambiguities.

Let $\mathcal{I}_\alpha : A \rightarrow C_N$ be a function that maps each atom in A to a node color in C_N, and $\mathcal{I}_\lambda : L \rightarrow C_A$ another function that maps each label to an arc color in C_A.

Variable Assignment

A variable assignment in a situation s is a mapping $\mu : X \rightarrow \mathcal{G}^*$ which maps variables to ground C-dags. We extend the variable assignment to other terms with the following clauses:

- If a is an atom, $\mu(s,a) = g$ *s.t.* $N_g = \{x\}, A_g = \emptyset$, and $color^N(x) = \mathcal{I}_\alpha(a)$.
- If l is a label, $\mu(s,l) = g \sqsubseteq_{\mathcal{G}} s$ *s.t.* $\exists a \in A_s.\ color^A(a) = \mathcal{I}_\lambda(l)$ and $src(a) = root_s$ and $tgt(a) = root_g$.
- If l is a label, and e is a path, $\mu(s, l.e) = \mu(\mu(s,l), e)$.
- If e is a path and ϕ is a formula, $\mu(s, e@[\phi]) = \mu(s,e)$ if $s \models \phi$.
- $\mu(s, [\phi]) = g \sqsubseteq_{\mathcal{G}} s$ *s.t.* $g \models \phi$.

Formulas

The "truthness" of a formula is relative to a specific situation. We say that a situation s models a formula ϕ under a variable assignment μ (written $s \models_\mu \phi$) iff there is a subgraph of s with the properties specified by the formula.

$s \models_\mu e : t$ iff $\mu(s,e) \sqsubseteq_{\mathcal{G}} \mu(s,t)$.

$s \models_\mu e_1 = e_2$ iff $\mu(s,e_1) \cong_{\mathcal{G}^*} \mu(s,e_2)$.

$s \models_\mu \phi; \psi$ iff $s \models_\mu \phi$ and $s \models_\mu \psi$

5 Conclusion

This paper has shown how the notions of *situation* and *polymorphic objects* in Morphe can handle situated knowledge in open systems. We claim that the Morphe features shown here are suited to support incremental development of a complex system. When a set of constraints is added to a situation, the new formulas may conflict with the old ones. Morphe helps the developer to find the locus of inconsistency, and in the cases where the programmer wants a new version of the system, Morphe splits the inconsistent situation into new subsituations whenever it is possible. Some meta-rules based on domain-dependent heuristics may help the system to decide on which actions to take in the presence of conflict.

Syntactically, a situation was defined as a set of formulas which define a hierarchy of versions of the knowledge base. *Situation descriptors* can be used in programs in order to specify *a priori* the family of situations in which the program is expected to work. Once the system is provided with a way to determine the right situation, the associated *morphe* can be selected and then passed to the constraint solver in order to proceed with the evaluation of the program or the query.

Most existing typed programming languages impose a distinction between types and values syntactically, and types are usually associated with the variables in order to check whether the value assigned to a variable is compatible with the associated type. Morphe does not impose such a distinction at the syntactic level, though it bears both the notions of "types" and "values". An equal treatment of types and values was achieved in Morphe by imposing a partial order on the set of terms. This partial ordering was identified as the subsumption relation over directed acyclic graphs in the domain of interpretation.

In this work we have shown only those features that we find most interesting to capture the intuitive notion of relative knowledge, perspective, and situations. Problems concerning changes of situations in the presence of transaction updates, locality of information and sharing (i.e., unification), database querying facilities, and the operational semantics were not treated here. We hope however that the contents of this article have given the readers an insight on the problems and solutions concerning relative representations of objects in open systems.

Acknowledgments

Sony Computer Science Laboratory has been a privileged environment for discussing the problems and requirements of open distributed systems. Discussions with the

other members of this laboratory have provided the underlying motivations for developing Morphe. In particular, we wish to thank Ei-Ichi Osawa, for his collaboration at the initial phase of Morphe, and Akikazu Takeuchi and Chisato Numaoka for their helpful comments on the formalisms presented in this work. Watari thanks the members of Next-Generation Database Working Group promoted by ICOT. Discussions in the group promoted a better understanding of the requirements for advanced data base programming languages.

References

[Barwise 83] Jon Barwise and John Perry. *Situations and Attitudes.* The MIT Press, 1983.

[Barwise 89] Jon Barwise. *The Situation in Logic.* Center for the Study of Language and Information, 1989.

[Bond 88] Alan H. Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann, 1988.

[Cellary 90] Wojciech Cellary and Geneviève Jomier. Consistency of Versions in Object-Oriented Databases. In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Proceedings of 16th International Conference on Very Large Databases*, August 1990.

[Cooper 90] Robin Cooper, Kuniaki Mukai, and John Perry, editors. *Situation Theory and its Applications – Volume I.* Center for the Study of Language and Information, 1990.

[Corradini 90] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, and Michael Löwe. Graph Grammars and Logic Programming. In *Proc. of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer-Verlag, March 1990.

[Ehrig 86] Hartmut Ehrig. Tutorial Introduction to the Algebraic Approach of Graph Grammars. In *Proc. of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer-Verlag, December 1986.

[Ehrig 90] Hartmut Ehrig and Michael Löwe Martin Korff. Tutorial Introduction to the Algebraic Approach of Graph Grammars Based on Double and Single Pushouts. In *Proc. of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer-Verlag, March 1990.

[Hewitt 84] Carl Hewitt and Peter de Jong. Open Systems. In J. Mylopoulos and J. W. Schmidt M. L. Brodie, editors, *On Conceptual Modeling*, Springer-Verlag, 1984.

[Honda 92] Yasuaki Honda, Shigeru Watari, and Mario Tokoro. Compositional Adaptation: A New Method for Constructing Software for Open-ended Systems. *JSSST Computer Software*, Vol.9, No.2, March 1992.

[Jaffar 87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the Fourteenth ACM Symposium of the Principles of Programming Languages (POPL'87)*, January 1987.

[Katz 90] Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, Vol.22, No.4, December 1990.

[Kifer 89] Michael Kifer and Georg Lausen. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, ACM, 1989.

[Numaoka 90] Chisato Numaoka and Mario Tokoro. Conversation Among Situated Agents. In *Proceedings of the Tenth International Workshop on Distributed Artificial Intelligence*, October 1990.

[Osawa 91] Ei-Ichi Osawa and Mario Tokoro. *Collaborative Plan Construction for Multiagent Mutual Planning.* Technical Report SCSL-TR-91-008, Sony Computer Science Laboratory, August 1991.

[ParisiPresicce 86] Francesco Parisi-Presicce, Hartmut Ehrig, and Ugo Montanari. Graph Rewriting with Unification and Composition. In *Proc. of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer-Verlag, December 1986.

[Shieber 86] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* Center for the Study of Language and Information, 1986.

[Yokota 92] Kazumasa Yokota and Hideki Yasukawa. Towards an Integrated Knowledge Base Management System. In *Proceedings of the FGCS'92*, ICOT, June 1992.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

On the Evolution of Objects in a Logic Programming Framework

F. Nihan Kesim Marek Sergot

Department of Computing, Imperial College
180 Queens Gate, London SW7 2BZ, UK
fnk@doc.ic.ac.uk, mjs@doc.ic.ac.uk

Abstract

The event calculus is a general approach to the representation of time and change in a logic programming framework. We present here a variant which maintains a historical database of changing objects. We begin by considering changes to the internal state of an object, and the creation and deletion of objects. We then present separately the modifications that are necessary to support the mutation of objects, that is to say, allowing objects to change class and internal structure without loss of identity. The aims are twofold: to present the modified event calculus and comment on its relative merits compared with the standard versions; and to raise some general issues about object-orientation in databases which do not come to light if dynamic aspects are ignored.

1 Introduction

There has been considerable research on combining logic-based and object-oriented systems, and reasoning with complex objects. Many proposals have been put forward for incorporating features of object-oriented systems into logic programming and deductive databases [Abiteboul and Grumbach 1988, Zaniolo 1985, Chen and Warren 1989, Kifer and Lausen 1989, Dalal and Gangopadhyay 1989, Maier 1986, Bancilhon and Khoshafian 1986]. But opinions vary widely as to what are the characteristic and beneficial features of objects and comparatively little attention has been given to the dynamic aspects of objects. Yet change in internal state of an object as it evolves over time is often seen as a characteristic feature of object-oriented programming; and the ability of object-oriented representations to cope gracefully with change has often been cited as a major advantage of this style of representation. It is these dynamic aspects that we wish to address in this paper.

We are not concerned with object-oriented *programming*, but with object-oriented representation of data in (deductive) databases. We address such problems as how objects change state, how deletion and creation of objects can be described and how an evolving object can change its class over time.

In order to avoid the discussion of destructive assignment, we formulate change in the context of a historical database which stores all past states of objects in the database. Historical databases are logically simpler than snapshot databases because change is then simply ***addition*** of new input. A snapshot of the historical database at any given time is an object-oriented database in the sense that it supports an object-based data model.

In this paper we present an object-based variant of the event calculus[Kowalski and Sergot 1986] which is a general approach to the treatment of time and change within a logic programming framework. We use this modified event calculus to describe changes to objects. The objectives of this paper are twofold: to present the object-based variant of the event calculus; and to raise some general issues about object-orientation in databases that we believe do not come to light if dynamic aspects are ignored. These more general points are touched upon in the course of the presentation, and identified explicitly in the concluding section.

In the following section we give a brief summary of the original event calculus. Section 3 presents the basic data model that is supported by the object-based variant. In section 4 we present this object-based variant and discuss how it can be applied to describe changes in objects. In section 5 we address the mutation of objects, where objects are allowed to change their classes during their evolution. We conclude the paper by summarising, and making some remarks about object-based representations in general.

2 The Event Calculus

The primitives of the event calculus are *events* together with some kind of temporal *ordering* on them, *periods* of time, and *properties* which are the facts and relationships that change over time. Events initiate and terminate periods of time for which properties hold. The effects of each type of event are described by specifying which properties they initiate and terminate. Given a set of events and the times at which they occurred, the event calculus derives (computes) which facts hold at which times. As an example, consider a fragment of a departmental database. An event of type

promote(X, New)

initiates a period of time for which employee X holds rank *New* and terminates whatever rank X held at the time of the promotion:

initiates(promote(X,New), rank(X,New)).
terminates(promote(X,New), rank(X,_)).

Given a fragment of data :

happens(promote(jim,assistant), 1986).
happens(promote(jim,lecturer), 1988).
happens(promote(jim,professor), 1991).

the event calculus computes answers to queries such as :

?- *holds_at(rank(jim,R), 1990).*
R=lecturer
?- *holds_for(rank(jim,lecturer), P).*
P=1988-1991

The original presentation of the event calculus showed how a computationally useful formulation can be derived from general axioms about the properties of periods. It gave particular attention to the case where events (changes in the world) are not necessarily reported in the order in which they actually occur. For the purpose of this paper, it is sufficient to consider only the simplest case, where the assimilation of events into a database is assumed to keep step with the occurrence of changes in the world, and where the times of all event occurrences are known. Under these simplifying assumptions, the event calculus can be formulated thus:

holds_at(R, T) ←
happens(Ev, Ts), Ts ≤ *T,*
initiates(Ev, R),
not broken(R, Ts, T).

broken(R, Ts, T) ←
happens(Ev, T*),*
Ts < *T** ≤ *T,*
terminates(Ev, R).*

We have omitted the clauses for *holds_for* which are similar. The interpretation of *not* as negation by failure in the last condition for *holds_at* gives a form of default persistence: property R is assumed to hold at all times after its initiation by event Ev unless there is information to the contrary.

The event calculus has been developed and extended in various different ways (see for instance [Sripada 1991, Eshghi 1988]). But what is important for present purposes is to stress that the underlying data model in all of these applications is relational. The *properties* that events initiate and terminate are facts like *rank(jim,professor)*. In database terms they are tuples of relations; in logic programming terms they are ground unit clauses or ground atoms or standard first order terms, depending on what is taken as the semantics of *holds_at*. A snapshot of the historical database at any given time is a relational database. In this paper we modify the event calculus in order to describe changes to a database which supports an object-oriented data model.

Before moving on to present this modification, we wish to make one further remark about the representation of events. One of the most common motivations for introducing object-oriented extensions to logic programming languages [Chen and Warren 1989, Ait-Kaci and Nasr 1986, M. Kifer and Wu 1990] is to overcome the restrictions imposed by the fixed arity of predicates and functors. These restrictions are particularly evident in the representation of events: Jim was promoted to professor in 1989, Jim was promoted from lecturer, Jim was promoted by his department in 1989 could all be descriptions of the same promotion recording different amounts of information about the event. In general, it is difficult or impossible to devise a fixed arity representation for events, because these representations cannot cope gracefully with the range of descriptions that can be expected even for events of the same type. (The philosopher Kenny refers to this phenomenon as the 'variable polyadicity' of events.) The standard way of dealing with 'variable polyadicity' is to employ binary predicates. Thus [Kowalski and Sergot 1986] represents events in the following style:

event(e1).
act(e1, promote).
object(e1, jim).
newrank(e1, prof).
happens(e1,1989).

Chen and Warren [Chen and Warren 1989] have developed this usage of binary predicates and have given it a formal basis. Their language C-logic allows the use of structured terms which can be decomposed into subparts. These terms are record-like tuples with

named labels. In the syntax of C-logic (also resembling the syntax of *LOGIN* [Ait-Kaci and Nasr 1986] and O-logic [Maier 1986]) the event *e1* can be described thus:

happens(event : e1[act ⇒ promote, object ⇒ jim, newrank ⇒ prof], 1989).

e1 is an identity which uniquely determines the event, and the labels are used to complete the specification of the event. Chen and Warren give a semantics to C-logic directly, and also by transformation to an equivalent first-order (Prolog) formulation that uses unary predicates for types and binary predicates for attributes. In this paper we use C-logic syntax as a convenient shorthand for describing events, and we exploit C-logic's transformation to Prolog by mixing C-logic and standard Prolog syntax freely. Thus we shall also write, for example,

event:e1[act⇒promote, object⇒ jim, newrank⇒prof].
happens(e1,1989).

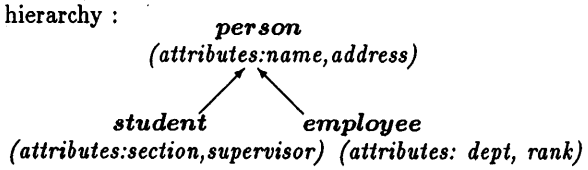
Chen and Warren's transformation to Prolog make all of these formulations equivalent.

3 The Data Model

Our objective in this paper is to focus attention on the dynamic aspects of objects. For this purpose, we take a very simple data model which exhibits only the most basic features associated with object orientation. As will be illustrated, this simple data model already raises a number of important problems; further elaborations of this data model are mentioned in the concluding section of the paper.

The basic building block of the model is the concept of an object. An object corresponds to a real world entity. Each object has a unique identity to distinguish it from other objects. The objects have attributes whose values can be other objects (i.e. their identities). We assume that all attributes are single-valued.

Objects are organized into class hierarchies, defined explicitly by *is_a* relationships among classes. A class denotes a set of object identities; the class-subclass relation (*is_a*) is the subset relation. A class describes the internal structure (state) of its instances by attribute names. The state of an instance is determined by the values assigned to these attributes. A subclass inherits the structure (attribute names) of its superclass(es). As an example consider the following class hierarchy :



The instances of the class *student* have the internal structure described by the attributes *name, address, section* and *supervisor*. Similarly the state of an *employee* instance is described by *name, address, dept* and *rank*. The class hierarchy is represented by *is_a* relations as:

is_a(student, person).
is_a(employee, person).

The relation between a class and its instances is represented by the *instance_of* relation. The instances of a class *C* are also instances of the superclasses of *C*. The *instance_of* relation can be represented thus:

instance_of(tom, student).
instance_of(mary, employee). etc.

together with

instance_of(X, Class) ←
is_a(Sub, Class), instance_of(X, Sub).

These definitions will be adjusted in later sections when we consider time dependent behaviour.

Multiple inheritance without overriding can be expressed by the *is_a* and *instance_of* relationships. This type of multiple inheritance causes no additional difficulty and is not mentioned again.

4 Object-Based Event Calculus

Database applications require an ability to model a changing world. The world changes as a result of the occurrence of events and hence it is very natural to describe such a changing world using a description of events. Given a description of events, it is possible to construct the state of the world using the the event calculus.

4.1 State Changes

One way of dealing with the evolution of an object over time (as suggested to us by several groups, independently) is to view the changing object as a collection of different though related objects. Thus, if we have an employee object *jim* in the database, which changes over time, *jim* at time t_1, *jim* at time t_2, *jim* at time t_3 are all different objects. Their common time-independent attributes are inherited from *jim* by some kind of 'part_of' mechanism. This approach has a certain appeal, but a moment's thought reveals it must be rejected for practical reasons. Each time an object is modified a new object is created. This new object becomes the most recent state of the object with a different identity. In this case, all other objects referring to the modified object should also be modified to refer to the new version. However updating them means creating other new objects in turn,

which results in an explosion in the number of objects in the database. In [M. Kifer and Wu 1990] a system of this type is described. They have to use equality in order to make certain denotations (i.e. object ids) in fact refer to the same object and provide some navigation methods through versions in order to get appropriate versions of an object.

The alternative is to have one object *jim* and to parametrize its attributes with times at which these attributes have various values. A state change in an object now corresponds to changing the value of any of its attributes. For instance if a person moves to a new place, the value of the address attribute changes; if an employee is promoted the rank attribute changes accordingly. Formulation of this idea in the spirit of the event calculus is straightforward. Instead of

happens(promote(jim,professor), 1991).

it is convenient to separate out the object that has been affected by the event :

happens(jim, promote(professor), 1991).

Now events are indexed by object; every object has associated with it the events that affected it. Events initiate and terminate periods of time for which a given attribute of a given object takes a particular value :

initiates(Obj, promote(NewRank), rank, NewRank).

Given a set of event descriptions which are indexed by object identities, the modified event calculus constructs the state of an object. We can ask queries to find out the value of an attribute of an object at a specific time or we can access the state of an object at any time by querying all of its attributes :

?- *holds_at(jim, rank, R, 1989).*
?- *holds_at(jim, Attr, Val, 1989).*

The following is the basic formulation of the object-based event calculus used to reason about the changing state of objects :

holds_at(Obj, Attr, Val, T) ←
happens(Obj, Ev, Ts), Ts ≤ *T,*
initiates(Obj, Ev, Attr, Val),
not broken(Obj, Attr, Val, Ts, T).

broken(Obj, Attr, Val, Ts, T) ←
happens(Obj, Ev, T*),*
Ts < *T** ≤ *T,*
terminates(Obj, Ev, Attr, Val).*

terminates(Obj, Ev, Attr, _)* ←
initiates(Obj, Ev, Attr, _).*

Informally, to find the value of an attribute of an object at time *T*, we find an event which happened before time *T*, and initiated the value of that attribute; and then we check that no other event which terminates that value has happened to the object in the meantime. The last clause for *terminates* is to satisfy the functionality constraint of the attributes. Since we are considering only single-valued attributes we can simply state that the value of an attribute is terminated if an event initiates it to another value. (The usage of the anonymous variable '_' in this clause is not a mistake).

The original event calculus can compute the periods of time for which a property holds. We can have the same facility for the attributes of objects. The following compute the periods of time for which an attribute takes a particular value :

holds_for(Obj, Attr, Val, (S - E)) ←
happens(Obj, Ev, S*),*
initiates(Obj, Ev, Attr, Val),
terminated(Obj, Attr, Val, S, E).

terminated(Obj, Attr, Val, S, E) ←
happens(Obj, Ev, E),
terminates(Obj, Ev, Attr, Val), S < *E,*
not broken(Obj, Attr, Val, S, E).

holds_for is used to find the period of time for which an attribute has a particular value. The time period is represented by its start (*S*) and end (*E*) points. We also require another clause for *holds_for* to deal with periods that have no end-point (i.e. an attribute is initiated but there is no event which terminated its value). This can be written in a similar style, which we omit.

Since objects are organized into classes, it is natural and convenient to structure the specification of the effects of a given event according to the class of object it affects. If an event is defined to affect the instances of a class, then the same event specification applies to the instances of subclasses. For example, consider a departmental database in which objects are organized according to the class hierarchy given in section 3. We can specify the effects of these events in the following way :

initiates(Obj,move(Address),address,Address) ←
instance_of(Obj,person).
initiates(Obj,promote(NewRank),rank,NewRank) ←
instance_of(Obj,employee).

The effects of changing the address are valid for all persons (i.e. all students and employees as well). However promotion is a type of event which can happen to employee objects only. In the formulation as presented here, it is possible to assert that an object of class person was promoted - but this event has no effect (does not initiate or terminate anything) unless the object is also an instance of class employee. An

alternative is to arrange for event descriptions to be checked and rejected at input if the class conditions are not satisfied. This alternative requires more explanation than we have space for; it is peripheral to our main points, and we omit further discussion.

We have discussed how event calculus can be used to describe changes to the values of attributes of objects. Apart from the events that cause state changes of existing objects, there are other kinds of events which cause the creation of new objects or deletion of objects.

4.2 Creation of Objects

The creation of a new object of a given class means adding new information about an entity to the database. In the real world being modeled, there are events which create new entities. Birth of a person, manufacturing of a vehicle or hiring a new employee are examples of such events. We can think of describing object creation by events whose specification will provide the necessary information about the initial state of the object.

For creation, we need to say what the class of an object is and specify somehow its initial state. In a practical implementation, generation of a unique identity for a newly created object can be left to the system; conceptually, all object identities exist, and the 'creation' of an object is simply assigning it to a chosen class. Assigning the new identity to the class initiates a period of time for which the new object is a member of that class. This makes it necessary to treat class membership as a time-dependent relationship. We introduce a new predicate *assigns* to describe instance addition to classes. For the time being we assume that once an object is assigned to a class it remains an instance of this class throughout its lifetime. Class changes are discussed separately in section 5.

We can handle creation of objects by specifying which events assign objects to which classes. We use the same event description to initialize the state of the object. As an example consider registration of a student *ali*, which causes the creation of a new student object in the database. The specification of the event and the necessary rules to describe creation are as follows :

event : e23 [act ⇒ register,
object ⇒ ali,
section ⇒ lp,
supervisor ⇒ bob].

assigns(event:E[act⇒register, object⇒Obj],
Obj, student).

initiates(Obj, E , section, S) ←
event : E[act⇒register, object⇒Obj, section⇒S].
initiates(Obj, E, supervisor, S) ←
event : E[act⇒register,object⇒Obj,supervisor⇒S].

The *assigns* statement is used to assign the identity of the object *Obj* to the class *student*; the *initiates* statements are used to initialize the object's state. Now the occurrence of the event is recorded by :

happens(e23, 1991).

To specify that the event has happened to the object *ali* we use the rule:

happens(Obj, Ev, T) ←
happens(event:Ev[act⇒register, object⇒Obj], T).

Note that we have two *happens* predicates: one binary (for asserting that events happened at a given time), and one ternary (to index events by objects affected).

We have to notice also that creating a new object of class C, creates a new instance of the superclasses of C as well. There are several ways to formulate this. The simplest is to write:

assigns(Ev, Obj, Class) ←
is_a(Sub, Class), assigns(Ev, Obj, Sub).

4.3 Deletion of Objects

There are two kinds of deletions that we are going to discuss in this paper. One is absolute deletion of an object where the object is removed from the database. The other one deletes an object from its class but keeps it as an instance of another class, possibly one of the superclasses. The second case is related to mutation of objects as they change class, which will be discussed in section 5.

For the purposes of this section, we assume that when an object is deleted it is removed from the set of instances of its class and the superclasses, and that all its attribute values are terminated. For example, if a person dies, all the information about that person is deleted from the database. We use a new predicate *destroys* to specify events that delete objects and write the following :

terminates(Obj, Ev, Attr, _) ← destroys(Ev, Obj).

This rule has the effect that all attributes *Attr* defined in the class of the object and also those inherited from superclasses are terminated. If an event destroys an object O which is an instance of class C, then that event removes O from class C and all superclasses of C.

There is one point to consider when deleting objects in object-oriented databases. If we delete an object x, there might be other objects that have stored

the identity of x as a reference. The deletion therefore can lead to dangling references. A basic choice for object-oriented databases is whether to support deletion of objects at all [Zdonik and Maier 1990]. We choose to allow deletion of objects and we eliminate dangling references by adding another rule for the *broken* predicate:

broken(Obj, Attr, Val, Ts, T) ←
happens(Val, Ev, T*),*
$Ts < T^* \leq T$,
destroys(Ev, Val).*

We obtain the effect that the value *Val* of the attribute *Attr* is terminated by any event which destroys the object *Val.*

4.4 Class Membership

As we create and delete objects the instances of a class change. Class membership, which is described by the *instance_of* relation, is a dynamic relation that changes over time. We can handle the temporal behaviour by adding a time parameter. We now have events that initiate and terminate periods of time for which an object *O* is an instance of a class *C.* The *instance_of* relation is affected when a new object is assigned to a class or when an object is destroyed. By analogy with *holds_at*, the following finds the instances of a class at a specific time :

instance_of(Obj, Class, T) ←
happens(Ev, Ts), $Ts \leq T$,
assigns(Ev, Obj, Class),
not removed(Obj, Class, Ts, T).

removed(Obj, Class, Ts, T) ←
happens(Obj, Ev, T*),* $Ts < T^* \leq T$,
destroys(Ev, Obj).*

With this time variant class membership we can ask queries to find the instances of a class at a specific time. For example:

?- *instance_of(Obj, employee, 1980).*

We can also write the analogue of *holds_for* to compute periods, which we omit here.

In the example we have been using, we have represented the rank of an employee object by including an attribute *rank* whose value might change over time. But suppose that instead of using an attribute *rank*, we had chosen to divide the class of employees into various distinct subclasses:

is_a(lecturer, employee).
is_a(professor, employee).

It is at least conceivable that this alternative representation might have been chosen, assuming that all employee objects have roughly the same kind of internal structure. Is the choice between these two representations simply a matter of personal preference? Not if we consider the evolution of objects over time. The first representation allows for change in an employee's rank straightforwardly, since this just changes the value of an attribute. The second does not, since no object can change class in the formulation of this section. The only way of expressing, say, a promotion from lecturer to professor, is by destroying (deleting) the lecturer object and creating a new professor object. But then how do we relate the new professor object to the old lecturer object, and how do we preserve the values of unchanged attributes across the change in class? In the next section we will examine the problem of allowing the class of an individual object to change.

5 Mutation of Objects: Changing the Class

The ability to change the class of an object provides support for object evolution [Zdonik 1990]. It lets an object change its structure and behaviour, and still retain its identity. For instance, consider an object that is currently a *person.* As time passes it might naturally become an instance of the class *student* and then later an instance of *employee.* This kind of modification is usually not directly supported by most systems. It may be possible to create another object of the new class and copy information from the old object to it, but one loses the identity of the old object.

We want to describe this kind of evolution by event specifications. For example graduation causes a student to change class. If we delete student *ali* from class student, then he will lose all the attributes he has by virtue of being a student, but retain the attributes he has by virtue of being a person. The effects of this event can be described by removing *ali* only from class student and terminating his attributes selectively. The attributes that are going to be terminated can be derived from the schema information. For dealing with this type of class change we use a new predicate *removes* in place of the predicate *destroys* of section 4.3:

removes(event: Ev[act ⇒ graduate, object ⇒ Obj],
Obj, student).
terminates(Obj, E, Attr, _) ←
event:Ev[act⇒graduate, object⇒Obj],
attribute(student, Attr).

The clauses for the time-dependent *instance_of* relation must be modified too, to take *removes* into account:

removed(Obj, Class, Ts, T) ←
happens(Obj, Ev, T*), Ts < T* ≤ T,*
removes(Ev, Obj, Class).*

The graduation of the student *ali* corresponds to moving him up the class hierarchy. Now consider hiring *ali* as an employee. This will correspond to moving down the hierarchy. The specification of an event causing such a change will likely include values to initialize the additional attributes associated with the subclass. So the effects of hiring *ali* will be to assign him to the employee class and initiate his employee attributes. The event might be:

event : e21[act ⇒ hire,
object ⇒ ali,
dept ⇒ cs,
rank ⇒ lecturer]

And we can declare the following:

assigns(event:E[act⇒hire, object⇒Obj], Obj, employee).
initiates(Obj, E, dept, D) ←
event:E[act⇒hire, object⇒Obj, dept⇒D].
initiates(Obj, E, rank, R) ←
event:E[act⇒hire, object⇒Obj, rank⇒R].

Note that in changing class first from student to person, then from person to employee, *ali* retains all the attributes he has as a person.

We have described this class change by two separate events: graduation and hiring. We can also imagine a single event which would cause an object to change its class from student to employee directly, say *hire-student* event. We could then describe the changes using the description of this event:

removes(event: E[act ⇒ hire-student,
object ⇒ Obj], Obj, student).
assigns(event: E[act ⇒ hire-student,
object ⇒ Obj], Obj, employee).

The initial values of the additional attributes will again be given in the event specification. As in the case of having two separate events, we have not lost the values of the attributes as a person, and we have not removed the object from class person.

We have illustrated three kinds of class changes: changing from a class *C* to a direct superclass of *C*, changing from *C* to a direct subclass of *C* and changing from *C* to a sibling class of *C* in the hierarchy. In general, changing an object from class *C1* to class *C2* involves removing from *C1* and assigning to *C2* and specifying in the event description how the initial values of *C2* attributes are related to the values of old *C1* attributes.

6 Concluding Remarks

We have presented a variant of the event calculus which maintains an object-based data model where the standard versions maintain a relational one. Section 4 considered state changes of objects in this framework, and the creation and deletion of objects. Section 5 discussed the modifications that are necessary to support also the mutation of objects - change of an object's class and its internal structure without loss of its identity.

There are other object-oriented features that can usefully be incorporated into the object-based data model. Removing the restriction that attributes are all single-valued causes no great complication. We are currently developing other extensions, such as the inclusion of methods in classes for defining the value of one attribute in terms of the values of other attributes, and we are investigating what additional complications arise when the schema itself is subject to change.

In object-oriented terminology, event types - like promote, change-address, and so on - correspond to methods: their effects depend on the class of object that is affected; the predicates *intiates* and *terminates* for attribute values, and *assigns*, *destroys* and *removes* for objects and classes are used to implement the methods (they would be replaced by destructive assignment if we maintained only a changing snapshot database). Of course, execution of this event calculus in Prolog does not yield an object-oriented style of computation. At the implementational level, objects are not clustered (except by Prolog's first argument indexing), and the computation has no element of message-passing. The implementation and the computational behaviour can be given a more object-oriented flavour by using for example the techniques described by [Chen 1990] for C-logic, or the class templates of [McCabe 1988]. We are currently investigating what added value is obtained by adjusting these implementational and computational details.

The object-oriented version of the event calculus offers some (computational) advantages over the standard relational versions, that we do not go into here for lack of space. Whatever the merits of the object-based variant of the event calculus, we believe that its formulation forces attention to be given to important aspects of object-orientation that are otherwise ignored. We limit ourselves to two general remarks:

1) In the literature, the terms *type* and *class* are often used interchangeably. Sometimes type is used in its technical sense, but then it is common to see illustrative examples resembling 'Mary is of type stu-

dent'. If we consider the dynamics of object-oriented representations, then these examples are either badly chosen or the proposals are fundamentally flawed. 'Mary' might be a student now but this will not hold forever. We could surely not contemplate an approach where an update to a database requires a change to the type system, and hence to the syntax of the representational language. These remarks do not apply to object-oriented programming where there is no need to make provision for updates that change the type of an object.

The static notion of a type corresponds to the treatment of a class we presented in section 4: an object may or may not exist at a given time, but when it exists it is always an instance of the same class. If we wish to go beyond this, to allow objects to mutate, then a dynamic notion of class is essential. This is not to say that types have no place in object-oriented databases. A student can become an employee over time, but a student cannot become a filing cabinet, and a filing cabinet cannot become an orange. Both static types and dynamic notions of class are useful. The consideration of the dynamics of objects - how they are allowed to evolve over time - suggests one immediate and simple criterion for choosing which notion to use: the type of an object cannot change.

2) In section 4.3, we assumed that all attributes of an object are terminated when the object is destroyed; in section 5, removal of an object from the class C terminates all attributes the object has by virtue of being an instance of the class C. The reasoning behind this assumption is this: attributes are used to represent the, possibly complex, internal state of an object. When an object ceases to exist, it is not meaningful to speak any more of its internal state. Of course, some information about an object persists even after it ceases to exist. It is still meaningful to speak of the father of a person who has died, but it is not meaningful to ask whether this person likes oranges or is happy or has an address. The development of these ideas suggests that we should distinguish between what we call 'internal attributes' and 'external relationships'. Internal attributes describe the state of a complex object, and they cease to hold when the object ceases to exist or ceases to be an instance of the class with which these attributes are associated. External relationships continue to hold even after the object ceases to exist. We are being led to a kind of hybrid data model together with some tentative criteria for choosing between representation as attribute and representation as relationship with other objects. The analysis given here is rather superficial, but it indicates the general directions in which we are planning to pursue this work.

Acknowledgements. F.N. Kesim would like to acknowledge the financial support by TUBITAK, the Scientific and Research Council of Turkey.

References

[Abiteboul and Grumbach 1988] S. Abiteboul and S. Grumbach. COL : A logic-based language for complex objects. In *International Conference on Extending Database Technology- EDBT'88*, pages 271–293, Venice, Italy, March 1988.

[Ait-Kaci and Nasr 1986] H. Ait-Kaci and R. Nasr. Login: A logic programming language with built-in inheritance. *The Journal of Logic Programming*, 1986.

[Bancilhon and Khoshafian 1986] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Proceedings of the 5th ACM-SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 53–59, Cambridge, Massachusetts, March 1986.

[Chen 1990] Weidong Chen. *A General Logic-Based Approach to Structured Data.* PhD thesis, State University of New York at Stony Brook, 1990.

[Chen and Warren 1989] W. Chen and D. Warren. C-logic of complex objects. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, 1989.

[Dalal and Gangopadhyay 1989] M. Dalal and D. Gangopadhyay. OOLP: A translation approach to object-oriented logic programming. In *The First International Conference on Deductive and Object-Oriented Databases*, pages 555–568, Kyoto,Japan, December 4-6 1989.

[Eshghi 1988] K. Eshghi. Abductive Planning with the Event Calculus. In *Proc. 5th International Conference on Logic Programming*,1988.

[Kifer and Lausen 1989] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the ACM-SIGMOD Symposium on the Management of Data*, pages 134–146, 1989.

[Kowalski and Sergot 1986] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[M. Kifer and Wu 1990] G. Lausen M. Kifer and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical report, Department of Computer Science, SUNY at Stony Brook, June 1990.

[Maier 1986] D. Maier. A logic for objects. In *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington D.C., August 1986.

[McCabe 1988] F.G. McCabe. *Logic and Objects: Language Application and Implementation.* PhD thesis, Department of Computing, Imperial College, 1988.

[Sripada 1991] S. M. Sripada. *Temporal Reasoning in Deductive Databases.* PhD thesis, Department of Computing, Imperial College, 1991.

[Zaniolo 1985] C. Zaniolo. The representation and deductive retrieval of complex objects. In *Proceedings of Very Large Databases*, page 458, Stockholm, 1985.

[Zdonik 1990] S. B. Zdonik. Object-oriented type evolution. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 277–288. ACM Press, 1990.

[Zdonik and Maier 1990] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*, chapter 4, page 239. Morgan Kaufmann, 1990.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

The panel on a future direction of new generation applications

Fumio Mizoguchi

Science University of Tokyo
Intelligent System Laboratory
Noda, Chiba 278, Japan

1 Introduction

This paper introduces a panel to be held at the application track of FGCS'92 conference. This panel will be devoted to a future direction of new generation applications. The goal is to discuss about the applications with various paradigms which have been explored in the areas of knowledge representation, logic programming, machine learing and parallel processing. It is my hope that by expressing different perspectives of the panelists, we will understand the importance of the underlying paradigms, the real problem areas, and a direction of next generation applications. The word paradigm itself is originally come from T. Kuhn's book called "The Structure of Scientific Revolution (1962)". Recently, this word is referred by the AI researchers because of its sophisticated meanings which indicates a current research trend or a future direction. Here, I will use this word in this context that implies new bases and views for exploration of applications without too much philosophical discussion.

In this short paper, I will attempt to outline the perspectives represented by the panelists. Althouth the ideas and the positions papers will be represented in the following pages in the proceedings, I will try to guide the rough views which will be necessary for this panel discussion. The context is my subjective impressions on the current trends and research directions.

2 KR paradigm

Ronald J. Brachman will talk about his knowledge representation language called Classic and his experiences through the use of Classic for the developments in applications. He might refer the knowledge representation as KR which follows his research community. KR might be the starting point for any AI based application system. KR is one of the main paradigms of AI researches including natual language understanding and coginitive science. There are a lots of attempts in the design of KR language and systems such as KRL, FRL and KLONE in the late 1970's. The 1980's was the following productive period for KR system developments and theories. The first dedicated international KR conference was held recently, and many important ideas and foundations were presented in the conference. This state of art has been reviewed by R.Brachman at the AAAI meeting in 1990. He has presented KR and issues which are related to the field, history, development of the 1980's, the future of KR and open research problems. I am especially interested in his highlights for the future of KR which predicts the current trends of common knowledge base and ontology. Now, KR should be standardized for the further developments for any knowledge systems. The related paper for Classic will be presented at the technical session and he will talk about his position based upon his paper presentation. The panel will start with KR and related topics.

3 CLP paradigm

Catherine Lassez will represent the constraint logic programming(CLP) which is a new face for handling constraints in Operations Research, Computational Geometry, Robotics and Qualitative Physics. Reasoning with constraint is very important for these application areas. These problems are sometimes required heavy computational resource and are related to combinatorial characteristics. The novel aspects of CLP is the unified framework of knowledge representation for numeric and non-numeric constraints, solution algorithm and data query system. Also, CLP has been implemented as the programming languages such as CLP(R), CHIP, CAL, Prolog-III and Triton. These languages are used for the various application domains which are linkage between AI and OR. As for the financial applications, CLP is very good affinity for describing the financial equations and relations. Constraint is also useful to the handling qualitative knowledge in Computational Geometry and Naive Physics. In order to show the expressive power of CLP, it is necessary to demonstrate the speed and performance for the same problems which are OR people's proposed. This is challenging for any AI researchers and Logic programmers to persuade other field researchers through the recent progress on programming which can avoid the brute forces of numerical calculation. She will

present her experiences on the developments on the theories and applications. The details will be shown in her very intensive long position paper in this panel.

4 ILP paradigm

Stephen Muggleton will represent his recent notion of inductive logic programming(ILP) which uses the inverse resolution and relative least general generalisation. ILP is newly formed research area in the integration of machine learning and logic programming. Machine learning is very attractive paradigm for knowledge acquisition and learning which any AI system is addressed. With the advent of machine learning research, there are a lots of developments in tools for classifying large data using concepts learning and neural network methods. Muggleton's recent development for his ILP is called GOLEM which is a first order induction algorithm for generating rules from given examples. Each example is a first order ground atom and each rule is a first order Horn clause. Rules can be used to classify new examples. GOLEM is implemented in SUN's using C and very efficient for inducing rules from examples. Another example of ILP will be presented in the invited speaker, Ivan Bratko and he will talk about learning qualitative model of dynamic system using GOLEM learing program. ILP is different from CLP, but in its spirit, idea is come from the logic programming paradigm. As is well known, Shapiro's work on Model Inference System(MIS) is implemented using Prolog and it is very clear logical model for learning. Using logic programming paradigm, ILP is unified approach to induction and deduction which provides knowledge system with more powerfull inference facilities. Namely, as for inductive component, IPL is very useful for inducing rules from data and then, using the rules, system infers deductively data into known diagnostic states. Therefore, ILP is new approach to application with very large data which are further classified into categorization. These kinds of applications are found in the area of protein engineering and fault diagnosis for satellites. He was the organizer of the first ILP workshop and the second workshop which will be held after the FGCS conference. ILP is very young paradigm for machine learning and there will be another exploration in theory and application. He will talk about the recent research with the relationship between Valiant's PAC-Learning framework. Machine learning is most active research area and it will be the next stage that it will deal with realistic problems.

5 PP paradigm

Kazuo Taki will represent the Parallel Processing(PP) paradigm which the Fifth Generation Computer System Project aims to explore and to develop both sides of hardware and software derived from the concurrent logic programming which shows affinity for both expressing concurrency and executing in parallel. With the continious efforts in langualge and implementation research in the FGCS project, KL-1 has expressive for describing many complex applications programs with efficient performance. Most important aspects in the use of the concurrent system are to built large scale parallel software which is further accumulated as the experiences in parallel programming. A new style of programming requires a new thinking way of programming and the model of computation. This is also true for KL-1 language and for applying it to complex applications such as VLSI-design, DNA analysis and legal reasoning system. Basing upon these experiences, he will focuss on the parallel language culture which is necessary for the next generation computer like multi-PSI and PIM. The hardware progress has made rapidly compared with software technology and the accumulation of parallel programming experiences are very important for the re-use and the economy of coding. The current issue of parallel programming is how to transfer knowledge in software technology developed by the FGCS project in order to explore the culture of the concurrent system. Therefore, as for the future directions, PP paradigm is how to use in the widely adopted computational environment. He will talk about the issue of the parallel programming culture and the experiences in the use of KL-1 for applications.

6 Future directions

I will introduce the various paradigms for knowledge information processing starting from KR to PP. Each paradigm has distinctive and novel features for exploration of applications. As for my position, I am interested in the research on the fusion of paradigms which is the integration of CLP and ILP for example. I will call this paradigm as Inductive Constraint Logic Programming(ICLP not conference name!) which is the natural extension of constraint logic programming into inductive inference for constraints in Spacial Geometry and Robotics. This framework is also useful for the Naive Physics and qualitative reasoning system without large amount of background knowledges for rules generations. We will examine our approach to Naive Kinematics and simple image processing for spacial reasoning. At this stage, the application domain is very simple, but for the research on Robotics that learns, the inductive component is very important in the knowledge acquisition on the constraints and then deductively use the constraints for the further moves. The fusion of paradigms will be necessary foundation for the next generation applications. We should re-examine the current paradigms for the different problems areas such as OR, Robotics and Computational Geometry.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Knowledge Representation Theory Meets Reality: Some Brief Lessons from the CLASSIC Experience

Ronald J. Brachman

AT&T Bell Laboratories, 600 Mountain Ave.,
Murray Hill, NJ 07974-0636, U.S.A.
rjb@research.att.com

Abstract

Knowledge representation is one of the keys to Artificial Intelligence, and as a result will play a critical role in many next generation computer applications. Recent results in the field look promising, but success on paper may be misleading: there is a significant gap between a theoretical result or proposal and its ultimate impact in practice. Our recent experience in converting a fairly typical knowledge representation design into a usable system illustrates how many aspects of "reduction" to practice can significantly influence and force important changes to the original theoretical foundation. I briefly motivate our work on the CLASSIC representation system and outline a handful of ways in which practice had significant feedback to theory. The general lesson for next generation applications is the need for us in our research on core technology to take more seriously the influence of implementation, applications, and users.

1 Knowledge Representation

Representation of knowledge has always been the foundation on which research and development in Artificial Intelligence has rested. While no single representation framework has come to dominate the field, and while there are important challenges to the utility of conventional representation techniques from "connectionists" and others, it is very likely that the next generation of AI and AI-related applications will still subscribe to the hypothesis that intelligent behavior can arise from formal reasoning over explicit symbolic representations of world knowledge.

The centrality of the need to represent world knowledge in AI systems, expert systems, robots, and Fifth Generation applications has helped increase interest in formal systems for representation and reasoning—so much so that over the last decade, the explicit subfield of "Knowledge Representation" (KR) has taken on its own identity, with its own international conferences, IFIP working group, etc. This subfield has been prolific. It has attracted the attention of the greater AI community with highly visible problems like the "Yale Shooting Problem" and systems like CYC. It has collected its own set of dedicated researchers, and has increasing numbers of graduate students working on formal logics, nonmonotonic reasoning, temporal reasoning, model-based diagnosis, and other important issues of representation and reasoning.

It is probably fair to say that in recent years, formal and theoretical work has become preeminent in the KR community.[1] Concomitantly, it appears to be generally believed that when the theory is satisfactory, its reduction to practice will be relatively straightforward. This transition from theory to practice is usually considered uninteresting enough that it is virtually impossible to have a technical paper accepted at a conference that addresses it; it seems to be assumed that all of the "hard" work has been done in developing the theory.

This attitude is somewhat defensible: it is common in virtually all other areas of AI; and there often really isn't anything interesting to say further about a KR formalism as it is implemented in a system. However, my own group has had substantial recent experience with the transition of a knowledge representation system from theory to practice that contradicts the common wisdom, and yields an important message for KR research and its role in next generation applications. In particular, our view of what we thought was a clean and clear—and "finished"—formal representation system was substantially influenced by the complexity and constraint of the process of turning the logic into a usable tool.

2 The CLASSIC Effort

As of several years ago, we had developed a relatively small, elegant representation logic that was based on many years of experience with description hierarchies and a key inference called *classification*. As described in a companion paper at this conference [Brachman *et al.*, 1992], the CLASSIC system was a product of many years of effort on numerous systems, all descended from the KL-ONE knowledge representation system. Work on KL-ONE and its successors grew to be quite popular in the US and Europe in the 1980's, largely because of the semantic cleanliness of these languages, the appeal of object-centered (frame) representations, and their provision for some key forms of inference not available in other formalisms (e.g., description classification). The reader familiar with KR research will note that numerous publications in recent years have addressed formal and theoretical issues in "KL-ONE-like" languages, including formal semantics and computational complexity of variant languages. However, the key prior efforts all had some fundamental flaws, and work on CLASSIC was in large part launched to design a formalism that was free of these defects.

Another central goal of CLASSIC was to produce a compact logic and ultimately, a small, manageable imple-

[1] This has happened for numerous reasons, and while it may have some negative consequences (as addressed here), it is positive in many respects. The early history of the field was plagued by vague and inadequate descriptions of ad hoc solutions and computer programs; recent emphasis on formality has encouraged more thorough and rigorous work.

mented representation and reasoning system. A small system has important advantages in a practical setting, such as portability, maintainability, and comprehensibility. Our intention was to eventually put KR technology in the hands of non-expert technical employees, to allow them to build their own domain models and maintain them. CLASSIC was also designed to fill a small number of application needs. We had had experience with a form of deductive information retrieval (most recently in the context of information about a large software system [Devanbu *et al.*, 1991]), and needed a better tool to support this work. We also had envisioned CLASSIC as a deductive, object-oriented database system (see [Borgida *et al.*, 1989]; success on this front was eventually reported in [Selfridge, 1991]).

After analyzing the applications, assessing recent progress in KL-ONE-like languages, and solving a number of the technical problems facing earlier systems, we produced a design for CLASSIC that felt complete; the logic was presented in a typical academic-style conference paper in 1989 [Borgida *et al.*, 1989]. In this design, some small concessions were made to potential users, including a procedural test facility that would allow some escape to the host implementation language for cases that CLASSIC could not handle. Given the clarity and simplicity of this original design of CLASSIC, we ourselves held the traditional opinion that there was essentially no research left in implementing the system and having users use it in applications. At that point, we began a typical AI programming effort, to build a version of CLASSIC in COMMON LISP.

3 Influences in the "Reduction" to Practice

As the research LISP version neared completion, we began to confer with colleagues in a development organization about the potential distribution of CLASSIC within the company. Despite the availability of a number of AI tools in the marketplace, an internal implementation of CLASSIC held many advantages: we could maintain it and extend it ourselves, in particular, tuning it to real users; we could assure that it integrated with existing, non-AI environments; and we could guarantee that the system had a well-understood, formal foundation (in contrast to virtually all commercially available AI tools). Thus we undertook a collaborative effort to create a truly practical version of CLASSIC, written in C. Our intention was to develop the system, maintain it, create a training course, and eventually find ways to make it useful in the hands of AI novices.

To make a long story short, it took at least as much work to get CLASSIC to the point of usability as it did to create the original logic that we originally thought was the culmination of our research. Our view of the language and knowledge base operations supporting it changed substantially as a result of this undertaking, in ways that simply could not be anticipated when consider a paper design of the logic.

The factors that influenced the ultimate shape of CLASSIC were quite varied, and in most cases, were not influences that we—or most other typical researchers, I suspect—would have expected to have forced more research before the logic was truly finished. These ranged from the need to be reasonable in the release and maintenance of the software itself to some specific needs for key applications that could not really have been anticipated until the system was actually put into practical use. Here is a brief synopsis of the five main types of issues that influenced the ultimate shape of the CLASSIC system:

- the constraints of *creating and supporting a system for real users* caused numerous compromises. For one thing, upward compatibility of future releases is a critical issue with real software, and it meant that any construct in the language in which we were not completely confident might better be left out of the released system. Issues of run-time performance (which also dictated the exclusion of some features) also had surprising effects on what we could realistically include in the released version.

- certain detailed *implementation considerations* played a role in determining what was included in the system. These included certain tradeoffs that affected the design, such as the tremendous space consequences an inverse relationship ("inverse roles") feature would have had, or the consequences of certain fine-grained forms of truth maintenance (to allow for later retraction of asserted facts). Some features (our **SAME-AS** construct, for example) were just so complex to implement that they were better left out of the initial release.

- *concern for real users* alerted us to issues easily ignored with a pure logic. These involved the sheer learnability and usability of the language and the system. Error-handling, for example, was of paramount concern to our real consumers, and yet the very idea never arose when considering the initial CLASSIC language. Similarly, the uniformity of abstractions and the simplicity of the interface were critical to acceptability of our system. The potential consequences of user "escapes" with side-effects was another related concern. Finally, explanation of the system's behavior—again, not an issue when we designed the logic—might make the difference between success and failure in using the system.

- as soon as a system is put to any real use, mismatches in its capabilities and *specific application needs* become very evident. In this respect, there seems to be all the difference in the world between the few small examples given in typical research papers and the details of real, sizable knowledge bases. In the case of CLASSIC, our lack of attention to the details of numbers and strings in the logic meant substantial more work before implementation. Another issue that plagued us was the lack of attention to a query language for our KR system (a common lack in most AI KR proposals).

- finally, *what looked good (and complete) on paper* did not necessarily hold up under the fire of real use. Even with a formal semantics, certain operators prove tricky to understand in practice, and subtle interactions between operators that arise in practice are rarely evident from the formal work. Simply being forced by an implementation effort to get every last detail right certainly caused us to re-examine several things we thought we had gotten correct in the original logic, and I suspect this would be the case with virtually every sufficiently complex KR logic that ends up being implemented.

4 Some Lessons

The main lesson to be learned here is that despite the ability to publish pure accounts of logics and their theoretical properties, the true theoretical work on knowledge representation systems is not really done until issues of implementation and especially of use are addressed head-on. The "theory" can hold up reasonably well in the transition from paper to system, but the typical KR research paper misses many make-or-break issues that determine a proposal's true value in the end. Arguments about needed expressive power, the impact of complexity results, the naturalness and utility of language constructs, etc., are all relatively hollow until made concrete with specific applications and implementation considerations.

For example, in our context, the right decision was clearly to start with a small version of the system for release, and extend it only as needed. Given the complexity of software maintenance, it may never make sense to try to anticipate in advance all possible ways that all possible users might want to express concepts.[2] A small core with an extension mechanism might in reality be better than a large, extraordinarily expressive—and complex—system. In the case of CLASSIC, we have been able to place in the hands of relatively naive users a fairly sophisticated, state-of-the-art inference system with a formal semantics and well-founded inference mechanism, and have them use it successfully, needing only to make a small number of key extensions to meet their real needs.

There are several consequences here for next generation applications of knowledge representation research. First, it is important that the research community recognize as legitimate and important the class of issues that arise from implementation efforts—issues relating to size, for example, that have always been the legitimate concern of the database community; issues relating to implementation tradeoffs and complexities; and issues relating to software release and maintenance. Second, unless our KR proposals are put to the test in real use on real problems, it is almost impossible to assess their real value. So much seems to be different when a proposal is reduced to practice that it is unclear what the original contribution really is. Third, it is quite critical that at least some fraction of the community address directly the needs of users and the constraints and issues in their applications. Too much research with only mathematics as its driving force will continue to lead KR (and other areas of AI research) farther afield. Not only that, it is clear that truly interesting research questions arise when driven from real rather than toy or imagined needs.

References

[Borgida *et al.*, 1989] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A Structural Data Model for Objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 59–67, June 1989.

[Brachman *et al.*, 1992] R. J. Brachman, A. Borgida, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Resnick. The CLASSIC Knowledge Representation System, or, KL-ONE: The Next Generation. In *Proceedings of the International Conference on Fifth Generation Systems*, Tokyo, June 1992.

[Devanbu *et al.*, 1991] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. LaSSIE: A Knowledge-Based Software Information System. *CACM*, 34(5):34–49, May 1991.

[Selfridge, 1991] P. G. Selfridge. Knowledge Representation Support for a Software Information System. In *Proceedings of the Seventh IEEE Conference on AI Applications*, pages 134–140, Miami Beach, Florida, February 1991.

[2] Ironically, the ongoing and sometimes virulently argued debate over how much expressive power to allow in KR systems may in the end be settled by simple software engineering considerations.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Reasoning With Constraints

Catherine Lassez

IBM T.J. Watson Research Center
P.O.Box 704, Yorktown Heights, NY 10598, USA
lassez@watson.ibm.com

Constraints are key elements in areas such as Operations Research, Constructive Solid Geometry, Robotics, CAD/CAM, Spreadsheets, Model-based Reasoning and AI. Languages have been designed specifically to solve constraints problems. More recently, the reverse problem of designing languages that use constraints as primitive elements has been addressed. Constraints handling techniques have been incorporated in programming languages and systems like CLP($\Re$), CHIP, CAL, CIL, Prolog III, 2LP, BNR-Prolog, Mathematica and Trilogy.

In the rule-based context of Logic Programming, the CLP scheme [5] provides a formal framework to reason with and about constraints. The key idea is that the important semantic properties of Horn clauses do not depend on the Herbrand Universe or Unification. These semantic properties and their associated programming methodology hold for arithmetic constraints and solvability (and in many other domains including strings, graphs, booleans,...). The CLP scheme is a main example of the use of constraints as the primitive building blocks of a class of programming languages, since logic formulae can be themselves considered as constraints.

In the same spirit constraints have been introduced in committed choice languages in Maher [14], and in the work of Saraswat [15], and in Database querying languages by Kanellakis, Kuper and Revesz [6]. The link between classical AI work on constraints, and Logic Programming has been described by van Hentenryck [17].

Not surprisingly there are many different paradigms reflecting the integration of constraints and languages. The main differences come from the aims of the language: general purpose programming language, database or knowledge based query language, or a tool for problem solving. In mathematical programming the focus is on optimization, in artificial intelligence the focus is on constraint satisfaction and constraint propagation, in program verification the focus is on solvability. This should be reflected in the design of appropriate languages, but constraint programming should also have its own focus and theory.

We have developped a general framework for a systematic treatment of specific domains of constraints. We recall that a logic formula is viewed as an implicit and concise representation of its set of logical consequences and that the answer to a query Q is a set of substitutions which establish a relationship between the variables of Q, satisfied if and only if Q is a logical consequence of the formula. The key point is that a single algorithm, *Resolution*, is sufficient to answer all queries. These properties of logic formulae have counterparts in other domains. In particular, Tarski's theorem for quantifier elimination in closed fields[16] establishes that an arithmetic formula can be viewed as representing the set of all its logical consequences, that is the set of all arithmetic formulae it entails. Furthermore, a single algorithm, *Quantifier Elimination*, is required in analogy with logic formulae and resolution.

At the design and implementation level, however, the problems are far more difficult than for logic formulae. To try and circumvent these problems one must make heavy use of results and algorithms from symbolic computation, operations research, computational geometry etc... Also, as in the case of logic formulae, we have to sacrifice generality to achieve acceptable efficiency by carefully selecting sets of constraints for which suitable algorithms can be found.

Parametric queries Applying the paradigmatic aspects of reasoning with logic formulae to linear arithmetic, we have that:

- a set of constraints is viewed as an implicit representation of the set of all constraints it entails
- there is a query system such that an answer to a query Q is a relationship that is satisfied if and only if the query is entailed by the system.
- there exists a *single* algorithm to answer all queries.

Given a set S of arithmetic constraints as a conjunction of linear equalities, inequalities and negative constraints (disjunctions of inequations), we define a *parametric query* [7] as:

$$\exists \alpha_1, \alpha_2, \ldots, \beta \; \forall x_1, x_2, \ldots : S \Rightarrow \alpha_1 x_1 + \alpha_2 x_2 + \ldots \leq \beta \\ \wedge R(\alpha_1, \alpha_2, \ldots, \beta)?$$

where S is the set of constraints in store and R is a set of linear relations on the parameters $\alpha_1, \alpha_2, \ldots, \beta$.

Parametric queries provide a general formalism to extract information from sets of constraints and to express standard operations. For instance:

1. is S solvable? If not, what are the causes of unsolvability?
2. does S contains redundancies or implicit equalities?
3. is S equivalent to S'?
4. is it true that $x = 2$ is implied by S?
5. does there exist α such that $x = \alpha$ is implied by S?
6. does there exist a linear relation $\alpha x + \beta y + \ldots = \gamma$ implied by S?
7. does there exist $\alpha_1, \alpha_2, \ldots \beta$ such that $S \Rightarrow \alpha_1 x + \alpha_2 y \ldots \leq \beta$ and $\alpha_1 = 2\alpha_2 - 1$?

The solvability query is typical of linear programming and corresponds to the first phase of the Simplex method. Finding the causes of unsolvability is a typical problem of constraints manipulation system where the constraints in store can be modified to restore solvability using feedback information provided by the solver. Queries 2 and 3 both address the problem of constraint representation. Redundancy is a major factor of complexity in constraints processing and the removal of redundancies and the detection of implicit equalities are key steps in building a suitable canonical representation for the constraints [10] [12]. Queries 4 and 5 are classic Constraint Satisfaction Problems (CSP) and queries 6 and 7 are generalization of CSP to linear relations: variables are bound to satisfy given linear relations instead of simply values.

A priori, there does not seem to be any real connections between these various queries. However, they can all be expressed as parametric queries which ask under what conditions on the parameters $\alpha_1, \alpha_2, \ldots, \beta$, the constraint in the query is implied by the constraints in store. By varying the parameters, specific queries can be formulated. For instance,

- *is x bound to a specific value a?*
  $\exists \alpha_1, \alpha_2, \ldots, \beta$, s.t. $S \Rightarrow \alpha_1 x_1 + \alpha_2 x_2 + \ldots = \beta$ with $\alpha_1 = 1, \alpha_2 = 0, \ldots, \beta = a$.
- *is x ground?*
  same as above but with β unconstrained.
- *does S implies* $2x_1 + 3x_2 \leq 0$?
  as above with $\alpha_1 = 2, \alpha_2 = 3, \ldots, \beta = 0$.
- *what are the constraints implied by the projection of S the* $\{x_1, y_2\}$*-plane?*
  All parameters except $\alpha_1, \alpha_2, \beta$ set to 0

The test for solvability and the classic optimization problem can also be expressed in this way:

- *is S solvable?*
  as above with all parameters $\alpha_1, \alpha_2, \ldots$ set to 0 except $\beta \geq 0$.
  (by Fourier's theorem, which states that a set of constraints is solvable if and only if the elimination of all the variables results in a tautology)
- *what are the upper and lower bounds of* $f = x_1 + x_2 + x_3$*?*
  as above with $\alpha_1 = 1, \alpha_2 = 1, \alpha_3 = 1$, all other parameters are set to 0 except $\beta \geq 0$. The answer gives the upper and lower bounds for β that correspond to the minimum and maximun of f.

Parametric queries generalize logic programming queries which ask if there exists an assignment of values to the variables in the query so that the query becomes a logical consequence of the program clauses. They also generalize CSP's queries which are restricted to constraints of the type $x = a$.

We now must address the problem of finding a finite representation for the answers to the queries. Parametric queries are more complex than simple conjunctions of constraints as they involve universal quantifiers, non linearity and implication. However by using a result linked to duality in linear programming [8], we can reduce the problem to a case of conjunction of linear constraints. The Subsumption Theorem states that *a constraint is implied by a set of constraints S iff it is a quasi-linear combination of constraints in S.* A quasi-linear combination of constraints is a positive linear combination with the addition of a positive constant on the righ-hand side. For instance, let S be the set

$$\{2x + 3y - z \leq 1,\ x - y + 2z \leq 2,\ x - y + z \leq 0\}$$

and Q be the query

$$\exists \alpha, \beta, \forall x, y, S \Rightarrow \alpha x + \beta y \leq 1?$$

The following relations express that the constraint in Q is a quasi-linear combination of the constraints in S.

$$\begin{aligned} &2\lambda_1 + \lambda_2 + \lambda_3 = \alpha \\ &3\lambda_1 - \lambda_2 - \lambda_3 = \beta \\ &-\lambda_1 + 2\lambda_2 + \lambda_3 = 0 \\ &\lambda_1 + 2\lambda_2 + q = 1 \\ &\lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0, q \geq 0 \end{aligned}$$

where the λ_i's are the multipliers of the constraints in S. It is from this simpler formulation that variables are eliminated.

Variable elimination is the key operation to obtain answers to queries. It plays the role of resolution in Logic Programming. With inequalities, the complexity problems are far more severe than in Logic Programming, even in the restricted domain of conjunctions of positive linear constraints.

Fourier's method The basic algorithm is Fourier's[2]. The severity of the problem is illustrated by the table below:

Number of variables eliminated	Number of constraints generated	Actual number of constraints needed
0	32	18
1	226	40
2	12,744	50
3	39,730,028	19
4	390,417,582,083,242	2

The middle column gives the size of the output of Fourier's method to eliminate between 1 to 4 variables from an initial set of 32 constraints. The right most column gives the minimum size of equivalent outputs. Fourier's elimination is in fact doubly exponential as it generates an enormous amount of redundant information. Even if we remove redundancy on the fly, we are still left with exponential size for intermediate computation and potential exponential size for output. To solve this problem, one must look for output bound algorithms (an important area of study in computational geometry), that will guarantee an output when its size is small, bypassing the problem of intermediate swell. Also in the case where the size of the output is unmanageable, there is no point in computing it. However, we may sacrifice completeness and search for an approximation of reasonable size. That brings us back to avoiding intermediate swell.

The extreme points method This method, derived from the formalism of parametric queries, is interesting as it shows that variable elimination can be viewed as a straightforward generalization of a linear program in its specification and as a generalization of the simplex in its execution. Let $S = Ax \leq b$ and let V be the set of variables to be eliminated, the associated generalised linear program GLP is defined as:

$$extr(\Phi(\Delta))$$

$$\Phi = \begin{cases} \sum \lambda_i a_{i_1} = \alpha_1 \\ \vdots \\ \sum \lambda_i a_{i_k} = \alpha_k \\ \sum \lambda_i b_i = \beta \end{cases}$$

$$\Delta = \begin{cases} \sum \lambda_i a_{i_{k+1}} = 0 \\ \vdots \\ \sum \lambda_i a_{i_m} = 0 \\ \sum \lambda_i = 1 \\ \lambda_i \geq 0 \end{cases}$$

where $extr$ denotes the set of extreme points. Δ represents the conditions to be satisfied by a combination of constraints of S that eliminates the required variables. The normalization of the λ's ensures that Δ is a polytope. $extr(\Phi(\Delta))$, solutions of GLP, determine a finite set of constraints which defines the projection of S. The coordinates of the extreme points of $\Phi(\Delta)$ are the coefficients of a set of constraints that define the projection. The objective function in the usual linear program can be viewed as a mapping from R^n to R, the image of the polyhedron defined by the constraints being an interval in R. The optimization consists in finding a maximum or a minimum, that is one of the extreme points of the interval. In a GLP, the objective function represents a mapping from R^n to R^m and instead of looking for one extreme point, we look for the set of all extreme points. At the operational level, we can execute this GLP by generalizing the simplex method. The extreme points of $\Phi(\Delta)$ are images of extreme points of Δ. So we compute the set of extreme points of Δ, map them by Φ and eliminate the images which are not extreme points. It is important to note that although the extreme points method is better that Fourier in general because it eliminates the costly intermediate steps, there are still two main problems: the computation of the extreme points of Δ can be extremely costly even when the size of the projection is small and also the method produces a highly redundant output [1].

The convex hull method Variable elimination has long been treated as algebraic manipulations based on the syntax of the constraints rather than their semantics. Fourier's Procedure and EPM are no exceptions. Consequently, the complexity of these methods is tied to the initial polyhedral set instead of to the projection itself. Quantifier elimination can also be viewed as an operation of projection. Exploiting this remark in a systematic way leads to more output bound algorithms which guarantee an output when its size is reasonable and an approximation otherwise [9]. In the bounded case, the idea is trivial: by running linear programs we compute constraints whose supporting hyperplanes bound both the polytope to be projected and its projection. The traces of these hyperplanes on the projection space provide an approximation containing the projection. At the same time the extreme points provided by the linear programs project on points of the projection. The convex hull of these points is a polytope that is included in the projection. Iterating this process leads to the projection. Whether we have an output bound algorithm or not will however depend on the choice of points. The difficulties that remain are that we do not want to make any assumption on the input polyhedral set which can be bounded or not, full dimensional or not, redundant or not, empty or not. Standard linear programming techniques can be used to determine solvability and to transform the input if required into a set of equations defining its affine hull and a set of inequalities defining a full-dimensional polyhedral set in a smaller space. A straightforward variable elimination in the set of equations gives the affine hull of the projection which will be part of the final output.

This simplification based on geometrical considerations allows us to eliminate as many variables as possible by using only linear programming and gaussian elimination before getting into the costly part of elimination.

In the bounded case, the algorithm works directly on the input constraints. The projection is computed by successive refinements of an initial approximation obtained by computing with linear programs enough extreme points of the projection so that their convex hull is full-dimensional. Successive refinements consist in adding new extreme points and updating the convex hull. The costly convex hull construction is done in the projection space thus the main complexity of the algorithm is linked to the size of the output. The process stops when either the projection has been found or the size of the approximation has reached a user-supplied bound.

In the unbounded case, the problem is reformulated using the generalised linear program representation which is bounded by definition. $\Phi(\Delta)$ is computed by projection. The output will consist of the convex hull of $\Phi(\Delta)$ but also the set of its extreme points, from which the constraints defining the projection are derived. The advantage over the extreme points method is that we compute directly the extreme points of the projection. We do not need to compute the extreme points of Δ, this computation being the source of enormous intermediate computation and high redundancy in the output.

Implicit equalities and causes of unsolvability Fourier's algorithm can be used to trace all subsets of constraints in S that cause unsolvability or that are implicit equalities [11].

By using the *quasi-dual* formulation, we can acheive the same effects by running linear programs. The quasi-dual formulation which corresponds to Fourier's algorithm is

$$\Phi: \ \beta = b^T\lambda$$
$$\Delta: \begin{cases} A^T\lambda = 0, \\ \Sigma\lambda_i = 1, \\ \lambda_i \geq 0 \ \forall i. \end{cases}$$

Here Φ maps $\Re^m$ to $\Re$, where m is the number of constraints in S. Since we want to compute the minimum of Φ subject to Δ we need to solve the following linear program D:

$$\begin{array}{ll} \text{minimize} & b^T\lambda \\ \text{subject to} & A^T\lambda = 0 \\ & \Sigma\lambda_i = 1 \\ & \lambda_i \geq 0 \ \forall i. \end{array}$$

It is obvious that, in general, solving S in this manner is far more efficient than using Fourier's algorithm. Since D is a variant of the dual simplex in Linear Programming, it inherits nice properties from the standard dual simplex such as good incremental behavior, no need to introduce slack variables and no restriction to positive variables. More importantly as a side effect of the solvability test we obtain information about the algebraic properties of the constraints and about the geometric structure of the associated polyhedron. The properties of D are summarized in the table.

Quasi-Dual D	Properties of S
Unsolvable	• *Strongly* solvable • Full dimensional • No implicit equalities • Unbounded and • no projection has parallel facets • Inscribed spheres have unbounded radius
$min(b^T\lambda) > 0$	• Solvable • Full dimensional • No implicit equalities • Bounded or • exists projection with parallel facets • Inscribed spheres have bounded radius
$min(b^T\lambda) = 0$	• *Weakly* Solvable • Not full dimensional • Exists implicit equalities • An evident minimal subset of implicit equalities • Inscribed spheres have radius zero
$\exists\lambda : b^T\lambda < 0$	• Unsolvable • An evident minimal infeasible subset

Conclusion Much of the existing work on constraints has been done in diverse domains with their own distinctive requirements. Even in the restricted domain of linear arithmetic constraints, there is a wealth of knowledge and algorithms. To build systems to reason with constraints requires borrowing and synthesizing various notions and this led to the emerging concept of a unified framework of a single representation, the parametric query, and solution technique, variable elimination, for handling all the different operations on constraints. This approach shares key aspects with Logic Programming, with variable elimination playing the rule of resolution. The viability of this approach, both from a knowledge representation and knowledge processing aspects, is beeing tested with applications in the domain of spatial reasoning [3] and graphic user-interface [4]. Empirical results with an initial implementation have shown that a variety of small (about a hundred inequalities in two dimensions) and fairly large problems (up to about 2,000 inequalities over 70 variables) can be processed in times ranging from less than a second to a few minutes. Ongoing work includes the design and implementation of an integrated system based on the proposed framework and incorporating several solvers. The potential applicability of more recent interior points method is also investigated. Many properties of linear arithmetic constraints hold for constraints in other domains. These properties have been abstracted and generalized in [13].

References

[1] T. Huynh, C. Lassez and J-L. Lassez, Practical Issues on the Projection of Polyhedral Sets, *to appear Annals of Maths and AI.*

[2] T. Huynh, C. lassez and J-L Lassez, Fourier Algorithm Revisited, *2nd International Conference on Algebraic and Logic Programming* Springer-Verlag Lecture Notes in Computer Sciences, 1990.

[3] T. Huynh, L. Joskowicz, C. Lassez and J-L. Lassez, Reasoning About Linear Constraints Using Parametric Queries, *in Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Sciences*, Springer-Verlag vol. 472 December 1990.

[4] R. Helm, T. Huynh, C. Lassez and K. Marriott, A Linear Constraint Technology for User Interfaces, to appear *Proceedings of Graphics Interface'92*

[5] J. Jaffar and J.L. Lassez, Constraint Logic Programming, *Proceedings of POPL 1987*, Munich.

[6] P. Kanellakis, G. Kuper and P. Revesz, Constraint Query Languages, *Proceedings of the ACM Conference on Principles of Database Systems*, Nashville 90.

[7] J-L. Lassez, Querying Constraints, *Proceedings of the ACM conference on Principles of Database Systems*, Nashville 1990.

[8] J-L Lassez, Parametric Queries, Linear Constraints and Variable Elimination *Proceedings of DISCO 90*, Springer-Verlag Lecture Notes in Computer Sciences.

[9] C. Lassez and J-L. Lassez, Quantifier Elimination for Conjunctions of Linear Constraints via a Convex Hull Algorithm, IBM research Report, T.J. Watson Research Center, RC 16779 (1991), *to appear Academic Press.*

[10] J-L. Lassez, T. Huynh and K. McAloon, Simplification and Elimination of Redundant Arithmetic Constraints, *Proceedings of NACLP 89*, MIT Press.

[11] J-L. Lassez and M.J. Maher, On Fourier's Algorithm for Linear Arithmetic Constraints, IBM Research Report, T.J. Watson Research Center, RC 14114 (1988). To appear *Journal of Automated Reasoning.*

[12] J-L. Lassez and K. McAloon, A Canonical Form for Generalized Linear Constraints, IBM Research Report, T.J. Watson Research Center, RC 15004 (1989), to appear *Journal of Symbolic Computation.*

[13] J-L Lassez, K. McAloon, A Constraint Sequent Calculus *LICS 90.* Philadelphia.

[14] M. Maher, A Logic Semantics for a class of Committed Choice Languages, *Proceedings of ICLP4*, MIT Press 87.

[15] V. Saraswat, Concurrent Constraint Logic Programming, *to appear MIT Press.*

[16] L. Van Den Vries, Alfred's Tarski's Elimination Theory for Closed Fields, *The Journal of Symbolic Logic*, vol.53 n.1, March 1988.

[17] P. van Hentenryck, Constraint Satisfaction in Logic Programming, *The MIT Press*, 1989.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Developments in Inductive Logic Programming

Stephen Muggleton
The Turing Institute,
36 North Hanover Street,
Glasgow G1 2AD,
UK.

Abstract

Inductive Logic Programming (ILP) is a research area formed at the intersection of Machine Learning and Logic Programming. ILP systems develop predicate descriptions from examples and background knowledge. The examples, background knowledge and final descriptions are all described as logic programs. A unifying theory of Inductive Logic Programming is being built up around lattice-based concepts such as refinement, least general generalisation, inverse resolution and most specific corrections. In addition to a well established tradition of learning-in-the-limit results, recently some results within Valiant's PAC-learning framework have been demonstrated for ILP systems. Presently successful applications areas for ILP systems include the learning of structure-activity rules for drug design, finite-element mesh analysis design rules, primary-secondary prediction of protein structure and fault diagnosis rules for satellites.

1 Introduction

Deduction and induction have had a long strategic alliance within science and philosophy. Whereas the former enables scientists to predict events from theories, the latter builds up the theories from observations. The field of Inductive Logic Programming [6, 8] unifies induction and deduction within a logical setting, and has already provided notable examples of the discovery of new scientific knowledge in the area of molecular biology [5, 7].

2 Theory

In the general setting an ILP system S will be given a logic program B representing background knowledge and a set of positive and negative examples $\langle E^+, E^- \rangle$, typically represented as ground literals. In the case in which $B \not\models E^+$, S must construct a clausal hypothesis H such that

$$B \wedge H \models E^+$$

where B, H and E^- are satisfiable. In some approaches [16, 13] H is found via a general-to-specific search through the lattice of clauses. This lattice is rooted at the top by the empty clause and is partially ordered by θ-subsumption (H θ-subsumes H' with substitution θ whenever $H\theta \subseteq H'$). Two clauses are treated as equivalent when they both θ-subsume each other. Following on from work by Plotkin [12], Buntine [1] demonstrated that the equivalence relation over clauses induced by θ-subsumption is generally very fine relative to the the equivalence relation induced by entailment between two alternative theories with common background knowledge. Thus when searching for the recursive clause for *member/2*, infinitely many clauses containing the appropriate predicate and function symbols are θ-subsumed by the empty clause. Very few of these entail the appropriate examples relative to the base case for *member/2*.

Specific-to-general approaches based on Inverse Resolution [9, 14, 15] and relative least general generalisation [1, 10] maintain admissibility of the search while traversing the coarser partition induced by entailment. For instance Inverse Resolution is based on inverting the equations of resolution to find candidate clauses which resolve with the background knowledge to give the examples. Inverse resolution can also be used to add new theoretical terms (predicates) to the learner's vocabulary. This process is known as *predicate invention*.

Several early ILP authors including Plotkin [12] and Shapiro [16] proved learning in the limit results. Recently, ILP learnability results have been proved

within Valiant's PAC framework for learning a single definite clause [11] and in [3] for learning a multiple clause predicate definition assuming the examples are picked from a simple-distribution.

3 Applications

ILP is rapidly developing towards being a widely applied technology. In the scientific area, the ILP system Golem [10] was used to find rules relating the structure of drug compounds to their medicinal activity [5]. The clausal solution was demonstrated to give meaningful descriptions of the structural factors involved in drug activity with higher acuracy on an independent test set than standard statistical regression techniques.

In the related area of predicting secondary structure of proteins from primary amino acid sequence [7] Golem rules had an accuracy of 80% on an independent test set. This was considerably higher than results of other comparable approaches.

Golem has also been used for building rules for finite-element-mesh analysis [2] and for building temporal fault diagnosis rules for satellites [4].

4 Conclusion

Inductive Logic Programming is developing into a new logic-based technology. The field unifies induction and deduction within a well-founded theoretical framework. ILP is likely to continue extending the boundaries of applicability of machine learning techniques in areas which require machine-construction of structurally complex rules.

References

[1] W. Buntine. Generalised subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.

[2] B. Dolsak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S.H. Muggleton, editor, *Inductive Logic Programming*, London, 1992. Academic Press.

[3] S. Dzeroski, S. Muggleton, and S. Russell. Pac-learnability of determinate logic programs. TIRM, The Turing Institute, Glasgow, 1992.

[4] C. Feng. Inducing temporal fault dignostic rules from a qualitative model. In S.H. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[5] R. King, S. Muggleton R. Lewis, and M. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences (to appear)*, 1992.

[6] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[7] S. Muggleton, R. King, and M. Sternberg. Predicting protein secondary-structure using inductive logic programming, 1992. submitted to Protein Engineering.

[8] S.H. Muggleton. *Inductive Logic Programming*. Academic Press, 1992.

[9] S.H. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In S.H. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[10] S.H. Muggleton and C. Feng. Efficient induction of logic programs. In S.H. Muggleton, editor, *Inductive Logic Programming*, London, 1992. Academic Press.

[11] D. Page and A. Frisch. Generalization and learnability: A study of constrained atoms. In S.H. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[12] G.D. Plotkin. *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University, August 1971.

[13] R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[14] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. In S.H. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[15] C. Sammut and R.B Banerji. Learning concepts by asking questions. In R. Michalski, J. Carbonnel, and T. Mitchell, editors, *Machine*

Learning: An Artificial Intelligence Approach. Vol. 2, pages 167–192. Kaufmann, Los Altos, CA, 1986.

[16] E.Y. Shapiro. *Algorithmic program debugging.* MIT Press, 1983.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Towards the General-Purpose Parallel Prosessing System

Kazuo Taki

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, JAPAN
taki@icot.or.jp

1 Introduction

Processing power of the recent microprocessors grows very rapidly; It almost gets over the power of mainframe computers. Trends of the continuous improvement of the semi-conductor technology suggest that the processing power of one-chip processor devices will reach 2000 MIPS until the end of 1990s, and that the parallel computer system with 1000 processors will be installed in a cabinet which will realize 2 TIPS (tera instructions per second) peek speed.

Such a gigantic power hardware is no longer hard to imagine because recent large-scale parallel computers for scientific processing, just appeared in the market, suggests a trend of large parallel computers.

However, the software technology on the scientific parallel computers focuses on very limited application domains. Hardware design is also shifted to the applications somewhat. The parallel processing paradigm on those systems is the *data parallelism*. Problem modeling, language specification, compiling technique, a part of OS design, etc. are all based on the *data parallelism*.

The characteristics of the *data-parallel computation* are regular computation on uniform data or synchronous computation in other word. The coverage of this paradigm is limited to non-wide area of application domains, such as dense matrices computation, image processing, and other problems with regular algorithms on uniform data.

To make full use of the gigantic power parallel machines in the future, the other parallel processing paradigms, that cover much wider range of application domains, are longed to be developed.

2 New Domain of Parallel Application

Knowledge processing is the target application domain of FGCS project. Characteristics of knowledge processing problems are different much from that of scientific computations based on the data-parallel paradigm.

Dynamic and non-uniform computation often appear in the knowledge processing. For example, when a heuristic search problem is mapped on a parallel computer, workload of each computation node changes drastically depending on expansion and pruning of the search tree. Also, when a knowledge processing program is constructed from many heterogeneous objects, each object arises non-uniform computation. Computation loads of these problems are hardly estimated before execution.

These large computation problems with dynamism and non-uniformity are called *the dynamic and non-uniform problems* in this paper. When a system supports the new computation paradigm suitable for *the dynamic and non-uniform problems*, its coverage of the application domain must expand not only to the knowledge processing but also to some classes of large numerical and symbolic computation that have less data-parallelism.

3 Research Themes

The dynamic and non-uniform problems arise *new* requirements mainly on the software technology. They need more complex program structure and more sophisticated load balancing scheme than that of the data-parallel paradigm.

These items, listed below, have not been studied enough for *the dynamic and non-uniform problems* with large computation.

1. Modeling scheme to realize large concurrency
2. Concurrent algorithms
3. Programming techniques
4. Load balancing schemes
5. Language design
6. Language implementation
7. OS implementation
8. Debug and performance monitoring supports

The latter five items should be included in the topics of design and implementation of the system layer. The former three items should be included in the application layer or more general framework of software development.

4 Approach

Such an approach has been taken in the FGCS project that the system layer (including the topics 5 to 8 in section 3) was carefully tailored to suit *the dynamic and non-uniform problems* and topics of the upper layer (1 to 4) were studied on the system.

Key Features in the System Layer : The system layer satisfies these items to realize efficient programming and execution of the target problems.

1. Strong descriptive power for complex concurrent programs
2. Easy to remove bugs
3. Ease of dynamic load balancing
4. Flexibility for changing the load allocation and scheduling schemes to cope with difficulty on estimating actual computation loads before execution

Mainly, the language feature realizes these characteristics and the language implementation supports efficiency. The key language features are listed below.

- **Small-grain concurrent processes** : A lot of communicating processes with complex structure can be easily described, realizing large concurrency.
- **Implicit synchronization/communication** : They are performed between concurrent processes even in remote processors, which helps to write less buggy programs.
- **Separation of concurrency description and mapping** : Programmers firstly describe concurrency of the program without concerning with mapping (load allocation). Mapping can be specified with a clearly separated syntax after the concurrency description is finished. Runtime support for the implicit remote synchronization enables it.
- **Handling a scheduling** without destroying the clear semantics of the single-assignment language
- **Handling a group of small-grain processes** as a task

The language implementation realizes an efficient execution of these features, including a efficient OS kernel implementation of memory management, process scheduling, communication, virtual global name space, etc. [Taki 1992]. The other OS functions, which are written in the language, realize an research and development environment of parallel software including a programming system, task management functions, etc.

Research for the Upper Layer : Research topics of 1 to 4 in section 3 have been studied. After toy problems have been tested enough, R & D on practical large applications become important.

Strong cooperation of experts on application domain and on parallel processing is indispensable for those R & D. Several R & D teams have been made for each application development. Firstly, the research topics have been studied focusing on each application, then commonly applicable paradigms and schemes are extracted and supported by the system as libraries, OS functions or programming samples.

5 Current Status

System Implementation : A concurrent logic programming language KL1, which has those features listed in section 4, has been efficiently implemented on the parallel inference machine PIM. A parallel operating system PIMOS, which is written in KL1, supports an R & D environment for parallel software.

Very low-cost implementation of those features [Taki 1992] encourages the research of load balancing schemes. The language features helps the research of various concurrent algorithms and programming techniques.

Application Development : Practical large applications have been implemented [Nitta 1992], such as:

- LSI-CAD system : Logic simulation / Placement
- Genome analysis system : Protein sequence analysis / folding simulation / structure analysis
- Legal reasoning system
- *Go* game playing system
- Other eight application programs with different knowledge processing paradigms

Most of them arise dynamic or non-uniform computation. Some measurements show very good speedup and absolute speed by parallel processing.

Common Paradigms and Schemes : Efforts on extracting common paradigms and schemes from each application development have been continuing. Categorizing dynamic process structures and load distribution schemes have been carried on. Performance analysis methodologies have also been studied [Nitta 1992].

A multi-level dynamic load distribution scheme for search problems is already supported as a library program. A modeling, programming and mapping scheme based on *a lot of small concurrent objects* have been commonly used among several application programs.

6 Conclusion

New paradigms of parallel processing, that can cover *the dynamic and non-uniform problems*, are expected to expand application domains of parallel processing much larger than ever.

The dynamic and non-uniform problems must be a large application domain of parallel processing, coming next to the applications based on the *data-parallelism*. Parallel processing systems, that support efficient programming and execution of the dynamic and non-uniform problems, will get close to the general-purpose parallel processing system.

The KL1 language system, developed in the FGCS project, realize many useful features for efficient programming and execution of that problem domain. Many application developments have been proving effectiveness of the language features and their implementation.

R & D of problem modeling schemes, concurrent algorithms, programming techniques and load balancing schemes for that problem domain have started in the project, and still have to be continued. The accumulation of those software technology must make the true general-purpose parallel processing system.

References

[Nitta 1992] K. Nitta, K. Taki and N. Ichiyoshi. Experimental Parallel Inference Software. In *Proc. of the Int. Conf. on FGCS*, 1992.

[Taki 1992] K. Taki. Parallel Inference Machine PIM. In *Proc. of the Int. Conf. on FGCS*, 1992.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Hybrid Reasoning System For Explaining Mistakes In Chinese Writing

Jacqueline Castaing
Univ. Paris-Nord, Lipn / . Csp., Avenue J-B Clément
93430 Villetaneuse France
jc@lipn.univ-paris13.fr

Abstract

We present in this paper a hybrid reasoning system for Explaining Mistakes In Chinese Writing, called EMICW. The aim of EMICW is to provide students of the chinese language with a means to memorize characters. The students write down from EMICW 's dictation. In case of graphic errors, EMICW will explain the reasons of this error by using either the etymology of characters or some efficient mnemonic techniques.

EMICW has multiple representations associated to multiple reasoning methods. The coherence of the reasoning is ensured by means of a common logic formalism, the FLL-theories, derived from Girard's linear logic.

1 Introduction

The main aim of the system EMICW is to provide students of the chinese language with a means to memorize chinese characters without losing heart. The first obstacle for people accustomed to an alphabet is indeed the great number of characters to sink in. We propose to them to write down from EMICW's dictation. In the case where students are mistaken about a character, the system will explain the reasons of this graphic error either by using the origin of the character [Henshall 1988], [Ryjick 1981], [Wieger 1978], or by invoking an efficient mnemonic technique.

EMICW is a hybrid knowledge representation and reasoning system [Brachman, et al. 1985], [Kazmareck et al. 1986], [Nebel 1988]. It has multiple representations - a semantic network associated to inference rules expressed in the formalism of Gentzen's calculus [Gentzen 1969] - associated to multiple reasoning methods. The set of inference rules defines the main cases of mistakes that the author of this article and school fellows could make during their own initiation into the chinese writing. The learning methods used are given in [Bellassen 1989], [De Francis 1966], [Lyssenko and Weulersse 1987] [Shanghai Press 1982].

To ensure a coherent reasoning, EMICW has a common logic formalism, the FLL-theories [Castaing 1991], borrowed from Girard's linear logic [Girard 1987, 1989]. The system essentially performs monotonic abduction [Bylander 1991]. So, let a be the correct chinese character the student should write down from EMICW 's dictation. Let b be the actual answer given by the student. If the student is mistaken, it means that the character a is different from b, the binary predicate Error (a, b) is then set to the value true. An explanation of a graphic error consists in finding a set of first-order formulas Sigma such that a proof of the linear sequent Sigma $\vdash$ Error (a,b) can be carried out in a FLL-theory. The set of the formulas of Sigma shows the different causes of the confusion of the characters a with the character b. For example, the two characters a and b may have the same sound (they are homophonic), or they may share the same graphic components, and so on.

In this paper, we first briefly outline the history of chinese characters [Alleton 1970], [Henshall 1988], [Li 1991] [Ryjick 1981], [Wieger 1978], so the reader can appreciate how a character is made up, how it acquired its structure and will make himself an opinion on the difficulties of the chinese writing. We also give the terminology we use. In the third section, we discuss the problem of characters representation and recognition which explains the limitation of our system. Then, after describing the system EMICW (section 4), we will give in section 5, an example of explanation in the FLL-theory T. The essential point of the section 6 is the proof of the tractability of our system.

2 Chinese Writing

The chinese characters originated between 3000- 2000 B.C in the Yellow River of China. They have been the subject of numerous studies. In this paper, we limit ourselves to mentioning what is essential for a good understanding of our work.

The chinese characters, also called **sinograms** (letters from China) are written in square form with the help of strokes, for example, horizontal stroke, vertical stroke. A set of 24 strokes standardized by the Foreign Languages Institute of Beijing are now of general use (see section 3.1). Strokes must be written down according to established principles of stroke order (generally from top to bottom, and from left to right) called **calligraphic order.** A knowledge of these principles is important in order to achieve the proper shape and to write in the cursive style or semi-cursive style (the writing style of the chineses). Sinograms are monosyllabic, and each syllable has a definite tone. There are **four basic tones** in the official national language (called mandarin chinese too). The transliteration used in this article is based on the official Chinese phonetic system, called **pinyin,** which is a representation of the sounds of the language in the Latin alphabet. We mark tones with numbers from 1 to 4. Sinograms have traditionally been classified into six categories. However, in many cases the categorization is

open to difference of opinion, and one sinogram can legitimately belong to more than one category. We list below the main categories that shed considerable light on the nature of sinograms. The students should consider these categories as guides to remembering sinograms.

1. The simple pictogram: essentially a picture of simple physical object. For example, woman 女 nu3, child 子 zi3.

2. The complex pictogram: a picture of several physical objects normally indissociable. For example, good 好 hao3.

3. The ideogram: a meaningful combination of two or more pictograms chosen for their meanings. For example, from pictograms sun 日 ri4, and moon 月 yue4, the ideogram intelligent is derived: 明

4. The ideo- phonogram: the largest category, containing about 90% of the sinograms. E ssentially a combination of a semantic element with a phonetic element. For example, the ideo-phonogram seed 籽 zi3 obtained by combining the semantic element cereal 米 mi3, with the phonetic element child 子 zi3, which gives to the character its reading. In fact, only about 30% of sinograms have a real phonetic component as in the example. Chinese (as any other language which is still spoken) has changed since the origin, so the phonetic element has lost its property.

The classification of sinograms in dictionaries can be done with the help of several methods. The number of strokes method and the alphabetical order (based on the pinyin romanization) method are easy to apply. The four corner method considers particular strokes located at the four corner of the sinogram. These strokes are codified with the help of four (or five) digits, and the sinogram is located at the position given by its numerical representation. The radical method uses a particular element in a sinogram, the key element, which indicates the general nature of the character. For instance, the ideo-phonogram 籽 zi3 is located under the radical 米.

The character dictionary Xin Hua Zi Dian (eds.. 1979) lists the sinograms with respect to 189 radicals.

About five to seven thousand sinograms of up to ten or so strokes are needed in order to master the Chinese writing. The usual technique for learning consists in writing down a sinogram until it sinks in. We believe that the key to successful study of sinograms does not lie in rote learning. We propose a way to make the task a lot easier. For each case of mistake, our EMICW system gives an explanation based on the etymology of the characters. For instance, the character 天 tian1 (sky) can be confused with the following one 夫 fu4 (adult), because they have similar graphics. In fact, the character 天 comes from 大 da4 (tall), and from the graphic 一 yi1 (one), which represents a hat, while the character 夫 comes from 大, and from the graphic 一 which means a hairpin. The position of the strokes can be meaningful. If such an explanation is given to the students in case of error, they progressively will be able to correct their own mistakes by reasoning, without relying heavily on memory. Moreover, they can consider these explanations as an introduction to the history of Eastern Asia.

We list below the main cases of mistakes we have met in our study of the chinese language:

1. **Confusion of homophonic sinograms:** about 50000 sinograms share four hundred syllables. According to official statistics each syllable with its tone corresponds to an average of five distinct sinograms. So, the first difficulty for students is to distinguish the homophonic sinograms.

For example, ten 十 shi2, moment 时 shi2, and to know 识 shi2 which are homophonic sinograms can be confused in a dictation.

2. **Confusion of sinograms with similar graphics:** For example, 己 ji3, 已 yi3, 巳 si4 have similar graphics, 天 tian1, 夫 fu4 adult have similar graphics too. It happens that the mistaken graphic is not a sinogram. For example, instead of half 半 ban1, the student (the author of this article) wrote 羊.

3. **Confusion of sinograms which share the same components:** For example, 土地 di4, and 池 chi2, which share the component 也.

4. **Confusion of sinograms which form a word:** The sinograms are monosyllabic, but the chinese words are generally dissyllabic. For example, the words 身体 shen1ti3 (body), 共同 gong4tong2 (together), and 说话 shuo1hua4 (to talk). The students usually learn dissyllabic words. So, they happen to confuse a sinogram with another.

We can also mention the case of confusion of simplified forms with non simplified forms of sinograms, of missing strokes: very complex sinograms may have about thirty strokes, so missing strokes is a very frequent mistake.

3 The Graphics Capture

Students write down sinograms from EMICW's dictation. A "good" method for representing graphics should allow the system to rapidly recognize the graphics drawn which are **not automatically sinograms**, because students can be mistaken. The different classification and search techniques in dictionaries that we have mentioned in the previous paragraph, permit to locate a character, but not to correct it. For instance, the four corners method does not take into account all the strokes drawn by the student, so, cannot be used to correct mistakes. The recognition problem of sinograms has been the subject of numerous studies. The last results can be found in [Wang 1988] [Yamamoto 1991].

3.1 Data Capture

In our particular application, we have to "understand" graphics drawn by students in order to help them in case of error. Each graphic drawn is characterized by the type of strokes used, the calligraphic order of strokes, and their positions in a square. In order to capture all these data, the system displays the set of 24 standardized strokes. In fact, only six strokes are primary ones: the point 丶 (pt), the horizontal stroke 一 (hr), the vertical stroke 丨 (vt), the top to left bottom stroke 丿 (dg), the top to right bottom stroke 乀 (dd), and the back up stroke 乁 (rt). All other strokes derived from these primary ones. These strokes are implemented by means of graphical primitives such as line drawing, rectangle and arc drawing. The students arrange strokes to draw graphics inside a square, the pictures may be expanded or shrunk to fit their destination square. For instance, the sinogram 天 tian1 (sky) can be written down in the following square by means of strokes of types hr, dg, and dd, according to the calligraphic order of writing (hr hr dg dd):

3.2 Graphic Feature of a Sinogram

As the position of strokes can be meaningful, we propose to locate each stroke in terms of coordinates on a plane (the coordinate plane is a two-dimension grid, which corresponds to the square drawn above, the coordinate origin (0, 0) being at the left top corner of the square). We sort out strokes with respect to their coordinates: from top to bottom (top-down order) , from bottom to top (bottom-up order), from left to right (left-right order), from right to left (right-left order). So, every graphic is characterized by the set of following codifications: the calligraphic order of strokes, the top-down, the bottom-up, the left-right and the right-left orders of strokes. For instance, the graphic feature of the sinogram 天 tian1 is given by the calligraphic order of strokes (hr hr dg dd), the top-bottom order (hr dg hr dd), the bottom-top order (dg dd hr hr), the left-right order (hr hr dg dd), and the right-left order (hr hr dd dg). We show now how all that knowledge can be used to explain graphic errors.

4 Knowledge Representation

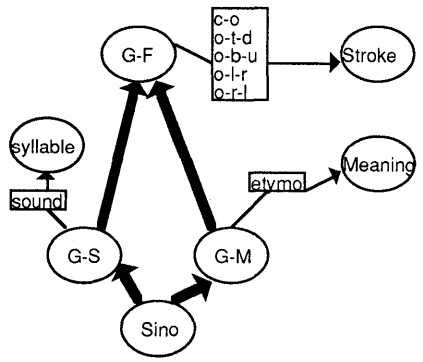
The representation language of EMICW is a restricted version of the frame-based language KL-ONE [Brachman and Smolze 1985] - for instance it does not support structural dependency relations.
EMICW has a terminological component, the data base associated to an assertional component. The assertional component is a set of rules expressed in terms of predicates which are defined in the terminological component. Let us first justify our choice, then we will describe the language.

In order to deal with all the cases of mistakes listed in section 2, we need for a representation system which allows us to define all the links of "proximity" between the objects manipulated, i.e. graphics which are (or which are not) sinograms. For instance, homophonic links between two different sinograms, or graphic similarity between a graphic and its components. The inheritance link IS-A (B IS-A A means intuitively that all instances of B are also instances of A), and the properties which correspond to roles fit very well our problem. For efficiency reasons, we have to find a trade-off between the expressive power of the representation language and the computational tractability of the relation IS-A (called subsumption relation). In [Castaing 1991], we analysed the relation B IS-A A, and we proved that providing some restrictions, a subsumption criterion can be defined. A matching algorithm based on this criterion computes subsumption in polynomial time. In the system EMICW, we increase the expressive power of our language by adding to the system an assertional component, which only deals with existential rules. In section (6), we will discuss the computational complexity of our system.

4.1 Terminological component

Concepts are labelled collections of (attribute, value) pairs. The main concepts are the following ones:
Stroke, Graphic-Feature (abbreviated as G-F),
Graphic-Meaning (abbreviated as G-M), Graphic-Sound (abbreviated into G-S), Syllable, Meaning.
Individual concepts denoted by small letters are instances of concepts denoted by capital letters.
Attributes are classified into the structural link IS-A and properties.
The **IS-A link** is used for inheritance. So, if two concepts B and A are linked by means of the IS-A link, we say that A subsumes B, and that the concept B is of **type** A.
Properties are related to the intrinsic features of concepts. The attribute values are concepts too. The main properties in the system are the following ones: c-o (abbreviation of stroke calligraphic order), o-t-d (abbreviation of top-down order), o-b-u (abbreviation of bottom-up order), o-l-r (abbreviation of stroke order from left to right), o-r-l (abbreviation of stroke order from right to left), sound (pronunciation), etymo (abbreviation of etymology).

We give below a general view of the classification of the main concepts in EMICW taxonomy. To make clear the presentation, we use an ordering graph (semantic network), where the bold arrow -> represents the IS-A relation, and the arrow -> represents the roles.



In the taxonomy given above, there are only individual concepts of type Meaning. For instance, the words tall and hat are instances of Meaning. The concepts of type Syllable correspond to the syllables of the chinese language without tone. For instance, the concept Tian is of type Syllable. An instance of the concept Tian may be tian1 (first tone). The concepts of type Stroke correspond to ordered sequences of strokes. Let Sa and Sb be two concepts of type Stroke. Sb IS-A Sa if and only if the strokes in Sa also appear in Sb in the same order. For instance, the concept Sa which corresponds to the sequence of strokes (hr dg dd) subsumes the concept Sb given by the sequence (hr hr dg dd). Intuitively, this relation means that the graphics drawn by means of the ordered sequence of strokes (hr hr dg dd) have been partially drawn by means of the ordered sequence (hr dg dd) too. The concepts of type G-F give the graphic features of sinograms. The meaning of a sinogram

is given by the property etymo, and its reading is given by the property sound. It may happen that two different sinograms have the same graphic feature. For instance, to love 好 hao4, and good 好子 hao3. So, we define concepts of type G-M (Graphic Meaning), and G-S (Graphic Sound), such that each sinogram in the data base can be considered as an instance of the concepts G-M and G-S. We now give an example of a sinogram representation.

Example-1:
Let a100 be the sinogram 天 tian1. Its graphic feature can be defined by means of the concept G-F100 which is characterized by the following (attribute, value) pairs:
G-F100 = { (c-o, (hr, hr, dg, dd)), (o-t-d, (hr, dg, hr, dd)), (o-b-u, (dg, dd, hr, hr)), (o-l-r, (hr, hr, dg, dd)), (o-r-l, (hr, hr, dd, dg))}. The sinogram a100 inherits its meaning (sound) from the concept G-M100 (G-S100) partially defined by the following sets of (attribute, value) pairs:
G-M100 = {(IS-A G-F100), (etymo, sky) }
G-S100 = {(IS-A G-F100), (sound, tian)}
So, the sinogram a100 is an individual concept of type G-F100 defined by the (attribute, value) pairs: a100 ={(IS-A, G-F100), (etymo, sky), (sound, tian1) }.
End of the Example-1.

The graphics drawn by the student during a dictation are not automatically sinograms. So, we first consider them as concepts of type G-F (Graphic Feature). We solve the recognition problem of graphics by means of a **classifier** [Brachman and Levesque 1984].

4.1.1 Classifier

Usually the role of the classifier in a KL-ONE taxonomy consists in placing automatically a concept at its proper location. For classifying concepts in EMICW taxonomy, we proceed in two steps:
1. From the graphic drawn by the student, we define the concept CG (Complete- Graphic) related to the properties c-o, o-t-d, o-b-u, o-l-r, o-r-l of the components
2. We look for the concepts A and B, such that A subsumes CG, CG subsumes B, and there does not exist a concept A' which can be located between A and CG, and a concept B' which can be located between CG and B. We place CG, and we say that CG is at its **optimal location** in EMICW taxonomy. It means that CG inherits from all its ancestors. A is said to be a **father** of CG. B is said to be a **son** of CG. In case the concepts A and B are identical, we say that CG has been **identified** with A (or with B).

4.1.2 Recognition Problem

The recognition problem consists in discovering an individual concept b of type Sino, which has the same graphic feature than CG. We proceed as follows:
1. By means of the classifier, we place the concept CG at its optimal location.
2. If CG can be identified with a concept G-Fn of type G-F, it means that there exists at least a sinogram which is an instance of G-Mn and G-Sn. Let cf be this particular instance of G-Mn and G-Sn. We identify CG with cf, and CG "wins" all the properties of cf, for example, the properties sound, and etymo. We give an example.

Example-2
Let us suppose that the graphic drawn by the student is 天 tian1 (sky). The concept CG has the following properties (after sorting out the strokes with respect to their coordinates)
CG = { (c-o, (hr hr dg dd)), (o-t-d, (hr dg hr dd)), (o-b-u, (dg dd hr hr)), (o-l-r, (hr hr dg dd)), (o-r-l, (hr hr dd dg))}. The concept CG placed at its optimal location can be identified with the concept G-F100 (see the Example-1):
G-F100 = { (c-o, (hr hr dg dd)), (o-t-d, (hr dg hr dd)), (o-b-u, (dg dd hr hr)), (o-l-r, (hr hr dg dd)), (o-r-l, (hr hr dd dg))}, and so, can be identified with the instance a100 of G-M100 and G-S100. The concept CG gains the properties sound and etymo of a100.
End of the example-2.

Our recognition procedure is a little drastic. It may happen in sinograms with multiple components that some strokes in a component have no link with those in another component. By sorting out all strokes, we consider that they are necessarily linked, so, we detect a graphic error and reject the graphic proposed by the student. Our recognition procedure suits sinograms (simple or complex) whose components are specified by the students.

4.2 Rules

The rules of the assertional component deal with the different cases of error in chinese writing. All the predicates manipulated are defined in the terminological component either as unary predicates (concepts) or as binary predicates (roles), except for the predicates Error, ≠ (different), and = (equivalence). We explain now how the confusion of sinograms can be interpreted by means of the predicate Error.

Let a be the sinogram of the dictation, and CG be the complete concept obtained from the graphic drawn by the student. The student's answer is considered correct (there is no error) if and only if:
1. The concept CG is recognized as a sinogram denoted by b.
2. The individual concepts a and b share exactly the same properties.
Two cases of error are possible:
1. The concept CG cannot be identified with a concept of type Graphic- Feature of a sinogram. It means that the graphic drawn is not a sinogram.
2 The concept CG is recognized as a sinogram denoted by b, but the sinograms a and b do not share the same properties.

In the first case, the concept CG is located at its optimal position, and has a father that we denote by B. We consider an individual concept b of type B, and we propose to explain the confusion of a with b.The choice of an individual b may depend on a strategy. For the time being in our application, we identify CG with an individual which has the **same** graphic feature as B. In the second case, we propose to directly explain the confusion of a with b. The individual concepts pointed out by our system during an explanation are the **witnesses** of the error.
The rules of the assertional component have a limited syntax. Their general form is: "If there-exists x such that P (x) then Error (a, b)", where x is a vector of variables, and P is a finite conjunction of predicates. For instance, the

rule: "If there-exists z such that Syllable(z) & Sound(a, z) & Sound(b,z) then Error (a, b)", can be used in order to explain a mistake between two sinograms a and b which are homophonic. We give some examples of rules expressed in sequent calculus formalism.

rule-1: $\exists$ z Syllable(z) & Sound(a,z) & Sound(b, z) $\vdash$ Error (a, b)

rule-2: $\exists$ u z m1, m2, G-M(u) & G-M(z) & Meaning (m1) & Meaning (m2) & m1$\neq$m2 & u $\neq$ a & z $\neq$b & Etymo (u, m1) &Etymo(z, m2) & Etymo (a, m1) & Etymo (b, m2) & Error (u, z) $\vdash$ Error (a, b)

rule-3: $\exists$ s Stroke (s) & c-o(a, s) & c-o(b, s) $\vdash$ Error (a, b)

The rule-1 deals with errors due to homophonic sinograms. The rule-2 explains that the confusion of a with b may come from a misunderstanding of the etymologies of some components of the sinograms a and b.The rule-3 stresses the importance of the calligraphic order: two sinograms with the same calligraphic order can be confused.

5 Explanation in term of Proofs

In this section, we first present a formal description of EMICW by means of the FLL-theory T, then we will give an example of explanation. The FLL-theories use a fragment of linear logic (see also [Cerrito 1990], and [Masseron et al. 1990] for some particular applications of this logic). We suppose the reader familiar with sequent calculus. In the next chapter, we will discuss the tractability of EMICW.

5.1 Formal Description of EMICW

The FLL-theories are built from the linear fragment which consists of the connectives & (conjunction), the connective γ (disjunction), and the linear negation denoted by $(\,)^\circ$. The essential feature of the fragment used is the **absence of the contraction and weakening rules** listed below:

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ (C\text{-}l) \qquad \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\ (C\text{-}r)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ (W\text{-}l) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\ (W\text{-}r)$$

The axiom and the rules of the fragment are the following ones:

Axioms : $A \vdash A$

Cut : $$\frac{\Gamma \vdash \Delta, A \quad A, \Gamma' \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ (C)$$

Exchange rules:

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, B, A \vdash \Delta}\ (Ex\text{-}l) \qquad \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, B, A}\ (Ex\text{-}r)$$

Logical rules:

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, A^\circ \vdash \Delta}\ (^\circ\text{-}l) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash A^\circ, \Delta}\ (^\circ\text{-}r)$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', (A\ \gamma\ B) \vdash \Delta, \Delta'}\ (\gamma\text{-}l)$$

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash (A\ \gamma\ B), \Delta}\ (\gamma\text{-}r)$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, (A \,\&\, B) \vdash \Delta}\ (\&\text{-}l1) \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, (A \,\&\, B) \vdash \Delta}\ (\&\text{-}l2)$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash (A \& B), \Delta}\ (\&\text{-}r)$$

$$\frac{\Gamma, A(t/x) \vdash \Delta}{\Gamma, \forall x\, A \vdash \Delta}\ (\forall\text{-}l) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \forall x\, A, \Delta}\ (\forall\text{-}r)$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, \exists x\, A \vdash \Delta}\ (\exists\text{-}l) \qquad \frac{\Gamma \vdash A(t/x), \Delta}{\Gamma \vdash \exists x\, A, \Delta}\ (\exists\text{-}r)$$

In rules ($\forall$-r) and ($\exists$-l), x must not be free in Γ and Δ

A FLL-theory can be obtained from the above fragment by adding a finite set of proper axioms S, which are sequents closed under substitution. In the cut rule given above, the formula A is the **cut-formula**. A proof in a FLL-theory is said to be **cut-free**, if all cut-formulas involved occur in some sequent of S.

In our particular application, the set of proper axioms S which completely defines the FLL-theory T is made up of two subsets S1 and S2. The subset of proper axioms **S1** corresponds to the **terminological component**. They have the general form $A \vdash B$, where A and B are literals which interprete either concepts or roles. So, the terminological component of EMICW can be formally described by the FLL-theory T1 limited to the set of proper axioms S1. The subset of proper axioms **S2** is given by the rules of the **assertional component**.

5.2 To Explain is To Prove

EMICW combines the two following different reasoning methods:

1. The **classifier** which performs inferences by means of the subsumption operation.
2. A **theorem prover** which applies the cut-rule by only using the cut-formulas which appear in the rules of the set S2.

An explanation of a graphic error consists in finding a finite conjunction of **ground formulas** Sigma = P1 &...& Pn such that a proof of the linear sequent Sigma $\vdash$ Error (a, b) can be carried out in the FLL-theory T. Let us show how we proceed generally.

1. First case: the cut-formula doesn't contain the predicate Error.

$$\frac{\text{axiom of S2}}{\text{Sigma} \vdash \exists x\ P(x) \quad \exists x\ P(x) \vdash \text{Error}(a, b)} \text{ (Cut)}$$
$$\text{Sigma} \vdash \text{Error}(a, b)$$

The proof of the sequent Sigma $\vdash \exists x\ P(x)$ consists in instantiating the existential quantifier. We define a component called **instantiation component** which performs the following operations:
1. it defines a concept CP by using the properties given in the predicates P.
2. it locates the concept CP at its optimal position with the help of the classifier, such that there exists a witness c which satisfies P in the taxonomy of EMICW.

We obtain the new sequent to be proved, Sigma $\vdash$ P(c). We "force" the proof of this sequent by setting Sigma = P(c) & P2 ...&Pn. The proof of the sequent P(c) & P2 ...&Pn $\vdash$ P(c) is now straightforward by means of the (&-l1) rule.

2. Second case: the cut-formula contains the predicate Error. We are left with the following tree:

$$\frac{\text{Sigma} \vdash \exists x\ y\ P(x,y) \ \&\ \text{Error}(x, y)}{\text{Sigma} \vdash \text{Error}(a, b)}$$

In the same way as indicated above, we use the instantiation component to point out two witnesses c and d which satisfy P. We obtain the following sequent to be proved : Sigma $\vdash$ P(c,d) & Error (c,d). We apply the (&-r) rule and we obtain the new tree:

$$\frac{\text{Sigma} \vdash P(c,d) \qquad \text{Sigma} \vdash \text{Error}(c,d)}{\text{Sigma} \vdash P(c,d)\ \&\ \text{Error}(c,d).} \text{ (\&-r)}$$

We set Sigma = P(c,d) & P2...& Pn, so, we are now left with the proof of the sequent : P(c,d) & P2...& Pn $\vdash$ Error(c,d). We progressively makes appear all the formulas of Sigma by iterating the same process.

The sequent Sigma $\vdash$ Error (a,b) may have several proofs. In this case, the system can give multiple explanations to the students. The best explanation must allow the students to better memorize the sinogram a. We think that a good criterion for the choice of the best explanation can be :
1. The presence of the predicate Etymo in the explanation with the meanings of the components
2. The shorter proof (a proof which applies the smaller number of rules).

5.3 An Example of Explanation

Let us explain the confusion of the sinogram 天 tian1 (sky) with the sinogram 夫 fu4 (adult) by means of proofs. Etymologists give the following explainations: the sinogram sky comes from a person standing with arms spread out to look as **tall** as possible 大 with a big head (or a **hat**) symbolised by the stroke 一 . The sinogram adult comes from tall 大 with an ornamental **hairpin** through his hair (a sign of adulthood in ancient China) symbolised by the stroke 一 . So, we propose the following taxonomy:
1. The concepts G-F90 and G-M90 give the graphic feature of the sinogram 大 da4 (tall), and its etymology
G-F90 = { (c-o, (hr dg dd)), (o-t-d, (dg hr dd)), (o-b-u, (dg dd hr)), (o-l-r, (hr dg dd)), (o-r-l, (hr dd dg))}.
G-M90 = {(IS-A, G-F90), (etymo, tall)}.
2. The concept G-F01 corresponds to the graphic feature of the sinogram 一 yi1,
G-F01 = {(c-o, (hr)), (o-t-d, (hr)), (o-b-u, (hr)), (o-l-r, (hr)), (o-r-l, (hr))}. As the sinogram 一 yi1 has (at least) two different origins, hat and hairpin, we define two concepts of type G-M:
G-M010 = {(IS-A, G-F01), (etymo, hat)}
G-M011 = {(IS-A, G-F01), (etymo, hairpin)}
3. The concept G-M100 defined as {(IS-A, G-F100), (etymo, sky)} (see the example-1 of section 4.1) can be located now as:
G-M100 = {(IS-A , G-M90), (IS-A, G-M010)}.
The sinogram 天 tian1 represented by the individual concept a100= {(IS-A, G-M100), (IS-A, G-S100), (sound, tian1), (etymo, adult)} inherits the properties (etymo, tall) and (etymo, hat) from the concepts G-M90 and G-M010.
In the same way, the sinogram 夫 adult is represented by the individual concept b100= {(IS-A, G-M110), (IS-A, G-S110), (sound, fu4), (etymo, adult)}, and inherits the properties (etymo, tall), (etymo, hairpin) from the concepts G-M90 and G-M011.

In order to prove the sequent Sigma $\vdash$ Error (a100, b100), we propose to apply the cut-rule (C) with the cut-formula appearing in the rule-2:

$\exists$ u z m1 m2 G-M(u)& G-M(v) & Meaning (m1) & Meaning (m2) & m1 $\neq$ m2 & u $\neq$ a & z$\neq$ b & Etymo(u, m1) &Etymo(z, m2) & Etymo (a, m1) & Etymo (b, m2) & Error(u, z) $\vdash$ Error (a, b). We are left with the following sequent to be proved :

Sigma $\vdash \exists$ u z m1 m2 G-M(u) & G-M(v) & Meaning (m1) & Meaning (m2) & m1 $\neq$ m2 & u $\neq$ a100 & z$\neq$ b100 & Etymo(u, m1) & Etymo(z, m2) & Etymo (a100, m1) & Etymo (b100, m2) & Error(u, z).
The instantiation component instantiates the variable m1 to hat , and the variable m2 to hairpin (it has only this possibility), and defines the two individual concepts uC and vC whose etymologies correspond to these meanings: uC ={(IS-A, G-M010), (etymo, hat)}, and vC= {(IS-A, G-M011) , (etymo, hairpin)}.
Then, Sigma contains the following main ground formulas: Etymo(uC, hat) & Etymo(vC, hairpin) which shows that the reason of the confusion of a100 with b100 comes from a misunderstanding of the origins of the component 一 yi1 which appears in these two sinograms.
We invite the reader to try to apply the rule-3 in place of the rule-2. He will find that the confusion of a100 with b100 may come from the fact that these two sinograms have the same calligraphic order.

6 Computational Complexity

In this chapter, we prove that EMICW is tractable. The main problem comes from subsumption. The subsumption opeartion has been particularly analysed in [Levesque and

Brachman 1987] and in [Schmidt-Schaub 1989]. Their approach are mainly based on semantics. In [Castaing 1991], we have characterized a subsumption criterion by means of proofs in FLL-theories as T1 (see section 5.1). We briefly explain how we have proceeded.

6.1 Tractability of Subsumption

Let A and B be two concepts. We interpret A and B by means of first-order formulas, as in Brachman-Levesque's interpretation, then, we replace all classical connectives with linear ones. Let Ac = $\exists$xA1(x) &...An(x), and Bc = $\forall$zB1(z)&...Bm(z), (where z and x can be vectors of variables, and Ai(x) = $A_{i1}(x)$ γ...γ $A_{ip}(x)$, Bj(z) = $B_{j1}(z)$ γ...γ $B_{jq}(z)$) be the conjunctive normal forms obtained. A subsumes B iff there exists a cut-free proof in T1 of the sequent Bc $\vdash$ Ac. In the absence of contraction and weakening, we proved the following result :

Theorem (subsumption criterion): A subsumes B iff Ac and Bc satisfy the following **condition (C)**: there exists a, a substitution for x such that for each Ai, 1$\leq$ i $\leq$n, there exists some Bj, 1$\leq$ j $\leq$m, and b, a substitution for z, such that there exists a cut-free proof of the sequent Bj.b $\vdash$ Ai.a in the FLL-theory T1.

A matching algorithm can be easily derived from the condition C. It computes subsumption in polynomial time proportional to the length of the concepts, and to the cardinality of the set of proper axioms S1.

Without contraction and weakening, FLL-theories are decidable. There exists other decidable first-order theories which are based on classical logic [Ketonen and Weyhrauch 1984], or [Patel-Schneider 1985, 1988]. The originality of our approach comes from the way we deal with the universal quantifiers (or with the existential ones). Let us show how we can explain the rise in complexity of subsumption by means of contraction. We consider the following cases:

1. B_C and A_C satisfy condition (C) (the contraction rule is absent): the sequent $B_C \vdash A_C$ is provable in polynomial times, then the complexity of subsumption is **polynomial**.

2. Bc and Ac do not satisfy condition (C): let us suppose that the sequent $B_C \vdash A_C$ is provable (for example, by means of an approach based on semantics), and the proof of the sequent $B_C \vdash A_C$ necessitates the use of the contraction rule, (and possibly of the weakening rule): the search procedure for a proof can make sequents of the form $\forall z\, B(z,a) \vdash \Delta$, (or of the form $\Gamma \vdash \exists x\, A(x, a)$) appear at the nodes of the search-tree. Let us consider the case, where the sequent $\forall zB(z,a) \vdash \Delta$ appears at a node of the search-tree: the search procedure can go back-up the tree by applying the universal and contraction rules. We can be left with the following tree:

$$\dfrac{\dfrac{B\,(b\,/\,z,\,a),\ \forall z\ B(z,a)\ \vdash \Delta}{\forall z\ B(z,a),\ \forall z\ B(z,a) \vdash \Delta}(\forall\text{-l})}{\forall zB(z,a) \vdash \Delta}(\text{C-l})$$

The use of contraction may open a branch which terminates with a failure. Some back-tracking is then necessary. The complexity of the subsumption in this case is NP-hard.

3. $B_C \vdash A_C$ is not provable, then the use of the contraction rule may lead to duplicate infinitely the same formulas in the case where the set of instantiation terms (such as b) is infinite (for example in presence of functions) :

$$\dfrac{\dfrac{\dfrac{\vdots}{B(b/z,a),\forall zB(z,a),\forall zB(z,a)\vdash\Delta}(\text{C-l})}{B(b/z,\,a),\forall zB(z,a)\vdash\Delta}(\forall\text{-l})}{\forall z\ B(z,a),\forall z\ B(z,a) \vdash \Delta}(\text{C-l})$$
$$\forall zB(z,a) \vdash \Delta$$

The subsumption turns to be **undecidable.**

6.2 Tractabilty of EMICW

The terminological component of EMICW has a restricted syntax. The condition (C) defined above gives an adequate subsumption criterion. In order to locate a concept at its optimal location, the classifier performs the subsumption operations in number limited by the diameter of the semantic network. Its computational complexity is then limited. The theorem prover applies the cut-rule, with cut-formulas in some sequents of S2 (see section 5.2). Without contraction, the existential formulas which appear are **never duplicated**, and so, are only instantiated by means of the classifier. The cardinality of S2 is finite. Then, the proof of the sequent Sigma $\vdash$ Error (a,b) can also be carried out in limited time depending on the cardinaly of the set of proper axioms S= S1 + S2. The tractability of our system is then ensured.

Conclusion

A prototype of our EMICW system is implemented in LISP. For the time being, if the student writes down a graphic which is not recognised as a sinogram, the system has no particular strategy for discovering a "good" witness of the error. We are now investigating a strategy of choice of witnesses, which can take the context of the dictation, (the sinograms that the student have already drawn during the dictation) into account. Providing adequate rules, EMICW can also help students to learn japanese characters (kanjis) with the chinese or the japanese reading, or to learn classical vietnamese characters (nôm).

Acknowledgements

I would like to thank the four FGCS 's referees for their comments which contributed to clarify the presentation of my work. Discussions with my colleagues of PRC-IA were very helpful. The contribution of J-L. Lambert and C. Tollu to this work was invaluable. Thanks to both.

References

[Brachman R.J and Levesque H.J.1984]: "The Tractability of Subsumption in Frame-Based Descrition language" Proceedings AAAI-84, August 84, pp34-37.

[Brachman R.J and Smolze J.G. 1985] : "An overwiew of the KL-ONE Knowledge Representation System". Cogitive Sci. 9(2) (1985) 171- 216.

[Brachman R.J and Gilbert V.P and Levesque H.J. 1985]: "An Essential Hybrid Reasoning System:Knowledge and Symbol Level Accounts of KRYPTON " Proc. 9th IJCAI (1985) Los Angeles. pp 532-539.

[Bylander T. 1991]: "The Monotonic Abduction Problem: A Functional Characterization on the Edge Of Tractability". Principles Of Knowledge Representation and reasoning Proceedings of the Second Internatipnal Conference. Cambridge, Massachusetts. April 1991.

[Cerrito S. 1990]: " A linear semantics for Allowed Logic Programs" Proc. 5th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1990, 219-227.

[Castaing J. 1991]: "A New Formalisation Of Subsumption In Frame-Based Representation Systems". Principles Of Knowledge Representation and Reasoning Proceedings of the Second International Conference. Cambridge, Massachusetts. April 1991.

[Girard J. Y. 1987] : "Linear Logic" Theorical Computer Science 50 (1987) .pp1-102.

[Girard J. Y. 1989] :" Towards a Geometry of Interaction" Proc. AMS Conference on Categories, Logic and Computer, Contemporary Mathematics 92, AMS 1989).

[Gentzen G.1969] :" The Collected Papers of Gerhard Gentzen" Ed. E; Szabo, North-Holland, amsterdam (1969).

[Kazmareck T.S and Bates R and Robbins G.1986]: "Recent Developments in NIKL" Proc. AAAI-86. Philadelphia, pp 978-985.

[Ketonen J. and Weyhrauch R. 1984] : "A decidable fragment of predicate calculus" Theorical Computer Science 32:3, 1984.

[Levesque H.J and Brachman R.J. 1987]: "Expressiveness and Tractability in Knowledge Representation and Reasoning, Comp. Intell. 3 (2) (1987) pp 78-93.

[Masseron M. and Tollu C. and Vauzeilles J. 1990] : "Generating plans in linear logic" Proc. FST & TCS 10, Bengalore (India), Dec. 1990.

[Nebel B. 1988]: "Computational Complexity of Terminological Reasoning in BACK". Artificial Intelligence 34 (1988) pp371-383.

[Patel-Schneider P.F. (1985)] : "A Decidable First-Order Logic for Knowledge Representation" Proceedings 9th. IJCAI (1985). Los Angeles. pp 455-458.

[Patel-Schneider P.F 1988]: "A Four-Valued Semantics for Terminological Logics" Artificial Intelligence. 36 (1988) pp 319- 353.

[Schmidt-Schaub M. 1989]: "Subsumption in KL-ONE is Undecidable" First International Conference on Principles of Knowledge Representation.1989. pp 421-431.

[Wang P.S.P 1988]: "On Line Chinese Character recognition" 6th IGC Int. Conference on Electronic Image pp209-214 1988.

[Yamamoto Y. 1991]: "Two-Dimensional Uniquely Parsable Isometrc Array Grammars". Proceedings of the International Colloquium On Parallel Image Processing Paris June 1991.

[Alleton V.1970]: "L'Ecriture Chinoise". Que sais-je N° 1374

[Bellassen 1989]: "Méthode d'Initiation à la Langue et à l'Ecriture chinoises". Eds. La Compagnie/ Bellassen 1989.

[De Francis]: "Character Text for Beginning Chinese" Yale Language Series. New haven and London, Yale University Press.

[Henshall Kenneth G 1988]. : "A Guide to Remembering Japanese Characters" Charles E. Tuttle Company, Inc. of Ruland, Vermont & Tokyo, Japan 1988.

[LI XiuQin 1991]: "Evolution de l'Ecriture Chinoise". Librairie You Feng Paris1991.

[Lyssenko N. and Weulersse D.]: "Méthode Programmée du Chinois Moderne" Eds. Lyssenko Paris 1987.

[Ryjick K. 1981] : "L'Idiot Chinois". Payot 1981

[Shangai Foreign Language Institute 1982]: "A Concise Chinese Course For Foreign Learners" (Books 1 and 2). Shangai Foreign Language Institute Press 1982.

[S.S Wieger 1978]: "Les caractères chinois" Taichung 1978

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Automatic Generation of a Domain Specific Inference Program for Building a Knowledge Processing System

Takayasu Kasahara, Naoyuki Yamada*, Yasuhiro Kobayashi*,*

*Katsuyuki Yoshino**, Kikuo Yoshimura***

**Energy Research Laboratory, Hitachi, Ltd., 1168 Moriyama-cho, Hitachi-shi*

Ibaraki-ken, Japan 316 Tel. (0294) 53-3111

***Software Development Center, Hitachi, Ltd., 549-6, Shinano-cho, Totsuka-ku*

Yokohama-shi Japan 244 Tel. (045) 821-4111

Abstract

We have proposed and developed an expert system tool ASPROGEN(Automatic Search Program Generator) having a built-in the automatic generation function of a domain specific inference program. This function was based on search-based program specification and an abstract data type of search. ASPROGEN has interfaces for domain knowledge using an object-oriented approach and constraints which represent control knowledge. It is described by using domain knowledge and it can cover a detailed problem solving strategy

We applied ASPROGEN to produce three kinds of scheduling systems. These generated systems have equivalent performance in comparison with knowledge processing systems implemented by the conventional tool. Further, a two-thirds reduction of the program step numbers required as programmers' input was realized.

1. Introduction

Current expert system tools based on production rule and/or frame representation provide an environment to generate expert systems through formalizing and describing problems by production rules. They are powerful tools, and many practical expert systems have been produced by using them. Industrial field applications of expert system tools have sometimes met problems, the most important one being that tools based on the production system only prepare a rule based language, not a problem solving strategy. So, mapping the problem solving strategy to the production rules is difficult for users who are not knowledge engineers.

Domain shells[1], tools based on generic task method[2], half weak method[3], and SOAR[4] have been developed to overcome this difficulty. Domain shells are expert system tools which are restricted to the specified problem regions such as diagnosis, scheduling, and design. They have spread sheet type user interfaces and problem-specific inference programs. But, actual industrial problems include particular conditions, constraints, or problem solving knowledge, and domain shells do not have enough flexibilities to cover all of them. This leads to a conflict between tool flexibility and easy use. In general, the tool becomes more specific to some regions, so it becomes easy to use it, but loses flexibility.

The generic task method and half weak method also have this conflict. The generic task method classifies problem solving methods into several types which are called generic tasks, and prepares generic task tools to provide them. Tool users select an appropriate generic task and supply domain knowledge to develop the knowledge processing system. The half weak method regards problem solving as a search and provides pre-defined search modules. Tool users select an appropriate search module and add domain knowledge to the module. However, these methods, based on classification, do not necessarily give directions for systematic preparation of building blocks of knowledge processing systems. So, tool users must reformulate the problem definition according to the prepared building blocks.

SOAR has more flexibility for defining the problem solving strategy. It can generate a search program by defining several search control rules. But, lack of functions to relate the search program and domain knowledge restricts the applicability of SOAR to toy problems.

Then, we developed ASPROGEN(Automatic Search Program Generator). ASPROGEN is an expert system tool having a built-in automatic generation function of a domain specific inference program was built. To specify the problem to be solved, it has interfaces for describing the problem solving strategy as a search strategy, domain knowledge in an object-oriented way, and the detailed problem solving strategy as constraints among the attribute values of the domain objects.

2. Overview of ASPROGEN

2.1 Building expert systems based on search

ASPROGEN has no embedded inference mechanism. Instead, as shown in Fig.1, its parts include the search program and search program generating mechanism which produces inference programs according to user specifications of the search program, domain knowledge, and detailed constraints.

The reason why we use search as the inference program specification is that it covers almost every inference mechanism required for expert systems, and it is simple. But, a search is not easy to describe nor is it easy to prepare controls tightly directed to a particular problem by the search strategy.

To describe detailed control strategies, ASPROGEN includes an interface for domain knowledge. Using the domain knowledge, the detailed controls or problem solving strategy can be described as constraints between attribute values of domain knowledge. The detailed control programs are complicated in the case of a scheduling system or CAD systems, and it is important to support their generation. In general, domain specific inference programs which have functional operators have complicated constraints. ASPROGEN combines these constraints to global search strategies, and generates domain specific inference programs.

ASPROGEN users develop expert systems by following this procedure:

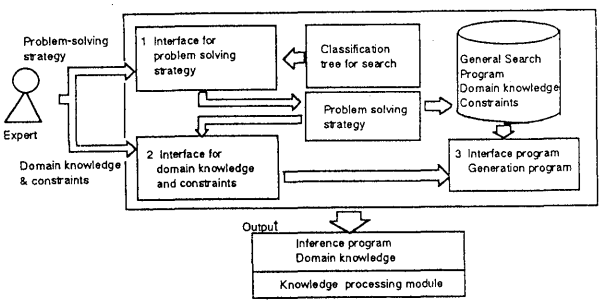


Fig. 1 Overview of ASPROGEN

(1)Users specify a problem solving strategy from the viewpoint of a search strategy.

(2)Users input the search strategy by selecting the classification items of the search classification tree which the tool prepares. This step is executed with the help of the tool interface.

(3)Users input domain knowledge and constraints with the help of the tool interface.

(4)ASPROGEN generates a domain specific inference program and data structures for domain knowledge.

Although (1) is an interesting problem, we limit the present discussion to (2)-(4).

2.2 Specification of problem solving strategy

To specify the problem solving strategy as a search, we define a classification tree for the search strategy and a template of the search program.

Figure 2 shows the classification tree. It comes from analyzing search trees used in various kinds of problem solving. A search tree consists of nodes and operators. We retrieve the classification items from the characteristics of the nodes and operators. The first classification item comes from the characteristics of the operators. There are two operator types. One is a functional operator which creates new nodes from parent nodes and adds them to the search tree. In the scheduling search program, a functional operator is used. The other type is a link operator. The link operator is used in the diagnosis search program which selects suitable diagnosis nodes for the observed state.

The second classification item comes from the characteristics of the nodes. They are evaluation functions to select nodes in the search procedures, pruning functions, establishment conditions, and so on. The evaluation functions define a global search strategy, for example one which prefers the deepest nodes of the search tree corresponds to the depth first search. The characteristics of the search nodes are described by specification values of the nodes in the search tree which are depth, breadth, parent relations, sibling relations, and node attributes values. Their values are retrieved from the structure of the search tree, and we can prepare these specification values or functions to calculate them. On the other hand, node attribute values cannot be retrieved from the structure of the search tree, and it is difficult to cover all attribute values of the nodes to specify the problem solving strategy.

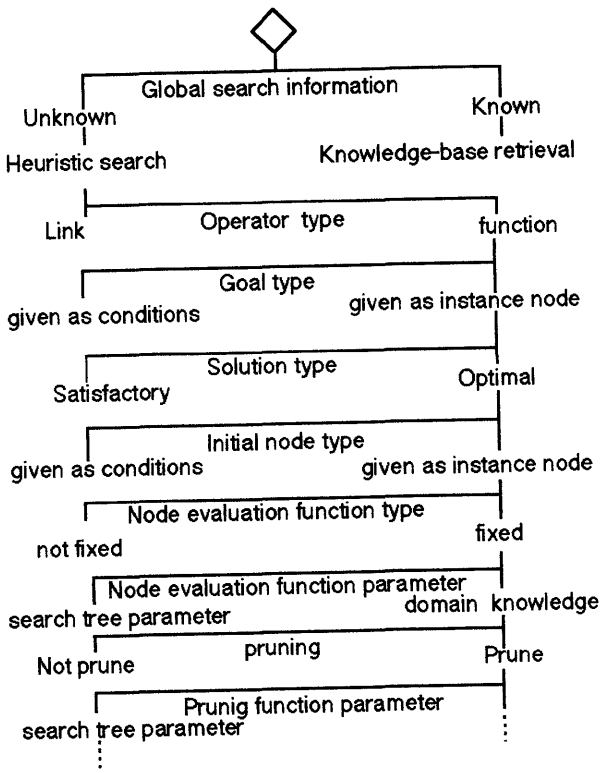


Fig. 2 Search classification tree

To mitigate this difficulty, we rank the attribute values from the viewpoint of their relation to the search tree operators. The attributes which search for the operator directly are called first-order attributes. For example, in the scheduling system starting time and ending time of each job are first order, and the resource constraints are not, if search operators are functions which adjust job scheduling. To describe programs in detail, not only first-order attributes but also multi-order attributes or variables are required. The first-order attributes and the multi-order attributes are domain knowledge. We do not embed detailed domain knowledge in ASPROGEN, instead an

interface is prepared to describe the domain knowledge and constraints of attributes of domain knowledge and global search strategy. By combining the global search strategy, described as a search strategy, and domain knowledge, ASPROGEN covers not only toy problems, but also applications for industrial uses.

2.3 *Representation of domain knowledge and constraints*

ASPROGEN has an interface for describing the domain knowledge. Domain knowledge is described by objects and attributes, attribute value ranges, and attribute constraints. There are two types of objects. One is a class objects which defines attributes, and relations between other objects. The other type is an instance object which has instantiated attribute values.

Figure 3 shows a representation scheme of domain knowledge for ASPROGEN. Nodes of the search tree are also objects. Node objects are related to other objects. The relations among objects are of three types.

(1) Class-instance relations: Instance objects have the same attributes as class objects, and the values of the attributes are inherited from the class objects.

(2) Attribute-value relations: The value region of the attributes can be described by he class objects. Thus, the attribute value region is a set of instance objects of the class objects.

(3) Attribute-object relations: The attributes of the objects can be described by the class objects. Thus, the attributes of the nodes are instance objects of the class objects, and attribute values are those of the instance objects.

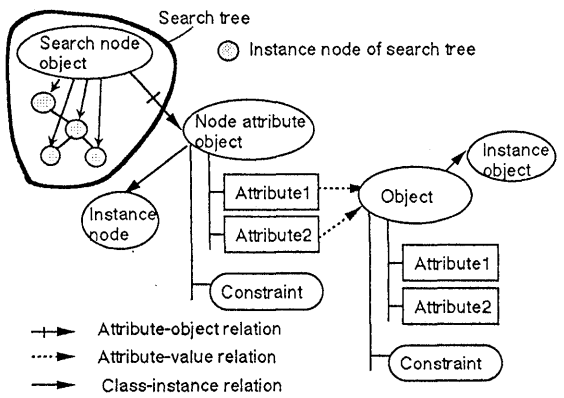


Fig. 3 Scheme of knowledge reprsentation in ASPROGEN

On the basis of these definitions of domain knowledge, ASPROGEN users describe constraints. ASPROGEN prepares a simplified language which can describe constraints by using object names and attributes.

2.4 *Generation of problem-directed inference program*

The inference program generated by ASPROGEN consists of two parts, the search program which corresponds to the global problem solving strategy, and constraint satisfaction programs which correspond to the domain knowledge. Figure 4 shows an outline of the inference program. The control program is embedded into ASPROGEN, and the global search program and constraint satisfaction programs are generated according to user input. If the inference program is completed, it behaves as follows. Using the global search strategy, the inference program activates an operator and generates or selects new node. Then, the constraint satisfaction programs activate and adjust the attribute values of the objects for every constraint. According to the result of the constraint satisfactions, the operator is activated again. This process continues until termination conditions are satisfied. The generating process of the inference program consists of three steps.

(1) Generate the search program which represents a global search strategy

ASPROGEN has a general search program which is independent of domain and includes six search sub-functions as shown in Fig. 5. When completed, it becomes the global search program of Fig. 4. Constraint satisfaction programs are activated in the sub-function of 'Apply operator'. The difference between each search strategy is reflected in the difference of the six element functions.

ASPROGEN prepares two reference tables and abstract data types for search[5],[6]. Parent function parent(c,k) which returns the parent of the node k of search tree c, and Left_most_child(c,k) which returns a child node which was first generated or selected are examples of abstract data types for search. Here, an abstract data type of search makes up the functions for the search program. The first reference table is a table intended for generation of search element functions.

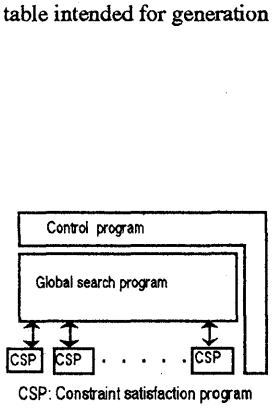


Fig. 4 Outline of the inference program

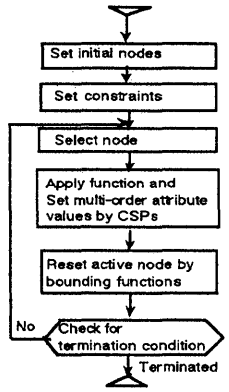


Fig.5 General search program

Search element functions are program parts of sub-functions and Fig.6 shows some. They consist of the abstract data type of search and domain-dependent search functions. Examples functions are node evaluation function for domain dependent corresponding search element functions to the userspecified problem solving strategy.

Figure 7 shows the generating process of the problem-directed inference program. Referring to the problem solving strategies using the reference table, the system decides the search element function. This is done by domain dependent search control functions such as evaluation function for node.

Then, the same as in the definition process, sub-functions are defined by abstract data types of the search tree.

Functions concerning search tree configuration	
(1) parent(c,k)	:returns parent of c in search tree k.
(2) Leftmose_child(c,k)	:returns eldest son of c in search tree k.
(3) Right_sibling(c,k)	:returns next younger brother in search tree k.
(4) Label(c,k)	:returns label of c in search tree k.
(5) Root(k)	:returns root node of k.
(6) Clear(k)	:makes search tree k null set.
(7) Deep(c,k)	:returns depth of c in search tree k.
(8) Height(c,k)	:returns height of c in search tree k.
(9) Leaf(c,k)	:if c has no children returns yes; otherwise, return no.
Functions concerning search tree operations(search element functions)	
(10) Evaluate(c,k)	:evaluate c, and returns evaluation value.
(11) Change_e(n_t,S,k)	:change evaluation function of node type n_t to S.
(12) Search state(c,k)	:if c is an open node returns Current; if c is a removed node returns Finished; otherwise returns Yet.
(13) Jumping(c,k)	:returns the node, when c is established.
(14) Back_tracking(c,k)	:returns the node, when c is not established.
(15) Initial(c,k)	:if node c is initialized state returns yes, otherwise returns no.
(16) Kill(c,k)	:removes node c from the active node.
(17) Active_c(k)	:returns active node of search treeof k.
(18) Goal(c,k)	:if c is a goal node returns yes.
(19) Cond_node(n_t,c,k)	:if node c satisfies establish condition of node type of n_t, returns yes, otherwise returns no.
(20) Cond_node_type(n_t,c_t,c,k)	:if node c satisfies establish conditions of node type of n_t, returns yes, othewise returns no.
(21) Establish(c,k)	:if c is established returns yes, otherwise returns no.

Fig. 6 Search element functions

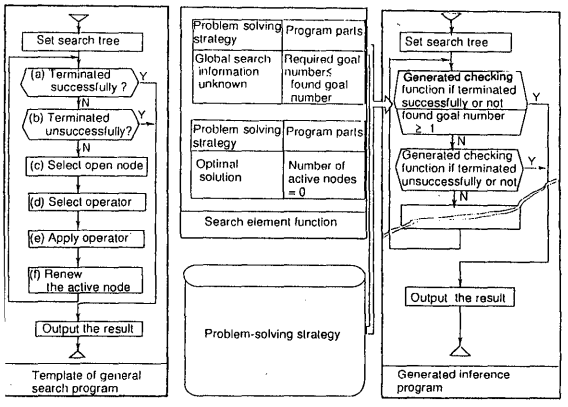


Fig. 7 Generating function of program-directed inference program

Figure 8 shows an example of the element function generating process, which exemplifies the node evaluation function. According to the user-input problem solving strategy that the depth of the tree has a high evaluation value, the tool selects the depth function from abstract data types and completes the node evaluation function.

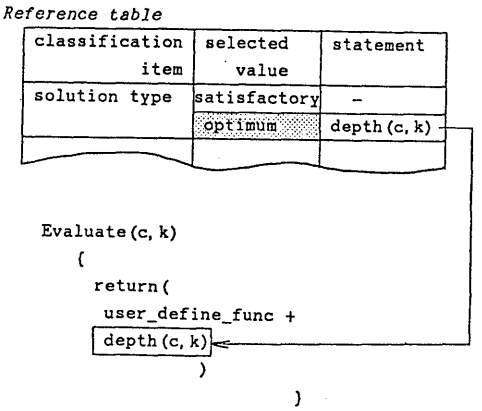


Fig. 8 Generation Example for search element functions

Figure 9 shows an example of a sub function generating process, which exemplifies function (a) in Fig.5 which is named SUCCES_END(c,k) here. Since the optimal solution is requested in the problem-solving strategy, the tool generates the checking successful termination function which terminates the inference program only if an optimal solution is found.

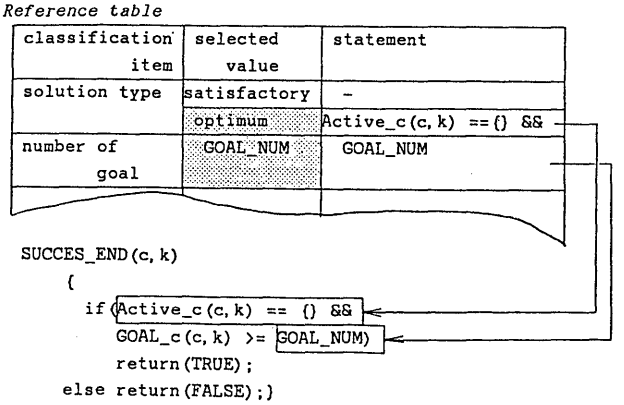


Fig. 9 Generation Example for search sub-functions

(2) Generate the constraint satisfaction programs, according to the user specifications in simplified language

The object names and attribute names of the objects which the tool users input are registered in the ASPROGEN as key words for simplified language constraints description. We call the language, SCRL (Simple Constraint Representation

terminal symbols. The SCRL compiler accepts only the following style sentence.

[value clause] [comparing key words] [value clause]

A value clause consists of object name, attribute name and object relation key words.

Table 1 lists key words and their meaning in SCRL. There are set operation key words, comparison key words, and object relation key words such as 'of'. Figure 10 shows an example of a constraint described by SCRL in which the number of persons required for each time span is less than the available personnel number.

```
Constraint name: personnel

sum((time_span of job), person )
    >
time_span of  avairable_person
```

Fig. 10 Example of constraint

Table 1 Example of key words of the SCRP

Set operation keywords

- A include B: $A \supseteq B$
- A have e: $e \in B$
- SUM(A,B): sum up attribute value of B of all instance object of A

Comparing keywords

- x > y
- x = y

object relation key words

- A of y: value of attribute y of object A.

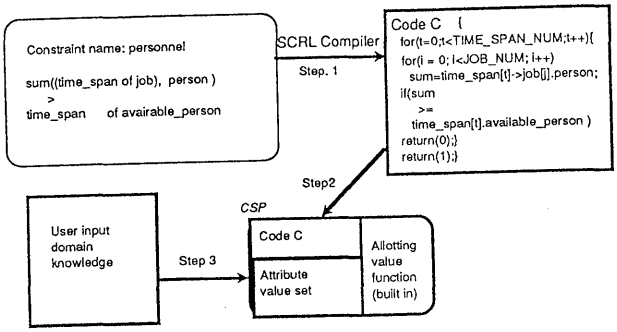


Fig. 11 Generating process of constraint satisfaction program

ASPROGEN generates constraint satisfaction programs from the kernel of the constraint satisfaction programs. This kernel of the constraint satisfaction programs comes from the relation of attributes and their value range. The allotting mechanism of the attribute values is built to ASPROGEN. The mechanism selects values from the value range. If constraints are not satisfied, other values are selected. ASPROGEN generates each constraint satisfaction program by setting the object attribute values and their range.

Figure 11 shows an example of the constraint satisfaction program generating process. The constraint is like in Fig. 10, a personnel constraint. First, the sentence is pursed into the C language by the SCRL compiler (code C in Fig. 11). And attribute value range and code C are set to the allotting function which ASPROGEN prepares, and the constraint satisfaction program is completed.

(3) Synthesize all constraint satisfaction programs and the search program

Finally, ASPROGEN synthesizes all the constraint satisfaction programs and the search programs, and generates the domain specific inference program. The key point of the synthesis is to ensure consistency of the attribute values of the objects which the tool users define. To make the argument clear, we define the identity of the search node and scope of the attribute values.

Identity of the search node

The identity of the node is defined by equality of the value set of the first-order attributes (cf. Section 2.2). Search tree operators operate them directly. So, it is possible that the inference programs generate different results, though the problem solving strategies are the same.

Scope of the attribute values

We define scope of the attribute values in the search tree node. The attribute value of the objects should have a consistency in the tree node, and the change of the attribute values in the process of the constraint satisfaction must propagate to other constraints.

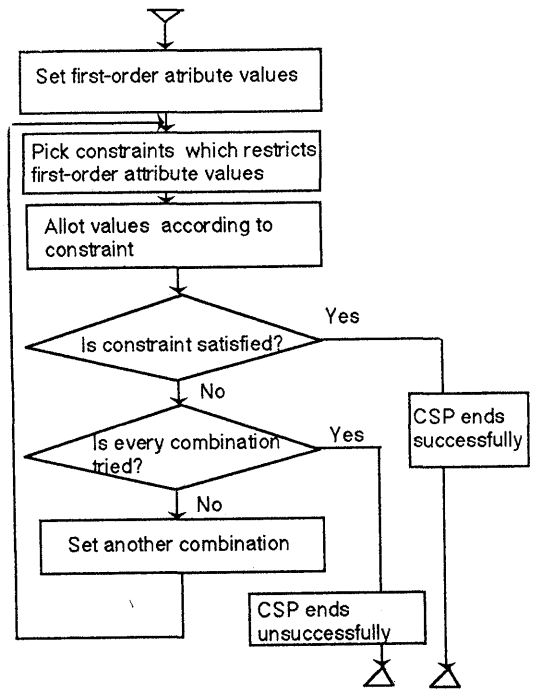


Fig. 12 Simplified procedure for constraint satisfaction

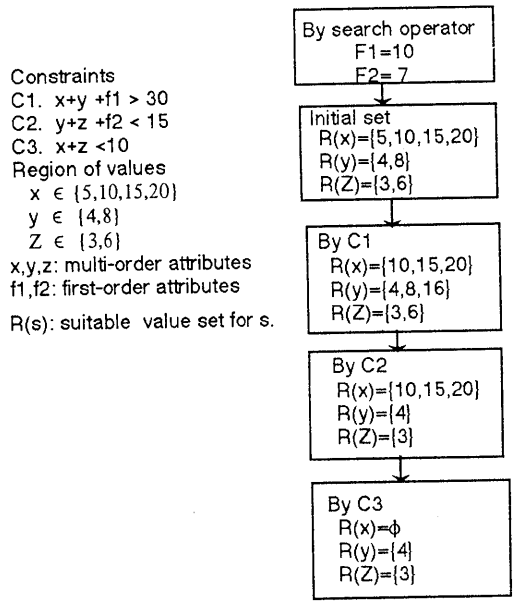


Fig. 13 Example of filtering process

Figure 12 shows a simplified mechanism to assure consistencies of the attribute values. At first, using the search tree operator, first-order attribute values are instantiated. In the next step, attributes which are constrained by the first-order attributes are instantiated by the allotting mechanism of the attributes. This process continues to survey all constraints. If the set of attribute values is found, then the first-order attribute set is suitable, and if not so, the node is unsuitable. But, this simple algorithm has a fatal defect, i.e. ineffectiveness of the allotting process. If global consistency among the constraints does not exist, the algorithm searches for every combination of the attributes until no solution is found.

To avoid this ineffectiveness ASPROGEN deals with attributes as a set. In the first stage, using the search tree operator, first-order attribute values are instantiated. Then the available value set of multi-order attributes are filtered by the constraints. Figure 13 shows a simple example of the filtering process. At first the region of the attributes value set is a candidate for solution. Filtering by the constraints, inconsistent values are retrieved from the candidate set. The process continues until no retrieval value exists/or no suitable value exists for some attributes.

4. Example and Result

Using ASPROGEN, we built three kinds of scheduling systems. They were a maintenance scheduling system (Problem A), construction scheduling system (Problem B), and jobshop scheduling system (Problem C).

Problem A is a scheduling system for maintenance scheduling of a nuclear power plant[7]. The generated program produces a schedule under constraints of maintenance personnel limitations and interferences between tasks. Problem B is a plant construction scheduling program. The generated program produces a schedule under previous relations between tasks and personnel limitations. Problem C is a jobshop scheduling system. The generated program produces a schedule under constraints of resource limitations and appointed date of delivery.

Table 2 Test problems

Problem	Characteristics of solution	Evaluation function	Constraints	Variety of resources
Maintenance scheduling	optimal	working time	· task interference	2
Construction scheduling	satisfactory		· task execution order	1
Jobshop sche	satisfactory			3

The problems are shown in Table 2. Table 3 summaries the problem solving strategy for each scheduling problem. These problems differ regarding solution type and resource numbers.

Figure 14 shows the domain model of each problem. They are the basis of ASPROGEN input. The framework of these problems is the same. This means that global search strategies are the same. First-order attribute values are the starting and ending times of each jobs. Preference of the node is total scheduling time. There are interference constraints that some jobs cannot be executed simultaneously. Domain knowledge differs. For example, problem A and problem B have personnel limitations, and problem C has machine constraints.

Table 3 Specification of the test problem-- task specific knowledge

Items	Definition of problem solving method		
	maintenance scheduling	construction scheduling	jobshop scheduling
Number of goal	1	all	1
Initial number	1	1	1
Global search information	none	none	none
Type of operator	function of adjusting schedule		
Type of initial state	state of representing work schedule		
Type of goal state	conditions that satisfies all constraints		
Solution type	optimal	satisfactory	satisfactory
Establish conditions about tree configuration	none	none	none
Evaluation function	fixed	fixed	fixed

Figure 15 shows program step numbers which programmers input. Comparing the inference programs implemented by using a conventional tool[8], equivalent performance is realized with two-thirds reduction in number of program steps required as programmer input. Of course, the reduction rate depends on the applications, for example, a diagnosis system has more domain knowledge, and the reduction rate may be smaller than for a scheduling system. But, overall some reduction of programmers load will result by the tool.

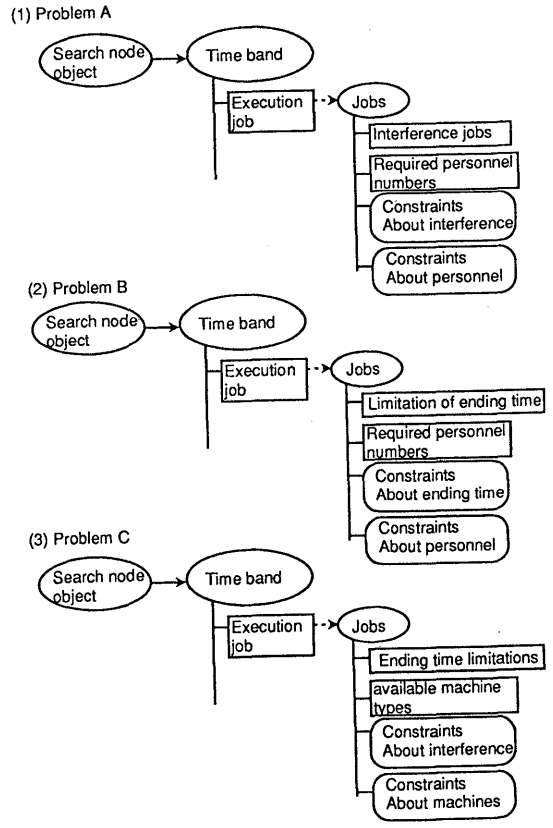


Fig.14 Domain model of the problems

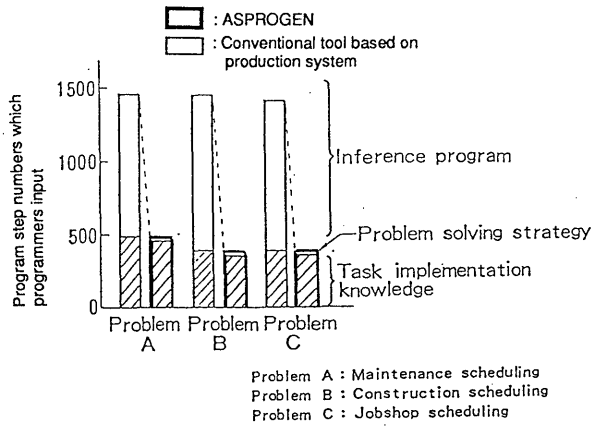


Fig.15 Program step numbers which programmers input

5. *Conclusions*

We have proposed and developed an expert system tool ASPROGEN(Automatic Search Program Generator) in which the automatic generation function of a domain specific inference program was built in. This function was based on search-based program specification and an abstract data type of search. ASPROGEN has interfaces for domain knowledge using an object-oriented approach and constraints which represent control knowledge. It is described by using domain knowledge and it can cover a detailed problem solving strategy

We applied ASPROGEN to produce three kinds of scheduling systems. These generated systems have equivalent performance in comparison with knowledge processing systems implemented by the conventional tool, and two-thirds reduction of the program step numbers required as programmer input was realized by ASPROGEN.

We have applied ASPROGEN only to scheduling systems, we are now going to check its applicability to CAD systems and diagnosis systems.

References

[1] K. Okuda et al.: Model Based Process Monitoring and Diagnosis, Proc. of IEEE Pacific Rim International Conference on Artificial Intelligence'90,pp.134-139, Nagoya, Japan (1990).

[2] B. Chandrasekaran: Towards Functional Architecture for Intelligence Base on Generic Information Processing Tasks, Invited Talk of IJCAI-87(1987).

[3] J. McDermott: Using Problem-Solving Methods to Impose Structure on Knowledge, Proc. of IEEE International Workshop on Artificial Intelligence for Industrial Applications, pp.7-11,Hitachi, Japan(1988).

[4] J.Laird, et al.: Universal Subgoaling and Chunking, Klur Academic Publishers(1987).

[5] E. W. Dijakstra et al. : Structured Programming, Academic Press, London(1979).

[6] A.V. Aho, et al.: Data Structure and Algorithms, Addison-Wesley Publishing Company, Inc., Reading Mass.(1983).

[7] T.Kasahara, et al.: Maintenance Work Scheduling Aid for Nuclear Power Plants, Proc. of IEEE International Workshop on Artificial Intelligence for Industrial Applications, pp.161-166 Hitachi, Japan(1988).

[8] S. Tano, et al.: Eureka-II A Programming Tool for Knowledge-Based Real Time Control Systems, International Workshop on Artificial Intelligence for Industrial Applications, pp.370-378, Hitachi, Japan(1988).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Knowledge-Based Functional Testing For Large Software Systems

Uwe Nonnenmann and John K. Eddy
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

Abstract

Automated testing of large embedded systems is perhaps one of the most expensive and time-consuming parts of the software life cycle. It requires very complex and heterogeneous knowledge and reasoning capabilities. The *Knowledge-based Interactive Test Script System* (KITSS) automates functional testing in the domain of telephone switching software. KITSS uses some novel approaches to achieving several desirable goals. Telephone feature tests are specified in English. To support this KITSS has a statistical parser that is trained in the domain's technical dialect. KITSS converts these tests into a formal representation that is audited for coverage and sanity. To accomplish this, KITSS uses a customized theorem prover-based inference mechanism and a hybrid knowledge base as the domain model that uses both a static terminological logic and a dynamic temporal logic. Finally, the corrected test is translated into an in-house automated test language that exercises the switch and its embedded software. This paper describes and motivates the approach taken and also provides an overview of the KITSS system.

1 Functional Testing Problem

There is an increasing amount of difficulty, effort, and cost that is needed to test large software development projects. It is generally accepted that the development of large-scale software with zero defects is not possible. A corollary to this is that accurate testing that uncovers all defects is also not possible [Myers, 1979]. This is because of the many inherent problems in the development of large projects [Brooks, 1987]. As just a few examples, a large project provides support for many interacting features, which makes requirements and specifications complex. Also, many people are involved in the project, which makes it difficult to ensure that each person has a common understanding of the meaning and functioning of features. Finally, the project takes a long time to complete, which makes it even harder to maintain a common understanding because the features change through time as people interact and come to undocumented agreements about the real meaning of features.

The consequence of these problems is that programs that do not function as expected are produced and therefore extensive and costly testing is required. Once software is developed, even more testing is needed to maintain it as a product. The major cost of maintenance is in re-testing and re-deployment and not the coding effort. Estimates, as in [Myers, 1976] and [McCartney, 1991], are that at least 50%, and up to as much as 80%, of the cost in the life cycle of a system is spent on maintenance.

We believe that the only practical way to drastically reduce the maintenance cost is to find and eliminate software problems early and within the development process. Therefore, we designed an automated testing system that is well integrated into the current development process [Nonnenmann & Eddy, 1991]. The focus of our system is on "functional testing" [Howden, 1985]. It corresponds directly to uncovering discrepancies in the program's behavior as viewed from the outside world. In functional testing the internal design and structure of the program are ignored. This type of testing has been called *black box* testing because, like a black box in hardware, one is only interested in the input and how it relates to the output. The resulting tests are then executed in a simulated customer environment. This corresponds to verifying that the system fulfills its intended purpose.

KITSS achieves a good integration into the current development process by using the same expressive and unobtrusive input medium (English functional tests) as is used currently as well as generating tests in the existing automated test language as output. Additionally, KITSS checks the tests for consistency with its built-in extensive knowledge base of "telephony".

Therefore, KITSS helps the test process by generating more tests of better quality and by allowing more frequent regression testing through automation. Furthermore, tests are generated earlier, *i.e.*, *during* the development phase not *after*, which should lead to detecting problems earlier. The result is higher quality software at a lower cost.

In this section, we motivated the need for the approach

chosen in KITSS. In the next section, we will describe KITSS in more detail.

2 KITSS Overview

The *Knowledge-based Interactive Test Script System* (KITSS) was developed at AT&T Bell Laboratories to reduce the increasing difficulty and cost involved in testing the software of DEFINITY®PBX switches[1]. Although our system is highly domain dependent in its knowledge base and inference mechanisms, the approach taken is a general one and should be applicable to any functional software testing task.

DEFINITY supports hundreds of complex features such as call forwarding, messaging services, and call routing. Additionally, it supports telephone lines, telephone trunks, a variety of telephone sets, and even data lines. At AT&T Bell Laboratories, PBX projects have many frequent and overlapping releases over their multi-year life cycle. It is not uncommon for these projects to have millions of lines of code.

GOAL: Activate CF[2] using CF Access Code.
ACTION: Set station B without redirect notification[3]. Station B goes offhook and dials CF Access Code.
VERIFY: Station B receives the second dial tone.
ACTION: Station B dials station C.
VERIFY: Station B receives confirmation tone. The status lamp associated with the CF button at B is lit.
ACTION: Station B goes onhook. Place a call from station A to B.
VERIFY: No ring-ping (redirect notification) is applied to station B. The call is forwarded to station C.
ACTION: Station C answers the call.
VERIFY: Stations A and C are connected.

Figure 1: Example of a Test Case

2.1 Testing Process

Before KITSS, the design methodology involved writing **test cases** in English. They describe the details of the external design and are written before coding begins. The cases, which are written by developers based on the requirements, constitute the only formal description of the external functioning of a switch feature. The idea is to describe how a feature works without having coding in mind.

Figure 1 shows a typical test case. Test cases are structured in part by a *goal/action/verify* format. The goal statement is a very high-level description of the purpose of the test. It is followed by alternating action/verify statements. An action describes stimuli that the tester has to execute. Each stimulus triggers a switch response that the tester has to verify (*e.g.*, a specific phone rings, a lamp is lit, a display shows a message etc).

Overall, there are tens of thousands of test cases for DEFINITY. All these test cases are written manually, just using an editor, and are executed manually in a test lab. This is an error prone and slow process that limits its test coverage and makes regression test intervals too long.

Some 5% of the above test cases have been converted into **test scripts** written in an in-house test automation language. Tests written in this language are run directly against the switch software. As this software is embedded in the switching system, testing requires large investments in test equipment (computer simulations are not acceptable as they do not address the real-time aspects of the system). Running and re-running test scripts becomes very time consuming and actually controls the rate at which projects are completed.

Although an improvement over the manual testing process, test automation has several problems. The current tools do not support any automatic semantic checking. The conversion from test case to test script takes a long time and requires the best domain experts. There are only limited error diagnosis facilities available as well as no automatic update for regression testing. Also, test scripts are cluttered with test language initialization statements and are specific to switch configurations and software releases. Test scripts lack the generality of test cases, which are a template for many test scripts. Therefore, test cases are easier to read and maintain.

2.2 KITSS Architecture

KITSS takes English test cases as its input. It translates all test cases into formal, complete functional test scripts which are run against the DEFINITY switch software. To make KITSS a practical system required novel approaches in two very difficult and different areas.

First, a very informal and expressive language needed

[1] A PBX, or private branch exchange, switch is a real-time system with embedded software that allows many telephone sets to share a few telephone lines in a private company.

[2] CF is an acronym for the call-forwarding feature, which allows the user to send his/her incoming calls to another designated station. The user can activate or deactivate this feature by pressing a button or by dialing an access code.

[3] Redirect notification is a feature to notify the user about an incoming call when he/she has CF activated. Instead of the phone ringing it issues a short "ring-ping" tone.

to be transformed into formal logic. Test cases are written in English. While English is undeniably quite expressive and unobtrusive as a representation medium, it is difficult to process into formal descriptions. It also requires theoretically unbounded amounts of knowledge to satisfactorily resolve incompleteness, vagueness, ambiguity, etc. In practice, however, test cases are written in a style that is considerably more restrictive than most English text. The test case descriptions are circumscribed in terms of the vocabulary and concepts to which they refer. Syntactic and semantic variations do occur, but the language is a technical dialect of English, a naturally occurring *telephonese* language that is less variable and less complex. These limits to a specific domain and style make it possible to transform the informal *telephonese* representation into a formal one.

Second, incomplete test cases needed to be extended. Even though humans find it easier to write test cases in natural language as opposed to formal language, they still have difficulties specifying tests that are both complete and consistent. They also have difficulties identifying all of the interactions that can occur in a complex system. This is analogous to the difference between trying to define a word and giving examples of its use. Creating a good definition, like creating a complete test case with all the details, is usually the more challenging task; giving word-usage examples, like describing a test case in general terms, is easier. Therefore, the input test cases need to be translated into a formal representation and then analyzed to be corrected and/or extended.

Both tasks have been attempted for more than a decade [Balzer *et al.*, 1977] with only limited success. Most difficulties arise because of the many possible types of imprecision in unrestricted natural language specifications, as well as by the lack of a suitable corpus of formalized background knowledge to guide automated reasoning tools for most application domains.

To address these two difficulties (see also [Yonezaki, 1989]), KITSS provides a natural language processor that is trained on examples of the *telephonese* sub-language using a statistical approach. It also provides a completeness and interaction analyzer that audits test coverage. However, these two modules have been feasible only due to the *domain-specific* knowledge-based approach taken in KITSS [Barstow, 1985]. Therefore, both modules are supported by a hybrid knowledge-base (the "K" in KITSS) that contains a model of the DEFINITY PBX domain. Concepts that are used in telephony and testing are available to both processes to reduce the complexity of their interpretive tasks. If, for example, a process gets stuck and cannot disambiguate the possible interpretations of a phrase, it interacts (the "I" in KITSS) with the test author. It presents the context in which the ambiguity occurs and presents its best guesses and asks the author to pick the correct choice. Finally,

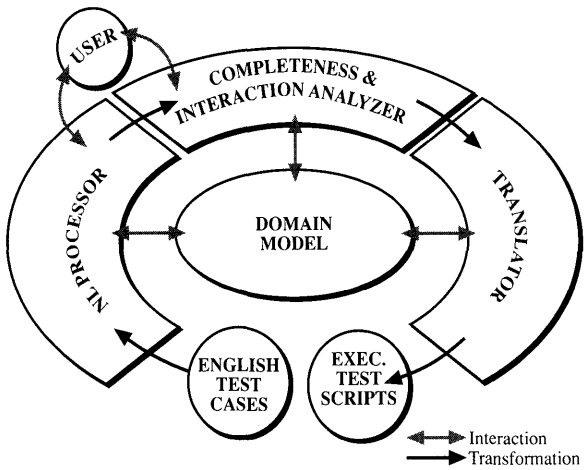


Figure 2: KITSS Architecture

KITSS also provides a translator that generates the actual test scripts (the "TS" in KITSS) from the formal representation derived by the analyzer.

The two needs described above led to the architecture shown in Figure 2. It shows that KITSS consists of four main modules: the domain model, the natural language processor, the completeness and interaction analyzer, and the translator. The domain model (see Section 3) is in the center of the system and supports all three reasoning modules (see Section 4).

3 Domain Knowledge

A domain model serves as the knowledge base for an application system. Testing is a very knowledge intensive task. It involves experience with the switch hardware and testing equipment as well as an understanding of the switch software with its several hundred features and many more interactions. There are binders full of papers that describe the features of DEFINITY PBX software, but no concise formalizations of the domain were available before KITSS. One of the core pieces of KITSS is its extensive domain model. The focus of KITSS and the domain model is on an end-user's point of view, *i.e.*, on (physical and software) objects that the user can manipulate.

The KITSS domain model consists of three major functional pieces (see Figure 3):

Core PBX model: It is split into two major parts. The static model is used by all reasoning modules. The dynamic model is used mainly by the analyzer.

Test execution model: It includes details about the current switch configuration and all the necessary

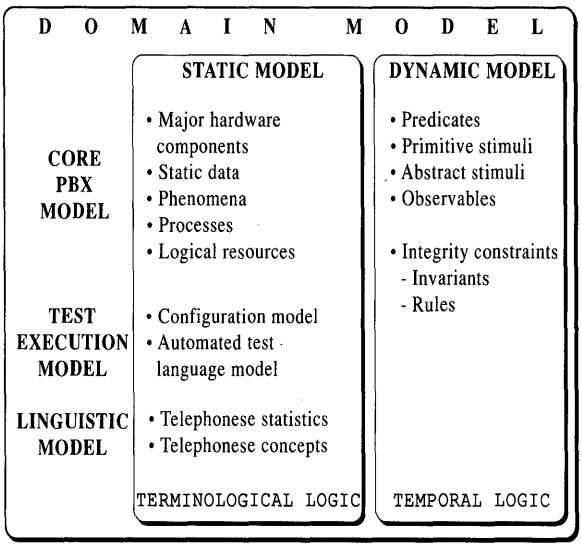


Figure 3: KITSS Domain Model

specifics of the automated test language. This model is used mainly by the translator.

Linguistic model: It is specific to the input language (*telephonese*) and is used mainly by the natural language processor.

From a knowledge representational point of view, we distinguish between static properties of the domain model and dynamic ones [Brodie *et al.*, 1984]. *Static properties* include the objects of a domain, attributes of objects, and relationships between objects. All static parts of the domain model are implemented in a terminological logic (see Section 3.1). *Dynamic properties* include operations on objects, their properties, and the relationships between operations. The applicability of operations is constrained by the attributes of objects. *Integrity constraints* are also included to express the regularities of a domain. The dynamic part of the core PBX model is represented in temporal logic (see Section 3.2).

3.1 Static Model

This part of the domain model represents the static aspects of KITSS. By static we mean all objects, data, and conditions that do not have a temporal extent but may have states or histories.

The **static PBX model** includes the following pieces:

- *Major hardware components*, such as telephones and switch administration consoles as well as smaller subparts of theses components, *e.g.*, buttons, lamps, and handsets.
- *Static data*, *e.g.*, telephone numbers, routing codes and administrative data such as available features, and current feature settings.
- *Phenomena*, such as tones and flashing patterns which are occurrences at points in time.
- *Processes*, such as static definition of types of calls (*e.g.* voice calls, data calls, priority calls) and types of sessions (*e.g.* calling sessions, feature sessions).
- *Logical resources*, such as lines and trunks required by processes.

The **test execution model** is divided as follows:

- The *configuration model* describes the current test setup, *i.e.*, how many simulated phones and trunk lines are available or which extension numbers belong to which phones/lines, etc. It also contains the dial plan and the default feature assignments.
- The *automated test language model* defines the vocabulary of the test script language.

The **linguistic model** supports two pieces:

- *Telephonese statistics*, which are frequency distributions of syntactic structures, help the natural language processor by disallowing interpretations of phrases and concepts that are possible in English but not likely in *telephonese*.
- *Telephonese concepts* make it easier to paraphrase KITSS' representations for user interactions.

We used CLASSIC [Brachman *et al.*, 1989] to represent the knowledge in our domain. CLASSIC belongs to the class of terminological logics (*e.g.* KL-ONE). It is a frame-based description system that is used to define structured concepts and make assertions about individuals. CLASSIC organizes the concepts and the individuals into a hierarchy by classification and subsumption. Additionally, it permits inheritance and forward-chaining rules. CLASSIC is probably the most expressive terminological logic that is still computationally tractable [Brachman *et al.*, 1990]. Queries to CLASSIC are made by *semantics* not by syntax.

The static model incorporates multiple views of an object from the various models into one (*e.g.*, a station might have one name in the English test case, another in the automated test language and a third in the actual configuration). Thus, although each reasoning module might have a different view on the same object, CLASSIC will always retrieve the same concept correctly.

3.2 Dynamic Model

This unique part of the domain model represents all dynamic aspects of the switch's behavior. It basically defines constraints that have to be fulfilled during testing as well as the predicates they are defined upon.
The **dynamic PBX model** includes the following pieces:

- *Predicates*, such as offhook, station-busy, connected, or on-hold, define a state which currently holds for the switch. The different phases of a call are described with predicates such as requesting-connection, denied-connection, or call-waiting-for-timeout. Each of the predicates has defined *sorts* that relate to objects in the static model. Synonyms (*e.g.*, on-hold is a synonym for call-suspended) are allowed as well.

- *Stimuli* can be either primitive or abstract. Stimuli appear in the action statements of test cases.

  A *primitive stimulus* defines an action being performed by the user (*e.g.*, dials-extension, goes-offhook) or by the switch (*e.g.*, timeout-call). The necessary pre- and postconditions (before and after the stimulus) are also specified. For instance, for a station to be able to go offhook the precondition is that the station is not already offhook and the postcondition is that the station is offhook after the stimulus[4].

  An *abstract stimulus* is not an atomic action but may have pre- and postconditions like a primitive stimulus. However, there are several primitive stimuli necessary to achieve the goal of a single abstract stimulus (*e.g.*, place-call, busy-out-station, or activate-feature). The steps necessary for an abstract stimulus are defined in one or many *abstract stimulus plans*. The abstract stimulus defines the conditions that need to be true for the goal to succeed whereas the abstract stimulus plans describe possible ways of achieving such a goal.

- *Observables* are states that can be verified such as receives-tone, ringing, or status-lamp-state. Observables appear in the verify statements of test cases.

Additionally, the dynamic model includes two different types of **integrity constraints**:

- *Invariants* are assertions that are true in all states. These are among the most important pieces of domain knowledge as they describe basic telephony behavior as well as the *look & feel* of the switch. The paraphrases of a few of the invariants are as follows: "Only offhook phones receive tones" or "You only get ringing of any kind when you are alerting" or "A forwarded call always alerts at the forwardee, never at the forwarder" or "You can't be be talking to an on-hold call".

- *Rules* also describe low-level behavior in telephony. These are mainly state transitions in signaling behavior like "A tone must stop whenever another begins" or "Stop dial-tone after dialing an extension" or "An idle phone starts to ring when the first incoming call arrives".

Representing the dynamic model we required expressive power beyond CLASSIC or terminological logics. For example, CLASSIC is not well-suited for representing plan-like knowledge, such as sequences of actions to achieve a goal, or to perform extensive temporal reasoning [Brachman *et al.*, 1990]. But this is required for the dynamic part of KITSS (see above examples). We therefore used the WATSON Theorem Prover (see Section 4.2), a linear-time first-order resolution theorem prover with a weak temporal logic. This non-standard logic has five modal operators *holds, occurs, issues, begins,* and *ends* which are sufficient to represent all temporal aspects of our domain. For example, the abstract stimulus plan for activating a feature is represented in temporal logic as follows.

```
(abstract-stimulus-plan activate-feature-1
 ((:plan-goal activate-feature)
  (:sorts
   ((station s1) (feature f) (station s2)))
  (:preconditions
   ((holds (onhook s1))))
  (:plan-steps
   (((occurs (initiate-feature-session s1 f))
     (begins (receives-tone s1
                        second-dial-tone)))
    ((occurs (dials-destination s1 s2))
     (issues (receives-tone s1
                        confirmation-tone)))
    ((occurs (terminate-feature-session s1 f))
)))))
```

The theorem proving is tractable due to the tight integration between knowledge representation and reasoning. Therefore, we specifically designed the analyzer using the WATSON Theorem Prover and targeting them for this domain. The challenging task in building the dynamic model was to understand and extract what the invariants, constraints, and rules were [Zave & Jackson, 1991]. Representing them then in the temporal logic was much easier.

[4]Note the difference between the *state* of being offhook and the *action* goes-offhook.

3.3 Domain Model Benefits

In choosing a hybrid representation, we were able to increase the expressive power of our domain model and to increase the reasoning capabilities as well. The integration of the hybrid pieces did produce some problems, for example, deciding which components belonged in which piece. However, this decision was facilitated because of our design choice to represent all dynamic aspects of the system in our temporal logic and to keep everything else in CLASSIC.

There were other benefits to building a domain model. It ensures that a standard terminology is used by all of the test case authors. The domain model also simplifies the maintenance of test scripts. In automated testing environments without a domain model, the knowledge is scattered throughout thousands of scripts. With the domain model a change in the functioning of the software is made in only one place which makes it possible to centralize knowledge and therefore centralize the maintenance effort. Additionally, the domain model provides the knowledge that reduces and simplifies the tasks of the natural language processor, the analyzer, and the translator modules.

4 Reasoning Modules

4.1 Natural Language Processor

The existing testing methodology used English as the language for test cases (see Figure 1) which is also KITSS' input. Recent research in statistical parsing approaches [Jones & Eisner, 1991] provided some answers to the difficulty of natural language parsing in restricted domains such as testing languages. In the KITSS project, the parser uses probabilities (based on training given by *telephonese* examples) to prune the number of choices in syntactic and semantic structures. Unlikely structures can be ignored or eliminated, which helps to speed up the processing. For instance, consider the syntax of the following two sentences[5]:

Place a call to station troops in Saudi Arabia.
Place a call to station "4623" in two minutes.

Both examples are correct English sentences. Although the second sentence on the surface matches in many parts the first one, their structure is very different. In the first sentence "station" is a verb, in the second a noun; "to" is an infinitive and a preposition respectively. "In Saudi Arabia" refers to a location whereas "in two minutes" refers to time. It is hard to come up with correct parses for both but by restricting ourselves to the *telephonese* sublanguage this is somewhat easier. In *telephonese*, the structure of the first sentence is statistically unlikely and can be ignored while the second sentence is a common phrase.

The use of statistical likelihoods to limit search during natural language processing was used not only during parsing but also when assigning meaning to sentences, determining the scope of quantifiers, and resolving references. When choices could not be made statistically, the natural language processor could query the domain model, the analyzer, or the human user for disambiguation. The final output of the natural language processor are logical representations of the English sentences, which are passed to the analyzer.

4.2 Completeness & Interaction Analyzer

The completeness and interaction analyzer represents one of the most ambitious aspects of KITSS. It is based on experience with the WATSON research prototype [Kelly & Nonnenmann, 1991]. Originally, WATSON was designed as an automatic programming system to generate executable specifications from episodic descriptions in the telephone switching software domain. This was an extremely ambitious goal and could only be realized in a very limited prototype. To be able to scale up to real-world use, the focus has been shifted to merely checking and augmenting given tests and maybe generating related new ones rather than generating the full specification.

Based on the natural language processor output, the analyzer groups the input logical forms into several *episodes*. Each episode defines a stimulus-response-cycle of the switch, which roughly corresponds to the action/verify statements in the original test case. These episodes are the input for the following analysis phases. Each episode is represented as a logical rule, which is checked against the dynamic model. The analyzer uses first-order resolution theorem proving in a temporal logic as its inference mechanism, the same as WATSON.

The analysis consists of several phases that are specifically targeted for this domain and have to be re-targeted for any different application. All phases use the dynamic model extensively. The purpose of each phase is to yield a more detailed understanding of the original test case. The following are the current analysis phases:

- The structure of a test case is analyzed to recognize or attribute purpose to pieces of the test case. There are four major pieces that might be found: administration of the switch, feature activation or deactivation, feature behavior, and regression testing.

[5]This example was given by Mark Jones.

- The test case is searched for connections among concepts, *e.g.*, there might be relations between system administration concepts and system signaling that need to be understood.
- Routine omissions are inserted into the test case. Testers often reduce (purposefully or not) test sequences to their essential aspects. However, these omissions might lead to errors during testing and therefore need to be added.
- Based on the abstract plans in the dynamic model, we can enumerate possible specializations, which yield new test cases from the input example.
- Plausible generalizations are found for objects and actions as a way to abstract tests into classes of tests.

During the analysis phases, the user might interact with the system. We try to exploit the user's ease at verifying or falsifying examples given by the analyzer. At the same time, the initiative of generating the details of a test lies with the system. For example, some test case might violate the *look & feel* of the system, *i.e.*, there is a conflict with an invariant. However, the user might want this behavior intentionally which will lead to a change in the *look & feel* itself.

The final output of the analyzer is a corrected and augmented test case in temporal logic. As an example of the analyzer's representation after analysis, the following shows the logical forms for the first few episodes in Figure 1. Notice that the test case is expanded since the analyzer applied abstract stimulus plans.

```
...
((OCCURS (GOES-OFFHOOK B))
 (BEGINS (RECEIVES-TONE B NORMAL-DIAL-TONE)))
((OCCURS (DIALS-CODE B
            (ACTIVATE-ACCESS-CODE CF)))
 (BEGINS (RECEIVES-TONE B SECOND-DIAL-TONE)))
((OCCURS (DIALS-EXTENSION B C))
 (ISSUES (RECEIVES-TONE B CONFIRMATION-TONE))
 (BEGINS (STATUS-LAMP-STATE B (BUTTON CF)
                              STEADY)))
...
```

This representation is passed to the translator.

4.3 Translator

To make use of the analyzer's formal representation, the translator needs to convert the test case into an executable test language. This language exercises the switch's capabilities by driving test equipment with the goal of finding software failures. One goal of the KITSS project was to extend the life of test cases so that they could be used as many times as possible. To accomplish this, it was decided to make the translator support two types of test case independence.

First, a test case must be test machine independent. Each PBX that we run our tests on has a different configuration. KITSS permits a test author to write a test case without knowing which particular machine it will be run on and assuming unlimited resources. The translator loads the configuration setup of a particular switch into the *test execution model*. It uses this to make the test case concrete with respect to equipment used, system administration performed, and permissions granted. Thus, if the functional description of a test case is identical in two distinct environments, then the logical representation produced by the earlier modules of KITSS should also be identical.

Second, a test case must be independent of the automated test language. KITSS generates test cases in an in-house test language. The translator's code is small because much of the translation information is static and can be represented in CLASSIC. If a new test language replaces the current one then the translator can be readily replaced without loss of test cases, with minimal changes to the KITSS code, and without a rewrite of most of the domain model.

5 Status

The KITSS project is still a prototype system that has not been deployed for general use on the DEFINITY project. It was built by a team of researchers and developers. Currently, it fully translates 38 test cases (417 sentences) into automated test scripts. While this is a small number, these test cases cover a representative range of the core features. Additionally, each test case yields multiple test scripts after conversion through KITSS. The domain model consists of over 500 concepts, over 1,500 individuals, and more than 80 temporal constraints. The domain model will grow somewhat with the number of test cases covered, however, so far the growth has been less than linear for each feature added.

All of the modules that were described in this paper have been implemented but all need further enhancements. System execution speed doesn't seem to be a bottleneck at this point in time. CLASSIC's fast classification algorithm's complexity is less than linear in the size of the domain model. Even the analyzer's theorem prover, which is computationally the most complex part of KITSS, is currently not a bottleneck due to continued specialization of its inference capability. However, it is not clear how long such optimizations can avoid potential intractability.

The current schedule is to expand KITSS to cover a few hundred test cases. To achieve this, we will shift our

strategy towards more user interaction. The version of KITSS currently under development will intensely question the user to explain unclear passages of test cases. We will then re-target the reasoning capabilities of KITSS to cover those areas. This rapid-prototyping approach is only feasible since we have already developed a robust core system. Although scaling-up from our prototype to a real-world system remains a hard task, KITSS demonstrates that our knowledge-based approach chosen for functional software testing is feasible.

6 Conclusion

As we have shown, testing is perhaps one of the most expensive and time-consuming steps in product design, development, and maintenance. KITSS uses some novel approaches to achieving several desirable goals. Features will continue to be specified in English. To support this we have incorporated a statistical parser that is linked to the domain model as well as to the analyzer. Additionally, KITSS will interactively give the user feedback on the test cases written and will convert them to a formal representation. To achieve this, we needed to augment the domain model represented in a terminological logic with a dynamic model written in a temporal logic. The temporal logic inference mechanism is customized for the domain. Tests will continue to be specified independent of the test equipment and test environment and the user will not have to provide unnecessary details.

Such a testing system as demonstrated in KITSS will ensure project-wide consistent use of terminology and will allow simple, informal tests to be expanded to formal and complete test scripts. The result is a better testing process with more test automation and reduced maintenance cost.

Acknowledgments

Many thanks go to Van Kelly, Mark Jones, and Bob Hall who also contributed major parts of the KITSS system. Additionally, we would like to thank Ron Brachman for his support throughout the project.

References

[Balzer *et al.*, 1977] Balzer R., Goldman N., and Wile D.: Informality in program specifications. In *Proceedings of the 5th IJCAI*, Cambridge, MA, 1977.

[Barstow, 1985] Barstow, D.R.: Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, November 1985.

[Brachman *et al.*, 1989] Brachman, R.J., Borgida, A., McGuinness, D.L., and Alperin Resnick, L.: The CLASSIC knowledge representation system, or, KL-ONE: The next generation. In preprints of *Workshop on Formal Aspects of Semantic Networks*, Santa Catalina Island, CA, 1989.

[Brachman *et al.*, 1990] Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F., Alperin Resnick, L., and Borgida, A.: Living with CLASSIC: When and how to use a KL-ONE-like language. In *Formal Aspects of Semantic Networks*, J. Sowa, Ed., Morgan Kaufmann, 1990.

[Brodie *et al.*, 1984] Brodie, M.L., Mylopoulos, J., and Schmidt, J.W.: *On conceptual modeling: Perspectives from Artificial Intelligence.* Springer Verlag, New York, NY, 1984.

[Brooks, 1987] Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. *Computer*, Vol. 20, No. 4, April 1987.

[Howden, 1985] Howden, W.E.: The theory and practice of functional testing. *IEEE Software*, September 1985.

[Jones & Eisner, 1991] Jones, M.A., and Eisner J.: A probabilistic chart-parsing algorithm for context-free grammars. *AT&T Bell Laboratories Technical Report*, 1991.

[Kelly & Nonnenmann, 1991] Kelly, V.E., and Nonnenmann, U.: Reducing the complexity of formal specification acquisition. In *Automating Software Design*, M. Lowry and R. McCartney, eds., MIT Press, 1991.

[McCartney, 1991] McCartney, R.: Knowledge-based software engineering: Where we are and where we are going. In *Automating Software Design*, M. Lowry and R. McCartney, eds., MIT Press, 1991.

[Myers, 1976] Myers, G.J.: *Software Reliability.* John Wiley & Sons, New York, NY, 1976.

[Myers, 1979] Myers, G.J.: *The Art of Software Testing.* John Wiley & Sons, Inc. New York, NY, 1979.

[Nonnenmann & Eddy, 1991] Nonnenmann, U., and Eddy J.K.: KITSS - Toward software design and testing integration. In *Automating Software Design: Interactive Design - Workshop Notes from the 9th AAAI*, L. Johnson, ed., USC/ISI Technical Report RS-91-287, 1991.

[Yonezaki, 1989] Yonezaki, N.: Natural language interface for requirements specification. In *Japanese Perspectives In Software Engineering*, Y. Matsumoto and Y. Ohno, eds., Addison-Wesley, 1989.

[Zave & Jackson, 1991] Zave, P, and Jackson, M.: Techniques for partial specification and specification of switching systems. In *Proceedings of the VDM'91 Symposium*, October 1991.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Diagnostic and Control Expert System Based on a Plant Model

Junzo SUZUKI* Chiho KONUMA Mikito IWAMASA Naomichi SUEDA
Systems&Software Engineering Laboratory, Toshiba Corporation
70, Yanagi-cho, Saiwai-ku, Kawasaki 210, Japan

Shigeru MOCHIJI Akimoto KAMIYA
Fuchu Works, Toshiba Corporation
1, Toshiba-cho, Fuchu 183, Japan

Abstract

A conventional expert system for plant control is based on heuristics, which are a priori knowledge stored in a knowledge base. Such a system has a substantial limitation in that it cannot deal with "unforeseen abnormal situations" in a plant due to the lack of heuristics. To realize a flexible plant control system which can overcome this limitation, we focus on model-based reasoning. Our system has three major functions: 1) model-based diagnosis for unforeseen abnormal situations, 2) model-based knowledge generation for plant control, and 3) knowledge-based plant control both with generated and a priori stored knowledge.

In this paper, we focus on the function of model-based knowledge generation. First, we show an overview of our system which has an integrated architecture of deep reasoning with shallow reasoning. Next, we explain the theoretical aspects of model-based knowledge generation. Finally, we show the experimental results of our system, and discuss the system's capabilities and some open problems.

1 Introduction

Currently in the field of diagnosis and control of thermal power plants, the more intelligent and flexible systems become, the more knowledge they need. Conventional diagnostic and control expert systems are based on heuristics stored a priori in knowledge bases, so they cannot deal with unforeseen abnormal situations in the plant. Such situations could occur if knowledge engineers forgot to implement some necessary knowledge.

A skilled human operator is able to operate the plant and somehow deal with such unforeseen abnormal situations because he has fundamental knowledge about the structure and functions of component devices of a plant, the principles of plant operations, and the laws of physics. His thought process is as follows.

- Diagnosis of an unforeseen abnormal situation
- Generation of plant control knowledge
- Verification of generated knowledge

A skilled human operator can deal with unforeseen abnormal situations by repeatedly executing these steps using the fundamental knowledge mentioned before. Therefore, the concepts of our diagnostic and control expert system are based on the same steps.

In this paper, we focus on the generation and verification of plant control knowledge. First, we show an overview of our system. Next, we explain the model representations and the model-based reasoning mechanisms. After that, we describe the experimental results and discuss the system's capabilities. Finally, we discuss some open problems and related work.

2 A System Overview

The model-based diagnostic and control expert system (Figure 1) consists of two subsystems: the *Shallow Inference Subsystem (SIS)* and the *Deep Inference Subsystem (DIS)*.

The *SIS* is a conventional plant control system based on heuristics, namely the shallow knowledge for plant control. It selects and executes plant operations according to the heuristics stored in the knowledge base. The *Plant Monitor* detects occurrences of unforeseen abnormal situations, and then activates the *DIS*.

The *DIS* consists of the following modules: the *Diagnosor*, the *Operation-Generator*, the *Precondition-Generator*, and the *Simulation-Verifier*. The *Diagnosor* utilizes the *Qualitative Causal Model* for plant process parameters to diagnose unforeseen abnormal situations. The *Operation-Generator* figures out which plant operations are necessary to deal with these unforeseen abnormal situation. It utilizes the *Device Model* and the

*Email:suzuki%ssel.toshiba.co.jp@uunet.uu.net

Operation Principle Model. The *Precondition-Generator* attaches the preconditions to each plant operation above, and as a result, generates rule-based knowledge for plant control. The *Simulation-Verifier* predicts plant behavior which is to be observed when the plant is operated according to the generated knowledge. It utilizes the *Dynamics Model*, verifies the generated knowledge using predicted plant behavior, and gives feedback to the *Operation-Generator* to refine the knowledge if necessary.

The knowledge compiled from models by the *DIS* is transmitted to the *SIS*. The *SIS* executes the plant operations accordingly, and as a result, the unforeseen abnormal situations should be handled properly.

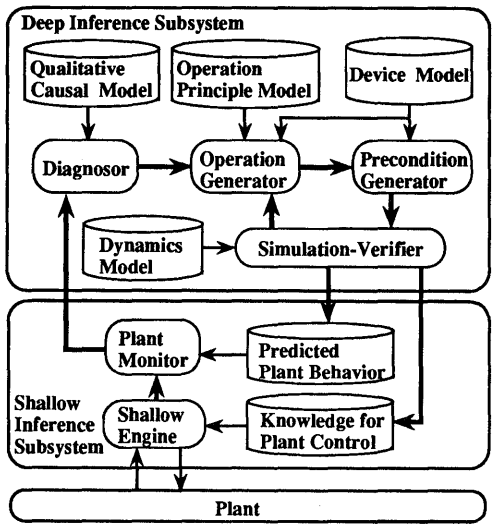


Figure 1: An overview of the system

3 Model-Based Generation and Verification of Knowledge

The main purpose of this section is to present a generation and verification procedure for plant control knowledge to deal with unforeseen abnormal situations. This knowledge is in IF-THEN format.

3.1 Model Representation

The *Device Model* and the *Operation Principle Model* are used to generate the knowledge. The *Dynamics Model* is used to verify the knowledge. We explain these models briefly.

1. Device Model

   The *Device Model* represents the fundamental knowledge about the functions, structure and characteristics of a plant. Because a plant consists of component devices, a *Device Model* can be defined for each component device. Figure 2 shows the *Device Model* representation for a *boiler-feeding-water-pump*, which supplies water to a boiler.

name :	**a_bfp**
demand :	**a_bff = 360 [ton/hr]**
goal :	**a_bff =< capacity(a_bff)**
states :	**on ; capacity(a_bff) = 615 [ton/hr]** **off ; capacity(a_bff) = 0 [ton/hr]**
operation :	**off →on ; time-lag = 0.1 [hr] , d/dt(a_bff) = +** **on →off ; time-lag = 0.1 [hr] , d/dt(a_bff) = -**
quality :	**d/dt(a_bff) = d/dt(a_bfif)**
flow_in :	***(defined at system)***
flow_out :	***(defined at system)***
system :	**bfp_system(a_bff , a_bfif)**

Figure 2: An example of the Device Model

The demands for each component device are described in the **demand** slot, and their constraints to be satisfied are described in the **goal** slot.

The functions of each component device are described as possible states of each device in the **states** slot. The operations of a device are defined by the change of its state.

Direct and indirect influences to plant processes by operations are described in the **operation** and **quality** slots respectively.

The structure of a plant is described in the **flow_in** and **flow_out** slots. In addition, hierarchical modeling can be done as shown in Figure 3.

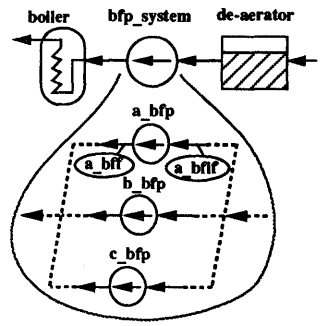


Figure 3: Hierarchical modeling of plant devices

2. Operation Principle Model

   The *Operation Principle Model* is concerned with the principles for safe and economical plant control. It consists of the following two rules.

   - Strict Accordance Rule

     The purpose of this rule is to ensure plant safety throughout a series of plant operations. It consists of the following two components: a rule to use a device within its own allowable range, and a rule to keep a faulty device out of service.

- Preference Rule

  The purpose of this rule is to ensure an economical plant operation. It consists of the following two components: a rule to keep the number of in-service devices to a minimum, and a rule to equalize the service-time of each device.

3. Dynamics Model

   The *Dynamics Model* represents the dynamic characteristics of the plant. In the area of plant control, the *Dynamics Model* is concerned either with the functions of traditional plant controllers based on *PID-control* or with the characteristics related to physical laws. Figure 4 shows the model of a *water-flow-controller*. Kp and T are constants. $1/s$ means the integral operator.

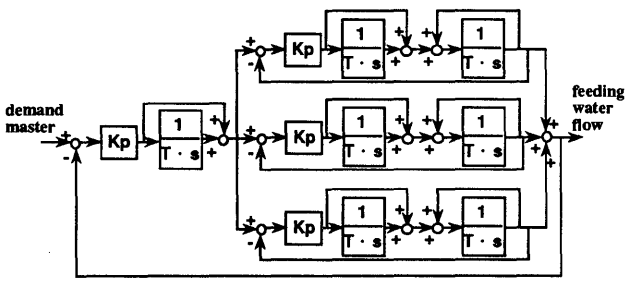


Figure 4: An example of the Dynamics Model

3.2 Model-Based Reasoning Mechanism

We briefly explain the model-based reasoning mechanism of these modules: the *Operation Generator*, the *Precondition Generator* and the *Simulation-Verifier*.

1. Operation Generator

   This module determines the *goal-state* where all of the constraints defined by the *Device Model* and the *Operation Principle Model* are satisfied. Generally, an unforeseen abnormal situation causes a state change of a plant, and this change can make the above constraints unsatisfied. To estimate this unsatisfied constraints, the following functions are needed.

   (a) Verification of Constraints

   All the constraints defined by the *Device Model* should be verified to see if they are still satisfied after the unforeseen abnormal situation. This function (Figure 5) consists of the following two sub-functions: propagating the change at each device to the others according to the connections of devices, and locally verifying the constraints at each device.

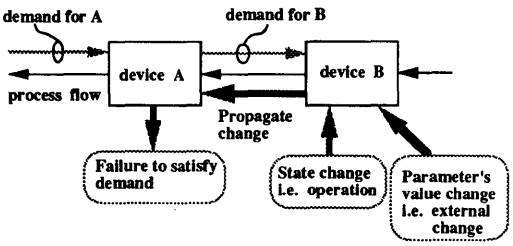


Figure 5: Constraints verification function

   (b) Update of the Goal-State

   If some of the constraints at a certain device are proved not to be satisfied, a new state for this device should be sought in order to satisfy them. This function (Figure 6) consists of the following sub-functions: searching for a state of each device where all of its demands can be satisfied, distributing the demands for a device of higher hierarchy to devices of lower hierarchy according to the constraints defined by the *Operation Principle Model*, and generating new demands for connected devices according to the *Device Model* and propagating them. The plant operations are deduced by taking the difference between the initial *goal-state* and the updated one.

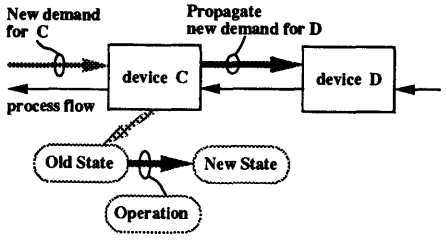


Figure 6: Goal-State update function

2. Precondition Generator

   In the domain of thermal power plant control, preconditions of each plant operation can be classified into the following five generic classes [Konuma 1990].

   - Preconditions for the state before an operation
   - Preconditions for the order of operations

- Preconditions for safety during an operation
- Preconditions for the timing of an operation
- Preconditions for completion of an operation

This module generates the above preconditions for each operation by analyzing the *goal-state* according to the constraints defined by the *Device Model*. An image of their generation process is shown in Figure 7.

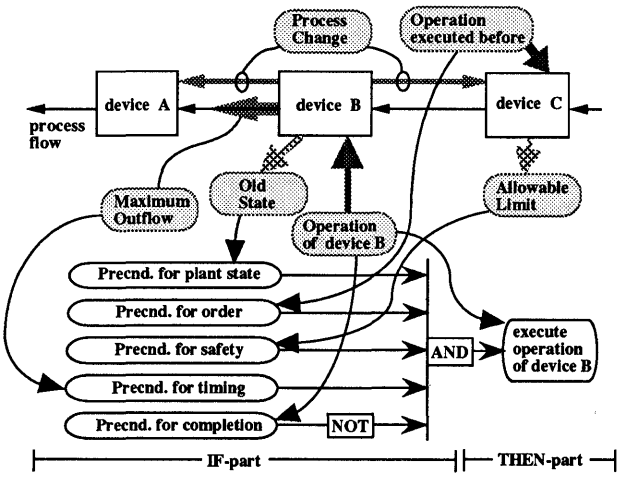


Figure 7: Generation process of preconditions

3. Simulation Verifier

This module predicts plant behavior using the *Dynamics Model* to verify the knowledge compiled from models by the *Operation Generator* and *Precondition Generator*. The prediction of plant behavior can be realized through simulation methods [Suzuki 1990]. After the prediction, the module examines whether or not undesirable events have occurred. Undesirable events can be defined by several criteria, but one of the most important is the transient violation of the allowable range for each process parameter's value. The execution of plant operations usually causes the transient change of processes due to the dynamic characteristics of a plant. If this change is beyond the allowable range of a current plant state, it is detected as a violation.

The *Simulation-Verifier* supports the *Generate&Test* algorithm of knowledge [Suzuki 1990] as illustrated in Figure 8. This process can be formalized as updating the *goal-state* according to the degree of the violation.

```
procedure Generate&Test ( M or D0 , S0 )
  begin
      [ Se, Op ] <= Operation_Generate ( M or D0, S0 ) ;
            K1 <= Precondition_Generate ( S0, Se, Op ) ;
            PS <= Simulate ( S0, K1 ) ;
    [ NG, D1, S1 ] <= Verify ( PS ) ;
    if NG =\= constraint_violation
    then return( K1, Se ) ;
    else
        [ K2, S3 ] <= Generate&Test ( D1, S1 ) ;
        [ K3, Se ] <= Generate&Test ( M, S3 ) ;
            K4 <= FIX( K1 ) + K2 + K3 ;
        return( K4, Se ) ;
    endif
  end.
```

NOTATION :	
Si, Se : plant state	M : output of Diagnosor
Di : demand for a device	Op : plant operations
PS : plant behavior	NG : flag for allowable range violations
Ki : plan of plant operations	
[ ] : list expression	<= : substitution expression

Figure 8: Generate&Test algorithm of the knowledge

4 Experiments

We have implemented the expert system on Multi-PSI [Taki 1988]. To realize a rich experimental environment, we have also implemented a plant simulator instead of an actual plant on a mini-computer G8050. Both computers are linked by a data transmission line. This section describes the results of some experiments.

4.1 Configuration of a Thermal Power Plant

Figure 9 shows the configuration of the thermal power plant. It consists of controllers (hatched rectangle) and devices. The *condenser* is a device for cooling the *turbine's* exhaust steam; the steam is reduced to water using cooling water taken from the sea. The reduced water is moved through the *de-aerator* to the *boiler* by the *condensation-pump-system* and the *boiler-feeding-pump-system*. The cooling water is provided by the *circulation-pump-system*. The *fuel-system* supplies pulverized coal to the boiler.

4.2 Experimental Results

The total of the *Device Models* in the system amounts to 78 (Table 1). In this table, the difference between the numbers in the left and right columns is due to hierarchical modeling.

The experiments were performed as follows.

1. First, we selected appropriate faults of the following devices: a *coal-pulverizer*, a *boiler-feeding-pump*, a *condensation-pump*, a *circulation-water-valve*, and a *water-heater*. We made these faults the malfunctions of the plant simulator. We also set them up for multiple faults.

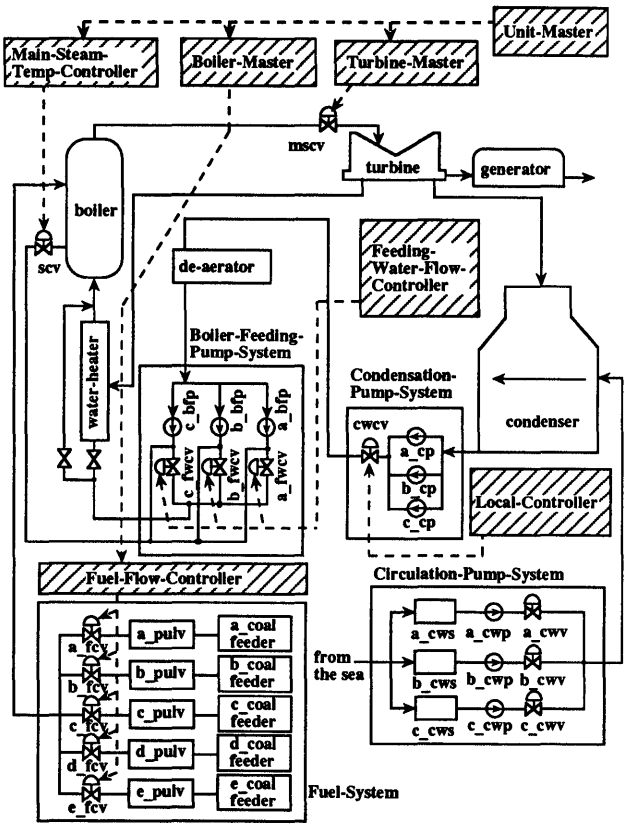


Figure 9: Configuration of a thermal power plant

2. Next, we extracted some specific knowledge for plant control from the knowledge base in the *SIS*. This specific knowledge was necessary to deal with the selected faults. As a result, the selected faults were equal to unforeseen abnormal situations.

3. Finally, after activating the malfunctions of the plant simulator, we confirmed that the *DIS* compiled the knowledge from the models and that the *SIS* executed the operations accordingly.

We explain the quality of generated knowledge for a single fault, because the results in multiple faults are the same as in a single fault. In the experiments, the contents of generated knowledge are concerned with switching from a faulty device to a backup one. Table 2 summarizes all the generated plant operations. In the case of a *water-heater* fault, the system failed to generate plant operations. In other cases, the system succeeded in generating plant operations. We estimate the quality of the generated knowledge in terms of its preconditions. This table lists columns consisting of the following items for each operation: the number of the preconditions encoded by a human expert ($N1$), the number of the essential ones

Table 1: The amount of devices and controllers

	Amount in the plant	Amount in the Device Models
Devices	43	63
Controllers	7	15
Total	50	78

in $N1$ ($N2$), the number of generated ones by the system ($N3$), the covered ratio of $N2$ by the system ($CR1$), and the uncovered ratio of $N2$ (ER), namely, the ratio of $N2$ missed being generated or incorrectly generated by the system. ($CR2$ will be explained in Section5.)

The difference between $N1$ and $N2$ is due mainly to the following reason. Although a human expert specifies the preconditions of the knowledge as generally as possible, the system generates specialized preconditions for each occurring unforeseen abnormal situation. With this point in mind, we determine $N2$ by eliminating unnecessary preconditions from $N1$. CRi (i=1,2) and ER are calculated by the following formulas.

$$CRi = \frac{Success(N2)}{N2}$$

$$ER = \frac{Miss(N2) + Fail(N2)}{N2}$$

$Success(N2)$ denotes the number of $N2$ generated by the system; $Miss(N2)$ the number of $N2$ which were not generated; and $Fail(N2)$ the number of $N2$ incorrectly generated.

We also consider the following in evaluating $Success(N2)$.

- Although the generated preconditions enumerate the individual state of each device, a human expert often represents them succinctly. For example, the conjunctive precondition "a_bfp = on" $\wedge$ "b_bfp = on" $\wedge$ "c_bfp = off" are represented as "the number of activated bfp = 2".

- The system often generates superfluous preconditions that a human expert does not mention.

- Although a human expert encodes preconditions for the selection of an in-service device, the system never generate them because they are already estimated in applying the *Operation Principle Model*.

None of the above devalues the quality of generated knowledge because the system is required only to generate specific preconditions for an occurring unforeseen abnormal situation. For this reason, we regard generated preconditions applicable to any of the above as $Success(N2)$.

We carried out the experiments under the following conditions.

Table 2: Quality of generated knowledge

Unforeseen situation	Precond. (OP) Operation	The number of preconditions					
		Knowledge in KB (N1)	Knowledge in KB (N2)	Generated (N3)	Covered (CR1) ratio [%]	Error (ER) ratio [%]	C.R. after (CR2) refinement [%]
Pulverizer Fault	1. activate a Pulverizer	12	6	11	100	0	100
	2. halt a Pulverizer	8	8	12	100	0	100
BFP Fault	3. activate a BFP	42	18	8	22	78	100
	4. open a FWCV	26	10	8	40	60	90
	5. set FWCV auto	32	8	8	38	62	75
	6. set FWCV hand	12	4	7	75	25	100
	7. close a FWCV	14	6	9	83	17	100
	8. halt a BFP	23	8	11	87	13	100
CP Fault	9. activate a CP	17	7	6	57	43	100
	10. halt a CP	13	7	8	86	14	100
CWV Fault	11. activate a CWP	8	7	7	57	43	100
	12. open a CWV	4	3	7	67	33	100
	13. close a CWV	4	3	8	67	33	100
	14. halt a CWP	8	7	8	71	29	100
HTR Fault	15. open a HTR Bypass VLV	4	3	failed to generate OP	failed to generate OP	failed to generate OP	failed to generate OP
	16. close a HTR VLV	4	3	failed to generate OP	failed to generate OP	failed to generate OP	failed to generate OP

- Once *DIS* was activated, no further unforeseen abnormal situation occurred.
- The *Diagnosor* deduced the exact diagnostic results.

Because of the above conditions, *SIS* interpreted all the generated knowledge and handled the unforeseen abnormal situations. Figure 10 shows the generated knowledge and its corresponding knowledge encoded by a human expert for the operation no.5 in Table 2. We also show some additional information in Figure 10, which is referred to in the next section.

5 Discussion

In this section, we evaluate the system's capability to generate the necessary plant operations and to generate the correct preconditions for each operation. The former is concerned with performance of the *Operation Generator* and the latter is concerned with that of the *Precondition Generator*. In addition, we discuss the pros and cons of using Multi-PSI and some open problems.

1. Capability to generate plant operations

In the experiment, the system could generate all the necessary plant operations for each malfunction except the *water-heater* fault. We briefly explain the reason for this failure bellow.

At a *boiler*, the following approximation holds true for outlet steam pressure (P), inlet fuel flow (F), inlet water temperature (T) and inlet water flow (G).

$$P = \int (c_1 F + c_2 G(T - \alpha_1) + \alpha_2) dt$$

c_1, c_2 are positive constants, and α_1, α_2 are correction terms related to other process parameters.

The *Operation Generator* calculates F, G and T from P using this formula defined in the *Device Model*. P is the demand for the *boiler*. After that, the *Operation Generator* propagates F to the *fuel-system*, and G and T to the *water-heater* as a new demand respectively. In this time, the *Operation Generator* must evaluate the above formula from left side to right side, but possible value combinations of F,G and T cannot be decided using the single input value P. To deal with this undecidability, the *Operation Generator* utilizes the *Operation Principle*

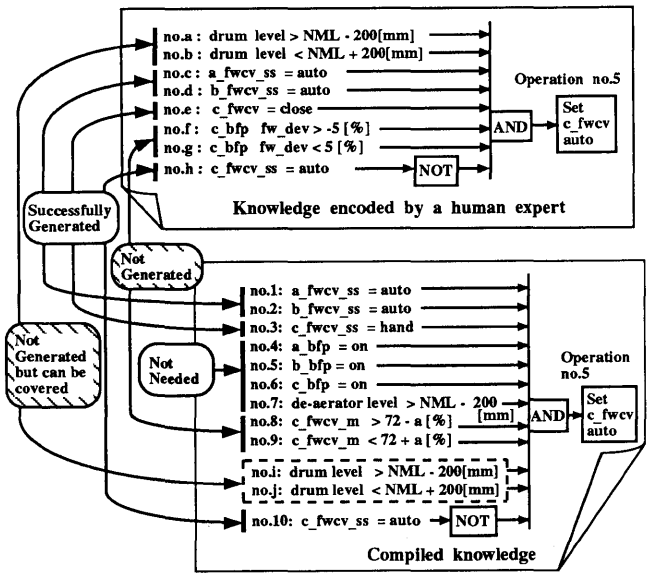


Figure 10: Knowledge for c_bfp controller

Model and approximation functions supplemented with the *Device Model*. The failure in *water-heater* fault is caused by this reasoning mechanism. We believe that additional principles are needed to evaluate such a process balance.

2. Capability to generate preconditions

   From $CR1$ and ER in Table 2, we can see that most of the generated preconditions are imperfect, namely $ER > 0$. The reasons are as follows.

   - The *Precondition Generator* failed to generate preconditions related to devices not modeled in *Device Model*. An example is the set of preconditions to establish the electric power supply for the pump. We can resolve this problem easily by augmenting the *Device Model*.
   - Although all the necessary preconditions could be checked in the *goal-state* search, the *Precondition Generator* missed analyzing them. No.i to no.j in Figure 10 illustrates this point. The system focuses only on the neighbor devices of the operated device. Because the system is required only to generate specific preconditions for an occurring unforeseen abnormal situation, we can resolve this problem easily by extending the focusing area.
   - The *Precondition Generator* generated incorrect preconditions for the timing of operations, as shown by no.8 to no.9 in Figure 10. Although the system is based on the concept that the timing of operations can be determined from the maximum outlet process flow of each device, this concept does not hold true for devices such as PID-controllers or devices placed under the control of PID-controllers.

   Although we can resolve the former two problems easily, the last problem is serious because it is closely related to the basic concept for the generation of preconditions. It is still an open problem. In Table 2, column $CR2$ represents the expected results after the refinements against the former two problems. The remaining uncovered parts for operations 4 and 5 (ER is 10% and 25% respectively) are related to the last mentioned problem above.

3. Real-time reasoning using Multi-PSI

   Although our system does not require of the severe real-time reasoning capability to cover either *PID-control* or *adaptive-control*, it requires at least the ability to compile the knowledge within a few minutes. To guarantee this performance, we have been investigating a parallel reasoning mechanism with Multi-PSI [Suzuki 1991]. We can use KL1 language on Multi-PSI, which is a profitable language to implement a multi-process system concisely. In particular, its process synchronization mechanism by "suspend" is an advantage for our system implementation. In spite of this point, it is very difficult to achieve a drastic speedup using KL1 and Multi-PSI. We have already demonstrated a threefold to fivefold improvement of reasoning time by using Multi-PSI with 16 processor elements. To achieve more improvement, we think we must make a more elaborate implementation.

4. Utility of the compiled knowledge

   In contrast to the classical approach by shallow knowledge, our proposed model-based reasoning mechanism succeeded to deal with unforeseen abnormal situations in a plant. This point is the utility of the compiled knowledge.

   Although our proposed mechanism is powerful to deal with unforeseen abnormal situations, it is weak with respect to the acquisition of knowledge which is reusable in the *SIS*. Because the system generates specific knowledge only for occurring unforeseen abnormal situations, the generated knowledge is either too general with respect to the lack of some conjunctive preconditions or too specific with respect to their enumerative representations from the viewpoint of its reusability.

5. Facility of model acquisition

   The system utilizes the *Qualitative Causal Model*, the *Device Model*, the *Operation Principle Model* and the *Dynamics Model*. These models could be

built from the plant design, and should be consistent with each other. In the current implementation of the system, each model is built and implemented separately. Therefore, model sharing is not yet realized.

In a diagnostic task, Yamaguchi[Yamaguchi 1987] refers to the facility of model acquisition. Some other related works are in the area of the qualitative reasoning. Crawford [Crawford 1990] attempted to maintain and support the qualitative modeling environment by *QPT*.

6. Over-sensitive verification of the plant behavior

   In the current implementation of the *Generate&Test* algorithm for the knowledge, the priority of each allowable range is not considered at all. Therefore, even though the violation of the range is slight enough to be ignored, the system tries to deal with this violation sensitively. This sensitivity is meaningless for all practical purposes because a plant would be designed with enough capacity to absorb the violation. For this reason, the system should check the range with some allowable degree of violation. We are now investigating the mechanism.

7. Monitoring the execution of the generated knowledge

   In this paper, we supposed that the *Diagnosor* can diagnose unforeseen events exactly. However, in general, this supposition can be invalid. Diagnostic results should be estimated by plant monitoring following the plant operations.

   As for the related work, Dvorak [Dvorak 1989] utilizes the *QSIM* [Kuipers 1986] to monitor a plant. However, he does not refer to the generation of the knowledge for unforeseen events.

6 Conclusion

We proposed a diagnostic and control expert system based on a plant model. The main target of our approach is a system which could deal with unforeseen abnormal situations. Our approach adopts a model-based architecture to realize the thought process of a skilled human plant operator.

In this paper, we focused on model-based generation of plant control knowledge, and explained the details of the model-based reasoning. Our system utilizes the following models: the *Device Model*, the *Operation Principle Model* and the *Dynamics Model*. We also discussed its ability as demonstrated through some experimental results. The results encourage us to make sure the model-based reasoning capabilities in plant control.

Acknowledgements

This research was carried out under the auspices of the Institute for New Generation Computer Technology (ICOT).

References

[Crawford 1990] Crawford, J., Farquhar, A. and Kuipers, B. *"QPC: A Compiler from Physical Models into Qualitative Differential Equations"*, Proc. of AAAI-90, pp.365–372 (1990).

[Dvorak 1989] Dvorak, D. and Kuipers, B. *"Model-Based Monitoring of Dynamics Systems"*, Proc. of IJCAI-89, pp.1238–1243 (1989).

[Konuma 1990] Konuma, C., et. al. *"Deep Knowledge based Expert System for Plant Control – Development of Conditions Generation Mechanism of Plant Operations –"*, Proc. of 12th Intelligent System Symposium, Society of Instrument and Control Engineers, pp.13–18 (1990) (in Japanese).

[Kuipers 1986] Kuipers, B. *"Qualitative Simulation"*, Artificial Intelligence, 29, pp.289–338 (1986).

[Suzuki 1990] Suzuki, J., et al. *"Plant Control Expert System Coping with Unforeseen Events – Model-based Reasoning Using Fuzzy Qualitative Reasoning –"*, Proc. of Third International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-90), ACM, pp.431–439 (1990).

[Suzuki 1991] Suzuki, J., et al. *"Plant Control Expert System on Multi-PSI Machine"*, Proc. of KL1 Programming Workshop, pp.101–108 (1991) (in Japanese).

[Taki 1988] Taki, K. *"The parallel software research and development tool: Multi-PSI system"*, *Programming of Future Generation Computers*, North-Holland, pp.411–426 (1988).

[Yamaguchi 1987] Yamaguchi, T., et al. *"Basic Design of Knowledge Compiler Based on Deep Knowledge"*, Journal of Japanese Society for Artificial Intelligence, vol.2, no.3, pp.333–340 (1987) (in Japanese).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Semiformal Metatheory for Fragmentary and Multilayered Knowledge as an Interactive Metalogic Program

Andreas Hamfelt and Åke Hansson

Uppsala Programming Methodology and Artificial Intelligence Laboratory
Computing Science Dept., Uppsala University
Box 520, S-751 20 Uppsala, Sweden
+46-18-18 25 00
hamfelt@csd.uu.se or hansson@csd.uu.se

Abstract

The expressiveness of metalogic programming enables a logically pure, structure preserving, easily updated representation of fragmentary, multilayered knowledge, which, since neither fully formalisable nor static, requires assimilation of externally supplied knowledge as well as coping with changes. In legal reasoning, e.g., only schematic rule descriptions are available from which case specific rules are dynamically specialised. Such rule proposals must be accepted by proposals for metarules of legal interpretation which, in turn, must be accepted by proposals for metametarules, etc. Inferencing between two adjacent levels corresponds to upward reflection; in contrast downward reflection is disallowed. Typically, formalising involves three distinct theories: the informal theory, the formal theory, and the *informal* metatheory discussing the latter, but for multilayered, imprecise informal theories we propose instead a *semiformal* metatheory discussing both theories. It expresses legal knowledge as a Horn clause metaprogram, implemented in Prolog, which interactively constructs and presents metaproofs (proof terms) allowing users to assess, accept or reject derived conclusions.

1. Introduction

Metaprogramming is an important technique for the three interrelated topics "knowledge representation", "knowledge processing", and "knowledge assimilation" [Kowalski 1990]. The first deals with the apt choices of formalism and approach for building formal theories, the second with the construction of proofs of theorems from such theories and/or the identification of the existence of such proofs, and the third with the assimilation of new knowledge into existing theories. We describe how these three topics and their interrelationships are involved in a semiformal metatheory characterizing fragmentary, multilayered, not fully formalisable legal knowledge as a metaprogram. This metaprogram interactively constructs metaproofs and facilitates the study of the complexity of these interrelationships as well as the adequacy of the formalisation attempt.

This study shows how *upward reflection* can be used as a powerful reasoning method. The potentials of reflection have been demonstrated in artificial domains, e.g., for representing multiagent belief and knowledge [Kowalski 1990] and for exploiting properties of predicates such as symmetry [Costantini 1990]. To our best knowledge, however, our study is the first illustrating reasoning with realistic knowledge which requires upward reflection and we hope it will contribute to the understanding of reflection as a knowledge representation tool. For instance, it indicates that, while upward reflection has its informal counterpart in legal reasoning, applying downward reflection violates on the contrary the inherent structure of legal knowledge. Upward reflection in legal reasoning is connected to "rules of legal interpretation". Briefly, if we propose a legal rule for solving a legal case we must show that the rule's structure and content are in accordance with the (meta)rules of legal interpretation, otherwise the rule is legally invalid. Likewise in automatized legal reasoning, a formula A representing a legal rule can be assumed included in an object level theory OT representing legally valid rules if its inclusion accords with the metalevel theory MT of formulas representing rules of legal interpretation, i.e., assuming *Demo* defines provability we have $Demo(OT, name(A)) \leftarrow Demo(MT, name(Demo(OT, name(A))))$ where *Demo* holds for sentences belonging to or deducible from a theory, cf. Kowalski [1990].

Though inessential in principle, metaprogramming is often convenient in practice. Reasons may be its naturalness of representing the domain knowledge or even the impracticability of giving perfect object level representations. This is paralleled in law by the role of "rules of legal interpretation" which are "inessential in principle, in the sense that, although they are necessitated in practice by the imperfections and the dynamic character of the existing systems, they would not be needed in a perfect, unambiguously formulated, consistent, and complete legal system, conformable to a stable social reality. The actual function of rules of legal interpretation is to direct the identification of the existing system and its continuous construction and readjustment." ([Horovitz 1972], p. 94). Since (meta)rules of legal interpretation are imperfect as well, metametarules are also necessary etc., giving a whole multilayered hierarchy, which may roughly be axiomatized as a multilevel theory structure in a metalogic program.

Kleene ([1980], pp. 65, 69) introduces three separate and distinct "theories" involved in the process of a formalisation: (a) the informal theory of which the formal system constitutes a formalisation, (b) the formal system or object theory, and (c) the metatheory, in which the formal system is described and studied where (b), which is formal, is not a theory in the common sense, but a system of symbols and of objects built from symbols described from (c). Theories (a) and (c), which are informal, do not have an exactly determined structure,

as does (b). Consider the following two approaches for studying (b): (i) the formal theory (b) is "introduced at once in its full-fledged complexity" and investigated by methods without making use of an interpretation. (This is known as the metamathematical approach if the methods are finitary.) (ii) the formal theory (b) is studied by recognizing an interpretation of the theory under which it constitutes a formalisation of (a), i.e., we analyse existing informal theories (a), "selecting and stereotyping fundamental concepts, presuppositions and deductive connections, and thus eventually arrive at a formal system", i.e., at the formal theory (b).

Approach (i) presupposes that the complexity of the formal theory is fully understood. This does not hold in our domain where a realistic system can only have a partial axiomatization of the formal object theory. This axiomatization can gradually be extended, though, by consulting the user both for supply of metalinguistic entities representing objects of the formal object theory and for completing formal proofs in it. Thus, in a practical system we must adhere to approach (ii).

However, an isolated study of "knowledge processing" and "representation" within the formal object theory (b) itself can be carried out along approach (i) if the problem of supplying and assimilating external knowledge is disregarded from. In such a study [Hamfelt and Hansson 1991a] we devised a Horn clause metalanguage axiomatization (c) of a formal object theory (b), as a theory of an n-level language where each level $i+1$ is the formal metalanguage of the language of level i, and where the informal theory (a) being formalised in (b) was hierarchical fragmentary legal knowledge. The investigation could be carried out as though we had had an ideal full one-to-one axiomatization (c) of the underlying formal object theory (b) by simply assuming that sufficient fragments were available for the particular cases studied, fragments that must be supplied from the outside in a "real life" application since it is impossible anticipating what knowledge will be needed.

This simplifying assumption has been removed in the present work which delves into the knowledge assimilation problem. Although the non-logical axioms of the formal object theory (b) cannot be enumerated in advance its possible content can nevertheless be discussed in a theory (c) of a metalanguage which may be informal but also formal or both. To this end we have devised a semiformal metalanguage—whose object language is the n-level language of (b)—for a theory (c) which axiomatizes the "available" part of the formal object theory (b) and encodes rules for the assimilation into it of externally supplied knowledge fragments. Knowledge assimilation is dependent on the deductive structure of (b), which can be accounted for in (c) since its objects of discourse include formal proofs, i.e., sequences of formulas of (b). Below, $\mathcal{IT}$, $\mathcal{OT}$ and $\mathcal{MT}$ denote, respectively the (a), (b) and (c) of our system.

2. The Informal Legal Theory

Let us detail our conception of the structure of legal knowledge, i.e., our informal legal theory $\mathcal{IT}$, so as to illustrate how rules at different levels operate together, as shown in fig. 2.1. We refer to [Hamfelt 1990, Hamfelt and Hansson 1991a, 1991b] for a more thorough account

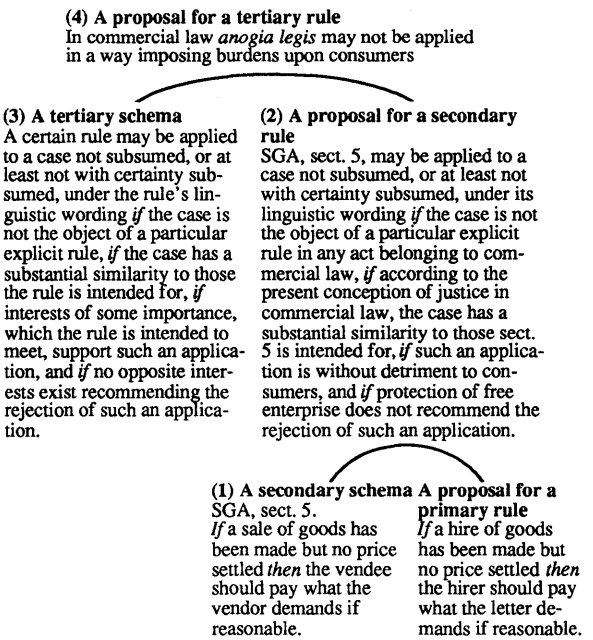


Fig. 2.1. Schemata and rules.

but the below description should suffice for this paper.

Consider the provision 1 in fig. 2.1, an ordinary statutory rule from the Swedish Sale of Goods Act (sect. 5, SGA). This provision is not only applicable to sale of goods. It could, e.g., be analogically applied to, e.g., hire of goods, or extensively interpreted, or interpreted by inversion (*e contrario*), etc., and embraces thus a lot of primary rules. One but only one of these is the rule given by a literal reading of the tokens building the provision and not even this rule has a legal validity which can be taken for granted. The provision 1 is a *schema* for all these rules and since this schema is about primary rules and thus conceptually belongs to the secondary level we call it a secondary schema.

The relation between secondary schemata and primary rules is given by secondary rules. For example, the relation between the schema 1 and real primary rules is given by secondary rules such as rule 2 in fig. 2.1 which is just an example of how a secondary rule for *analogia legis* in commercial law could possibly look. In the same way there exist tertiary schemata for secondary rules and tertiary rules that give the relation between these schemata and the secondary rules, etc. The secondary rule 2 originates from the tertiary schema 3 in fig. 2.1. Information about the relation between this schema and secondary rules such as rule 2 is given by tertiary rules, such as rule 4 in fig. 2.1.

Schematic descriptions of rules at various levels are important. We have argued elsewhere [Hamfelt 1990] that a lawyer only has a schematic knowledge of legal rules; each adjudication comprises an interpretation of schemata for legal rules and results in the construction of specialised rules applicable only to the case at hand. An obvious example is our secondary rule 2 for *analogia legis* in commercial law. It is not generally applicable. The rule is the result of an interpretation at levels

above the secondary and only "applicable" in an individual adjudication, i.e., in a particular legal case. In another legal case the interpretation at the levels above the secondary may yield another formulation of the rule 2. Rules, such as rule 2, are thus generated for each individual adjudication and there exists a diversity of possible formulations. What they have in common is that they all originate from a common schematic description 3 which in this case conceptually belongs to the tertiary level. The schematic description 3 originates from the legal literature. Rule 4 is thus also, in its turn a possible specialisation of a quaternary schema, proposed by quaternary rules which in their turn are specialisations of quinary schemata, etc. Fig. 2.2 illustrates the hierarchy of legal knowledge in which the levels have been made distinct by introducing, at each level i, the *names* for the rules of the level $i-1$.

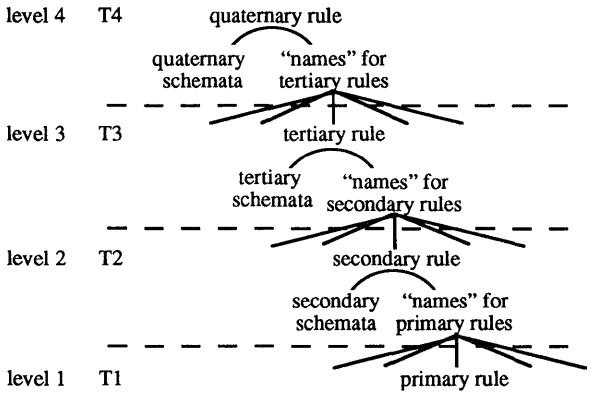


Figure 2.2. Levels of legal knowledge.

The specialisation of schemata (principles) must yield *meaningful* rules which are legally *acceptable* for the current case. The first means that the rule only consists of legal concepts embraced by the principle, the second that its content in addition is legally adequate and its premises fulfilled so it in fact applies. The latter is recursively settled, i.e., a sentence is legally acceptable if either its content directly is accepted by rules at the higher adjacent level (base case), or it follows from other acceptable sentences at its own level (recursive case). The base case is thus informal upward reflection where the upper level enforces a sentence on a lower level.

Till now rule specialisation has been described in a top-down perspective. Specialisations cannot, however, be carried out without interpretation data, the supply of which has a character of a bottom-up process, since these data mainly originate from a description of the legal case. If C is a description in legal terminology of a legal case and J is a suggestion for a judicial decision, then $J \leftarrow C$ is a proposed primary rule. Establishing that $J \leftarrow C$ is accepted by the (secondary) metarules of legal interpretation coincides then with resolving the case.

Level 1 in figure 2.2 may be understood as an object level, level 2 as its metalevel, level 3 as its metametalevel the object level of which is the pair of level 1 and level 2, etc. The structure of $\mathcal{IT}$ could be understood as a hierarchy of "theories" where the language of each level i consists of all meaningful rules expressed by schemata and conditions for their specialisation on the level $i+1$. Each T_i is characterized in T_{i+1} by schematic descriptions of its sentences and rules for deciding which specialisations of these are meaningful and legally acceptable. The same holds for the theory T_{i+1} with respect to a theory T_{i+2}, etc., and since no level has rules which do not require interpretation this proceeds *ad infinitum*. This can be handled by choosing some level n to be the "top-most" at which discretion is used for specifying schemata, thus making the validity of a primary rule proposed for resolving a case and of all rules on each level i, $1 \leq i < n$, ultimately depending on discretion.

$\mathcal{IT}$ can thus be characterized as a set of theories $\{T_1, T_2, \ldots, T_n\}$ where each theory T_i is understood as a collection of sentences fully determined by the higher adjacent theory T_{i+1}, i.e., if from theory T_{i+1} it can be deduced a theorem expressing that a sentence A_i is provable from theory T_i, then A_i belongs to theory T_i. We need a definition of the *provability relationship* between two adjacent levels $i+1$ and i ranging over n distinct levels (theories), and forming a hierarchy of dependent relations directed from the highest to the lowest level. On each level i this provability relation $T_i \vdash A_i$ should coincide with the rules of logic. The hierarchical dependence of the provability relationship, $1 \leq i < n$, may be characterized as follows $T_i \vdash A_i$ *iff* $T_{i+1} \vdash$ "$T_i \vdash A_i$", where "$T_i \vdash A_i$" names $T_i \vdash A_i$. With theory T_n specified, the hierarchy decides the content of the object level as well as of all the other levels. That is to say, T_n decides the contents of all the theories $T_1, \ldots, T_{n-1}$. The definition constitutes the *rules of acceptance* of $\mathcal{IT}$.

3. The Formal Object Theory

The provability definition in a formalisation $\mathcal{OT}$ of $\mathcal{IT}$ must support upward but prevent downward reflection. This is due to the imperfection of realistic legal systems. Both reflection principles would be allowed in Horovitz' unattainable perfect system where each T_i is an enumerable and decidable set which, for each of all the meaningful rules of its language, includes either the rule itself or its negation, but not both. Each theory T_i expresses T_{i-1} completely in a metalanguage exactly determining its content and conforms to both reflection principles since everything derivable from theory T_{i-1} can be "simulated" in theory T_i and vice versa; in principle, it would be sufficient to consider one of these theories in an ideal formalisation. In reality, however, the formalisation of each T_{i-1} can only be partial, imperfect and schematic and applying the rules in T_i is necessary for assessing, accepting or rejecting the rules in T_{i-1}. This corresponds to upward reflection, where something proved on an upper level is forced upon a lower level. Downward reflection is unsound however since something 'proved' from imperfect knowledge on a lower level cannot be forced upon an upper level thereby perhaps contradicting rules accepted by the legal principles on this level.

Thus, the amalgamated language of Bowen and Kowalski [1982] is inadequate for characterizing $\mathcal{IT}$ since its provability predicate *Demo* is founded on representability (equivalence between object language and metalanguage proofs) and requires both reflection rules.

Demo represents provability on the object level thus presupposing that an object level proof procedure exists, which is not the case in our domain whose counterpart to *Demo defines* rather than represents a provability relation for an object language. However, *Demo* predicates giving *definitional* non-conservative extensions of theories deviating from the representability notion are not unusual, e.g., metalevel definitions allowing upward reflection to enforce proofs on the object level which the object level theorem prover itself cannot carry out, cf. [Kowalski 1990]. This latter use of the *Demo* predicate accords with our understanding of $\mathcal{IT}$ and could be used to characterize its hierarchical structure.

The complexity caused by $\mathcal{IT}$'s hierarchical structure is reduced, if each T_i is characterized "locally" towards its immediate object of study, i.e., each meta/object language relation is represented separately. A link between adjacent levels is obtained by characterizing the *Demo* predicate as defining the provability relation on a lower level. For example, the formula $Demo(name(T_1), name((J \leftarrow C)))$ says (on level 2) that the primary rule $J \leftarrow C$ is provable from, and thus included in, the object theory T_1. The declarative reading of the formula is rendered by taking T_1 as a static theory implicitly consisting of all rules fulfilling the conditions for inclusion. Only the boundary between rules shown to satisfy the conditions and those not yet shown to satisfy the conditions moves just as in Sergot's "query the user" [1983] the boundary between the facts given to the computer by the user and those not yet given, moves. Stated as a goal the formula reads "is $J \leftarrow C$ provable from the theory T_1?" the proof of which corresponds to a line of arguments to the effect that the inclusion in T_1 of $J \leftarrow C$ should be regarded as in accordance with the (secondary) metarules of legal interpretation, thus showing, as hinted in sect. 2, that the rule is legally acceptable. The provability definition expresses the conditions for including $J \leftarrow C$ in T_1, i.e., the secondary rules whose inclusions in T_2 depend in their turn on theories of higher levels, whose provability definitions are characterized in a similar way yielding a whole hierarchy of interdepending provability definitions, still however allowing us to describe and consider each T_i as a separate theory.

Specifying the theory of primary rules T_1 metatheoretically in terms of what can be proved from T_1 seems thus natural. The topmost theory T_n gives an axiomatic definition of provability of theory T_{n-1} and indirectly of all theories T_i, $i < n$, hence, embracing all the non-logical axioms of these theories. We have chosen an alternative perspective where the hierarchical structure of $\mathcal{OT}$ is taken as a composite object language which can be characterized in a theory $\mathcal{MT}$ of a separate metalanguage. This metalanguage takes thus the whole n-level language of $\mathcal{OT}$ as its object language. In $\mathcal{MT}$ $Prover(Demo(name(T_1), name(A)), \ldots, \ldots, Proof)$, e.g., expresses that a formalisation of a primary rule A is included in the formal theory of $\mathcal{OT}$, which represents the informal theory T_1 of $\mathcal{IT}$. In the metalanguage this inclusion is verified by a sequence *Proof* of statements each of which names an "object/object"-inference or a "meta/object"-inference of the object language, thus constituting a metaproof in $\mathcal{MT}$ of a formal proof in $\mathcal{OT}$.

4. The Semiformal Metatheory

We now briefly describe our metatheory $\mathcal{MT}$, mainly as a program partially characterizing $\mathcal{OT}$ whose intended interpretation is $\mathcal{IT}$. Our metalogic consists of Horn clauses and the inference mechanism of Prolog. In this metalanguage $\mathcal{MT}$ is represented by formulas, i.e., in Horn clauses, and the language of $\mathcal{OT}$ by terms, i.e., terms of Horn clauses.

To the *formation rules* and *rules of inference* (including axiom schemata) of proof theory correspond the *rules of meaningfulness* and the *rules of acceptance*, respectively, both of which have a more vague character than their proof theory counterparts. It is a hard and sometimes impossible problem to state formation rules in the metatheory which can handle the vagueness of legal concepts. Therefore, it seems necessary, in contrast to proof theory, to complement with an external settlement of which rules are meaningful rules, and which rules are acceptable as true material implications. The specification of a formal theory is done from the starting-point of an informal theory which is the intended interpretation the formalism should capture. For example, a constant in the formal language of the formal theory is supposed to have a natural counterpart in the informal theory, e.g., to denote a specific individual or a specific category of individuals. In our case, we cannot predetermine this connection between a formal and an informal theory, because we cannot decide in advance all legal concepts that can be relevant in a legal system. This is something that has to be resolved from case to case. Therefore, we leave it to the user to decide the formal counterpart of his informal understanding of a legal concept, e.g., an informal legal concept "hirer" of $\mathcal{IT}$ will naturally get the constant "hirer" of the formal language of $\mathcal{OT}$. That is to say, we presuppose an immediate or autonymous relation between a symbol or symbols of the formal language and its informal counterpart. Thus, with external help a metatheory $\mathcal{MT}$ could be extended and some of the gaps in its schemata for non-logical axioms of $\mathcal{OT}$ be filled in yielding a formalisation of rules of $\mathcal{IT}$ obtained by specialising the available vague descriptions of these rules. In that way, a program representing $\mathcal{MT}$ could construct, interactively with the user, a metaproof in $\mathcal{MT}$ representing a formal proof in $\mathcal{OT}$; at least for the proposal of a legal case under consideration. In some sense this position could be understood as performing a specification in a metatheory of the formal language of a formal theory as well as deciding its non-logical axioms in parallel with a derivation of some conclusion from it. At the end of a session (of constructing metaproofs) the relevant fractions of this theory, and the formal proof of a particular legal case could be presented to the user for a final examination by extracting the formal proof out of the metaproof. How adequately these formal proofs, represented in the metatheory $\mathcal{MT}$, correspond to $\mathcal{IT}$ is a matter of external judgements. Observe, that $\mathcal{MT}$ is a *semiformal* theory in the sense that it has both a formal part, consisting of sentences represented as Horn clauses, and an informal part consisting of user interpreted sentences.

Thus, the *rules of meaningfulness* in $\mathcal{MT}$ can only partially characterize sentences of the object language, i.e., the language of $\mathcal{OT}$. In the representation in the

metalanguage, we have to assume a fixed structure for designating a class of rules, i.e., a schema. Within this structure local differences must be met, i.e., different specialisations of the schema have to give different representations of sentences (rules) of the object language. These local differences are expressed by metavariables which have to be filled in by a user and satisfy certain interactively investigated typing conditions. Let us illustrate this with the program clause that characterizes a provision schema whose linguistic wording is the one specified for the metavariable Text. In the clauses below n(...) is a shorthand for name(...) for which we postulate an inverse law of naming, i.e., n(A)=n(B)→A=B. As to the problem of naming in metaprogramming, observe that all variables are metavariables; there are no object variables.

```
meaningfulsent(t(1),RuleProp1,[ModAt1,unspec],LegSet1,Text):-
  RuleProp1 = (legalcons(pay,X,Y,goods,price):-
                  and(actor1(X,goods),and(actor2(Y,goods),
                  and(unsettledprice(goods),and(demands(Y,price),
                  reasonable(price,goods))))))),
  ModAt1 = [X/vendee,Y/vendor], Types = [actor(X),actor(Y)],
  LegSet1 = [[provisionno(sga(5))|_],LegSet0],
  Text = the same text as in rule schema 1 in fig. 2.1.
  propertyping(t(1),RuleProp1,ModAt1,Types,Text).
```

In $\mathcal{IT}$ this provision is assumed open with respect to the concepts 'vendee' and 'vendor'. So, the assumed fixed structure of this provision is represented in the metalanguage as the term specified for the metavariable RuleProp1 with open places expressed by the metavariables X and Y. These variables have to be specialised interactively with the user. The predicate propertyping/5 is defined for this interaction. The metavariable ModAt1 expresses the relation between the concepts of $\mathcal{IT}$, i.e., the text of Text and its open parts, i.e., 'vendee' and 'vendor', and its formal counterpart in $\mathcal{OT}$ partly specified in RuleProp1. Thus, a proper typing carried out by the user gives a meaningful rule of the object language of level 1, represented in the metalanguage by the specialised term of RuleProp1. The metavariable LegSet1 identifies what part of level 1 in $\mathcal{IT}$ is relevant for a particular case.

The *rules of acceptance* may also only be partially characterized in the metalanguage. However, a user can interactively add interpretation data, thereby extending the partial characterization of $\mathcal{OT}$ in the theory $\mathcal{MT}$ of the metalanguage. What is hard characterizing is the determination of whether or not a meaningful rule belongs to a theory of $\mathcal{IT}$, i.e., is legally acceptable, and thus should have a formal counterpart in $\mathcal{OT}$. Presently, this is solved by assuming in $\mathcal{MT}$ that a rule is acceptable when a user tries to apply it, and the conditions for its application are accepted, i.e., either follow by logic from other accepted rules or are included in the theory by rules at the higher adjacent level in co-operation with the user. So, we presuppose that it is only the user who can determine the relevance of a specific principle. Consequently, at the end of a session these assumptions should be possible for a user to examine.

These aspects are encoded in the prover clause [UP] (short for upward reflection). Observe that the prover clauses belong to $\mathcal{MT}$ which takes as object theory the whole multilayered $\mathcal{OT}$. Their first demo argument defines the formalisation in $\mathcal{OT}$ of logic provability between a theory T_i of $\mathcal{IT}$ and a sentence of $\mathcal{IT}$ but though e.g., the fourth proof term argument has a counterpart in $\mathcal{OT}$— a formal proof extending over the whole hierarchy of $\mathcal{OT}$—it includes expressions solely of $\mathcal{MT}$ as well.

```
[UP]
prover(demo(n(t(I)),n(SentPropI)),ModI,LegSetI,ProofI):-
  proposesent(t(I),SentPropI,ModI,LegSetI),
  J is I + 1,
  ground([SentPropI,ModI,LegSetI]),
  permissible(t(I),SentPropI),
  prover(demo(n(t(J)),n(demo(n(t(I)),n(SentPropI)))),
         [ModAtJ,ModI],[LegSetAtJ,LegSetI],ProofJ),
  ProofI = (sentenceof(theory(I),SentPropI):-
              proofof(theory(J),proved(theory(I),SentPropI),ProofJ)).

permissible(t(I),SentPropI):-I = 1.
permissible(t(I),SentPropI):-I ≥ 2,\+ SentPropI = (Head:-Body).
```

Clause [UP] encodes in $\mathcal{MT}$ upward reflection between two theories T_i and T_j of arbitrary adjacent levels in $\mathcal{IT}$, with formal counterparts t(I) and t(J) in $\mathcal{OT}$. A sentence is assumed to belong to a theory T_i if this accords with the rules of theory T_j of the higher adjacent level. In $\mathcal{MT}$, LegSetI and ModI identify and modify formula schemata corresponding to known fragments of sentences of the theory T_i. The predicate proposesent/4 is defined to specialise interactively with a user such meaningfulsent schemata. ProofI is a metaproof in $\mathcal{MT}$ of the existence of a sequence of formulas in $\mathcal{OT}$'s formalisation of $\mathcal{IT}$ constituting a formal proof of the proposed sentence.

Upward reflection must be constrained. If each sentence were upward reflected directly when proposed, the reasoning process would ascend directly to the topmost level since the metarule proposed for assessing the sentence would itself directly be upward reflected, etc. Therefore, at levels i, $i \geq 2$, only sentence proposals which are ground facts may be upward reflected, postponing the assessment of rules, which may only be proposed as non-ground conditional sentences, till facts are activated by their premises. Under this reasoning scheme the content of all sentences involved in the reasoning process will eventually be assessed. The restriction is maintained by the permissible subgoal.

Clause [ANDI] handles $\wedge$-introduction. In $\mathcal{MT}$ a theory T_i of $\mathcal{IT}$, with t(I) as formal counterpart in $\mathcal{OT}$, is assumed to include a sentence which is a conjunction if both its conjuncts may be assumed included in T_i.

```
[ANDI]
prover(demo(n(t(I)),n(and(G1,G2))),
       [[ModG1,ModG2],ModsBelow],LegSetI,ProofI):-
  I ≥ 2,
  prover(demo(n(t(I)),n(G1)),[ModG1,ModsBelow],LegSetI,ProofG1),
  prover(demo(n(t(I)),n(G2)),[ModG2,ModsBelow],LegSetI,ProofG2),
  ProofI =(sentenceof(theory(I),and(G1,G2)):-
             and(proofof(theory(I),G1,ProofG1),
             proofof(theory(I),G2,ProofG2))).
```

Clause [MP] encodes our version of modus ponens. In $\mathcal{MT}$ a theory T_i of $\mathcal{IT}$, with t(I) as formal counterpart in $\mathcal{OT}$, is assumed to include a sentence which is the consequence of a proposed implication of T_i whose antecedent can be assumed included in T_i.

```
[MP]
prover(demo(n(t(I)),n(HeadI)),ModI,LegSetI,ProofI):-
  I ≥ 2,
  proposesent(t(I),(HeadI:-BodyI),ModI,LegSetI),
  prover(demo(n(t(I)),n(BodyI)),ModI,LegSetI,ProofBodyI),
  ProofI = (sentenceof(theory(I),HeadI):-
              and(ruleof(theory(I),(HeadI:-BodyI)),
              proofof(theory(I),BodyI,ProofBodyI))).
```

The knowledge of rules in $\mathcal{IT}$ for assessing sentence proposals for the adjacent lower level theory T_i will at some level j be too rudimentary for composing a theory T_j. At this level, T_j is considered to be the user's opinion of the sentences proposed for T_i. This is encoded in $\mathcal{MT}$ in the clause [TOP].

```
[TOP]
prover(demo(n(t(J)),n(demo(n(t(I)),n(RulePropI)))),
       ModJ,LegSetJ,ProofJ):-
  J ≥ 2,
  \+ proposesent(t(J),(demo(n(t(I)),n(RulePropI)):-BodyJ),
                 ModJ,LegSetJ),
  externalconfirmation(t(I)),RulePropI,ModJ,LegSetJ),
  ProofJ = externallyconfirmed(sentenceof(theory(I),RulePropI)).
```

Let us now partially trace the computation of a sample query

```
>prover(demo(n(t(1)),n(RuleProp1)),Mod1,LegSet1,Proof1).
```

This query could be read as "is there a metaproof Proof1 stating that the theory T_1 of level 1 includes a primary rule which is represented in $\mathcal{OT}$ by RuleProp1 and modified by Mod1 in the legal setting LegSet1?" Since it is completely unspecified at this point what particular problem to solve the query can be stated in these general terms and be generated by the system. The goal resolves with the prover clause [UP] leading to six subgoals, the last of which builds the proof term to bind Proof1. Below, we refrain from discussing how the proof term is built during the computation. The first subgoal of [UP] is

```
proposesent(t(1),SentProp1,Mod1,LegSet1)
```

which through user interaction selects a legal rule and modifies it for the current case. The unifying clause

```
proposesent(Theory,RuleProp,Mod,LegSet):-
  (Theory = t(1);RuleProp=(demo(_,_):-Body)),[1]
  findlegalsetting(Theory,LegSet),
  meaningfulsent(Theory,RuleProp,Mod,LegSet,Text).
```

identifies the relevant part of the legal domain from which it retrieves a proposal for a rule provided it is meaningful. The latter is sorted out by meaningfulsent clauses, say, the one presented above. In this clause the propertyping condition is intended to promote that user proposed modifications preserve the rule's meaningfulness. Suppose now that the user interaction makes the first subgoal of [UP] return with the following ground argument bindings, i.e., the schemata from sect. 5 Sale of Goods Act is adapted into a primary rule proposal regulating a case of 'hire of goods',

```
LegSet1 =
  [[provisionno(sga(5)),
    provisioncategory('Determination of Purchase Money'),
    legalfield('Commercial Law')],unspec], call it ⟨legset1⟩
```

[1] The legal setting may be assumed unknown if any of these two conditions hold.

```
Mod1 = [[hirer/vendee,letter/vendor],unspec], call it ⟨mod1⟩
RuleProp1 =
  (legalcons(pay,hirer,letter,goods,price):-
     and(actor1(hirer,goods),and(actor2(letter,goods),
     and(unsettledprice(goods),and(demands(letter,price),
     reasonable(price,goods))))))), call it ⟨ruleprop1⟩
```

Now it must be established whether it accords with the higher adjacent level, i.e., the theory T_2, to assume a primary rule with this proposed content is included in the theory T_1. This is accomplished through "upward reflection". Before a formula with content information is upward reflected it must be checked for groundness (a hack) and permissibleness. These are the tasks of the third subgoal of [UP] (where ⟨*name*⟩ is shorthand for an occurrence of the term named by *name*).

```
ground([⟨ruleprop1⟩,⟨mod1⟩,⟨legset1⟩])
```

and of the fourth subgoal of [UP]

```
permissible(t(1),⟨ruleprop1⟩),
```

which permits a conditional rule on level 1 to be upward reflected. The fifth, "upward reflection", subgoal of [UP]

```
prover(demo(n(t(2)),n(demo(n(t(1)),n(⟨ruleprop1⟩)))),
       [ModAt2,⟨mod1⟩],[LegSetAt2,⟨legset1⟩],Proof2),
```

resolves with the prover clause [MP] leading to four subgoals (the first and last of which controls the index of the current level and builds the proof term, respectively). Now a secondary rule must be proposed for assessing the lower level expression. The second subgoal of [MP] is

```
proposesent(t(2),(demo(n(t(1)),n(⟨ruleprop1⟩)):-Body2),
            ⟨mod2⟩,⟨legset2⟩),
```

where ⟨mod2⟩ is [ModAt2,⟨mod1⟩], ⟨mod1⟩ is [⟨modat1⟩,unspec], ⟨modat1⟩ is [hirer/vendee,letter/vendor] and ⟨legset2⟩ is [LegSetAt2, ⟨legset1⟩].

Suppose the user chooses the *analogia legis* principle. The relation between primary rules of theory T_1 and secondary rules for analogia legis of theory T_2 is encoded in this clause:

```
meaningfulsent(t(2),RuleProp2,Mod2,LegSet2,Text):-
  RuleProp2 =
    (demo(n(t(1)),n(RuleProp1)):-
       analogialegis(n(RuleProp1),n(ModAt1),LegSet1)),
  Mod2 = [_,[ModAt1,_]],
  LegSet2 = [[interpretationtheory('analogia legis')|_],LegSet1],
  Text = '"A primary rule proposal is legally valid (i.e., belongs to the
          theory t1 of valid primary rules) if its inclusion accords with
          the secondary rule for analogia legis."...',
  propertyping(t(2),RuleProp2,[],[],Text).
```

The second subgoal of [MP] returns with its second argument bound to

```
(demo(n(t(1)),n(⟨ruleprop1⟩)):-
   analogialegis(n(⟨ruleprop1⟩),n(⟨modat1⟩),⟨legset1⟩))
```

and LegSetAt2 bound to [interpretationtheory('analogia legis')|_]. The third subgoal of [MP] is

```
prover(demo(n(t(2)),n(analogialegis(n(⟨ruleprop1⟩),
                                    n(⟨modat1⟩),
                                    ⟨legset1⟩)))),
       ⟨mod2⟩,⟨legset2⟩,ProofBody2),
```

which recursively calls [MP]. Now a meaningful proposal for an actual analogia legis secondary rule will, by the second proposesent subgoal of [MP], be retrieved from this clause

```
meaningfulsent(t(2),RuleProp2, _,LegSet2,Text):-
  RuleProp2 =
    (analogialegis(n((Cons:-Ante)),n(ModAt1),LegSet1):-
      and(not(casuisticalinterpretation(LegalField,
                                        n((not(Cons):-Ante)))),
      and(intendedfor(ProvisionNo,n(TypeCase)),
      and(substantialsimilarity(n(TypeCase),n(Ante),n(ModAt1)),
      and(intendedtomeet(ProvisionNo,Interests,LegalField),
      and(supports(ProvisionNo,n(ModAt1),ProInt,Interests),
      and(recommendrejection(ProvisionNo,n(ModAt1),
                             ContraInt,Interests),
      outweigh(ProInt,ContraInt))))))))),
  LegSet2 = [[interpretationtheory('analogia legis')|_],LegSet1],
  LegSet1 = [[provisionno(ProvisionNo), _,legalfield(LegalField)], _],
  Text = the same text as in rule schema 3 in fig. 2.1.
  propertyping(t(2),RuleProp2,[],[],Text).
```

with these bindings (where ⟨ruleprop1⟩ is (⟨consrule1⟩:-⟨anterule1⟩))

```
analogialegis(n((⟨consrule1⟩:-⟨anterule1⟩)),n(⟨modat1⟩),⟨legset1⟩):-
  and(not(casuisticalinterpretation('Commercial Law',
                                    n((not⟨consrule1⟩:-⟨anterule1⟩)))),
  and(intendedfor(sga(5),n(TypeCase)),
  and(substantialsimilarity(n(TypeCase),n(⟨anterule1⟩),n(⟨modat1⟩)),
  and(intendedtomeet(sga(5),Interests,'Commercial Law'),
  and(supports(sga(5),n(⟨modat1⟩),ProInt,Interests),
  and(recommendrejection(sga(5),n(⟨modat1⟩),ContraInt,Interests),
  outweigh(ProInt,ContraInt)))))).
```

Now it must be proved that with the proposed content the antecedent of the *analogia legis* rule (call it ⟨albody⟩) is included in T_2. The third subgoal of [MP] is

```
prover(demo(n(t(2)),n(⟨albody⟩)), _,
       [[interpretationtheory('analogia legis')],⟨legset1⟩], _),
```

and each of the conjuncts in ⟨albody⟩ will be demonstrated in turn by the prover clauses [ANDI], [MP], and [UP]. To illustrate how user proposed content for a sentence is accepted (or rejected) at higher levels let us focus on the fourth conjunct which gives rise to the goal

```
prover(demo(n(t(2)),
            n(intendedtomeet(sga(5),Interests,'Commercial Law'))),
       _,[[interpretationtheory('analogia legis')|_],⟨legset1⟩]).
```

An "intended to meet" sentence must be proposed by the user. The result may be a meaningful fact (unconditional sentence) whose inclusion in the theory T_2 must be accepted by the rules of theory T_3 or it may be a rule (conditional sentence) which is assumed included in T_2 directly after the user's acceptance. The resolving clauses in the respective cases are [UP] and [MP]. Thus, in the first case upward reflection occurs immediately. In the second case upward reflection is postponed until backward inferencing by modus ponens at the current level leads to the proposal of a fact. Note that this guarantees that the application of the originally proposed rule is not accepted unless all the components of its antecedent eventually are assessed and accepted.

Suppose a fact is proposed. The goal will resolve with the prover clause [UP], whose recursive fifth subgoal resolves with the prover clause [MP] leading to the application of tertiary rules for assessing the proposed (secondary) fact. Reasons of space force us to remove a part of the trace here. The inferencing at the tertiary level is similar to that just described for the secondary level. We conclude this section with a fragment of the trace in which a tertiary fact is proposed but no quaternary rules exist for assessing it. The upward reflected goal looks like

```
prover(demo(n(t(4)),
            n(demo(n(t(3)),
                   n(adequatetoequalize(
                       'actors with similar economical positions',
                       'consumer protection'/'hirer protection',
                       'Commercial Law'))))),
       Mod4,LegSet4, _).
```

For the theory T_4 proposesent fails however to return any quaternary rules which may assess the adequatetoequalize fact. The goal resolves with the prover clause [TOP] and the user may or may not accept the content of the "adequate to equalize" rule.

Provided the rule is accepted this completes the computation of the fourth conjunct in the antecedent of the analogia legis rule. The following three conjuncts in the antecedent of the analogia legis rule are computed likewise which completes the computation of the initial query. A conclusion is not considered as final before the line of arguments leading up to it has been considered and accepted by the user. To this end the user needs a comprehensible presentation of the proof term. We illustrate elsewhere [Hamfelt and Hansson 1991b] how derivations of goals can be entrusted to the user's acceptance or rejection by an interactive piecemeal unfolding of a term representing the proof of the goal.

5. Coping with Change

A program should be able to *cope with changes* in the frequently revised legal knowledge it formalises. Also it should be *structure preserving* ("isomorphic") modulo this knowledge, cf. [Sergot *et al.* 1986]. This is a conflict, Bratley *et al.* [1991] claim, since coping with changes requires modifying "implicit or explicit rules which do not correspond directly to paragraphs in the text of law".

Our metalogic program $\mathcal{MT}$, however, is a structure preserving formalisation of legal knowledge coping with changes. The schemata give a modular, direct and easily changed description of statutory rules and (meta ...)metarules of legal interpretation. $\mathcal{MT}$ is modular both horizontally and vertically entailing that adjustments can be made locally to the schemata for the (higher level) rules of legal interpretation as well as to the schemata for the ordinary (low level) statutory rules. The level of the knowledge is identified and the appropriate adjustment made to its rule schemata, which then control the computation of accepted rules assumed included in theories of the lower adjacent level. Also, since $\mathcal{MT}$ takes as its object language the whole n-level language of $\mathcal{OT}$, we can encode in the formal part of $\mathcal{MT}$, rules coping with global changes which are not possible to localize to rule schemata of a certain level. Furthermore, if the legal system has undergone an even more drastic revision, a large part of our system will nevertheless remain intact since the structure of principles such as *analogia legis* will hardly be affected. The structure

preserving model of the British Nationality Act [Sergot *et al.* 1986] is according to Kowalski and Sergot [1990] "of limited practical value" since it expresses a "layman's reading of the provision" but in our $\mathcal{MT}$ expert knowledge may be incorporated e.g., for verifying the correctness of $\mathcal{OT}$, modifying and augmenting it, and for suggesting promising ways for applying its rules.

6. Related Work

Allen and Saxon [1991] discuss, in contrast to our multiple semantic interpretations, assistance for multiple structural interpretation of components of provisions, such as "if", "not", "provided that", e.g., by changing which component is taken as the main connective of a sentence. The logical relationship between theories comprising interpretative knowledge and interpreted theories is not analysed.

Assessing and compiling persuasive lines of arguments pro and contra different, often contradictory, legal decisions is important in legal reasoning. Proof terms should thus be objects of discourse and be reasoned about, which they are in $\mathcal{MT}$. This is advocated also by Bench-Capon and Sergot [1988], who do not, however, propose a formalisation or a detailed informal theory, such as our $\mathcal{IT}$, concerning how these aspects are sorted out in informal legal reasoning.

7. Conclusions and Further Work

Above we have proposed a *novel approach* for representing fragmentary, multilayered, not fully formalisable knowledge, in which the informal metatheory of the usual formalisation approach is replaced by a semiformal metalogic program which interactively composes formal object theories to be accepted or rejected as formalisations of the knowledge by the user. Our representation easily copes with changes in the represented knowledge.

Imprecise knowledge requires advanced *user interaction* that promotes meaningful user answers and queries, constructs and intelligibly displays proof terms explaining derived conclusions, and makes the system pose its questions in a natural order. These aspects have been considered and to some extent solved in our program [Hamfelt and Hansson 1991b].

Multiple semantic interpretations of provisions is realised by allowing the user to fill schemata with meaningful content referring to his fact situation whereupon the system accepts or rejects the thus proposed rule. Including multiple structural interpretations, e.g., adding premises, should raise no real obstacles provided rules of acceptance for such alteration can be established.

In *case law* rules of legal interpretation are as important as in statute law and apart from the difficult problem of inducing schemata from precedent cases, we hypothesize that our framework needs only minor adaptations to catch the problem of case-based reasoning.

Proof terms should, since the notion of being a persuasive line of arguments is vague, not only be displayed for user communication but also reasoned about.

Acknowledgments

We like to thank Keith Clark and Leon Sterling for valuable comments.

References

[Allen and Saxon 1991] L. E. Allen and S. S. Saxon. More IA Needed in AI: Interpretation Assistance for Coping with the Problem of Multiple Structural Interpretations. In *Proc. Third Int. Conf. on Artificial Intelligence and Law*, ACM, New York, 1991. pp. 53–61.

[Bench-Capon and Sergot 1988] T. Bench-Capon and M. Sergot. Toward a Rule-Based Representation of Open Texture in Law. *Computer Power and Legal Language*, ed. C. Walter, Quorum Books, New York, 1988. pp. 39–60.

[Bowen and Kowalski 1982] K. A. Bowen and R. A. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. *Logic Programming*, eds. K. Clark and S.-Å. Tärnlund, Academic Press, London, 1982. pp. 153–72.

[Bratley *et al.* 1991] P. Bratley, J. Fremont, E. Mackaay and D. Poulin. Coping with Change. In *Proc. Third Int. Conf. on Artificial Intelligence and Law*, ACM, New York, 1991. pp. 69–75.

[Costantini 1990] S. Costantini. Semantics of a Metalogic Programming Language. In *Proc. Second Workshop on Metaprogramming in Logic*, ed. M. Bruynooghe, Katholieke Universiteit Leuven, 1990. pp. 3–18.

[Hamfelt 1990] A. Hamfelt. *The Multilevel Structure of Legal Knowledge and its Representation*, Uppsala Theses in Computing Science 8/90, Uppsala University, Uppsala, 1990.

[Hamfelt and Barklund 1990] A. Hamfelt, J. Barklund. Metaprogramming for Representation of Legal Principles. In *Proc. Second Workshop on Metaprogramming in Logic*, ed. M. Bruynooghe, Katholieke Universiteit Leuven, 1990. pp. 105–22.

[Hamfelt and Hansson 1991a] A. Hamfelt, Å. Hansson. Metalogic Representation of Stratified Knowledge. *UPMAIL TR 66*, Comp. Sci. Dept., Uppsala University, Uppsala, 1991.

[Hamfelt and Hansson 1991b] A. Hamfelt, Å. Hansson. Representation of Fragmentary and Multilayered Knowledge—A Semiformal Metatheory as an Interactive Metalogic Program. *UPMAIL TR 68*, Comp. Sci. Dept., Uppsala University, Uppsala, 1991.

[Horovitz 1972] J. Horovitz. *Law and Logic*. Springer-Verlag, Vienna, 1972.

[Kleene 1980] S. C. Kleene, *Introduction to Metamathematics*. North Holland, New York, 1980.

[Kowalski 1990] R. A. Kowalski. Problems and Promises of Computational Logic. *Computational Logic*, ed. J. W. Lloyd, Springer-Verlag, Berlin, 1990. pp. 1–36.

[Kowalski and Sergot 1990] R. A. Kowalski, M. J. Sergot. The Use of Logical Models in Legal Problem Solving. *Ratio Juris*, Vol. 3, No. 2 (1990), pp. 201–18.

[Sergot *et al.* 1986] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, H. T. Cory. The British Nationality Act as a Logic Program. *Comm. ACM* 29, (May 1986), pp. 370–86.

[Sergot 1983] M. J. Sergot. A Query-the-User Facility for Logic Programming. *Integrated Interactive Computer Systems*, eds. P. Degano and E. Sandewall, North-Holland, Amsterdam, 1983. pp. 27–41.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

HELIC-II:
A Legal Reasoning System on the Parallel Inference Machine

Katsumi Nitta (1) Yoshihisa Ohtake (1) Shigeru Maeda (1)
Masayuki Ono (1) Hiroshi Ohsaki (2) Kiyokazu Sakane (3)

(1) Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
(2) Japan Information Processing Development Center
(3) Nippon Steel Corporation

nitta@icot.or.jp

Abstract

This paper presents HELIC-II, a legal reasoning system on the parallel inference machine. HELIC-II draws legal conclusions for a given case by referring to a statutory law (legal rules) and judicial precedents (old cases). This system consists of two inference engines. The rule-based engine draws legal consequences logically by using legal rules. The case-based engine generates legal concepts by referencing similar old cases. These engines complementally draw all possible conclusions, and output them in the form of inference trees. Users can use these trees as material to construct arguments in a legal suit.

HELIC-II is implemented on the parallel inference machine, and it can draw conclusions quickly by parallel inference.

As an example, a legal inference system for the Penal Code is introduced, and the effectiveness of the legal reasoning and parallel inference model is shown.

1 Introduction

The primary knowledge source of a legal inference system is a statutory law. A statutory law is a set of legal rules. As legal rules are given as logical sentences, they are easily represented as logical formulae. Therefore, if a new case is described using the same predicates as those appearing in legal rules, we can draw legal conclusions by deductive reasoning.

However, legal rules often contain legal predicates (legal concepts) such as "public welfare" and "in good faith". Some legal concepts are ambiguous and their strict meanings are not fixed until the rules are applied to actual facts. Predicates which are used to represent actual facts do not contain such legal concepts. As there are no rules to define sufficient conditions for legal predicates, in order to apply legal rules to actual facts, interpreting rules and matching between legal concepts and facts are needed. To realize this, precedents (old cases) are often referenced because they contain the arguments of both sides (plaintiff vs. defendant or prosecutor vs. defendant) and the judge's opinions concerning interpretation and matching.

Consequently, legal reasoning can be modeled as a combination of logical inference using legal rules and case-based reasoning using old cases. Based on this model, several hybrid legal inference systems consisting of two inference engines have been developed [Rissland *et al.* 1989] [Sanders 1991(a)]. However, as practical legal systems contain many legal rules and old cases, it takes a long time to draw conclusions. Moreover, controlling two engines often requires a complex mechanism.

ICOT (Institute for New Generation Computer Technology) has developed parallel inference machines (Multi PSI and PIMs) [Uchida *et al.* 1988],[Goto *et al.* 1988]. These are MIMD-type computers, and user's programs written in parallel logic programming language KL1 [Chikayama *et al.* 1988] are executed in parallel on them.

The **HELIC-II** (Hypothetical Explanation constructor by Legal Inference with Cases by 2 inference engines) is a legal inference system based on the hybrid model. It has been developed on the parallel inference machine, and draws legal conclusions for a given case by quickly referencing statutory law and old cases.

In Section Two, we introduce the function and architecture of HELIC-II. In Section Three, we explain legal knowledge representation. In Section Four, we explain the reasoning mechanism of HELIC-II. In Section Five, a legal inference system of the Penal Code is explained.

2 Overview of HELIC-II

The function of HELIC-II is to generate all possible legal conclusions for a given case by referring to legal rules

and old cases. These conclusions are represented in the form of inference trees which include final conclusions and explanations of them.

HELIC-II consists of two inference engines - the rule based engine and the case-based engine - and three knowledge sources - a rule base, a case base and a dictionary of concepts (see Fig.1). The rule-based engine refers to legal rules and draws legal consequences logically. The case-based engine generates abstract predicates (legal concepts) from concrete predicates (given facts) by referring to similar old cases.

HELIC-II draws legal consequences using these two engines. Since the reasoning of these engines is data-driven, there are no special control mechanisms to manage this. A typical pattern of reasoning by HELIC-II is as follows. When a new case (original facts) is given to HELIC-II, the case-based engine initially searches for similar old cases and generates legal concepts which may hold in the new case. These concepts are passed to the rule-based engine by way of working memory(WM). Then, the rule-based engine draws legal consequences using original facts and legal concepts.

These results are gathered by an explanation constructor, which then produces inference trees.

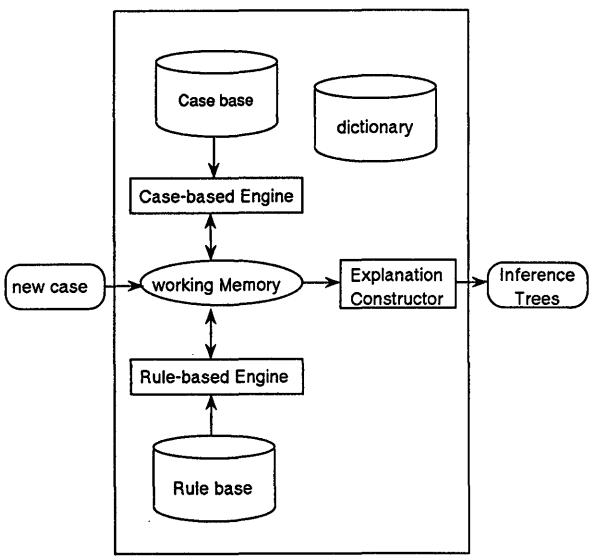


Figure 1: The architecture of HELIC-II

3 Knowledge Representation

In this section, we will explain the representation of legal knowledge in HELIC-II. We will show how to represent legal rules, old cases and legal concepts.

3.1 Representation of Legal Rules

A statutory law consists of legal rules. Each legal rule is represented as follows.

$$RuleName(Comment, RuleInfo, [A_1, A_2, ..., A_i] \rightarrow [[B_1, .., B_k], [C_1, .., C_l], ..]).$$

In this clause, *RuleName* is the rule identification, *Comment* is a comment for users and *RuleInfo* is additional information such as article number. The LHS ($[A_1, A_2, ..., A_i]$) is the condition part, and the RHS($[[B_1, .., B_k], [C_1, .., C_l], ..]$) is the consequence part. $[B_1, .., B_k]$ and $[C_1, .., C_l]$ are combined disjunctively. Each literal of the LHS and RHS is an *extended predicate* or its negation (denoted by "$\sim$" or "*not*"). An *extended predicate* consists of a predicate (concept), an object identifier and a list of *attribute* = *value* pairs. The following is an example of an extended predicate. An object "*drive*1" is an instance of a concept "*drive*". Two *attribute* = *value* pairs (*agent* = *tom* and *car* = *toyota*1) are defined.

$$drive(drive1, [agent = tom, car = toyota1]).$$

Internally, this extended predicate is treated as a set of the triplet $\{object, attribute, value\}$ as follows.

$$\{drive1, agent, tom\}$$
$$\{drive1, car, toyota1\}$$

In a clause, we can use "*not*" (negation as failure) in addition to "$\sim$" (logical not). By introducing "*not*", nonmonotonic reasoning is realized, and the representation of exceptional rules and presumed facts are easily represented [Sartor 1991].

The following are examples of legal rules.

$homicide01(\text{“}example\text{”}, [article = 199],$
$[person(A), person(B),$
$action(Action, [agent = A]),$
$intention(Intention, [agent = A, action = Action,$
$goal = Result]),$
$death(Result, [agent = B]),$
$caused(Caused, [event = Action, effect = Result2]),$
$death(Result2, [agent = B]),$
$not(\sim illegality(Illegal, [agent = A,$

$action = Action, result = Result2]))]$
$\rightarrow$
$[[crimeOfHomicide(Crime, [agent = A,$
$action = Action, result = Result2])]]).$

$legality01(\text{"}example\text{"}, [article = 38],$
$[action(Action, [agent = A]),$
$intention(Intention, [agent = A, action = Action,$
$goal = Result]),$
$selfDefence(Result, [object = Action]),$
$caused(Caused, [event = Action, effect = Result2])]$
$\rightarrow$
$[[\sim illegality(Illegal, [agent = A,$
$action = Action, result = Result])]]).$

The first rule is a definition of the crime of homicide, which is given by the Penal Code.

The meaning of "$not(\sim illegality(Illegal, [...]))$" is that illegality is presumed, in other words, if there isn't proof that "$\sim illegality(Illegal, [...])$" holds then "$not(\sim illegality(Illegal, [...]))$" is true.

The second rule is an exception of the first rule. If a person did some action in defense, "$illegality(Illegal, [...])$" is refuted.

3.2 Representation of Cases

A judicial precedent consists of the arguments of both sides, the opinion of the judges and a final conclusion. We represent a precedent (an old case) as a *situation* and some *case rules*, and represent a new case as a *situation*.

(1) Situation

A *situation* consists of a set of events/objects and their *temporal relations*. An event and an object are represented as an extended predicate as introduced in the previous section. The temporal relations are represented as follows.

$problem(CaseID, Comment, TemporalRelations).$

CaseID is the case identification, *Comment* is a comment for users and *TemporalRelations* is a list of relations between events. To represent temporal relations between events/objects, we use Allen's interval notation such as "$before$", "$meets$", "$starts$", and so on [Allen 1984].

The following is an example of a situation.

$problem(trafficAccident112, \text{"}example\text{"},$
$[before(dinner1, drive1), during(accident1, drive1)]).$
$dinner(dinner1, [agent = john, place = maxim's]).$
$drive(drive1, [agent = john, car = toyota1]).$
$accident(accident1, [agent = john]).$
$person(john, [sex = male]).$
$person(mary, [sex = female]).$
$restaurant(maxim's, [rank = 5stars]).$
$car(toyota1, [type = sportsCar]).$

The meaning of this example is that the case "$traffic\ accident$ 112" consists of three events such as "$dinner1$", "$drive1$" and "$accident1$". "$Dinner1$" occurred before "$drive1$", and "$accident1$" happened during "$drive1$". The event "$dinner1$" is a lower concept of "$dinner$", and it is acted by "$john$" in "$maxim's$", etc..

(2) Case Rules

Arguments by both sides are represented as a set of *case rules*. The following is the syntax of a case rule.

$RuleName(Comment, RuleInfo,$
$[A_1, A_2, ..., A_i] \rightarrow [B_1, B_2, .., B_k]).$

RuleName is the rule identification, *Comment* is a comment for users and *RuleInfo* is additional information such as a related article, index for the opposing side's case rules, relation to judge's decision and so on. The LHS ($[A_1, A_2, ..., A_i]$) is the context of the opinion, and the RHS ($[B_1, B_2, .., B_k]$) is the conclusion insisted on by one side.

The following is an example of a case rule.

$rule001(\text{"}example\text{"},$
$[article = 218, insisted = prosecutor,$
$result = lost],$
$[drive(drive1, [agent = john/important,$
$object = toyota1/trivial]),$
$person(john, [sex = male/trivial]),$
$person(mary, [sex = female/trivial]),$
$accident(accident1, [agent = john/important]),$
$caused(caused1, [event = accident1/important,$
$effect = injury1/important]),$
$injury(injury1, [agent = mary/trivial])]$
$\rightarrow$
$[responsibility(resp1, [agent = john,$
$object = ken, reason = accident1])]).$

The meaning of this case rule is:"In the case that a traffic accident caused by John injured Mary, John had a responsibility of care to Mary." This rule concerns article 218 of the Penal Code and was insisted on by the prosecutor, but the judge didn't employ this rule. On the LHS, "$effect = injury1$" is an important fact from

the legal point of view. Therefore, this fact is marked as "important". We can use "exact", "important" and "trivial" to represent levels of importance. This information is used to calculate the similarity between two situations.

Arguments in a case are sequences of case rules. As both sides try to draw contradictory conclusions, an old case contains case rules whose conclusions are inconsistent.

3.3 Representation of Concepts

All concepts in legal rules and cases must be contained in the dictionary. In other words, each event and object in a situation are instances of these concepts.

In the dictionary, a *super concept*, a *concept* and a list of *attributes* are defined as follows.

$object(creature, [])$.
$creature(person, [age, sex])$.
$person(person, [])$.
$person(infant, [])$.
$creature(lion, [])$.
$action(drive, [agent, car, destination])$.

The similarity between concepts is defined by the *distance* in the hierarchy (see Fig.2). For example, "*baby*" is closer to "$infant$" than to "*lion*" because it requires two steps for "*baby*" to reach "$infant$" but three steps to reach "*lion*" in this hierarchy.

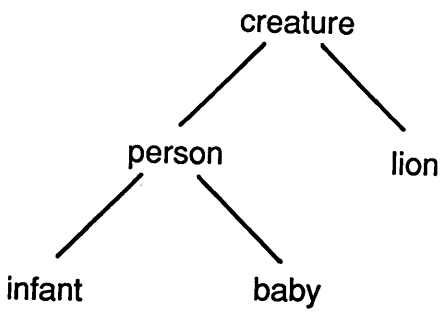


Figure 2: Hierarchy of concepts

4 Reasoning by HELIC-II

In this section, we will explain the reasoning mechanisms of the rule-based engine and the case-based engine. These engines are implemented in the parallel logic programming language KL1 and run on the parallel inference machine.

4.1 A Rule-based Engine

The function of the rule-based engine is to draw all legal consequences by the forward reasoning of legal rules, using original data (a new case) and results from a case-based engine.

The rule-based engine is based on the parallel theorem prover MGTP (Model Generation Theorem Prover)[Fujita *et al.* 1991] developed by ICOT.

MGTP solves range restricted non-Horn problems by generating models. For example, let's take the following clauses.

C1: $true \rightarrow p(a); q(b)$.
C2: $p(X) \rightarrow q(X); r(X)$.
C3: $r(X) \rightarrow s(X)$.
C4: $q(X) \rightarrow false$.

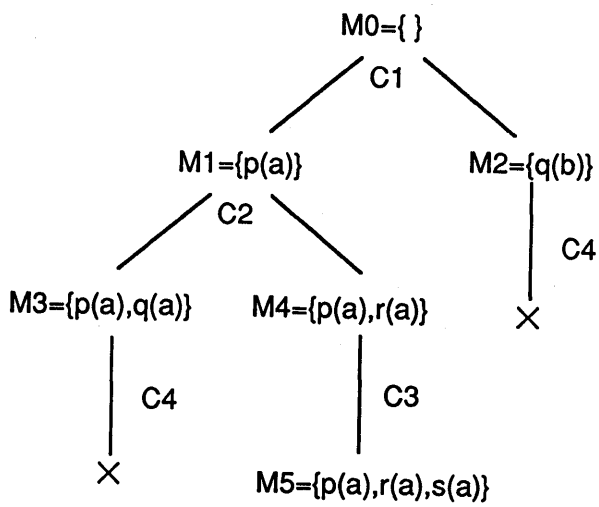


Figure 3: MGTP proof tree

MGTP calculates models which satisfy these clauses as follows (see Fig.3). The proof starts with null model $M0 = \{\phi\}$. By applying C1, $M0$ is extended into $M1 = \{p(a)\}$ and $M2 = \{q(b)\}$. Then, by applying C2, $M1$ is extended into $M3 = \{p(a), q(a)\}$ and $M4 = \{p(a), r(a)\}$. Using C4, $M3$ and $M2$ are discarded. By C3, $M4$ is extended to $M5 = \{p(a), r(a), s(a)\}$. $M5$ is a model which satisfies all clauses.

In MGTP, each clause is compiled into a KL1 clause, and each KL1 clause is applied in parallel on the parallel inference machine. In the problem in which the proof tree has many branches, parallel inference performance becomes high.

To use MGTP as a rule-based engine of HELIC-II, we extended the original MGTP as follows.

1. **Realization of "not (negation as failure)":** We made MGTP able to treat "negation as failure" based on [Inoue *et al.* 1991]. For example, the following C is treated as C', and the model is extended in two ways (see Fig.4). Here, "k" is a modal operator, and "k(r(X))" means that the model is believed to contain a datum which will satisfy r(X) in the future.

   C: $not(r(X)) \rightarrow s(X)$.
   C': $dom(X) \rightarrow k(r(X)); \sim k(r(X)), s(X)$.

   After MGTP generates models which satisfy all clauses, the rule-based engine examines each of them. For example, if a model contains both $\sim k(r(a))$ and $r(a)$, or if a model contains $k(r(a))$ and doesn't contain $r(a)$, the model is discarded.

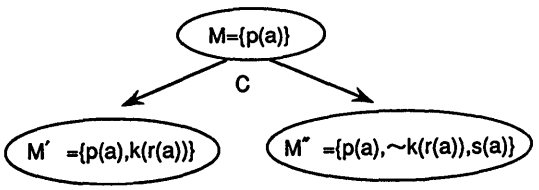


Figure 4: Negation as failure of MGTP

2. **Realization of the multiple context:** The rule-based engine uses both original facts (a new case) and results from the case-based engine as the initial model. The case-based engine may generate data which conflicts with each other such as "$q(b)$" and "$\sim q(b)$". Therefore, before reasoning, the rule-based engine has to split the initial model into several ones so that each model doesn't contain any conflicts (see Fig. 5).

   However, the case-based engine has not generated all results when the rule-based engine begins to reason because the reasoning of both engines is data driven. To obtain the pipeline effect, we developed a function to register predicates which may cause conflicts, and to split the model when such predicates reach the rule-based engine. For example, in Fig.5, if $\sim q(b)$ reaches the rule-based engine, the model is split before $q(b)$ is reached. We implemented this mechanism by using a similar modal operator as the "k-operator".

3. **Keeping justification:** To construct inference trees, the rule-based engine must keep the justifications for each consequence. A justification consists of a rule name and data which matches the LHS of the rule.

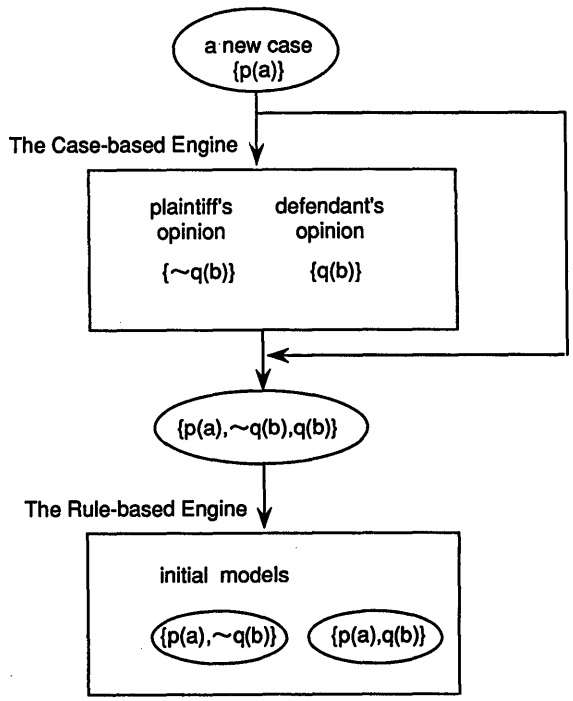


Figure 5: Splitting a model

4. **Temporal reasoning:** We prepared a small rule set of temporal reasoning [Allen 1984] to help in describing the temporal relation. The following are example rules.

   $before(A,B), before(B,C) \rightarrow before(A,C)$.
   $meets(A,B), overlaps(C,B) \rightarrow$
   $overlaps(A,C); during(A,C); starts(A,C)$.

With these extensions, the rule-based engine has many proof tree branches even if clauses don't have the disjunction such as C1 and C2 in Fig.3. Therefore, the rule-based engine has a lot of parallelisms in its reasoning.

4.2 A Case-based Engine

The function of the case-based engine is to generate legal concepts by using similar old cases. The reasoning of the case-based engine consists of two stages (see Fig.6).

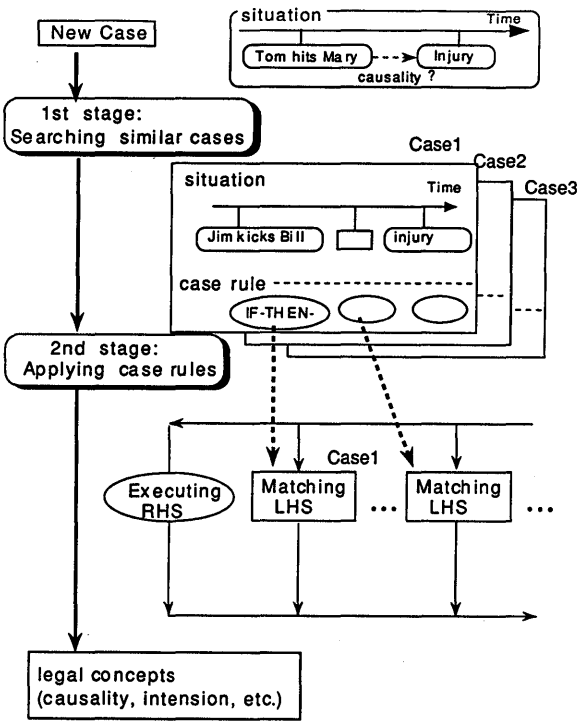


Figure 6: Reasoning by the case-based engine

1. **Searching similar cases:**

   The role of the first stage is to search for similar cases from the case base. At first, the case-based engine constructs a *sequence of events* for each case. As the situations of the new case and old cases are described as a set of events/objects and their temporal relations, it is easy to construct a *sequence of events* for each situation.

   Then, the case-based engine tries to extract common subsequences from event sequences of the new case and each old case. For example, let's take the following two sequences.

   $$\begin{aligned} \text{S1: } & [..., meets(strike1, injury1), \\ & \quad during(runAway1, injury1), ..] \\ \text{S2: } & [..., before(kick2, sneak2), ..] \end{aligned}$$

   In this example, the temporal relation between "strike1" and "runAway1" is the same as that of "kick2" and "sneak2". Furthermore, "strike1" and "kick2" have a common upper concept "violence", and "runAway1" and "sneak2" have a common upper concept "escape" in the dictionary. Therefore, we regard [strike1, runAway1] and [kick2, sneak2] as mapped subsequences of S1 and S2 (see Fig.7).

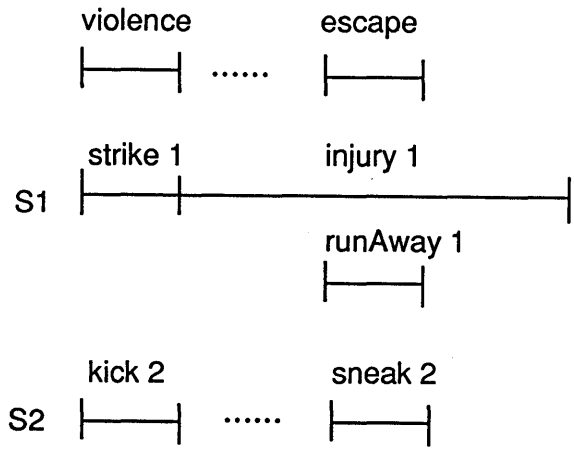


Figure 7: Subsequence of events

The similarity between two cases is evaluated by the length of the longest mapped subsequence. Several cases whose similarities are beyond a threshold are selected in the first stage.

2. **Applying case rules:**

   The role of the second stage is to apply the case rules of selected cases as follows [Branting 1989].

   At first, the similarity between the LHS of a case rule and a new case is evaluated. For example, let's take "rule001" in section 3.2 and the following new case.

   $$\begin{aligned} & person(bill, []). \\ & baby(jane, []). \\ & cycle(cycle2, [agent = bill, object = honda2]). \\ & collision(collision2, [agent = bill]). \\ & sprain(sprain2, [agent = jane]). \\ & intention(intention2, [goal = injury2]). \\ & injury(injury2, [agent = jane]). \end{aligned}$$

   The engine tries to map the LHS of "rule001" to a new case. As the following pairs of event/object have common upper concepts in the dictionary, we map these pairs (see Fig.8).

john	$\leftrightarrow$	bill
mary	$\leftrightarrow$	jane
drive1	$\leftrightarrow$	cycle2
toyota1	$\leftrightarrow$	honda2

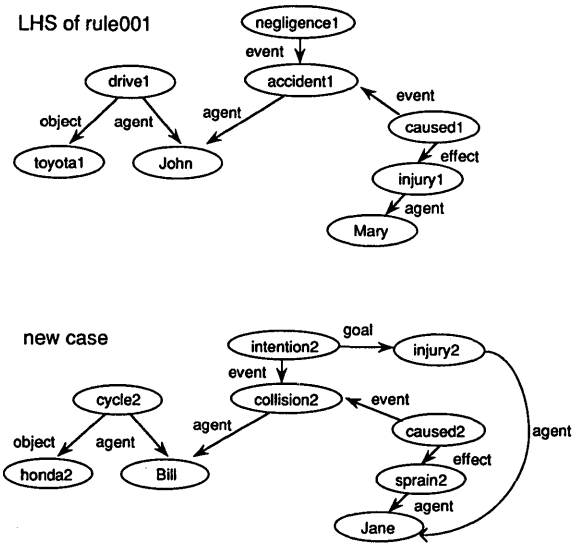


Figure 8: Mapping networks

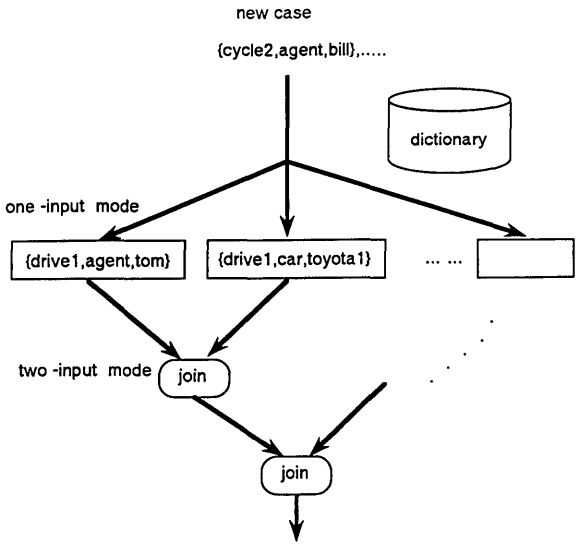


Figure 9: Rete-like networks of KL1 processes

accident1	$\leftrightarrow$	collision2
injury1	$\leftrightarrow$	sprain2
caused1	$\leftrightarrow$	caused2

The similarity is evaluated by counting the number of mapped links in Fig.8. As we explained in section 3.2, an annotation (exact, important, trivial) is attached to each link in the network. These annotations and the distances between concepts are used as weights to evaluate similarities. Even if some conditions of a case rule are not satisfied, but the important conditions are satisfied, then the LHS may be judged as similar to the new case. For example, in Fig.8, though there is no node which can be mapped to "negligence1", "rule001" may be selected as similar.

Next, the case-based engine selects case rules whose LHSes are similar to the new case, and executes their RHSes.

The matching and executing case rules are repeated until there are no case rules left to be fired.

On the parallel inference machine, each stage is executed in parallel. In the first stage, before searching, cases are distributed to processors (PEs) of the parallel inference machine, and then a new case is sent to each PE. Each PE evaluates similarities between the new case and old cases, and selects similar ones.

In the second stage, case rules are distributed to PEs, and the LHSes of each case rule are compiled into a Rete-like network of KL1 processes (see Fig.9). Then, triplets ($\{object, attribute, value\}$) which are facts of the new case are distributed to each PE as tokens. To realize matching based on similarity, each one-input node refers to the dictionary of concepts, and each two-input node not only examines the consistency of pairs of tokens but evaluates their similarities with the LHS.

5 A legal reasoning system for the Penal Code

We developed an experimental legal reasoning system for the Penal Code.

In the Penal Code, general provisions and definitions of crimes are given as legal rules. Though they seem to be strictly defined, the existence of *criminal intention* and *causality* between one's action and its result often becomes the most difficult issue in the court. The concept of *causality* in the legal domain is similar to the concept of *responsibility* and is different from physical causality. Therefore, to judge the existence of causality, we have to take into account various things such as social, political and medical aspect.

We show the function of the reasoning system of the Penal Code using Mary's case. We selected this case

from the qualification examination for lawyers in Japan.

Mary's Case:

On a cold winter's day, Mary abandoned her son Tom on the street because she was very poor. Tom was just 4 months old. Jim found Tom crying on the street and started to drive Tom by car to the police station. However, Jim caused an accident on the way to the police. Tom was injured. Jim thought that Tom had died of the accident and left Tom on the street. Tom froze to death.

The problem is to decide the crimes of Mary and Jim. The hard issues of this case are the following.

1. **Causality between Mary's action and Tom's death:**

   If Mary hadn't abandoned Tom, Tom wouldn't have died. Moreover, the reason for his death wasn't injury but freezing. Therefore, some lawyers will judge the existence of causality and insist she should be punished for the crime of "*abandonment by person responsible resulting in death*". On the other hand, other lawyers will deny any causality because causality was interrupted by Jim's action.

2. **Causality between Jim's action and Tom's death:**

   Jim did several actions such as "pick up", "drive", "cause accident" and "leave Tom". Among them, "cause accident" will be punished by the crime of "*injury by negligence in the performance of work*", and "leave Tom" will be punished by the crime of "*death by negligence*". Moreover, if there is causality between "cause accident" and Tom's death, Jim will be punished by the crime of "death by negligence in the performance of work" which is very grave. As the main reason of Tom's death is freezing, it is difficult to judge the causality.

Though the Penal Code has no definite rule for the causality, lawyers can get hints from old cases. For example, let's take Jane's case which was handled by the Supreme Court in Japan.

Jane's Case:

Jane strangled Dick to kill him. Though Dick only lost consciousness, Jane thought he was dead. Then, she took him to the seashore, and left him there. He inhaled sand and suffocated to death.

In the court, there were arguments between the prosecutor and Jane. The prosecutor insisted Jane should be punished by the crime of *homicide* because of the following reasons.

P1: "Strangling" and "taking to the seashore" should be considered the one action of performing the homicide. Therefore, it is evident that there was an intention to kill Dick and causality between her action and Dick's death.

P2: There is causality between "strangling" and "Dick's death" even though "strangling" wasn't the main reason for his death.

On the contrary, Jane insisted her actions didn't satisfy the condition of the crime of homicide because of the following reason.

J1: "Strangling" should be punished be the crime of "*attempted homicide*, and "taking to the seashore" should be punished by the crime of "*manslaughter caused by negligence*" because there isn't causality between strangling and Dick's death, and there wasn't an intention to kill him when taking him to the seashore.

We represent Mary's situation and Jane's case rule as follows.

Mary's situation

$problem(\text{"}mary's\ case\text{"}, \text{"}example\text{"}, ...).$
$abandon(aba1, [agent = mary, object = tom]).$
$pickup(pic2, [agent = jim, object = tom]).$
.....
$trafficAccident(acc1, [agent = jim]).$
.....

Jane's opinion

$rule002(\text{"}Jane's\ case\text{"},$
$[article = 218, insisted = defendant,$
$result = lost],$
$[\ suffocate(suf1, [agent = jane/trivial,$
$object = dick/trivial]),$
$intention(int1, [agent = jane/trivial,$
$object = act1/important,$
$goal = death1/important]),$
$death(death1, [agent = dick/trivial]),$

$caused(caused1, [event = act1/important,$
$effect = lost1/important]),$
.....
$\rightarrow$
$[\ \sim caused(caused1, [event = act1,$
$effect = death3])]).$

The case-based engine of HELIC-II generated "$\sim caused(ID, [event = acc1, effect = death9])$" by applying *rule002*.

In Mary's case, HELIC-II generated 12 inference trees. Some of them are based on the prosecutor's opinion and others are based on the defendant's opinion. The root of each tree is a possible crime such as *abandonment by a person responsible resulting in death*, *manslaughter caused by negligence*, etc.. The leaves are the initial data of the new case, and intermediate nodes are consequences by case rules or legal rules (see Fig.10).

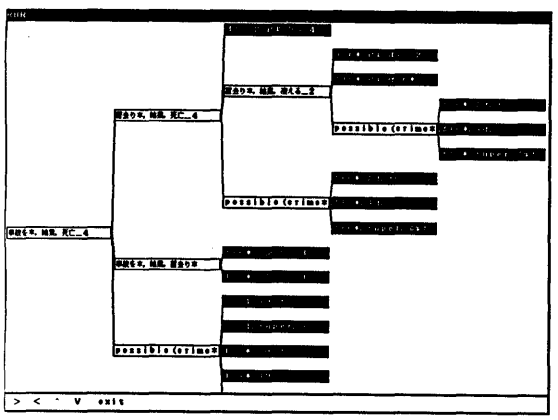


Figure 10: An Inference Tree

We measured the calculation time to draw a conclusion for Mary's case on the experimental parallel inference machine Multi-PSI. The number of rules used was about 20 and the number of cases used was about 30.

Fig.s 11 and 12 show the performance of the case-based engine, and Fig.13 shows the performance of the rule-based engine. These graphs show the effectiveness of the parallel inference.

6 Conclusion

We introduced the parallel legal reasoning system HELIC-II. The advantages of HELIC-II are as follows.

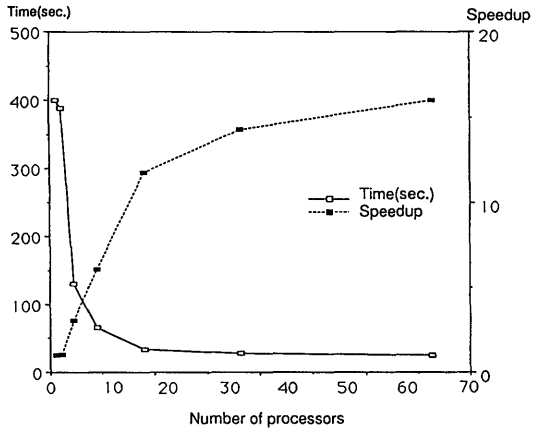


Figure 11: Performance of stage 1 of the case-based engine

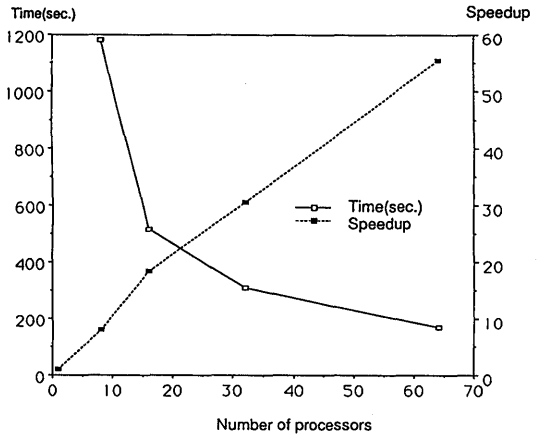


Figure 12: Performance of stage 2 of the case-based engine

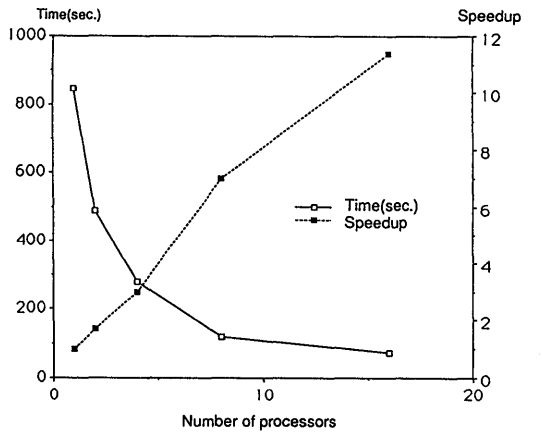


Figure 13: Performance of the rule-based engine

1. The hybrid architecture of HELIC-II is appropriate to realize legal reasoning. As the reasoning of both engines is data-driven, controlling these engines is easier.

2. The knowledge representation and inference mechanisms of HELIC-II are simple but convenient to represent legal rules and old cases.

3. By parallel inference, HELIC-II draws conclusion quickly. As the rule base and the case base of the legal domain are very large, quick searching and quick reasoning are important to develop practical systems.

4. Though it is troublesome to represent cases in detail, the rules of temporal reasoning help to describe cases.

There are many tasks for extending HELIC-II. The following are examples.

- Though the case-based engine is focusing on the similarity between two cases, we have to develop a mechanism to contrast two cases [Rissland *et al.* 1987],[Rissland *et al.* 1989]. By comparing two inference trees, it is possible to construct a debate system.

- To describe legal rules in detail, we have to integrate an extended logic system such as the logic of belief and knowledge with temporal logic on MGTP.

- To improve the power of the similarity based matching of the case-based engine, we have to introduce a derivational analogy mechanism.

- As inference trees are not suitable for allowing lawyers to understand the inference steps, they are represented in natural language.

References

[Uchida *et al.* 1988] Shunichi Uchida et al. . Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988. pp.16-36.

[Goto *et al.* 1988] Atsuhiro Goto et al. . Overview of the Parallel Inference Machine Architecture. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988. pp.208-229.

[Chikayama *et al.* 1988] Takashi Chikayama et al. . Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988. pp.230-251.

[Nitta *et al.* 1991] K. Nitta et al. . Experimental Legal Reasoning System on Parallel Inference Machine. In *Proc. PPAI Workshop of 12th IJCAI*, Sydney, Australia, 1991. pp.139-145.

[Rissland *et al.* 1987] E.L. Rissland et al. . A Case-Based System for Trade Secrets Law. In *Proc. Int. Conf. on Artificial Intelligence and Law*, Boston, USA, 1987. pp.60-66.

[Rissland *et al.* 1989] E.L. Rissland et al. . Interpreting Statutory Predicates. In *Proc. Int. Conf. on Artificial Intelligence and Law*, Vancouver, CANADA, 1989. pp.46-53.

[Sartor 1991] G. Sartor. The structure of Norm Conditions and Nonmonotonic Reasoning in Law. In *Proc. Int. Conf. Artificial Intelligence and Law*, Oxford, UK, 1991. pp.155-164.

[Branting 1989] L.K.Branting. Representing and Reusing Explanations of Legal Precedents. In *Proc. Int. Conf. on Artificial Intelligence and Law*, Vancouver, CANADA, 1989. pp.103-110.

[Sanders 1991(a)] K.Sanders. Representing and reasoning about open-textured predicates. In *Proc. Int. Conf. on Artificial Intelligence and Law*, Oxford, UK, 1991. pp.137-144.

[Sanders 1991(b)] K.Sanders. Planning in an Open-Textured Domain. A Thesis Proposal. Technical Report CS-91-08, Brown University, 1991.

[Fujita *et al.* 1991] H.Fujita et al. . A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm. ICOT TR-606. 1991.

[Inoue *et al.* 1991] K.Inoue et al. . Embedding negation as failure into a model generation theorem prover. ICOT TR-722. 1991.

[Allen 1984] J.F.Allen. Towards a general theory of action and time. *Artificial Intelligence*, Vol. 23, No.2 (1984),pp.123-154.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Chart Parsers as Proof Procedures for Fixed-Mode Logic Programs

David A. Rosenblueth

IIMAS, UNAM
Apdo. 20-726, 01000 Mexico D.F.
drosenbl@unamvm1.bitnet

Abstract

Logic programs resemble context-free grammars. Moreover, Prolog's proof procedure can be viewed as a generalization of a simple top-down parser with backtracking. Just as there are parsers with advantages over that simple one, it may be desirable to develop other proof procedures for logic programs than the one used by Prolog. The similarity between definite clauses and productions suggests looking at parsing to develop such procedures. We show that for an important class of logic programs (fixed-mode logic programs with ground data structures) the conversion of parsers into proof procedures can be straightforward. This allows for proof procedures that construct refutations that Prolog does not find and opens up opportunities for parallelism.

1 Introduction

A logic program consists of clauses that look like the productions of a context-free grammar. This suggests connections between proof procedures and parsers. In fact, Prolog's proof procedure can be regarded as a generalization of a simple parser with backtracking. Although this language has found numerous applications, its execution mechanism has several disadvantages. For instance, if such a mechanism finds an infinite branch of the derivation tree, it enters a nonterminating loop. Thus, it may be desirable to develop new proof procedures for logic programs.

Simple parsers with backtracking also enter nonterminating loops easily. This has motivated the design of other more sophisticated parsing methods. In contrast with proof procedures for logic programs, there already exists a great variety of parsers. The resemblance between definite clauses and productions suggests looking at parsers to develop new proof procedures.

Pereira and Warren [1983] have adapted Earley's [1970] parsing algorithm, but the result is inefficient compared with Prolog. It uses subsumption, which is NP-complete [Garey and Johnson 1979]. We show that by considering a restricted class of logic programs, parsers can be readily adapted to proof procedures. This class is important: it consists of fixed-mode logic programs with ground data structures. Moreover, our proof procedures do not use subsumption and may be more efficient than Pereira and Warren's.

Compositional programs. By using difference lists to represent strings, a logic program can be restricted to coincide with the productions of a context-free grammar. Hence, for this class of logic programs, parsers *are* proof procedures. Such a class, however, only has the expressive power of context-free grammars. Assuming that we are interested in having a programming language, this suggests generalizing such programs without losing the close similarity with grammars. We do so by allowing the body of clauses to denote the composition of arbitrary binary relations; we call such programs "compositional." Prolog programs are not normally written in compositional form. Thus, we consider programs in a larger class (fixed-mode programs with ground data structures) and transform [Rosenblueth 1991] them into compositional form.

Fixed-mode programs. A "mode" for a subgoal is the subset of arguments that are variables at the time the subgoal is selected. Thus, the mode depends on the derivation tree for a program and a query. When we refer to a "fixed-mode logic program," we actually mean a program and a query such that with Prolog's computation rule all subgoals with the same predicate symbol have the same mode. By further restricting these programs to have "ground data structures," we require all arguments in a subgoal that are not variables to be ground terms when the subgoal is selected. This class of program is important because it includes many programs occurring in practice.

At first glance, it seems that the presence of difference lists causes a program to have data structures with variables. However, by separating both components of a difference list it is possible to write some programs using

difference lists as programs with ground data structures. (The usual quicksort program is such an example; the sorted list is then built backwards.)

Overview of the paper. The rest of this paper is organized as follows. Section 2 reviews chart parsers. Section 3 shows that such parsers are also correct for compositional programs. Section 4 deals with a method for converting fixed-mode to compositional programs, thus making chart parsers proof procedures for the former class of programs. Section 5 compares these procedures with Pereira and Warren's. Section 6 concludes this paper with some remarks.

2 Chart parsers

Charts. Chart parsers [Gazdar and Mellish 1989] are methods for parsing strings of context-free languages that can be regarded as a generalization of Earley's algorithm. A *chart* is a set of "partially" applied productions, usually called *edges*. Each edge contains, in addition to the part of a production to be applied and the left-hand side of that production, two pointers to symbols of the string being parsed. The substring between these pointers corresponds to the part of that production that has already been applied.

It is useful to classify edges into those that have not been applied at all: *empty active edges*, those that have already been applied completely: *passive edges*, and all the others: *nonempty active edges*.

The fundamental rule. New edges are created according to the following rule, often called the *fundamental rule*.

If a chart contains:

1. *an active edge (either empty or nonempty) from point* **a** *to point* **b** *in which the next symbol to be applied is Q, and*
2. *a passive edge with left-hand side Q, from point* **b** *to point* **c**,

then create a new edge from **a** *to* **c** *in which the production is the same as the one in the active edge, but with Q applied.* Figure 1 illustrates this rule. In figures representing edges, we use the following notation. Each edge is labeled with an arrow, a symbol to the left of the arrow, and a possibly empty string to the right. The symbol is the left-hand side of the partially applied production. The string is the part of that production that remains to be applied.

Top-down and bottom-up parsing. The fundamental rule takes only existing edges to create new ones, and does not use information from the set of productions.

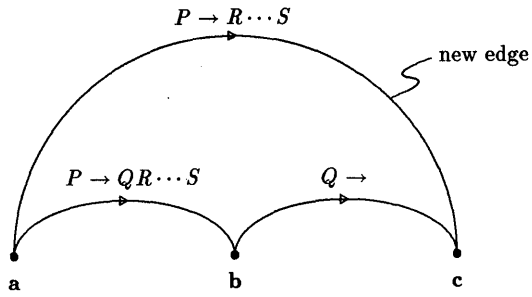


Figure 1: *The fundamental rule.*

Therefore, a mechanism is needed for building edges from productions. Two main mechanisms for this purpose are used, commonly called "top-down" and "bottom-up" rules. The former builds parse trees from the root towards the leaves, and the latter does so from the leaves towards the root.

The *top-down rule* creates edges as follows. *If an active edge from* **a** *to* **b** *is added to the chart, in which the next symbol to be applied is Q, then create one empty active edge from* **b** *to* **b** *for every production having Q as left-hand side and labeled with that production.* Figure 2 exemplifies this rule.

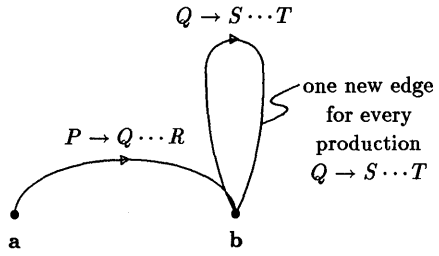


Figure 2: *The top-down rule.*

Given a parse tree having a leaf Q and a node P as parent of Q, this rule allows for Q to be expanded by creating an empty active edge with Q as left-hand side. Hence, parse trees are built by expanding the leaves with nonterminals, which is a construction of parse trees from the root towards the leaves.

The *bottom-up rule* creates edges as follows. *If a passive edge from* **a** *to* **b** *is added to the chart, in which the left-hand side symbol is Q, then create one empty active edge from* **a** *to* **a** *for every production having Q as first symbol on the right-hand side and labeled with that production.* This rule is depicted in Figure 3.

The bottom-up rule takes a passive edge, representing a parse subtree with Q as root. By creating an empty active edge with Q as first symbol to be applied, and P as left-hand side, Q becomes the child of a node P, which is the root of a new subtree. Thus, this rule builds parse trees from the leaves towards the root.

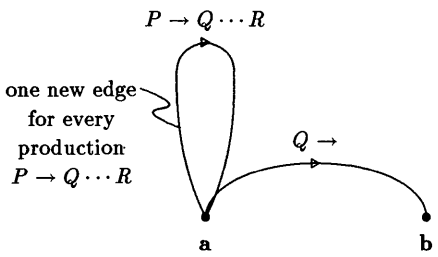


Figure 3: *The bottom-up rule.*

Base of the chart. The fundamental rule takes two edges. One of them is active and the other one passive. The next symbol to be applied in the former must be the left-hand side of the latter. This means that the case where the next symbol to be applied is a terminal is not covered (all left-hand sides of productions are nonterminals). We can remedy this situation by assuming that the productions have been written in such a way that each terminal occurs only in productions with exactly one symbol (that terminal) on the right-hand side. Now we can create certain edges as follows. For each production with a terminal occurring in the string being parsed, we create a passive edge from that terminal to the next one, labeled with that production. We can do so, because an edge represents a partially applied production (where "partially" may mean "completely") and all those productions can be immediately applied. Now we can rely only on the fundamental rule to operate existing edges. We shall call the set of all edges created from terminals the *base* of the chart.

Initialization. To initialize a parser using the bottom-up rule, it suffices to create the base. The reason is that the creation of edges in the bottom-up rule depends only on the existence of a passive edge. In a parser using the top-down rule, however, we must also create empty active edges from the first symbol of the string being parsed to itself labeled with productions having the start symbol of the grammar as left-hand side. This is because such a rule uses an *active* edge to create another one.

Agenda. The rules for producing edges that we have described only *create* edges, but do not *add* them to the chart. Normally, chart parsers store edges in two different data structures: the chart and an *agenda* of the set of edges to be added to it. The choice of the procedure for selecting edges from the agenda to be added to the chart is a degree of freedom relegated to the chart-parser designers. When an edge is removed from the agenda, it is added to the chart only if it has not been added before.

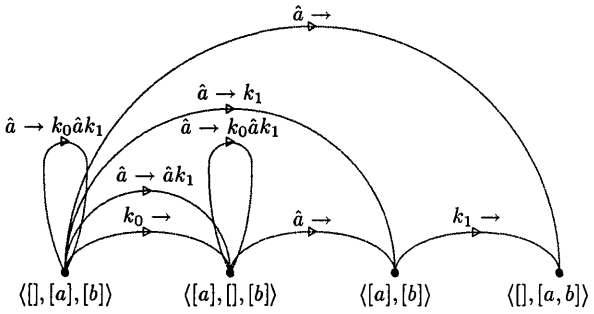


Figure 4: *A chart constructed with the top-down rule.*

Example. Figure 4 shows a chart created by a parser using the top-down rule for the grammar with productions:

$$\hat{a} \to k_0\ \hat{a}\ k_1$$
$$\hat{a} \to \langle [a], [], [b] \rangle$$
$$k_0 \to \langle [], [a], [b] \rangle$$
$$k_1 \to \langle [a], [b] \rangle$$

and the input string $\langle [], [a], [b] \rangle$ $\langle [a], [], [b] \rangle$ $\langle [a], [b] \rangle$ $\langle [], [a, b] \rangle$. Terminals have been enclosed in angled brackets. The last symbol $\langle [], [a, b] \rangle$ is not part of the string itself, but rather an end marker. This example will be used again to illustrate the chart created by a proof procedure when concatenating $[a]$ to $[b]$.

Phillips' variant of the bottom-up parser. Phillips observed [Simpkins and Hancox 1990] that the bottom-up chart parser can be modified so that some edges can be disposed of as the chart is built. The agenda, then, only keeps passive edges, *ordered with respect to the position of the symbol on the string they start from.* The chart only keeps active edges. When the first passive edge E is removed from the agenda and momentarily added to the chart, then

1. the fundamental rule is applied as many times as possible, and

2. the bottom-up rule is also applied if possible, followed by applications of the fundamental rule.

In both cases, if the resulting edges are active, they are added to the chart; otherwise they are added to the agenda. After this, E can be disposed of. The reason is that E cannot contribute to the creation of any more new edges.

3 Chart parsers as proof procedures

In this section we will show that chart parsers can be regarded as proof procedures for compositional programs.

State-oriented computations. The difference-list representation of strings associates a production

$$P_0 \rightarrow P_1 \cdots P_n \tag{1}$$

with a clause of the form

$$p_0(X_0, X_n) \leftarrow p_1(X_0, X_1), \ldots, p_n(X_{n-1}, X_n) \tag{2}$$

and a production with a single terminal on its right-hand side

$$P \rightarrow a \tag{3}$$

with

$$p([a|X], X) \leftarrow \tag{4}$$

With a programming language having only those clauses we cannot compute all computable functions. But if we generalize (4) to

$$p(t, t') \leftarrow \tag{5}$$

where t and t' are terms such that $\mathrm{var}(t') \subseteq \mathrm{var}(t)$, we can. (Throughout, $\mathrm{var}(t)$ denotes the set of variables occurring in term t.) This can be shown, for instance, by associating a logic program with a flowchart in such a way that both have the same set of computations [Clark and van Emden 1981]. A refutation for such a program and a query with a ground term in its first argument may be said to define a sequence of ground terms, resembling the sequence of states in a computation of a programming language using destructive assignment. Thus we shall say that such a logic program defines *state-oriented computations.*

Strings vs. state-oriented computations. There are two main differences between state-oriented computations and strings. One is that at a given point of a state-oriented computation, there may be more than one way to extend it. State-oriented computations are then said to be nondeterministic. This phenomenon does not occur in strings, which have a linear structure.

The other difference is that whereas we do know all the symbols of the string before it is parsed, we do not know initially all the states in a computation. A proof procedure could in principle compute some sequence of states before trying to build a chart. However, it may not be convenient to do so, because not all sequences of states form the base of a chart. A better idea is to extend the computations one step at a time, guided by the part of the chart built so far.

Chart parsers as proof procedures. We shall generalize chart parsers to proof procedures by establishing a correspondence between chart parsing and resolution.

The difference-list representation of languages suggests that clauses of the form (2) should play the role of productions with no terminals on the right-hand side (1). Clauses of the form (5) would then be the counterpart of productions with exactly one terminal on the right-hand side (3).

Given this correspondence, we now turn our attention to edges. The fundamental rule of chart parsing takes two edges and produces another one. Resolution, on the other hand, takes two clauses and produces another one. This suggests identifying edges with clauses and the fundamental rule with a resolution step.

The fundamental rule. If an edge from **a** to **b** labeled with $P_0 \rightarrow P_i \cdots P_n$ corresponds to a clause of the form

$$p_0(\mathbf{a}, X_n) \leftarrow p_i(\mathbf{b}, X_i), \ldots, p_n(X_{n-1}, X_n) \tag{6}$$

then the fundamental rule corresponds to a resolution step having (6) (which plays the role of the active edge) and

$$p_i(\mathbf{b}, \mathbf{c}) \leftarrow$$

(which plays the role of the passive edge) as input clauses. The resolving clause of this resolution step is

$$p_0(\mathbf{a}, X_n) \leftarrow p_{i+1}(\mathbf{c}, X_{i+1}), \ldots, p_n(X_{n-1}, X_n)$$

which corresponds to an edge from **a** to **c** labeled with $P_0 \rightarrow P_{i+1} \cdots P_n$. By correctness of resolution, the resolving clause is a logical consequence of the two input clauses. Thus, we have generalized the fundamental rule to a correct operation.

The top-down and the bottom-up rules. Given the above identification of clauses with edges, the top-down rule for parsing corresponds to the following. Let P be a program in compositional form. If a clause of the form

$$p_0(\mathbf{a}, X_n) \leftarrow p_i(\mathbf{b}, X_i), \ldots, p_n(X_{n-1}, X_n)$$

is added to the chart, then create a clause of the form

$$p_i(\mathbf{b}, X_m) \leftarrow q_1(\mathbf{b}, X_1), \ldots, q_m(X_{m-1}, X_m)$$

for every clause in P of the form

$$p_i(X_0, X_m) \leftarrow q_1(X_0, X_1), \ldots, q_m(X_{m-1}, X_m)$$

The created clause is an instance of a clause in P, which is a logical consequence of P. The bottom-up rule can be generalized in a similar way.

The base. The base can be extended one step at a time as follows. For each clause

$$p_0(\mathbf{a}, X_n) \leftarrow p_i(\mathbf{b}, X_i), \ldots, p_n(X_{n-1}, X_n)$$

that is created, create a clause

$$r(\mathbf{b}, t'\theta) \leftarrow$$

for each clause in P of the form

$$r(t, t') \leftarrow$$

such that $\mathbf{b}$ and t unify with unifier θ and there is a path from p_i to r. There is a *path* from p to r if

1. p is r or
2. there is a clause in P of the form

$$p(X_0, X_m) \leftarrow q(X_0, X_1), \ldots, s(X_{m-1}, X_m)$$

and there is a path from q to r.

4 Conversion of fixed-mode to compositional programs

We have seen that chart parsers can be regarded as proof procedures for compositional programs. However, logic programs are not normally written in compositional form. In this section we observe that it is possible to convert a fixed-mode logic program with ground data structures into compositional form. The resulting program is logically implied by an extension of the original one.

First we define the class of programs transformable by our method and the class produced by it. Then we prove, for a particular example, the correctness of the resulting program. We omit the proof for the general case, which can be found in [Rosenblueth 1991].

4.1 Directed and compositional programs

Directed form. The class of transformable programs has fixed modes. Thus we assume, without loss of generality, that in each predicate, all input arguments have been grouped into one argument, and all output arguments into another one. We write the input argument first, and the output argument second. A definite clause of the form

$$p_0(t_0, t'_n) \leftarrow p_1(t'_0, t_1), p_2(t'_1, t_2), \ldots, p_n(t'_{n-1}, t_n) \quad (n \geq 0)$$

where

1. $\mathrm{var}(t_i) \cap \mathrm{var}(t_j) = \emptyset$, for $i, j = 0, \ldots, n$ and $i \neq j$;
2. $\mathrm{var}(t'_i) \subseteq \mathrm{var}(t_0) \cup \cdots \cup \mathrm{var}(t_i)$, for $i = 0, \ldots, n$;
3. each variable occurring in t'_i occurs only once in t'_i, for $i = 0, \ldots, n$

is a *directed clause.* A *directed program* is a logic program having only directed clauses. Condition 1 causes the term constructed when a subgoal succeeds to have an effect only on the input of other subgoals. Condition 2 causes the input argument of all selected subgoals to be ground if the input of the initial query is also ground and subgoals are selected in a left-to-right order. We include Condition 3 only for technical reasons. This is a minor restriction that considerably simplifies both stating our transformation and proving it correct. We call these "directed programs" because we can visualize the binding of a variable as flowing from one occurrence to subsequent occurrences.

Compositional form. A *compositional clause* is a definite clause of the form

$$p(t, t') \leftarrow \quad \text{or}$$
$$p_0(X_0, X_n) \leftarrow p_1(X_0, X_1), \ldots, p_n(X_{n-1}, X_n) \quad (n > 0)$$

where t and t' are terms such that $\mathrm{var}(t') \subseteq \mathrm{var}(t)$, and the X_i are distinct variables. A logic program with only compositional clauses is a *compositional program.*

We shall need various axioms. As with program clauses, we assume that each axiom is implicitly universally quantified with respect to its variables.

Normally, an SLD-derivation is either successful, failed, or infinite. Sometimes, however, we shall use derivations that end in a clause that could possibly be resolved with a program clause. We shall refer to these derivations as *partial derivations.*

A partial derivation with a single-subgoal initial query yields a *conditional answer* [Vasey 1986]. Such an answer is a clause in which the head is the subgoal in the initial query of that derivation with the composition of the substitutions applied to it, and the body is the set of subgoals in the last query of that derivation.

4.2 Example

We illustrate our method with the following program for concatenating two lists. It defines the usual *append* relation, but its arguments have been grouped in such a way that its two inputs constitute the first argument, and its output, the second. $a(\langle X, Y\rangle, \langle Z\rangle)$ holds if Z is the concatenation of the list Y at the end of the list X. The angled brackets $\langle\ \rangle$ are an alternative notation for ordinary brackets [ ], that we use to group input and output arguments. We do this for clarity.

$$a(\langle[], Y\rangle, \langle Y\rangle) \leftarrow \tag{7}$$
$$a(\underbrace{\langle[W|X], Y\rangle}_{t_0}, \underbrace{\langle[W|Z]\rangle}_{t'_1}) \leftarrow a(\underbrace{\langle X, Y\rangle}_{t'_0}, \underbrace{\langle Z\rangle}_{t_1}) \tag{8}$$

We shall convert (8), which is directed, to compositional form. This process can be motivated as follows.

Assume that we wish to construct an SLD-derivation for (7) and (8) with a query having a ground input that unifies with the head of (8). It is necessary, then, to remember the term with which W unifies, to be able to add it to the front of the result of appending the lists that unify with X and Y. This lack of information in the arguments of the subgoal of (8) prevents us from representing a computation by the composition of the relation denoted by $a(X,Y)$ with itself. To be able to use relational composition for representing computations, we must provide the missing information to the arguments. A common technique in the implementation of state-oriented languages for recording values needed in subsequent steps of a computation is the use of a stack. This suggests storing the term unifying with W in a list that is treated as a stack. We thus define the predicate:

$$\hat{a}(\langle St_0|X\rangle,\langle St_1|Y\rangle) \leftrightarrow St_0 = St_1 \;\&\; a(X,Y) \qquad (9)$$

Although both St_0 and St_1 represent the same stack, it will be convenient to keep two names for this term, so that the input of this new predicate shares no variables with the output. Later we will see why we wish clauses in which the input and the output of their atoms share no variables.

We will also use of the *standard equality theory*. This theory consists of the following axioms:

$$\begin{aligned}
&X = X \leftarrow \\
&X = Y \leftarrow Y = X \\
&X = Z \leftarrow X = Y, Y = Z \\
&f(X_1,\ldots,X_n) = f(Y_1,\ldots,Y_n) \leftarrow \\
&\qquad\qquad X_1 = Y_1,\ldots,X_n = Y_n \\
&p(U,V) \leftarrow U = X, V = Y, p(X,Y)
\end{aligned}$$

which are called, respectively, reflexivity, symmetry, transitivity, function substitutivity, and predicate substitutivity. Note that the last two axioms are actually axiom schemas; an axiom is included for every function and predicate symbol respectively.

Next, we can derive another clause in which the input and the output of the atoms have no variables in common:

$$a(\langle[W|X],Y\rangle,\langle[W'|Z]\rangle) \leftarrow W = W', a(\langle X,Y\rangle,\langle Z\rangle) \qquad (10)$$

This clause can be obtained as a conditional answer, starting from the query $\leftarrow a(U,V)$ and using function substitutivity to disassemble the term $\langle[W|Z]\rangle$, and reflexivity to assemble it with W' instead of W.

Next we can proceed as follows. Unfolding[1] (10) on the "if" part of the definition of $\hat{a}$ (9) we obtain:

$$\hat{a}(\langle St_0,[W_0|X_0],Y_0\rangle,\langle St_1,[W_1|Z_1]\rangle) \leftarrow St_0 = St_1, W_0 = W_1, a(\langle X_0,Y_0\rangle,\langle Z_1\rangle)$$

Next we fold the "iff" version $[U|V] = [U'|V'] \leftrightarrow U = U' \;\&\; V = V'$ of the function substitutivity axiom for the list-constructor function symbol:

$$\hat{a}(\langle St_0,[W_0|X_0],Y_0\rangle,\langle St_1,[W_1|Z_1]\rangle) \leftarrow [W_0|St_0] = [W_1|St_1], a(\langle X_0,Y_0\rangle,\langle Z_1\rangle)$$

and fold the definition of $\hat{a}$:

$$\hat{a}(\langle St_0,[W_0|X_0],Y_0\rangle,\langle St_1,[W_1|Z_1]\rangle) \leftarrow \hat{a}(\langle[W_0|St_0],X_0,Y_0\rangle,\langle[W_1|St_1],Z_1\rangle) \qquad (11)$$

Now the head (W_0) of the first list in the original clause can be thought of as being removed from that list, and pushed onto the stack, then being removed from the stack with another name (W_1) and finally added to the front of the result of appending the tail of the first list to the second.

The fact that in (11) the inputs share no variables with the outputs allows us to fold the definitions of k_0 and k_1:

$$\begin{aligned}
k_0(U,V) \leftrightarrow \exists St_0\exists W_0\exists X_0\exists Y_0.[&\langle St_0,[W_0|X_0],Y_0\rangle = U \\
&\;\&\; \langle[W_0|St_0],X_0,Y_0\rangle = V] \\
k_1(U,V) \leftrightarrow \exists St_1\exists W_1\exists Z_1.[&\langle[W_1|St_1],Z_1\rangle = U \\
&\;\&\; \langle St_1,[W_1|Z_1]\rangle = V]
\end{aligned}$$

in the following clause:

$$\begin{aligned}
\hat{a}(U_0,U_3) \leftarrow\; &\langle St_0,[W_0|X_0],Y_0\rangle = U_0, \\
&\langle[W_0|St_0],X_0,Y_0\rangle = U_1, \\
&\langle[W_1|St_1],Z_1\rangle = U_2, \\
&\langle St_1,[W_1|Z_1]\rangle = U_3, \\
&\hat{a}(U_1,U_2)
\end{aligned}$$

which is a logical consequence of (11) and the standard equality theory. The resulting clause is:

$$\hat{a}(U_0,U_3) \leftarrow k_0(U_0,U_1),\hat{a}(U_1,U_2),k_1(U_2,U_3)$$

Using a result found, for instance, in [Shoenfield 1967 p. 57, 58] we can prove that the fold steps preserve all models of the program.

It may not be practical to transform a program with fold and unfold operations. The compositional form of a directed program may be obtained in a more straightforward manner based on the theorem in the Appendix.

4.3 Example (continued)

The compositional form of the *append* program used to concatenate lists is, then:

$$\begin{aligned}
&\hat{a}(U_0,U_3) \leftarrow k_0(U_0,U_1),\hat{a}(U_1,U_2),k_1(U_2,U_3) \\
&\hat{a}(\langle St,[],Y\rangle,\langle St,Y\rangle) \leftarrow \\
&k_0(\langle St,[W|X],Y\rangle,\langle[W|St],X,Y\rangle) \leftarrow \\
&k_1(\langle[W|St],Z\rangle,\langle St,[W|Z]\rangle) \leftarrow
\end{aligned}$$

[1] In program-transformation terminology, the "unfold" operation is a resolution step. The "fold" operation replaces the subgoals that unify with a conjunction of atoms by a single atom using a definition.

The chart created by a proof procedure using the top-down rule for this program and the query $\leftarrow \hat{a}(\langle [], [a], [b] \rangle, Z)$ was shown in Figure 4.

5 A comparison with Pereira and Warren's Earley deduction

Pereira and Warren [1983] have extended Earley's [1970] algorithm to a proof procedure for logic programs that they call "Earley deduction," and we shall now compare their work with ours. Their proof procedure has the advantage that it can be applied to any logic program.

Two rules produce new clauses; when none can be applied, the process terminates. Since chart parsers are a generalization of Earley's algorithm, we can give such rules using the chart-parsing terminology.

1. If the chart contains a clause C having a selected literal that unifies with a unit clause either in the chart or in the program, then create the resolvent of C with that unit clause. (This rule is the counterpart of the fundamental rule as well as the extension of the base.)

2. If the chart contains a clause having a selected literal that unifies with the head of a nonunit clause C in the program with most general unifier θ, then create the clause $C\theta$. (This rule parallels the top-down rule of chart parsing.)

A new clause is added to the chart only if there is no clause already in the chart that subsumes the new one. Subsumption, however, is NP-complete [Garey and Johnson 1979].

Earley deduction terminates for some programs if subsumption is replaced by a test for syntactic equality. This change results in a proof procedure that can be faster than the original Earley deduction and our methods. Our proof procedures, however, are preferable than this variant of Earley deduction in programs for which our methods terminate but such a variant does not. We now exhibit one such example. Given the directed program

$$p(\langle\rangle, X) \leftarrow p(\langle\rangle, f(X))$$

and a chart initialized with the clause $ans(Y) \leftarrow p(\langle\rangle, Y)$, Earley deduction with a syntactic equality test instead of subsumption produces the infinite sequence

$$p(\langle\rangle, Y) \leftarrow p(\langle\rangle, f(Y))$$
$$p(\langle\rangle, f(Y)) \leftarrow p(\langle\rangle, f(f(Y)))$$
$$\vdots$$

With subsumption, Earley deduction does terminate for this example. Our method, in contrast, does not require subsumption and yet also terminates.

We have implemented Earley deduction based on the top-down chart parser of [Gazdar and Mellish 1989, p. 211, 212]. and using Robinson's [1965] subsumption algorithm as modified in [Gottlob and Leitsch 1985]. We have also adapted both top-down and bottom-up parsers [Gazdar and Mellish 1989, p. 208–212] to proof procedures for compositional programs. In addition, we have modified Phillips' variant of the bottom-up chart parser as presented in [Simpkins and Hancox 1990]. The following table summarizes execution times for several programs and queries. The tests were performed on a SUN SPARC station 1 using SICStus Prolog.

	PW1	top-down		Phillips		PW2	
	time	time	su	time	su	time	su
perm	48	46	1.0	11	4.4	7	6.9
hanoi	36	21	1.7	9	4.0	2	18.0
append	49	22	2.2	5	9.8	6	8.2
qsort	249	30	8.3	7	35.6	17	14.6

"perm" computes all permutations (four elements), "hanoi" solves the Towers of Hanoi problem using difference lists to store the sequence of steps of the solution (five disks), "append" is the ordinary append used to concatenate lists (80 elements), and "qsort" is quicksort using difference lists (20 elements). "PW1" is Pereira and Warren's proof procedure, "top-down" and "Phillips" result from our method, and "PW2" is a variant of Pereira and Warren's proof procedure in which subsumption has been replaced by a syntactic equality test. "su" stands for "speedup." Times are in seconds.

6 Concluding remarks

Chart parsers work for a generalization of the difference-list representation of context-free grammars. This generalization replaces the clauses representing productions with exactly one terminal by clauses having terms subject to only one syntactic restriction: all variables in the second argument must appear in the first (compositional programs).

It is possible to transform [Rosenblueth 1991] fixed-mode logic programs into this generalization by adding arguments that play the role of a stack. Consequently, chart parsers can be used as proof procedures for fixed-mode logic programs transformed by this method. Strings correspond to sequences of ground terms.

Experiments have shown that programs so transformed can be executed several times faster than with the previous adaptation of Earley's parser to a proof procedure done by Pereira and Warren [1983].

Phillips has modified [Simpkins and Hancox 1990] the bottom-up chart parser so that portions of the chart being built can be disposed of. It is essential in the doctored parser to keep edges ordered with respect to the string

being parsed. In compositional programs, computations form sequences and Phillips' idea can also be applied. It is not clear how to apply it to Pereira and Warren's method.

Proof procedures obtained from chart parsers terminate for some programs for which Prolog does not. In addition, it is possible to build charts in parallel [Trehan and Wilk 1988].

Acknowledgments

We are grateful to Felipe Bracho, Carlos Brody, Warren Greiff, Rafael Ramírez, Paul Strooper, and Carlos Velarde. The anonymous referees also made valuable suggestions. We acknowledge the facilities provided by IIMAS, UNAM.

Bibliography

[Clark and van Emden 1981] Keith L. Clark and M.H. van Emden. Consequence verification of flowcharts. *IEEE Transactions on Software Engineering*, SE-7(1):52–60, January 1981.

[Earley 1970] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 14:453–460, 1970.

[Garey and Johnson 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[Gazdar and Mellish 1989] Gerald Gazdar and Chris Mellish. *Natural Language Processing in Prolog. An Introduction to Computational Linguistics.* Addison-Wesley, 1989.

[Gottlob and Leitsch 1985] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the ACM*, 32(2):280–295, 1985.

[Pereira and Warren 1983] Fernando C.N. Pereira and David H.D. Warren. Parsing as deduction. Technical Report 295, SRI, June 1983.

[Robinson 1965] J.A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[Rosenblueth 1991] David A. Rosenblueth. Fixed-mode logic programs as state-oriented programs. Technical Report Preimpreso No. 2, IIMAS, UNAM, 1991.

[Shoenfield 1967] Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[Simpkins and Hancox 1990] Neil K. Simpkins and Peter Hancox. Chart parsing in Prolog. *New Generation Computing*, 8:113–138, 1990.

[Trehan and Wilk 1988] R. Trehan and P.F. Wilk. A parallel chart parser for the committed choice nondeterministic logic languages. In K.A. Bowen and R.A. Kowalski, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 212–232. MIT Press, 1988.

[Vasey 1986] P. Vasey. Qualified answers and their application to transformation. In *Proceedings of the Third International Logic Programming Conference*, pages 425–432. Springer-Verlag Lecture Notes in Computer Science 225, 1986.

Appendix

Our method for converting fixed-mode programs to compositional form is based on the following theorem, which is proved in [Rosenblueth 1991].

Theorem 1 *Let C be a directed clause*

$$p_0(t_0, t'_n) \leftarrow p_1(t'_0, t_1), p_2(t'_1, t_2), \ldots, p_n(t'_{n-1}, t_n)$$

and let

$$\Pi_i = (var(t_0) \cup \cdots \cup var(t_{i-1})) \cap (var(t'_i) \cup \cdots \cup var(t'_n))$$

for $i = 1, \ldots, n$. Then the clause

$$\begin{aligned}\hat{p}_0(X_0, X_{2n+1}) \leftarrow\ & k_0(X_0, X_1), \hat{p}_1(X_1, X_2),\\ & k_1(X_2, X_3), \hat{p}_2(X_3, X_4), \ldots,\\ & \hat{p}_n(X_{2n-1}, X_{2n}), k_n(X_{2n}, X_{2n+1})\end{aligned}$$

is logically implied by C, the standard equality theory, the "iff" version of the function substitutivity axiom for the list-constructor function symbol, and the following axioms:

$$\hat{p}_i(\langle St|X\rangle, \langle St'|Y\rangle) \leftrightarrow St = St' \ \&\ p_i(X, Y) \qquad i = 0, \ldots, n$$

$$k_0(U, V) \leftrightarrow \exists Y_{1,0} \cdots \exists Y_{m_0,0}.[\langle St|t_0\rangle = U \ \&\ \langle \Sigma_1|t'_0\rangle = V]$$

$$k_1(U, V) \leftrightarrow \exists Y_{1,1} \cdots \exists Y_{m_1,1}.[\langle \Sigma_1|t_1\rangle = U \ \&\ \langle \Sigma_2|t'_1\rangle = V]$$

$$\vdots$$

$$k_{n-1}(U, V) \leftrightarrow \exists Y_{1,n-1} \cdots \exists Y_{m_{n-1},n-1}.[\langle \Sigma_{n-1}|t_{n-1}\rangle = U \ \&\ \langle \Sigma_n|t'_{n-1}\rangle = V]$$

$$k_n(U, V) \leftrightarrow \exists Y_{1,n} \cdots \exists Y_{m_n,n}.[\langle \Sigma_n|t_n\rangle = U \ \&\ \langle St|t'_n\rangle = V]$$

where $Y_{1,i}, \ldots, Y_{m_i,i}$ are the variables on the right-hand side of the definition of k_i, except for U and V, for $i = 0, \ldots, n$; Σ_i is any list of the form $[X_{1,i}, \ldots, X_{d_i,i}|St]$, and $\{X_{1,i}, \ldots, X_{d_i,i}\} = \Pi_i$, for $i = 1, \ldots, n$.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Discourse Structure Analyzer for Japanese Text*

K. Sumita, K. Ono, T. Chino, T. Ukita, and S. Amano

Toshiba Corp. R&D Center
Komukai-Toshiba-cho 1, Saiwai-ku, Kawasaki 210, Japan
sumita@isl.rdc.toshiba.co.jp

Abstract

This paper presents a practical procedure for analyzing discourse structures for Japanese text, where the structures are represented by binary trees. In order to construct discourse structures for Japanese argumentative articles, the procedure uses local *thinking-flow* restrictions, *segmentation rules*, and *topic flow* preference. The thinking-flow restrictions restrict the consecutive combination of relationships detected by connective expressions. Whereas the thinking-flow restrictions restrict the discourse structures locally, the segmentation rules constrain them globally, based on rhetorical dependencies between distant sentences. In addition, the topic flow preference, which is the information concerning the linkage of topic expressions and normal noun phrases, chooses preferable structures. Using these restrictions, the procedure can recognize the scope of relationships between blocks of sentences, which no other discourse structure analysis methods can handle. The procedure has been applied to 18 Japanese articles, different from the data used for algorithm development. Results show that this approach is promising for extracting discourse information.

1 Introduction

A computational theory for analyzing linguistic discourse structure and its practical procedure are necessary to develop machine systems dealing with plural sentences; e.g., systems for text summarization and for knowledge extraction from a text corpus.

Hobbs developed a theory in which he arranged three kinds of relationships between sentences from the text coherency viewpoint [Hobbs 1979]. Grosz and Sidner proposed a theory which accounted for interactions between three notions on discourse: linguistic structure, intention, and attention [Grosz and Sidner 1986]. Litman and Allen described a model in which a discourse structure of conversation was built by recognizing a participant's plans [Litman and Allen 1987]. These theories all depend on extra-linguistic knowledge, the accumulation of which presents a problem in the realization of a practical analyzer. The authors aim to build a practical analyzer which dispenses with such extra-linguistic knowledge dependent on topic areas of articles to be analyzed.

Mann and Thompson proposed a linguistic structure of text describing relationships between sentences and their relative importance [Mann and Thompson 1987]. However, no method for extracting the relationships from superficial linguistic expressions was described in their paper. Cohen proposed a framework for analyzing the structure of argumentative discourse [Cohen 1987], yet did not provide a concrete identification procedure for 'evidence' relationships between sentences, where no linguistic clues indicate the relationships. Also, since only relationships between successive sentences were considered, the scope which the relationships cover cannot be analyzed, even if explicit connectives are detected.

This paper discusses a practical procedure for analyzing the discourse structure of Japanese text. The authors present a machine analyzer for extracting such structure, the main component of which is a structure analysis using thinking-flow restrictions for processing of argumentative documents. These restrictions, which examine possible sequences of relationships extracted from connective expressions in sentences, indicate which sentences should be grouped together to define the discourse structure.

2 Discourse structure of Japanese text

2.1 Discourse structure

This paper focuses on analyzing discourse structure, representing relationships between sentences. In text, various rhetorical patterns are used to clarify the principle of argument. Among them, connective expressions, which state inter-sentence relationships, are the most significant. They can be divided into the categories described in Table 1.

Here, connective expressions include not only normal connectives such as "therefore", but also idiomatic

*This work was supported by ICOT (Institute for New Generation Computer Technology), and was carried out as a part of the Fifth Generation Computer Systems research.

expressions stating relations to the other part of text such as "in addition" and "here ... is described." The authors extracted 800 connective expressions from a preliminary analysis of more than 1,000 sentences in several argumentative articles [Ono *et al.* 1989]. Then, connective relationships were classified into 18 categories as shown in Table 1. Using these relationships, linguistic structures of articles are captured.

Table 1 is the current version of the relationship categories. The number of relationship categories necessary and sufficient to represent discourse structures must be determined through further experimentation. New categories will be formed as need becomes apparent; likewise, categories found to overlap in function will be merged. Final categorization can only be fixed after extensive analysis.

Sentences of similar content may be grouped together into a block. Just as each sentence in a block serves specific roles, e.g., "serial", "parallel", and "contrast", each block in text serves a similar function. Thus, the discourse structure must be able to represent hierarchical structures as well as individual relationships between sentences. In this paper, a discourse structure is represented as a binary tree whose terminal nodes are sentences; sub-trees correspond to local blocks of sentences in text.

Figure 1 shows a paragraph from an article titled "a zero-crossing rate which estimates the frequency of a speech signal," where underlined words indicate connective expressions. Figure 2 shows its discourse structure. Extension relationships are set to sentences without any explicit connective expressions. Although the fourth and fifth sentences are clearly the exemplification of the first three sentences, the sixth is not. Thus, the first five can be grouped into a block.

Discourse structure can be represented by a formula. The discourse structure in Figure 2 corresponds to the following formula.

```
[[[1 <EX> [2 <EX> 3]]
     <EG> [4 <EX> 5]] <SR> 6].
```

2.2 Local constraint for consecutive relationships

For analyzing discourse structure, a local constraint on consecutive relationships between blocks of sentences is introduced. The example shown in Figures 1 and 2 suggests that the sequence of connective relationships can limit the accepted discourse structures to those most accurately representative of original argumentative text. Consider the sequence `[P <EG> Q <SR> R]`, where `P`, `Q`, `R` are arbitrary (blocks of) sentences. The premise of `R` is obviously not only `Q` but both `P` and `Q`. Since the argument in `P` and `Q` is considered to close locally, the two should be grouped into a block. This is a local constraint on natural argumentation.

Table 1: Connective relationships.

RELATION	EXAMPLES and EXPLANATION
serial connection <SR>	だから (thus, therefore), よって (then) *dakara* *yotte*
negative connection <NG>	だが (but), しかし (though) *daga* *shikashi*
reason <RS>	なぜなら (because), *nazenara* その訳は (the reason is ...) *sono wake wa*
parallel <PA>	同時に (at the same time), *doujini* さらに (in addition) *sarani*
contrast <CT>	一方 (however), 反面 (on the contrary) *ippou* *hanmen*
exemplification <EG>	例えば (for example), *tatoeba* ... 等である (and so on) *... nado dearu*
repetition <RP>	というのは (in other words), *toiunowa* それは (it is ...) *sore wa*
supplementation <SP>	もちろん (of course) *mochiron*
rephrase <RH>	つまり, すなわち (that is ...) *tsumari* *sunawachi*
summarization <SM>	結局 (after all), まとめると (in sum) *kekkyoku* *matomeruto*
extension <EX>	これは (this is) *kore wa*
definition <DF>	ここで ... とする (... is defined as ...) *koko de ... to suru*
rhetorical question <RQ>	なぜ ... なのだろうか (Why is it ...) *naze ... nanodarouka*
direction <DI>	ここでは ... を述べる *kokode wa ... wo noberu* (here ... is described)
reference <RF>	図Xに ... を述べる (Fig.X shows ...) *zu X ni ... wo noberu*
topic shift <TS>	さて, ところで (well, now) *sate* *tokorode*
background <BG>	従来 (hitherto) *juurai*
enumeration <EN>	第一に (in the first place), *dai 1 ni* 第二に (in the second place) *dai 2 ni*

1 : In the context of discrete-time signals, zero-crossing is said to occur if successive samples have different algebraic signs.
2 : The rate at which zero crossings occur is a simple measure of the frequency content of a signal.
3 : This is particularly true of narrow band signals.
4 : For example, a sinusoidal signal of frequency F0, sampled at a rate Fs, has Fs/F0 samples per cycle of the sine wave.
5 : Each cycle has two zero crossings so that the long-term average rate of zero-crossings is Z = 2F0/Fs
6 : Thus, the average zero-crossing rate gives a reasonable way to estimate the frequency of a sine wave.

Figure 1: Text example 1.

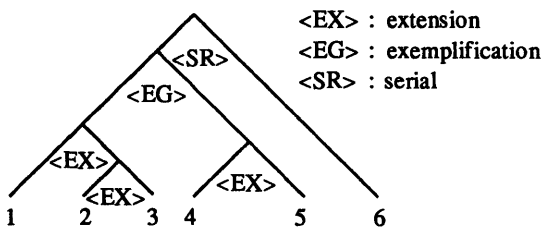


Figure 2: Discourse structure for the text example 1. (This structure can be represented as the form [[[1 <EX> [2 <EX> 3]] <EG> [4 <EX> 5]] <SR> 6].)

Thinking-flow is defined by a sequence of connective relationships and the way in which the sequence fits into the allowable structure. The authors have investigated all 324 (18 × 18) pairs of connective relationships and derived possible local structures for thinking-flow restrictions. The pairs of connective relationships can be represented by (r1, r2), where the relations r1 and r2 are arbitrary connective relationships. They can be classified into the following four major groups.

(1) POP-type : permitting [[P r1 Q] r2 R]
(eliminating [P r1 [Q r2 R]])
ex. [[P <EG> Q] <SR> R],
<EG> : exemplification,
<SR> : serial.

(2) PUSH-type : permitting [P r1 [Q r2 R]]
ex. [P <RS> [Q <SR> R]],
<RS> : reason.

(3) NEUTRAL-type : permitting both (1) and (2)
ex. [[P <PA> Q] <EG> R],
[P <PA> [Q <EG> R]],
<PA> : parallel.

(4) NON-type : permitting non-structure
[P r1 Q r2 R]
ex. [P <PA> Q <PA> R].

The relationship sequence of POP-type means that the local structure for the first two blocks should be popped up, because the local argument is closed. On the other hand, the relationship sequence of PUSH-type means that the local structure should be pushed down.

The relationship sequence of NON-type permits non-structure, which is of the form [P r1 Q r2 R]. Therefore, to be exact, the discourse structure which contains the sequence of this type is not a binary tree.

The thinking-flow restrictions can be used to eliminate structures expressing unnatural argumentative extensions, by examining their local structures. Although the thinking-flow restrictions define local constraints on relationships to neighbors, the scope of relationships is analyzed by recursively checking all local structures of a discourse structure.

2.3 Distant dependencies

The greater part of text can be appropriately analyzed, using the above local constraints on connective relationships to neighbors, if the relationships are extracted correctly. However, in real text, there are rhetorical dependencies concerning distant sentences, which cannot be detected by examining only the normal relationships to neighbors. Two kinds of linguistic clues to distant dependencies must be considered in the realization of a precise discourse analyzer: rhetorical expressions which cover distant sentences, and referential relations of words, in particular, *topics*.

2.3.1 Rhetorical expressions stating global structure

First, rhetorical expressions which relate to an entire article play an important role. Examples are :

"...? ...? The reason is, ...",
"... as follows. ...(TENSE=present).
...(TENSE=present).",
"... is not an exceptional case. ... ...".

Consider the text example in Figure 3, in which unnecessary words are omitted for expositional clarity. In this text the rhetorical expressions which relate to the entire paragraph affect its discourse structure. The expressions "first" and "second" in the last two sentences correspond to the expression "two pieces" in the first sentence; the second and the third sentences, therefore, cab be said to be connected by parallel relationship, as they have similar relations with the first sentence. Thus, the discourse structure in Figure 4 is a natural representation.

While, in real text, there is a wide variety of rhetorical expressions of this type, those that are often used in argumentative articles can be determined through analysis. A robust discourse analysis system must detect these rhetorical expressions to restrict discourse structures.

2.3.2 Topic flow

The other significant phenomenon concerning the distant dependencies is *reference*. While English uses pronouns and definite noun phrases in reference, in Japanese, a phrase that is identical to or a part of the original noun phrase is used when referring to some other part of the text. By analyzing the appearance of the same expressions, a restriction or a preference for building discourse structures can be determined. However, the same expressions tend to scatter in a text, and it is difficult to determine the referent for a reference without task-dependent knowledge [Sumita *et al.* 1991]. The author's aim is to create a system not dependent on such extra-linguistic knowledge; the reappearance of certain expressions is used as a preference for structure determination.

Figure 5 shows a text example in Japanese, where the underlined words are the same expressions. Note that many underlined words are followed by the character " は (*wa*)". This character is a postpositional particle topicalizing the preceding noun in a sentence.

A *topic* of a sentence is an object indicating what the sentence is about; it can localize the reader's attention in the area that the object relates to. In contrast to topic processing for English (cf. [Schank 1977], [Sidner 1983]), we can use a linguistic device to extract topics for Japanese; some postpositional words are said to indicate a topic of a sentence [Nagano 1986].

In this paper, topic information is used for preference judgment of discourse structures, but not as an element of the structures. To simplify explanation, let us denote a topic of the sentence Q by T^{Q}, and a case where T^{Q} refers to a word in the previous sentence P by $T^{Q} \Rightarrow$ P. In the case of the text shown in Figure 5, $T^{2} \Rightarrow$ 1, $T^{3} \Rightarrow$ 2, and $T^{4} \Rightarrow$ 3 hold. If a topic in a sentence refers to a word in the previous sentence, it is regarded as an elaboration of the earlier sentence. Thus, these sentences must be kept close together in their discourse structure; the structure depicted in Figure 6 is appropriate for this text.

In addition, relative importance of relationship connecting sentences in text must be considered for the topic flow analysis. Connective relationships can be classified into three categories according to their relative importance: left-hand, right-hand, and neutral type. For example, the exemplification relationship is a left-hand type; i.e., for [P <EG> Q], P strongly relates to the global flow of argumentation beyond the outside of this block, and in this sense P is more important than Q. In contrast, the serial relationship is a right-hand type, and the parallel relationship is a neutral type.

Consider the structure [[P r1 Q] r2 R], where 'r1' is a left-hand type relationship, and 'r2' can be any relationship. If $T^{R} \Rightarrow$ P, the above structure is natural, even if there is the same word as T^{R} in Q. However, if $T^{R} \Rightarrow$ Q, this structure is unnatural, in the sense of coherency. In this case, the structure [P r1 [Q r2 R]] is preferable to [[P r1 Q] r2 R].

On the contrary, in the case where 'r1' is a right-hand type, [[P r1 Q] r2 R] is a natural structure, even if $T^{R} \Rightarrow$ Q. In short, the naturalness of a discourse structure closely depends on the appearance position of topics and their referents, and the relative importance of the referred nodes.

1 : Two pieces of X are relevant.
2 : First, ...
3 : Second, ...

Figure 3: Text example 2 (X is a noun phrase.)

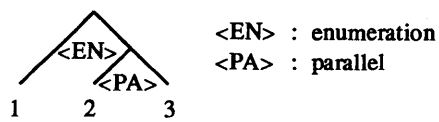


Figure 4: Discourse structure for the text example 2.

1 : AはBとCからなる。
A *wa* B *to* C *kara naru*
A consists of B and C.
2 : Cは ... DとEに分けられる。
C *wa* ... D *to* E *ni wakerareru*
C is divided into D and E.
3 : Dは ... Fを持つ。
D *wa* ... F *wo motsu*
D has ... F.
4 : Fは ... 。
F *wa* ...
F is ... .

Figure 5: Text example 3 (A - F are noun phrases.)

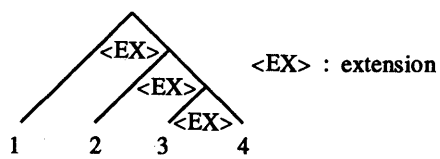


Figure 6: Discourse structure for the text example 3.

3 Discourse structure analyzer

3.1 System configuration

Figure 7 shows the discourse structure analyzer, which consists of five parts: pre-processing, segmentation, candidate generation, candidate reduction and preference judgement. If input text consists of multiple paragraphs or multiple sections, every section or every paragraph in the text is analyzed individually. Figure 8 outlines the input/output data of each stage for a paragraph. The outline of each stage of the discourse structure analyzer is described in the following sections.

3.1.1 Pre-processing

In this stage, input sentences are analyzed, character strings are divided into words, and the dependency structure for each sentence is constructed. The stage consists of the following sub-processes :

(1) Extracting the text of an article from chapters or sections.

(2) Accomplishing morphological and syntactic analysis.

(3) Extracting topic expressions and the reappearance of the targeted expression.

(4) Detecting connective relationships and constructing their sequence.

In Step (1), the title of an article is eliminated, and the body is extracted. Next, in Step (2), sentences in the body of the article, extracted in Step (1), are morphologically and syntactically analyzed. In Step (3), topic expressions are extracted, according to a table of topic denotation expressions. The following are examples of topic expressions.

"... *wa*" (as for ...),
"... *niwa*" (in ...),
"... *dewa*" (in ...),
"... *nioitewa*" (in ...).

In Step (4), a connective expression is detected based on an expression table consisting of a word and its part of speech for individual connective relationships. In this step, *connection sequence*, a sequence of sentence identifiers and connective relationships, is acquired. For example, a connection sequence is of the form

[1 <EN> 2 <EX> 3 <EX> 4 <EN> 5 <SR> 6],

as is shown as the final result in Figure 8.

3.1.2 Segmentation

In this stage, rhetorical expressions between distant sentences, which define discourse structure, are detected. They form restrictions on segmentation of text.

This stage is implemented as a rule-based proce-

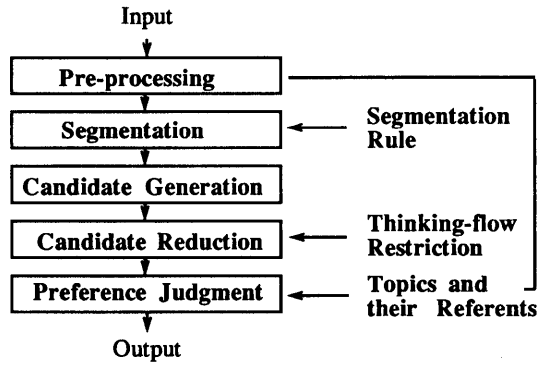


Figure 7: System overview.

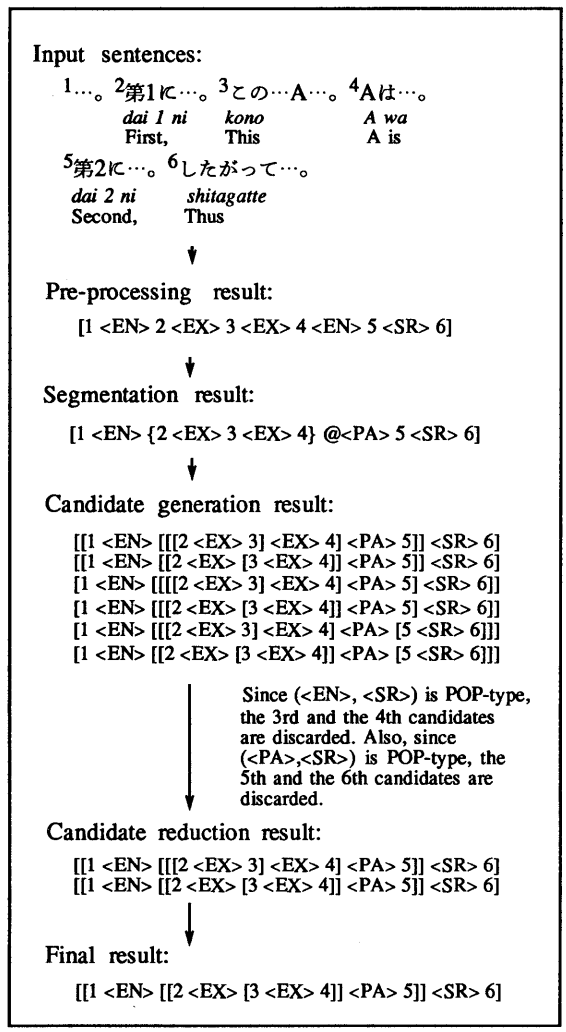


Figure 8: Output example of each process.

dure [Ono *et al.* 1991]. *If-then* rules, called *segmentation rules*, have been formulated in advance. The *if*-part of a segmentation rule corresponds to linguistic surface patterns to detect inter-sentence rhetorical expressions, e.g. "as follows. ... First ... ... Second ...". The *then*-part represents a connection sequence embedded with control operators discussed below. Also, the *then*-part can indicate an exchange of connective relationships.

There are three kinds of control operators. They are '{' and '}', '(' and ')', and '@'. Sentences enclosed by '{' and '}' must be grouped together as a block of sentences. Operators '(' and ')' are similar to '{' and '}'. They can be used singly, while the operators '{' and '}' must be used in pairs. The operator '@' means that the position must not be a boundary of a sentence block.

Figure 9 shows examples of the segmentation rules. The first example means that if the Nth sentence includes expression "*tashikani*" (of course), and the Mth sentence includes expression "*shikashi*" (though), then from N+1st to M-1st sentences must be grouped together.

For the input sentences and the connection sequence in Figure 8, the second rule is activated. The connection sequence is then converted into

```
[1 <EN> {2 <EX> 3 <EX> 4}@<PA> 5 <SR> 6].
```

This structure directs the next stage to generate discourse structure candidates whose second, third and fourth sentences are grouped into a block.

At present, approximately 100 rules are available in the system.

3.1.3 Candidate generation

All possible discourse structures, described by binary-trees which do not violate segmentation restrictions, are generated as discourse structure candidates. The generation is performed in a bottom-up manner of sentence parsing by the CYK algorithm. After the generation of sub-trees for blocks directed by segmentation restrictions, the whole trees are generated based on these sub-trees. In case of the example in Figure 8, only 6 candidates are generated, while 42 binary trees would be produced without the segmentation rules.

3.1.4 Candidate reduction

Local structures of generated structure candidates are checked by inspecting thinking-flow restrictions. The candidates including a local structure violating the restrictions are discarded. Only legal candidates are passed on to the next stage.

In order to show the effectiveness of the thinking-flow restrictions, consider the following connection sequence.

```
[1 <EX> 2 <EG> 3 <PA> 4 <SR> 5].
```

Figure 10 shows discourse structure candidates for the above sequence. There are 14 binary tree possibilities.

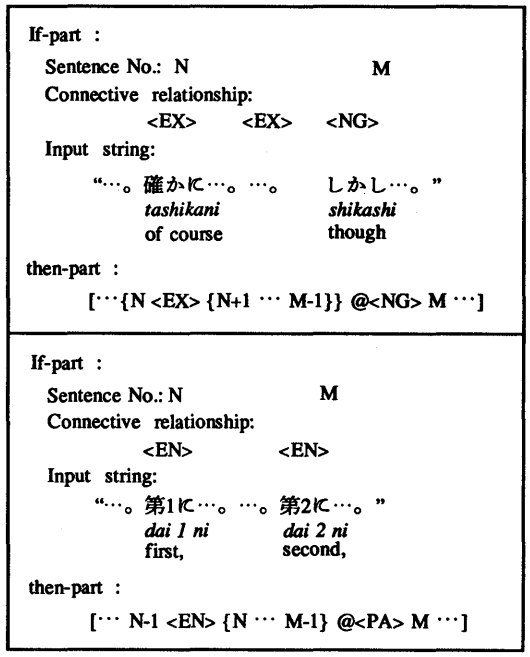


Figure 9: Segmentation rule example.

The candidates violating the thinking-flow restrictions are eliminated. For example, the first structure is discarded, because it contains the local structure [2 <EG> [3 <PA> 4]], and the pair (<EG>,<PA>) is POP-type. For the same reason, the seventh structure is also eliminated. This local structure would also be discarded after the exemplification relationship ("<EG> [[3 <PA>"). As a result of elimination through thinking-flow restrictions, 11 candidates can be discarded, and the second, the fourth and the tenth structures remain.

In the above example, in the case outlined in Figure 8, structure candidates unnatural from the viewpoint of thinking-flow are discarded. Since the third through sixth candidates violate thinking-flow restrictions, the candidates are reduced to two structures.

The thinking-flow restrictions are represented in the system as a table of the applicable pairs of consecutive relationships and their acceptable local structures.

3.1.5 Preference judgment

The final result of discourse analysis is the structure with the lowest *penalty score*, a value associated with topic-referent relationships.

A penalty is set against each arc of path on a discourse structure, which leads from a sentence containing a topic to a sentence referred to by the topic. The concrete arc of a discourse structure, on which a penalty is imposed, is either an arc to or from an unimportant node or an arc to an equally important node. For example, for the structure [[P <EG> Q] <EX> R] where $T^R \Rightarrow$ Q,

1: [[1 <EX> [2 <EG> [3 <PA> 4]]] <SR> 5] : NG
((<EG>,<PA>):POP, "<EG> [3 <PA>" :NG)

2: [[1 <EX> [[2 <EG> 3] <PA> 4]] <SR> 5]

3: [[[1 <EX> 2] <EG> [3 <PA> 4]] <SR> 5] : NG
((<EG>,<PA>):POP, "<EG> [3 <PA>" :NG)

4: [[[1 <EX> [2 <EG> 3]] <PA> 4] <SR> 5]

5: [[[[1 <EX> 2] <EG> 3] <PA> 4] <SR> 5] : NG
((<EX>,<EG>):PUSH,"<EX> 2] <EG>" :NG)

6: [1 <EX> [2 <EG> [3 <PA> [4 <SR> 5]]]] : NG
((<PA>,<SR>):POP, "<PA> [4 <SR>" :NG)

7: [1 <EX> [2 <EG> [[3 <PA> 4] <SR> 5]]] : NG
((<EG>,<PA>):POP, "<EG> [[3 <PA>":NG)

8: [1 <EX> [[2 <EG> 3] <PA> [4 <SR> 5]]] : NG
((<PA>,<SR>):POP, "<PA> [4 <SR>" :NG)

9: [1 <EX> [[2 <EG> [3 <PA> 4]] <SR> 5]] : NG
((<EG>,<PA>):POP, "<EG> [3 <PA>" :NG)

10: [1 <EX> [[[2 <EG> 3] <PA> 4] <SR> 5]]

11: [[1 <EX> 2] <EG> [3 <PA> [4 <SR> 5]]] : NG
((<PA>,<SR>):POP, "<PA> [4 <SR>":NG)

12: [[1 <EX> 2] <EG> [[3 <PA> 4] <SR> 5]] : NG
((<EG>,<PA>):POP, "<EG> [[3 <PA>":NG)

13: [[1 <EX> [2 <EG> 3]] <PA> [4 <SR> 5]] : NG
((<PA>,<SR>):POP, "<PA> [4 <SR>" :NG)

14: [[[1 <EX> 2] <EG> 3] <PA> [4 <SR> 5]] : NG
((<PA>,<SR>):POP, "<PA> [4 <SR>" :NG)

Figure 10: Discourse structure candidates.

a penalty is imposed on the arc from the parent node of P and Q to Q because the left node in an exemplification relationship is unimportant.

The penalty of a discourse structure is defined as a sum of penalties for all paths concerning all topics in the paragraph. By selecting the structure candidate with the lowest penalty, the most coherent discourse structure is obtained.

Of the two surviving structures of the candidate reduction process in Figure 8, the second structure is preferable. The difference is the structural relationship between the second and fourth sentences: the local structure for the first candidate is [[2 <EX> 3] <EX> 4], and that for the second candidate is [2 <EX> [3 <EX> 4]]. Since $T^4 \Rightarrow 3$, a penalty is imposed on the first structure, but not on the second structure. As a result, the second structure candidate is chosen.

While every paragraph can be analyzed respectively, a chapter or a section containing multiple paragraphs is analyzed in an analysis manner similar to that of a paragraph. In case of a discourse structure for a chapter or a section, paragraphs rather than sentences are used as the terminal nodes of the structure. The connective relationship expressed in the first sentence of each paragraph is used for making the connection sequence. After structure candidates are generated based on the connection sequence, candidates unnatural from the viewpoint of thinking-flow are eliminated. Since every paragraph is analyzed into a discourse structure, each node of the discourse structure for a section also forms the discourse structure for the corresponding paragraph.

3.2 Experiment

To evaluate the discourse structure analyzer, 18 journal articles, different from the data used for algorithm development or rule extraction, have been analyzed. The journal used is "Toshiba Review", which publishes short technical papers of three or four pages. An experiment has been carried out on every paragraph. Correct discourse structure for every paragraph was made manually in advance. The system's performance was evaluated by comparing the correct human-produced structures and the structures analyzed by the system,

Table 2 shows analysis results. There are a total of 554 paragraphs. Nearly 50% of them consist of only one sentence and are excluded from consideration. For 114 paragraphs consisting of more than three sentences, a correct analysis was produced for approximately seventy-four percent.

There were 15 errors for all of the processed paragraphs. Most of the errors are due to incorrect detection of relationships (60%), or incorrect candidate reduction (27%). For the former, the procedure failed to detect explicit connective expressions because of insufficient dictionary data, which can be improved by refining the dictionary data. Most of the latter type of errors occur in a paragraph in which the first or last sentence refers to information outside of the paragraph by such phrases as "as shown above" or "as follows." This suggests that the procedure should also take into account relationships to

Table 2: Analysis results

paragraph size (number of sentences)	correct* (unique)	correct* (other candidate)	incorrect*	Total*
1	-	-	-	293
2	-	-	-	147
3	53	8	6	67
4	12	5	7	24
5	7	1	2	10
6	3	0	0	3
7	5	0	0	6
8	2	0	0	2
9	2	0	0	2
Total	84	14	15	114+(554)

* Numbers indicate counts of paragraphs, except for the paragraph size.
\+ Total number of paragraphs consisting of more than 3 sentences.

neighboring paragraphs.

In the segmentation stage segmentation rules were activated for 35 paragraphs, with 85% of the rules correctly used; 65% have contributed to structure determination for itemized parts of text, and 20% to relationship determination. In addition, the preference judgment stage has increased the accuracy of the analysis by 3%. Except for the effects of these contributions, correct relationships have been detected in 73 paragraphs, and correct results have been obtained for 55 paragraphs. Thus, if correct connective relationships are detected, 73% of discourse structures can be appropriately analyzed using thinking-flow restrictions only.

4 Concluding remarks

A practical analyzer has been described for building discourse structures for Japanese argumentative or explanatory articles. To analyze structures, three types of knowledge are used: thinking-flow restrictions, segmentation rules, and topic-flow preference. They represent relative constraints between connective relationships or structural restrictions spanning a paragraph, as opposed to the relative importance between consecutive sentences on which other discourse structure analysis researchers depend. Using linguistic knowledge, global structures or the scope of relationships can be determined appropriately.

In addition, the above knowledge on which the procedure is based is detected from superficial linguistic clues independent of topic areas in analyzed articles. The authors are convinced that the method is effective for any articles whose aim is persuasion or assertion.

It should be noted that the relative importance of sentences can be evaluated, using the extracted discourse structure. For example, a left-hand node of a structure linked by exemplification relationship is more important than the right-hand node, as discussed in Section 2.3.2. By a recursive application of relative importance judgment from the top node of discourse structure analyzed from a paragraph, the key-sentence in the paragraph can be extracted.

In addition to the key-sentence extraction shown above, the extracted structure can be a promising clue to other various natural language processes, such as topic estimation and knowledge extraction. The authors intend to polish up the presented restrictions and rules, and refine the procedure toward these natural language processes.

References

[Cohen 1987] Cohen, R.: "Analyzing the Structure of Argumentative Discourse", *Computational Linguistics*, Vol.13, 1987, pp.11–24.

[Grosz and Sidner 1986] Grosz, B.J. and Sidner, C.L.: "Attention, Intentions and the Structure of Discourse", *Computational Linguistics*, Vol.12, 1986, pp.175–204.

[Hobbs 1979] Hobbs, J.R.: "Coherence and Coreference", *Cognitive Science*, Vol.3, 1979, pp.67–90.

[Litman and Allen 1987] Litman, D.J. and Allen, J.F.: "A Plan Recognition Model for Subdialogues in Conversations", *Cognitive Science*, Vol.11, 1987, pp.163–200.

[Mann and Thompson 1987] Mann, W.C. and Thompson, S.A.: "Rhetorical Structure Theory: A Framework for the Analysis of Texts", *USC/Information Science Institute Research Report* RR–87–190, 1987.

[Nagano 1986] Nagano, K.: *Bunshouron Sousetsu — Bunpouron-teki Kousatsu—* (An Introduction to Theory of Texts —Syntactic Consideration—), Asakusa Shoten, 1986, (in Japanese).

[Ono *et al.* 1989] Ono, K., Ukita, T., and Amano, S.: "An Analysis of Rhetorical Structure", *IPS Japan Technical Report NL* 70–2, 1989, (in Japanese).

[Ono *et al.* 1991] Ono, K., Sumita, K., Ukita, T., and Amano, S.: "Text Segmentation and Discourse Analysis", *Proc. IPS Japan '91* October, 4E–2, 1991, (in Japanese).

[Schank 1977] Schank, R.C.: "Rules and Topics in Conversation", *Cognitive Science*, Vol.1, 1977, pp.421–441.

[Sidner 1983] Sidner, C.L.: "Focusing in Comprehension of Definite Anaphora", M.Brady and R.C.Berwick (Eds.), *Computational Models of Discourse*, MIT Press, 1983, pp.267–330.

[Sumita *et al.* 1991] Sumita, K., Ukita, T., and Amano, S.: "Disambiguation in Natural Language Interpretation Based on Amount of Information", *IEICE Trans.*, Vol.E74, No.6, 1991, pp.1735–1746.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Dynamics of Symbol Systems — An Integrated Architecture of Cognition —

HASIDA, Kôiti
Institute for New Generation Computer Technology (ICOT)*

Abstract

To account for the diversity and partiality of information processing in the cognitive process, we need a design method for cognitive system without explicit stipulation of domain/task dependent information flow, together with a control scheme for partial information processing which does not commit us to global and crisp consistency or completeness.

A computational architecture is proposed which consists of a first-order logic program with a dynamics. Information flow is controlled not by any domain/task dependent procedures but by a control scheme emergent from the dynamics. The declarative semantics of the logic program is defined by formulating the degree of violation in terms of *potential energy*, and a control scheme for both analog and symbolic inferences is derived from an energy minimization principle. This inborn integration of the control scheme with the declarative semantics guarantees a natural reflection of semantic relevance in inferences. Ideas underlying inference mechanisms developed so far, such as weighted abduction and marker passing, are captured in terms of such a dynamics.

1 Introduction

It is practically impossible to delimit the information of the world potentially relevant to the benefit (typically, survival) of a cognitive agent, whereas the information-processing capacity of the cognitive agent is severely restricted. Here arises *partiality of information*: the information potentially relevant to the determination of a cognitive agent's action (including information processing) is only partially reflected in its actual behavior.

Only very relevant information must hence be selectively reflected in the behavior of the cognitive agent. However, the distribution of relevant information, together with the degree of relevance, drastically changes depending on the context. Since only a very small part of the potentially relevant information is exploited at each context, dramatically different parts of the information must be exploited at different contexts, in order for the whole information the cognitive agent uses in various contexts to encompass as much of the relevant information as possible.

This causes very diverse patterns of information flow, underlying the complex behavior of a cognitive agent. So cognition is complex, not entirely because the design of the cognitive agent itself is complex, but rather because it is situated in a complex world, which provides the diverse contexts of the cognitive agent's behavior. The cognitive agent is complex indeed, but still is far simpler than the behavior of the agent reflecting also the vastness of the world.

To capture this situatedness and relative simplicity of a cognitive agent, the design of the cognitive system should largely abstract away the directions of information flow (the temporal order of actions, among others). The models which stipulate the directions of information flow (that is, *procedural* programming) quickly become untractably complex, attributing too much of the complexity of cognitive process to the complexity of the cognitive system itself, and thus failing to capture the situatedness of cognition. For instance, production systems (Anderson 1983) fail to serve as the functional architecture of cognition. This is where *constraint* paradigm comes in. Constraint abstracts the direction of information flow away from the design of a cognitive model, keeping the model within tractable complexity, attributing most of the complexity to the world, and thus capturing the situatedness of cognition.

So the domain-dependent aspects of cognition (language, vision, etc.) should be designed basically in terms of declarative semantics rather than operational semantics. Symbolic logic is a typical formalism for declarative design. Some sort of logic at least as powerful as first-order predicate calculus is considered necessary to design a cognitive system capable of combinatorial behaviors such as language use.

However, such a powerful formalism commits us to untractable computation for maintaining global consistency, exhaustive examination of the possible hypotheses, and so on. This applies to whatever logics have ever been fabricated, including non-monotonic logic, probabilistic logic, fuzzy logic, paraconsistent logic, and so forth. There has been no formalism of logic which could support useful inferences under arbitrary sort of violation of the constraint in question. The problem here is

*From April of 1992, the author is at Natural Language Section, Electrotechnical Laboratory, 1-1-4 Umezono, Tukuba, Ibaraki 305 JAPAN.

essentially that symbolic logics provide no control over inferences other than closure operation (exhaustive inference).

We need a declarative formalism which inherently supports partial and hence tractable computation, while approximately preserving the first-order expressive power and supporting diverse flow of information. To be useful at all, that computation must be about only very relevant information, which will lead to a diverse information flow sensitive to the context.

To implement all this, the present paper considers a system of constraint represented as a first-order logic program, and postulates a *dynamics* of this constraint. The degree of violation is captured in terms of *potential energy*, which is a real-valued function of the state of the constraint. The constraint is thus provided with a fuzzy declarative semantics which is finer-grained than the usual crisp semantics. An operational semantics is also derived from the dynamics. That is, control schemes for analog and symbolic inferences are obtained on the basis of energy minimization principle. Such an inborn integration of declarative semantics and inference method not only supports concise design but also guarantees natural reflection of semantic relevance in inferences.

The rest of the paper proceeds as follows. In the next section we outline the combinatorial structure of the constraint. Section 3 provides a declarative semantics for this constraint. The components of the declarative semantics are each formulated in terms of potential energy. Section 4 discusses the field of force induced from the potential energy, and analog information processing driven by this field of force. It will be shown that associative inferences naturally emerge out of the dynamics. Section 5 defines a method of symbolic inference which is a sort of program transformation, and derives a control scheme for it on the basis of energy minimization principle. The proposed framework is pointed out to capture the ideas underlying some inference mechanisms tailored so far, such as weighted abduction (Hobbs et al. 1990, Stickel 1989) and marker passing (Charniak 1986, Norvig 1989). Section 6 concludes the paper.

2 Constraint Network

A constraint consists of *clauses*. A clause is a set of *literals*, and roughly means their disjunction, which is inclusive or exclusive to various degrees depending of their dynamical properties as discussed later. A *literal* is an *atomic constraint* preceded by a sign. An atomic constraint is an *atomic formula* such as p(X,Y,Z) or an *equation* such as X=Y. Signs are '+' and '−' and stand for affirmation and negation, respectively. '+' is omitted in cases discussed below. Names beginning with capital letters represent variables, and the other names predicates.[1]

[1]A binding is also regarded as an atomic formula. For example, X=f(Y) is an atomic formula with binary predicate =f.

A clause is written as a sequence of the included literals followed by a period. The order among literals is not significant. So (1) and (2) represent the same clause, which means (3) in a rough, crisp approximation.

(1) −p(U,Y) +q(Z) −U=f(X) −X=Z.

(2) +q(X) −p(f(X),Y).

(3) $\forall U, X, Y \{\neg p(U,Y) \vee q(X) \vee U \neq f(X)\}$

A clause containing a literal with empty sign is called a *definition clause* of the predicate of that literal. The meaning of such a predicate is defined in terms of completion based on its definition clauses. For instance, if the definition clauses of predicate p are those in (4), then p is defined as in (5).

(4) p(X) −q(X,a). p(f(X)) −r(X).

(5) $\forall A\{p(A) \Leftrightarrow \{\exists Y(q(A,Y) \wedge Y = a) \vee \exists X(A = f(X) \wedge r(X))\}\}$

A definition clause of a zero-ary predicate true is called a *top clause*. A top clause corresponds to the query in Prolog. That is, top clause (6) represents top-level hypothesis (7).[2]

(6) true −p(X) +q(X,Y).

(7) $\exists X, Y\{p(X) \wedge \neg q(X,Y)\}$

We postulate clause +true. to give rise to such a top-level hypothesis. The computation is to tailor the best hypothesis to explain a top-level one.

A constraint is regarded as a network. For instance, the following constraint may be graphically shown as in Figure 1.

(i) +true −p(A) −q(B).

(ii) +p(X) −r(X,Y) −p(Y).

(iii) +r(X,Y) −q(X).

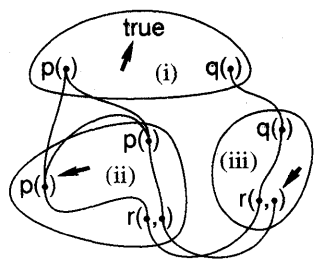


Figure 1: Constraint Network.

[2]Theoretically, Prolog uses false instead of true here so that the negation of the top clause amounts to the top-level hypothesis. In our formulation, a top clause itself directly means a top-level hypothesis.

In such a graphical representation, a clause is a closed domain containing the atomic constraints constituting that clause. Short thick arrows indicate references to the atomic constraints as positive literals in clauses. Atomic constraints without such indication are negative literals. An argument of an atomic formula is shown either as a '•' or as an identifier. Equations between arguments are links. Equations in clauses are called *intraclausal equations*, and those outside of clauses are called *extraclausal equations*.

We will write $\alpha \circ \beta$ to mean that atomic formulas α and β are unifiable. We regard each part of constraint network as a set of instances, and $\alpha \circ \beta$ as meaning that whether $I(\alpha) \cap I(\beta) = \emptyset$ or not is unknown. I is an interpretation function which maps those instances to objects (state of affairs, in the case of atomic formulas) in the world. So unifiability is not transitive. We assume two atomic formulas are unifiable if and only if their corresponding arguments are directly connected through an extraclausal equation, and that every extraclausal equation connects two corresponding arguments of two unifiable atomic formulas.[3] For each zero-ary predicate, the constraint network contains only one atomic formula with it.

3 Declarative Semantics

Now we move on to dynamics to define a declarative semantics for the constraint network described above.

Each atomic constraint α has an *activation value* x_α, which is a real number such that $0 < x_\alpha < 1$ and may be regarded as the truth value (or a subjective probability of the truth) of α. The *potential energy* of a constraint network is a function of the activation values, and represents the degree of violation of the constraint. The potential energy U of the entire constraint is the sum of the potential energy of the parts of the constraint.

The declarative semantics of the entire constraint is decomposed into several aspects. U is a sum of terms each representing one such aspect, so that U captures the whole declarative semantics. Each term of U is the degree of violation of the aspect of declarative semantics in question. These aspects are enumerated below and each formulated by a term of potential energy.

Normalization of activation value. In order to normalize the activation value of an atomic constraint α so that $0 < x_\alpha < 1$, let us employ a standard sigmoid function $\mathrm{sg}(x) = \frac{1}{1+\exp(x)}$ and postulate $x_\alpha = \mathrm{sg}(-F_\alpha/T)$ holds at equilibria of force, where F_α stands for the total force to α from outside of α, and T is a positive constant called the *temperature*. This amounts to assuming the following energy inherent in α.

[3]There can hence be $O(N^2)$ extraclausal equations, for N different atomic formulas sharing the same predicate. So an efficient encoding schema would be necessary to avoid that space complexity. We skip further details of this issue.

$$T\{x_\alpha \log x_\alpha + \overline{x_\alpha} \log \overline{x_\alpha}\} \tag{8}$$

Let us call this the *normalization energy* of α. For any v, $\overline{v}$ stands for $1 - v$. Here and henceforth, mathematical details are not very important; they are quite tentative indeed. The formulas are mainly motivated by convenience. In fact, the fancy outlook of (8) is for the computational ease of analog inference, though we do not go into details here.

Disjunction of literals in a clause. The *disjunction energy* of a clause implements the ordinary disjunctive meaning of the clause. For instance, consider the following clause.

(9) $\neg \mathsf{p} + \mathsf{q}$.

The ordinary disjunctive meaning of this clause is that $\neg \mathsf{p}$ or q is true. The disjunction energy of this clause as follows captures this meaning.

$$D x_{\mathsf{p}} \overline{x_{\mathsf{q}}} \tag{10}$$

D is a positive constant associated with clause (9). (10) is small iff either x_{p} or $\overline{x_{\mathsf{q}}}$ is small; keep in mind that the activation values are between 0 and 1 due to the normalization energy. The semantics of (9) may be depicted like Figure 2. D in (10) represents how large the area b

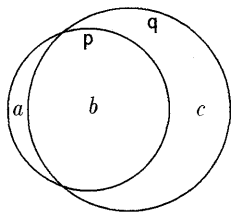


Figure 2: Venn Diagram for (9).

is in comparison with a in this figure.

Mutual exclusion of literals in a clause. By 'mutual exclusion' we mean that at most one literal may be true in a clause. In the case of (9), the mutual exclusion will allow us to abductively assume p when given q, and assume $\neg \mathsf{q}$ when given $\neg \mathsf{p}$. The following term, called the *exclusion energy* of (9), will take care of such inferences.

$$E \overline{x_{\mathsf{p}}} x_{\mathsf{q}} \tag{11}$$

E is a constant associated with clause (9). In Figure 2, E represents how large the area b is in comparison with c. If q means that you are in Japan, for instance, E is larger when p means that you are in Tokyo than when it means that you are in Imabari, a small city in the island of Sikoku.

In the general form, the disjunction energy and the exclusion energy of clause Φ consisting of literals $l_1, \cdots, l_m$ are (12) and (13), respectively.

(12) $D_\Phi \prod_i \overline{r_i y_i}$ (13) $E_\Phi \sum_{i \neq j} r_i y_i r_j y_j$

y_i is the activation value of l_i. For any atomic constraint α, the activation value of literal $+\alpha$ is defined to be x_α and that of $-\alpha$ is defined to be $\overline{x_\alpha}$. r_i is a constant such that $0 < r_i \leq 1$, and is called the *relevance coefficient* of l_i. In the digital approximation, (12) means that at least one literal should be true, whereas (13) means that at most one literal may be true. Incidentally, it is due to exclusion energy that top clause (6) means (7). In (9), (10) and (11), l_1 =−p, l_2 =+q, $y_1 = \overline{x_\mathsf{p}}$, $y_2 = x_\mathsf{q}$, and $r_1 = r_2 = 1$.

Completion of an atomic formula. We somewhat extend the notion of completion so that to complete atomic formula (not predicate) α positively (negatively) means that α ($\neg\alpha$) should be inferred either deductively or abductively[4] on the basis other than the one on which α ($\neg\alpha$) was first postulated. For example, if we have postulated q(X) (say, based on clause +p(X) −q(X)., abductively) and it is positively completed, then it must be inferred from another reason; typically, another atomic formula q(Y) could be closely related with q(X) (in the sense of assimilation to be discussed later) and is inferred on the basis of a clause such as +q(Y) −r(Y). deductively or −q(Y) +s(Y). abductively. As discussed later, completion implements assumability cost (Hobbs et al. 1990).

The positive and negative *completion energy* of an atomic formula α are defined by (14) and (15), respectively.

(14) $C^+_\alpha x_\alpha \prod_{\alpha \circ \beta} \overline{s_{\alpha\beta} x_\beta}$ (15) $C^-_\alpha \overline{x_\alpha} \prod_{\alpha \circ \beta} \overline{s_{\alpha\beta} \overline{x_\beta}}$

C^+_α and C^-_α are positive constants, and are called the *positive completion coefficient* and the *negative completion coefficient* of α, respectively. $s_{\alpha\beta}$ is a constant called the *subsumption coefficient* of α as to β. $s_{\alpha\beta}$ represents how close α is related to β, as seen also in the formulation of assimilation below. We say α *subsumes* β to mean $I(\alpha) \supseteq I(\beta)$. When $\alpha \circ \beta$, $s_{\alpha\beta} = 1$ if α subsumes β, and otherwise $s_{\alpha\beta} = s_0$ for a small positive constant s_0. In the digital approximation, the positive (negative) completion energy means that some β ($\neg\beta$ for some β) satisfying $\beta \circ \alpha$ should be true in order for α ($\neg\alpha$) to be true.[5] Since a subsumption coefficient usually equals to s_0, which is close to 0, completion energy and accordingly other types of energy often decrease if subsumption coefficients increase, which is caused by symbolic operation discussed in the next section.

The dynamics for definition clauses may be defined on the basis of exclusion energy and completion energy, but we do not go further into details here.

Assimilation between atomic formulas. Two unifiable atomic formulas are the same if they have the same arguments for the corresponding argument places. By relaxing this, we obtain the notion of assimilation: two unifiable atomic formulas should have similar truth values to the extent that they share the same assignment of the arguments. So for instance p(X) and p(Y) tend to have similar activation values if X and Y are linked with a strongly activated equation.

To capture this, we postulate *assimilation energy*. Suppose $\alpha \circ \beta$ for two atomic formulas α and β, and let δ be the extraclausal equation connecting their i-th arguments. Then the assimilation energy of δ is defined as follows.

(16) $-A_{\pi i} (s_{\alpha\beta} + s_{\beta\alpha}) x_\delta (x_\alpha - \frac{1}{2})(x_\beta - \frac{1}{2})$

$A_{\pi i}$ is a positive constant called the *assimilation coefficient* of the i-th argument place of the predicate π shared by α and β. The assimilation energy roughly means that x_α and x_β should be similar (both close to 0 or 1) if x_δ is close to 1, and vice versa.

Transitivity of equality. A *transitive cycle* is a cycle of equations $\Delta = \delta_0 \delta_1 \cdots \delta_{k-1}$ where either $\delta_{(i-1) \bmod k}$ or $\delta_{i \bmod k}$ is an intraclausal equation for every i. Note that no cycle of extraclausal equations is a transitive cycle. Transitivity of equality as to Δ is regarded as excluding the cases where just one equation in Δ is false. To capture this, we define the *transitivity energy* U_Δ of Δ as below.

(17) $U_\Delta = \begin{cases} -t \prod_i (e_i - \theta) & (e_i < \theta \text{ for at most one } i) \\ 0 & (\text{otherwise}) \end{cases}$

e_i is the activation value of δ_i, and θ is a constant such that $0 < \theta < 1$. t is a positive constant called the *transitivity coefficient*. Note that the transitivity energy is large when just one equation in Δ has a small activation value.

Since detection of cycles is a very costly computation, we will have to consider some approximate method for efficient processing of transitivity energy instead of guaranteeing perfect detection of transitive cycles. We do not go further into such implementation details.

4 Analog Inference

Potential energy gives rise to a field of force to change the state of the system so as to decrease the total potential energy. Suppose there are n distinct atomic constraints in the given constraint, and hence n activation values, x_1 through x_n. Then the current analog state of the system is regarded as a point (18) in the n-dimensional Euclidean space, and the global potential energy U defines a *field of force* (19).

(18) $\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$ (19) $\vec{F} = \begin{pmatrix} -\frac{\partial U}{\partial x_1} \\ \vdots \\ -\frac{\partial U}{\partial x_n} \end{pmatrix}$

[4] In this respect, only deduction is considered in Prolog.

[5] The completion in Prolog corresponds to our positive completion. In Prolog β must be deduced only.

$\vec{F}$ causes *spreading activation*: when $\vec{F} \neq \vec{0}$, a change of x_i so as to reduce U influences the neighboring parts of the constraint network, which causes further changes of activation values there, and thus state transition propagates across the network. In the long run, the assignment of the activation values will settle upon a stable equilibrium satisfying $\vec{F} = \vec{0}$, under an appropriate scheme of spreading activation. The resulting state gives a minimal value of U.[6] That is, it satisfies the constraint best in some neighborhood.

Let us look at some typical patterns of analog inferences emerging from the dynamics through spreading activation. First, the dynamics gives rise to associative inference based on syntactic similarity. Suppose for instance that, as in Figure 3, the extraclausal equation

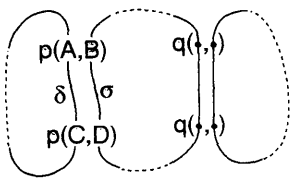


Figure 3: Association due to Syntactic Similarity.

δ connecting argument A of p(A,B) and argument C of p(C,D) is included in a transitive cycle as shown in the figure, and that the activation value of every equation in this cycle is greater than θ. Then δ is excited due to the transitivity energy. This raises the tendency (due to the assimilation energy of δ) for p(A,B) and p(C,D) to have similar activation values. Thus the assimilation energy of the extraclausal equation σ between B and D makes σ to have a high activation value, provided that the equations in the transitive cycle involving σ as shown in the figure are all highly activated. So each equation in a transitive cycle including σ could be excited even stronger due to the transitivity energy. This might make other pairs (such as the two q(•,•)s in Figure 3) of atomic formula with corresponding arguments on that transitive cycle have similar activation values, and so on. In general, two syntactically similar combinations of atomic constraints thus tend to have similar activation patterns, corresponding parts exciting each other or inhibiting each other.

Transitivity energy also enhances semantic association. Consider the following discourse.

(20) Tom took a telescope. He saw a man with it.

We assume that *he* and *it* in the second sentence are anaphoric with *Tom* and *the telescope*, respectively, in the first sentence. There is an attachment ambiguity in the second sentence, about whether the prepositional phrase *with it* modifies *saw* or *a man*. Let us assume that the structure of the constraint generated by processing this discourse looks like Figure 4. Each region

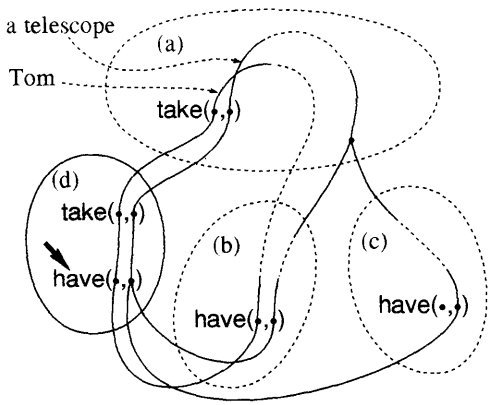


Figure 4: Semantic Association Concerning (20).

in a dashed closed curve represents a cluster of clauses. These clauses have been created by symbolic inference as described in the next section. (a) is a set of clauses including the top clause. (b) and (c) represent two alternative readings of the second sentence of (20), each derived by backward (abductive) inferences. The take(•,•) in (a) is a part of the hypothesis obtained by interpreting the first sentence. Its first argument stands for Tom and the second argument the telescope, so that the whole thing means that Tom takes the telescope at some time. Thus, reading (b) means that Tom has the telescope when he sees the man, and (c) that the man has it when Tom sees him. Clause (d) is an inference rule to the effect that if A takes B then A will have B.[7] Due to this inference rule, the take(•,•) in (a) can imply the have(•,•) in (b) but not that in (c), so (b) is more plausible than (c).

Note that there are two transitive cycles both going through the take(•,•)s in (a) and (d). So these two atomic formulas tend to strongly excite each other due to assimilation energy, provided that every relevant equation is excited. These two cycles also both go through the have(•,•)s in (b) and (d), making them tend to strongly excite each other, too. On the other hand, there is only one transitive cycle which goes through both the have(•,•)s in (c) and (d). Hence the associative inference based on the take(•,•) in (a) through (d) supports the have(•,•) in (b) more strongly than it supports the have(•,•) in (c).

[6] When $\vec{F}$ is not entirely attributed to potential energy, spreading activation is not guaranteed to converge into a stable equilibrium but may exhibit chaotic behaviors. Such a less restricted system may be more powerful and useful, but that is beyond the scope of the present discussion.

[7] We ignore the temporal relation between the taking and the having here.

5 Symbolic Inference

We consider just one type of symbolic operation called *subsumption*. It is a sort of program transformation to create a new subsumption *relation*. A subsumption *operation* concerns a pair of unifiable atomic formulas. As shown in Figure 5, subsumption operation from atomic

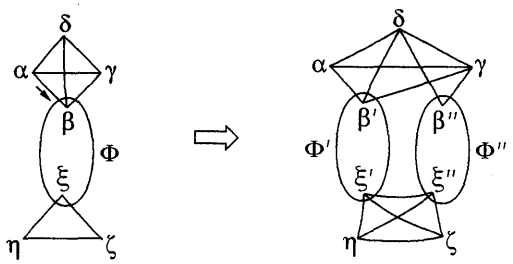


Figure 5: Subsumption Operation From Atomic Formula α to β.

formula α to β divides β into β' and β''. β' is the maximum subset of β subsumed by α, and β'' is the rest of β: $beta'' = \beta - \beta'$. Neither α nor β' is hence unifiable with β'', as indicated in the figure. If it is somehow known that α subsumes β from the beginning, then no copy (division) need to happen. When the division of β actually takes place, then it causes a duplication of the clause containing β, atomic formula ξ accordingly dividing into ξ' and ξ''.[8] Unlike in the division of β, ξ' and ξ'' are unifiable both with each other and with all the atomic formulas unifiable with ξ, because there is no reason to believe $I(\xi') \cap I(\xi'') = \emptyset$.and so on.

We omit further details of combinatorial aspects of symbolic inference, due to the space limitation, and go on to the dynamical aspect. Subsumption generates new atomic constraints and thus redefines U. $s_{\alpha\beta'}$ is set to 1, because α subsumes β'. $s_{\xi'\xi''}$ and $s_{\xi''\xi'}$ are both set to s_0, because we are not sure about the subsumption relation between these atomic formulas. The other coefficients are simply inherited along with the copy of the part of the constraint network.

Since subsumption is a local operation, it may take place in parallel at many different places. Now we consider how to guide such computation based on the dynamics, without recourse to any centralized control. As the preference score for a subsumption, we could use the expected contribution of that subsumption to reduction of U at the equilibrium of spreading activation. As mentioned above, a subsumption from atomic formula α to β divides β into β' and β'', setting $s_{\alpha\beta'}$ to 1. The expected influence of this to reduction of the total energy could be estimated by $-\frac{\partial P}{\partial s_{\alpha\beta}}$, where P is defined to be the minimal U_0 (U_0 under the condition $\vec{F} = \vec{0}$) in a neighborhood of the current $\vec{x}$. U_0 is a representative part of energy whose definition is not changed due to symbolic operations. For instance, it could be the disjunction energy of clause +**true**.. At any rate, the symbolic computation is controlled so as to minimize some part of energy, whereas the analog computation to minimize the whole energy. By employing generalized backpropagation (Pineda 1988), $\frac{\partial P}{\partial s_{\alpha\beta}}$ can be efficiently computed for all $s_{\alpha\beta}$. The space complexity of that computation is linear with regard to the size of the constraint network, and its parallel time complexity practically constant. See APPENDIX for mathematical details.

Our method implements some important features of other inference mechanisms proposed elsewhere. First, weighted abduction (Hobbs et al. 1990, Stickel 1989) emerges from our method. In weighted abduction, just as in the current framework, one attempts to tailor a best hypothesis to explain the observed fact. A hypothesis is a conjunction of (negated) atomic formulas. Each conjunct in a hypothesis is assigned an assumability cost, which is a cost of assuming the conjunct. A hypothesis is better when the total assumability cost is smaller. Assumability cost may be reduced by unifying the conjuncts. For instance, if the current hypothesis contains **p(A)** and **p(B)** one of which has a large cost, then this cost will be reduced by unifying them. Assumability cost is inherited through abduction. For example, a cost of **p(A)** in the current hypothesis is inherited down to **q(A)** and **r(A)** when **p(A)** is resolved by clause +**p(X)** −**q(X)** −**r(X)**..

Assumability cost is basically captured by our completion energy: the conjunct in question must be inferred otherwise than the way it was first postulated, or it would be inhibited due to its completion energy. So an *inherent* cost is encoded by a completion coefficient of atomic formula α. This gives rise to a high preference score of subsumption from α, because if α comes to subsume another atomic formula β then perhaps the completion energy of α is reduced due to $s_{\alpha\beta} = 1$, which will be indicated by a large value of $-\frac{\partial P}{\partial s_{\alpha\beta}}$. An *inherited* cost is captured along the same line. For example, when **p(A)** with a large cost subsumes **p(X)** in clause +**p(X)** −**q(X)** −**r(X)**., the completion energy of **p(A)** is probably still large, but it will decrease if **q(X)** and **r(X)** get more excited. So the preference score of subsumptions from **q(X)** and **r(X)** tend to be large, corresponding to the inherited cost in weighted abduction.

Our framework is more flexible and dynamic than weighted abduction. That is, we allow inferences concerning a hypothesis to influence the state of other hypotheses, whereas in weighted abduction assumability costs change only due to unification involving the atomic formulas carrying those costs. So our method is more appropriate to account for such phenomena as belief revision. In this connection, our dynamical semantics is much more general than the probabilistic semantics of

[8] If α and β belonged to the same clause, then α is also divided into α' and α''. If α' and β' belong to one clause and hence α'' and β'' belong to another, then α' and β'' subsume each other and α'' and β'' are not unifiable.

Charniak and Shimony (1990), which is restricted to propositional Horn logic.

Second, marker passing (Charniak 1986, Norvig 1989) may be also understood as an emergent property of the dynamics, along the same line as above. Consider the following discourse for example.

(21) Taro got a book. He paid one thousand yen.

Figure 6 shows the network involved in the abductive inference to assume that Taro bought the book. In the left is the marker-passing network encoded[9] by the constraint network, which is in the right. A node in marker passing network corresponds to an argument or a predicate in our constraint. An edge between an argument node and a predicate node represents that the argument satisfies the predicate, and an edge between two predicate nodes represent a clause referring to the two predicates. The directions of the arrows are static, and irrelevant to the direction of marker passing. **get(G1)** and **pay(P1)** are created upon reading/hearing (21), where G1 and P1 stand for the event of Taro's getting a book and that of his paying money, respectively.

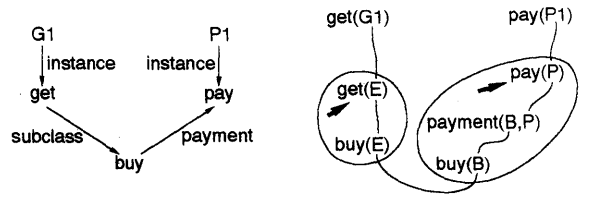


Figure 6: Marker-Passing for (21).

In marker passing, the abductive inference of Taro's buying the book will be suggested by a collision of markers passed down from G1 and P1 along the path between them. In our framework, the same abductive inference consists of three subsumption operations along the (copy of) extraclausal equations in the right of Figure 6. The preference scores of these subsumptions are probably all high, because of the path of clauses between **get(G1)** and **pay(P1)**. If the activation value of **get(G1)** is larger than $\frac{1}{2}$, then it excites **get(E)** due to assimilation energy, **get(E)** excites **buy(E)** due to exclusion energy, **buy(E)** excites **buy(B)** due to assimilation energy, and **buy(B)** excites **pay(P)** due to disjunction energy. **get(E)** is similarly excited indirectly by **pay(P1)**. So **get(E)**, **buy(E)**, **buy(B)** and **pay(P)** are excited stronger than when there were no such path. The subsumptions along the extraclausal equations in the right of Figure 6 are therefore very promising for reduction of positive completion energy, so that the abductive inference mentioned above is suggested.

A path of clauses between two very informative atomic formulas (ones with activation values close to 0 or 1) thus tends to raise the preference for subsumptions along it. This is what marker passing is designed to capture in general. Of course how much the preference for subsumption increases depends on the dynamical properties of the path. For instance, the path in Figure 6 would not indicate the above abductive inference if the exclusion coefficients of the two clauses are small.[10] Suggestion of inference also depends on the length of the path. Obviously, shorter paths more readily suggest inferences.

Subsumptions can also be promoted by associative inferences discussed in the previous section, because a subsumption between two atomic formulas will strongly affect P when some of the extraclausal equations between them are strongly excited owing to transitive cycles involving them. See Hasida (1991) for how generation of natural language sentence is controlled by heuristics regarded as approximating our control scheme taking such associations into account.

6 Concluding Remarks

We have discussed a framework of constraint for designing a cognitive system. To capture the partiality and the corresponding situatedness of cognition, the constraint is situated in a field of force derived from potential energy representing the degree of violation. This field of force gives rise to analog inference as spreading activation, and also controls symbolic computation to transform the constraint. Not only nearly logical inferences and abductive inferences but also associative inferences emerge out of such a dynamics.

A distinguished feature of our framework is that the control scheme for inference is derived from a dynamics which also provides the declarative semantics. In comparison, the other frameworks such as marker passing stipulate the inference control apart from the declarative semantics. The inborn integration of declarative semantics and inference control as in our method will not only provide a clear perspective of the design, but also guarantee emergent reflection of semantic relevance in information processing. In this connection, our method is integrated also in another sense that it controls analog and symbolic inferences based on the same dynamics. This is a strong advantage over the methods such as in Waltz and Pollack (1985) which separate the two inference schemes.

The current framework should be extended with respect to several points. First, some partial processing method is necessary for dealing with transitive cycles, although at any rate a massively parallel computational system is essential to implement our theory. Second, *deletion* should be incorporated in addition to subsumption, in order to prevent the constraint network from unlimited growth. Probably deletion is regarded as a reverse of subsumption, and hence the control of deletion

[9]Charniak (1986) employs a similar encoding scheme.

[10]What Charniak (1986) calls isa-plateau can be understood along the same line.

may be formulated along the same line as that of subsumption. Third, the control method should take into account consistency checking as well. Consistency checking pertaining to binding is discussed in Hasida (1991).[11] In order to handle consistency checking in general, we will have to give preferences not only to subsumptions which seem to decrease P but also to those which seem to increase P. Finally, learning is vitally necessary for both the coefficients (Suttner and Ertel 1990) and the symbolic structure of constraint. Further scrutiny is open with regard to the role of the dynamics in learning.

APPENDIX

The equilibrium condition of the spreading activation concerning $\vec{x}$ is regarded in general as $\vec{x} = \vec{y}$, where $\vec{y}$ is a vector function of $\vec{x}$ and $\vec{S}$.[12] Let s be a parameter in $\vec{S}$. When $\vec{y}$ is differentiable, we get (22) where $\frac{\partial \vec{y}}{\partial \vec{x}}$ is defined by (23).

$$\frac{\partial \vec{x}}{\partial s} = \begin{pmatrix} \frac{\partial x_1}{\partial s} \\ \vdots \\ \frac{\partial x_n}{\partial s} \end{pmatrix} = \frac{\partial \vec{y}}{\partial \vec{x}} \frac{\partial \vec{x}}{\partial s} + \frac{\partial \vec{y}}{\partial s} \tag{22}$$

$$\frac{\partial \vec{y}}{\partial \vec{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_n} \end{pmatrix} \tag{23}$$

(24) follows from (22), where I is the n-dimensional unit matrix.

$$\frac{\partial \vec{x}}{\partial s} = \left(I - \frac{\partial \vec{y}}{\partial \vec{x}} \right)^{-1} \frac{\partial \vec{y}}{\partial s} \tag{24}$$

Let H be a scaler function of $\vec{x}$ and $\vec{S}$, and P be a scalar function of $\vec{S}$ such that $P = H$ when $\vec{x} = \vec{y}$. Where H is differentiable, we get the following.

$$\begin{aligned} \frac{\partial P}{\partial s} = \frac{\partial H}{\partial \vec{x}} \frac{\partial \vec{x}}{\partial s} + \frac{\partial H}{\partial s} &= \frac{\partial H}{\partial \vec{x}} \left(I - \frac{\partial \vec{y}}{\partial \vec{x}} \right)^{-1} \frac{\partial \vec{y}}{\partial s} + \frac{\partial H}{\partial s} \\ &= \vec{z} \frac{\partial \vec{y}}{\partial s} + \frac{\partial H}{\partial s} \end{aligned} \tag{25}$$

Here $\vec{z}$ is defined by (26), from which we obtain (27).

$$\vec{z} = \frac{\partial H}{\partial \vec{x}} \left(I - \frac{\partial \vec{y}}{\partial \vec{x}} \right)^{-1} \tag{26}$$

$$\vec{z} = \vec{z} \frac{\partial \vec{y}}{\partial \vec{x}} + \frac{\partial H}{\partial \vec{x}} \tag{27}$$

Thus, $\vec{z}$ is computed via spreading activation based on (27). So as a whole we are to do double-layered spreading activation, the first layer for $\vec{x}$ and the next for $\vec{z}$. We omit mathematical discussions on the convergence of spreading activation. Finally, $\frac{\partial P}{\partial s}$ can be obtained from (25). We have avoided calculating $\frac{\partial \vec{x}}{\partial s}$, which would be a very complex computation. Note that $\frac{\partial x_i}{\partial s}$ is not zero for most x_i, whereas $\frac{\partial \vec{y}}{\partial \vec{x}}$ and $\frac{\partial \vec{y}}{\partial s}$ are sparse.

[11] Treatment of binding could probably be ascribed to the general case of consistency checking plus transitivity energy.

[12] In the current formulation, $y_i = \frac{1}{1+\exp(Y_i)}$, where Y_i is a polynomial not involving x_i.

Acknowledgments

The author would like to thank DEN Yasuharu, Mark Gawron, Jerry Hobbs, NAGAO Katashi, NAKASHIMA Hideyuki, Manny Rayner, Ivan Sag, TUTIYA Syun, and Rémi Zajac for valuable discussions and comments. He is also indebted to ASOH Hideki for drawing his attention to generalized backpropagation. Special thanks go to NISHIKAWA Noriko, NAGATA Kazumi, and MIYATA Takasi for implementation works.

References

Anderson, J. R. (1983). *The Architecture of Cognition.* Harvard University Press, Cambridge.

Charniak, E. and Shimony, S. E. (1990). Probabilistic semantics for cost based abduction. In *Proceedings of AAAI '90*, pp. 106–111.

Charniak, E. (1986). A neat theory of marker passing. In *Proceedings of AAAI '86*, pp. 584–588.

Hasida, K. (1991). Common heuristics for parsing, generation, and whatever ... In *Proceedings of the Workshop on Reversible Grammar in Natural Language Processing*, Berkeley.

Hobbs, J., Stickel, M., Appelt, D., and Martin, P. (1990). Interpretation as abduction. Tech. rep. Technical Note 499, SRI International.

Norvig, P. (1989). Marker passing as a weak method for text inferencing. *Cognitive Science, 13*, 569–620.

Pineda, F. J. (1988). *Generalization of Backpropagation to Recurrent and Higher Order Neural Networks*, pp. 602–611.

Stickel, M. E. (1989). Rationale and methods for abductive reasoning in natural-language interpretation. In Studer, R. (Ed.), *Proceedings, Natural Language and Logic, International Scientific Symposium*, No. 459 in Lecture Notes in Artificial Intelligence, pp. 233–252. Springer Verlag.

Suttner, C. B. and Ertel, W. (1990). Automatic acquisition of search guiding heuristics. In *Proceedings of the 10th International Conference on Automated Deduction (CADE)*, pp. 470–484.

Waltz, D. and Pollack, J. (1985). Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science, 9*, 51–74.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Mental Ergonomics As Basis For New-Generation Computer Systems

M.H. van Emden
Logic Programming Laboratory, Department of Computer Science
University of Victoria, Victoria, B.C., Canada V8W 3P6
vanemden@csr.uvic.ca

Abstract

Reliance on Artificial Intelligence suggests that Fifth Generation Computer Systems were intended as a *substitute for* thought. The more feasible and useful objective of a computer as an *aid to* thought suggests *mental ergonomics* rather than Artificial Intelligence as the basis for new-generation computer systems. This objective, together with considerations of software technology, suggest logic programming as a unifying principle for a computer aid to thought.

1 Introduction

When surveying the literature on computing, it is remarkably difficult to find work directly aimed at making computers usable as a *tool for thought.* Even when we go to publications specialized in Artificial Intelligence, we find mostly work aiming at simulating or automating human intellectual functions, but very little on how to use computers to *augment* the intellect in the way envisaged by pioneers such as Licklider, Engelbart, Taylor, and Kay.

Until recently it was understandable that the goal of augmenting the intellect had to be deferred, and that top priority had to be given to the development of hardware and systems software that provided a functional basis on which to proceed towards the main goal. I believe that this basis now exists and that therefore the top priority in computing should be to use the existing machinery to make computers available as tools for thought. It seems, however, that at present it is still top priority to make computers faster, bigger, and cheaper. This can only be explained as a form of inertia: we feel comfortable in an enterprise blessed with past and continuing success and it is painful to change emphasis, even if it is towards what is now *really* important.

Let us then take stock and see what progress has been made towards providing the basis for the goal of making a computer into a tool for thought. The hardware dreamed of by the pioneers has arrived: fast processors, large memories, sophisticated and comfortable displays, high-performance networks; all of this available in thousands of enterprises and institutions. Still, we do not use computers as tools for thought in the way the pioneers envisaged. Were they unrealistic in their expectations? Or is it the case that the remaining obstacles can be overcome?

I believe that the latter is the case, and that the remaining barrier is *the difficulty of using software.* What is from the larger viewpoint but a tool among many, such as, for example, a database program, is so complex that it comes with a fat manual and a programming language of its own, so that to become an effective user is almost a career in itself.

This is but one example of many, of a wider phenomenon I refer to as *concept fragmentation*: that each job seems to require its own special-purpose solution; even worse, that the same job in a different context requires a different solution. This, not hardware, is now the main barrier standing in the way of using computers to augment the intellect.

In this paper I will argue that the best bet for a unifying principle to overcome this barrier is *logic programming.* As Artificial Intelligence often comes up in discussions about how to make computers easier to use, it is important to distinguish the roles to be played by Artificial Intelligence and Mental Ergonomics[1]. For this reason, I sketch the argument from scratch: why there is reason to believe that computers can be tools for thought, before going on to explain in what way logic programming, as ergonomic principle, can help demolish the main barrier now holding up progress.

2 A synopsis of the argument

This section is what is sometimes called "executive summary." Each paragraph summarizes one of the following sections of the paper.

[1] Webster's Third International Dictionary: ergonomics — an applied science concerned with the characteristics of people that need to be considered in designing and arranging things that they use in order that people and things interact most effectively and safely.

Towards Computer-Aided Thought. Writing is *paper-aided thought*: while we can do simple sums in our head, we need help for more complex ones; help offered traditionally by writing on paper. Similarly, we can do simple thoughts in our head; to work out complex thoughts, such as plans, proposals, essays, reviews, critiques, we need writing.

Computers are now widely used as a more convenient writing tool than a pen on paper. The availability of programs for spreadsheets, databases, and communications provide a tantalizing glimpse of a more powerful aid to thought than the pen on paper ever was. Such a potent new mixture deserves a name. I chose one suggested by the familiar concepts of Computer-Aided Instruction (CAI) and Computer-Aided Design (CAD). I call it Computer-Aided Thought, CAT for short. Today's laptop computers already pack the hardware required to support a powerful CAT system. Thus you will be able to take it wherever you go, like a Sony *Walkman*. Let us call such a package "CATMAN."

And hardware trends suggest that CATMAN will be widely affordable, giving unprecedented power to intellectual workers of all ages: school children as well as professionals, business persons, and scientists.

Why Computer-Aided Thought is an underdeveloped area. In spite of spectacular advances in computing, both in large systems and in personal computers, no one, not even the most privileged researcher, has a computer available as a congenial tool for intellectual work. At best she[2] can call on a hodge-podge of language processors, databases and application packages requiring a bevy of system gurus at her beck and call if she is to avoid devoting her career to mastering the mechanics of the various systems.

Improvement is a matter of ergonomics, not AI. A congenial tool for intellectual work needs to handle a variety of tasks: including database work, text processing, communication, constraint exploration, developing algorithms. This diversity of tasks has caused a proliferation of languages so that logically identical tasks need to be done in an exasperatingly different way, just because they occur in a different application. The problem is one of *ergonomics* (in this case Mental Ergonomics): the lack of a unifying concept makes current program interfaces conceptually fragmented. This is where we should look in the first place for help, rather than to Artificial Intelligence.

Logic programming meets the requirements of Mental Ergonomics. I mention some ergonomic principles that help to make computers easier to use and review results showing how such principles can be implemented by means of logic programming.

[2]or "he." Here, as elsewhere in my writings, no gender is to be inferred when none is implied by the context.

3 Towards Computer-Aided Thought

What is "thought"; what is "intellect"? Why do I consider writing "paper-aided thought"? The analogy about complex thought spilling over to paper, just as complex sums do, is due to Susan Horton [1982], who took as starting point the familiar phenomenon that we don't sit down to write an essay with its main line of reasoning ready in our head. Instead, we only *discover* what we want to say as a result of initially unsuccessful and often frustrating attempts to write down inchoate, preliminary versions. In this way, Horton concludes, writing allows us to have thoughts too hard to do in our head.

Another way of expressing Horton's idea is to say that writing "augments the intellect." In 1963 Douglas C. Engelbart published the paper "A conceptual framework for the augmentation of man's intellect" [Engelbart 1963]. Its first paragraph, which I quote in full, contains a better description of "aids to thought" or "augmentation of intellect" than I can give.

> By "augmenting man's intellect" we mean increasing the capability of a man to approach a complex problem situation, gain comprehension to suit his particular needs, and to derive solutions to problems. Increased capability in this respect is taken to mean a mixture of the following: that comprehension can be gained more quickly; that better comprehension can be gained; that a useful degree of comprehension can be gained where previously the situation was too complex; that solutions can be produced more quickly; that better solutions can be produced; that solutions can be found where previously the human could find none. And by "complex situations" we include the professional problems of diplomats, executives, social scientists, life scientists, physical scientists, attorneys, designers — whether the problem situation exists for twenty minutes or twenty years. We do not speak of isolated clever tricks that help in particular situations. We refer to a way of life in an integrated domain where hunches, cut-and-try, intangibles, and the human "feel for a situation" usefully coexist with powerful concepts, streamlined terminology and notation, sophisticated methods, and high-powered electronic aids.

Conventional computer applications are much further developed in the area of what Engelbart calls "powerful

concepts, streamlined terminology and notation, sophisticated methods." I regard the area of "hunches, cut-and-try, intangibles and 'human feel for the situation' " the one where writing helps as an aid to thought in the sense of Horton. Engelbart's vision will be realized when a computer can be used as a congenial tool for writing, giving fluent access to spreadsheets, databases (local as well as remote), numerical and statistical libraries and so on.

Other prophetic early papers that improved upon much later thinking are by Bush [1945] and Licklider [1960]. For an overview of Engelbart's subsequent work in the Augmented Knowledge Workshop, see [Engelbart 1988].

4 Why Computer-Aided Thought is an underdeveloped area

A present-day personal computer can provide a word processor, a spelling checker and a thesaurus. This combination is a powerful advance over pen and paper, and therefore qualifies as a correspondingly powerful aid to thought. Personal computers can also run packages for databases, spreadsheets, and computer mail. But although these are potentially valuable extensions, the resulting combination is not easy enough to use to qualify as a computer tool for thought. The existence of the components conjures up a tantalizing vision of such a tool, but the reality of ergonomics turns the vision into a mirage.

To appraise the situation, let us consider what personal computers have given us, and what's lacking.

What we do have. The amazing thing about personal computers is that they have caused such large step in the direction of a computer tool for thought. The early computers were operated in a closed shop, to which users submitted their jobs, which were collected in batches and run.

Timesharing brought a dramatic change: from a turnaround time of hours to instantaneous interaction. Effectively, the timesharing user has the machine to himself. And even in the sixties, these machines were not small compared to contemporary personal computers. So it was not obvious that a dramatic change would result from the next step, the introduction of the personal computer. Yet they made an enormous difference and that was because of their low cost. This has two effects:

1. Small firms and individual software designers can afford machines of their own.

2. The potential financial rewards in the software market became much greater.

The result should convince sceptics of the power of a sufficiently free market: it resulted in an unprecedented improvement in user interfaces.

This is the more amazing when we realize that since the early sixties timesharing computers have been used for word processing. These installations commanded the best in programming talent and were largely devoted to research. Yet nothing was produced that can compare with the better word processing packages that appeared on the market soon after personal computers took off. For most of the 1980's, Unix-based workstations, with more powerful hardware than personal computers, had word processors worse than those on personal computers. Spreadsheets are an even more striking example. This type of software, now considered obvious and indispensable, was not even known before the advent of personal computers.

Even a loaded PC does not come close. Yet, even this progress still falls far short of what is necessary to make a personal computer a congenial tool for thought. Progress has been in the application packages separately, not in ways to integrate.

Consider, as an example of the need for integration, an engineer in his daily activities. He makes calculations, searches tables, standards, textbooks, draft reports, receives and sends mail, retrieves and studies drawings and textual library material, accesses databases (local and remote) and so on. In all these activities separately, computer programs exist that can help. The rapidly falling cost of hardware makes these programs potentially accessible to every engineer. But this is a mixed blessing: if he is to utilize the full potential of all available computer tools, he needs several specialists in attendance, to be available at a few seconds' notice.

Even then he will not be able to use the computer as a truly congenial tool: that is only possible when *no* intermediary is needed. At present he needs an intermediary for many applications because of the complex and idiosyncratic interface provided by the required software. And it does not help that every application package comes with its own, unique programming language, so that two logically identical jobs in different applications need to be done in an exasperatingly different way.

Of course the situation sketched here is not unique to engineers, but is shared by professionals in public or business administration and in scientific research.

What's lacking? A plan for improvement has to be based on a diagnosis of what is wrong. One common diagnosis can be summarized as:

> Computers are difficult to use because *they are not enough like humans.* To make progress we must make them more like humans. Therefore, before we work directly towards a congenial tool

> for mental work, we need progress in Artificial Intelligence.

But another diagnosis is possible:

> Computers are difficult to use because *they are not enough like automobiles*, that is, they are not a tool that one can easily learn to use as an extension of oneself. To make progress towards a congenial tool for mental work, we need to work on the ergonomics of interfaces to software.

The Japanese Fifth Generation Computer System project [Moto-Oka 1982] is based on the first diagnosis. I will argue that to make most rapid progress towards CATMAN we must work on ergonomics rather than on Artificial Intelligence. Moreover, that via ergonomics progress is predictable and will be rapid, as it will be a matter of elaborating existing software technology. In comparison, progress in Artificial Intelligence seems unpredictable: the required results may indeed be around the corner, or it may be a long time before they materialize (if at all).

5 Improvement is a matter of ergonomics, not AI

We saw that what stands in the way of CATMAN can be diagnosed as either a problem in Artificial Intelligence or as a problem in mental ergonomics. There are two episodes from the past that should help in deciding which diagnosis is more fruitful.

In the late 1940's influential administrators perceived an acute shortage of experts available to translate Russian scientific publications into English. In that period there existed considerable optimism about the feasibility of fully automatic high-quality translation, resulting in several well-funded research projects. Lack of progress in the fifties, combined with devastating criticism [Bar-Hillel 1964] of the scientific basis of machine translation, caused funding to be withdrawn.

Let us consider two possible reactions to this failure to get computers to alleviate the shortage of translators.

Reaction 1: The funds should have been spent on Artificial Intelligence. The consensus that emerged in the fifties and caused the demise of projects aiming at fully automatic high-quality translation, was that the text to be translated had to be understood, at least to a certain extent, by the translating agent, human or machine. Machine translation was therefore seen as a problem in Artificial Intelligence. Getting a computer to help in translation was therefore premature — progress in Artificial Intelligence was needed first.

Reaction 2: The funds should have been spent on ergonomics. In the fifties, when the objective was to use computers to help alleviate the shortage in translators, the technology available to translators consisted of a typewriter (electro-mechanical at best) and some well-thumbed reference books. In the early eighties, after machine translation had long been forgotten, and in response to different pressures, there evolved a set of computer tools that have enormously increased the productivity of translators: word processors, spelling checkers, thesauruses, dictionaries, checkers of style and diction. Such software could have been built soon after 1960 when the first time-sharing systems became available.

Thus in 1960, when it was clear that the approach to machine translation taken in the fifties was doomed to failure, it would have been possible to go on to achieve great increases in productivity at low cost. Instead, *it was concluded that the least tractable stage of translation was to remain to receive top priority and that research in Artificial Intelligence was to be motivated in part by the desirability to use computers to increase productivity of translators.*

A lesson to be learned from this episode. There is a similarity between the situation now, in which we suspect that computers can do more to help mental work than is actually the case and the situation in the fifties when it was hoped that computers could help in translation. In the case of translation, the least tractable aspect of the work was selected, leading to Artificial Intelligence. In retrospect, more tractable, even mundane, aspects (namely ergonomics) could have been selected with great success, not only to increase the productivity of translators, but of other office workers as well.

Similarly, when considering how to make a computer into a congenial tool for mental work, there seems to be a great temptation to fall into the same trap: to view Artificial Intelligence as panacea.

To end this section on a positive note, I will conclude with an episode from the past where the right alternative was selected. There was a time when automobiles were difficult to use, for several reasons: for example, because of frequent need for tuning, maintenance, and repair. At that time, a Plutocrat requiring transportation solved this problem by retaining a chauffeur and a mechanic (ideally, but not necessarily, the same person).

When considering obstacles preventing more widely available transportation by automobile, the following diagnoses are possible:

1. build robot chauffeur-mechanics

2. make automobiles easier to use, so that the chauffeuring can be done by the person to be transported and so that only an occasional visit to a garage is required for tuning, maintenance, and repair.

The Japanese FGCS project[3] has selected an alternative in the spirit of the first [Moto-Oka 1982].

6 Logic programming meets the requirements of Mental Ergonomics

In this section I review some of the basic principles of Mental Ergonomics and comment how logic programming can help implement them in CATMAN: *Avoid trying to do two things at a time*, *Allow the user to do the same thing in the same way (if desired)*, *Exploit useful conservatism*, and *Avoid harmful conservatism.*

Avoid trying to do two things at a time. It is bad ergonomics to define a programming language in such a way that the declarative and the imperative aspects of programming are not easy to separate. Rather than to attempt to define these aspects, I will illustrate them by the example of computing an arithmetic expression using register-to-register machine operations.

Two tasks have to be distinguished here:

- To make sure that the correct expression is evaluated (*what* is computed; this is the *declarative* aspect of programming). What the correct expression is, is only determined by the application, independently of the machine on which the computation is to be performed. Thus, this task can also be thought of as that of solving the *application problem.*

- To determine the sequence of register operations and transfers required to get the correct value in the desired location (*how* it is computed; this is the *imperative* aspect of programming). This task contains the above task. What belongs to this task over and above the application problem is to control the machine. To get this additional aspect right is to solve the *control problem.*

The example of arithmetical expressions is useful because every programming language allows these to be evaluated without having to solve a control problem. Thus, every programmer, at the level of assembler language and up, is familiar with declarative programming. The problem with conventional languages is that this type of programming is only possible with arithmetic expressions, on which the programmer spends only a small proportion of his time. Most of the work requires areas of the language where the application and control aspects of the task are intimately intertwined. As a result, it is possible for an error in control to cause a wrong answer. As a result, a programmer in such a language is forced to try to do two things at the same time.

Logic as a programming language allows a decomposition of an algorithm into what Kowalski [1979] calls its *logic* component (corresponding to the declarative aspect) and its *control* component (corresponding to the imperative aspect). A consequence of Kowalski's approach is that the declarative and imperative aspects are separated, so that *an error in control cannot cause an erroneous answer to appear*; at worst it will cause failure to find an answer. The advantage is that there is no need to solve the application and control problems at the same time.

Allow the user to do the same operation in the same way (if desired). In the existing personal computer systems, the closest approximations to CATMAN require the use of separate programming languages for databases, spreadsheets, intensive numerical computation, system programming, document preparation, the shell, and perhaps even other ones. This means that the same operation (such as procedure and data declaration, procedure call, case selection, iteration, and so on) has to be done in a different way in each of these different languages, violating a principle of ergonomics.

It has been shown that logic programming can be the basis of many different types of programming language: functional [Cheng *et al.* 1990], imperative [van Emden 1976, Rosenblueth 1989], object-oriented [Davison 1988], stream-oriented [Taylor 1989], as well as for database querying [Ceri *et al.* 1990]. Of course, within this framework there are still many opportunities for violating the ergonomic principle by undue proliferation of variety. But by using logic as common framework for whatever different languages are needed, improvement is made easier.

Exploit useful conservatism. A *conceptual interface* represents a beneficial kind of conservatism: it is an interface modelled on a familiar concept so that the known operations on the concept can serve as model for the computer operations that need to be learned. The prototypical example of a conceptual interface is the WYSIWYG editor, where the familiar concept is a sheet of paper. Examples of conceptual interfaces that fit well in logic programming are: *the conversational partner, the pocket calculator, spreadsheets and tables*, and *the filling in of blanks.* I elaborate on these below.

Lack of a conversational partner as conceptual interface can lead to the kind of frustration eloquently voiced by John McCarthy, who complained[4] that even to get a computer to acquire a simple symbol manipulation skill

[3] In its original 1982 version [Moto-Oka 1982]. What the project has actually done since then is more sensible. In fact, they have been a prolific contributor to logic programming, my proposed technical basis for CATMAN. But the preoccupation with parallelism and with big machines remains, and this can only be traced back to the initially intended role of Artificial Intelligence.

[4] In debate with Sir James Lighthill on BBC TV in 1974.

is like having to perform *brain surgery*. He explained that the goal of his research is to program computers in such a way that one just needs to *tell* them. That is, to model the interface on that of a conversational partner.

This ideal has been realized to a certain extent by MYCIN [Shortliffe 1976], an early expert system. The user interacts with it in the following way. If the user lacks information, he *asks* a question to which the machine may respond with an answer. If the user knows something that the computer doesn't, he *tells* a fact, or a rule. If the user is puzzled by an answer, then he can *request an explanation*, which comes in the form of facts and rules chaining the facts to the answer. Sergot [1982] and Shapiro [1983], have shown that this conceptual interface finds a natural home in logic programming. They added to the initial version embodied in MYCIN the possibility of making the computer and user play symmetrical roles.

The programming language LISP is an example of a conceptual interface, albeit in an inverted way. The basic interaction mode in LISP does not need to be learned because it is the same as that of a pocket calculator: enter expression to be evaluated, get in return its value. The curious inversion lies in the fact that the familiar concept, the pocket calculator, is of more recent origin than the beneficiary of the conceptual model, namely LISP itself.

In the early days, computers were used in a rigidly planned way. With the advent of time-sharing, users were given the illusion of having a machine of their own, allowing in principle an intimate, interactive, and spontaneous use. Software to exploit this possibility was slow in coming: only with the advent of programs modelled on the spreadsheet as conceptual interface for personal computers has this mode of use been convincingly demonstrated. A similar, but significantly different, interface is that of a *table*, where rows and columns play different roles. Both interfaces have been shown to be compatible with logic programming [van Emden *et al.* 1986, Cheng *et al.* 1988].

Filling in the blanks of a form is a useful, though not widely loved, conceptual interface. It has been exploited in Query-By-Example [Zloof 1977] to provide one of the more congenial query languages for databases. The queries of the logic programming language Prolog are similar [van Emden 1977, Kowalski 1979].

Avoid harmful conservatism. Exploiting a conceptual interface is a useful form of conservatism. Insisting that only English is fit for humans to communicate with our tools is not. The optimism about the utility of natural language for a user-computer interface is based in no small degree on the work of T. Winograd [1972], who himself, however, subsequently made the following observation [Winograd and Flores 1987]:

> The practicality of limited natural language systems is still an open question. Since the nature of the queries is limited by the formal structure of the data base, it may well be more efficient for a person to learn a specialized formal language designed for that purpose, rather than learning through experience just which English sentences will and will not be handled. When interacting in natural language it is easy to fall into assuming that the range of sentences that can be appropriately processed will approximate what would be understood by a human being with a similar collection of data. Since that is not true, the user ends up adapting to a collection of idioms – fixed patterns that experience has shown will work. Once the advantage of flexibility has been removed, it is not clear that the additional costs of natural language (verbosity, redundancy, ambiguity, etc.) are worth paying in place of a more streamlined formal system.

An interface where the user is confronted with seemingly random breakdowns and has to guess at what will work and what won't, is frustrating and inefficient — bad ergonomics.

A special-purpose notation can not only be a convenience, but even a genuine augmentation of the intellect. Such a notation should be seen as evolution of language, helping further development of the intellect. *Such co-evolution of language and intellect should be allowed to continue in the computer age and should not be stifled by doctrinaire insistence that only English is fit for humans.*

"Natural, easy-to-use" interfaces are to be approached warily when they are slower in use than other interfaces. Windows and a mouse can be extremely enticing when a novice finds that already after the first half hour he can get simple jobs done on a computer. But an interface that takes ten times as long to learn and allows the user to work twice as fast is worth the extra trouble after four and a half hours of use. As most users spend over a hundred, or even over a thousand hours with a computer every year, it is clear that preference for the "natural and easy-to-use" can be a form of harmful conservatism.

7 Concluding remarks

If a moratorium on hardware improvement were to go into effect today, it would take decades before software caught up far enough to exploit hardware to a reasonable extent. Such a degree of exploitation includes the use of a computer as a congenial tool for thought, and deserves to be primary focus of computer science.

To exploit the potential of computers as tools to augment the intellect, the Fifth-Generation Computer Systems Project has relied on expected advances in Artificial

Intelligence. Experience in attempts to use computers to increase productivity in translation between texts in natural language suggests that more mundane approaches, summarized under Mental Ergonomics, are more effective.

I have argued against the use of Artificial Intelligence and of natural language processing by computer. Lest I be misunderstood, let me emphasize that this concerned the particular applications addressed in this paper. Artificial Intelligence as cognitive science is as interesting and important as particle physics and cosmology. Natural language processing by computer has by now reached the stage where it can be a valuable aid to human translators, and to authors more generally. This is consistent with the reservations quoted from Winograd.

Acknowledgments

Generous support was provided by the British Columbia Advanced Systems Institute, the Canada Natural Science and Engineering Research Council, the Canadian Institute for Advanced Research, the Institute of Robotics and Intelligent Systems, and the Laboratory for Automation, Communication and Information Systems Research.

References

[Bar-Hillel 1964] Y. Bar-Hillel. *Language and Information.* Addison-Wesley, 1964.

[Bush 1945] Vannevar Bush. As we may think. In Adele Goldberg, editor, *A History of Personal Workstations*, pages 237–247. Addison Wesley, 1988. First published in *Atlantic Monthly*, July 1945.

[Ceri *et al.* 1990] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer, 1990.

[Cheng *et al.* 1988] M.H.M. Cheng, M.H. van Emden, and J.H.M. Lee. Tables as a user interface for logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 784–791, Tokyo, Japan, November–December 1988. Ohmsha, Ltd.

[Cheng *et al.* 1990] M.H.M. Cheng, M.H. van Emden, and B.E. Richards. On Warren's method for functional programming in logic. In David H.D. Warren and Peter Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference*, pages 546–560, Jeruzalem, 1990. MIT Press.

[Davison 1988] A. Davison. Polka: a parlog object-oriented language. Technical report, Department of Computing, Imperial College of Science and Technology, University of London, 1988.

[Engelbart 1963] D.C. Engelbart. A conceptual framework for the augmentation of man's intellect. In P.W. Howerton and D.C. Weeks, editors, *Vistas in Information Handling*, pages 1–29. Spartan Books, 1963.

[Engelbart 1988] Douglas Engelbart. The augmented knowledge workshop. In Adele Goldberg, editor, *A History of Personal Workstations*, pages 187–232. Addison Wesley, 1988.

[Horton 1982] S. Horton. *Thinking Through Writing.* Johns Hopkins University Press, 1982.

[Kowalski 1979a] R.A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22:424–436, 1979.

[Kowalski 1979] R.A. Kowalski. *Logic for Problem-Solving.* Elsevier North-Holland, 1979.

[Licklider 1960] J.C.R. Licklider. Man-computer symbiosis. In Adele Goldberg, editor, *A History of Personal Workstations*, pages 131–140. Addison Wesley, 1988. First published in *IRE Transactions on Human Factors in Electronics*, March 1960, pp. 4–11.

[Moto-Oka 1982] T. Moto-Oka, editor. *Fifth-Generation Computer Systems.* North-Holland, 1982.

[Rosenblueth 1989] D. Rosenblueth. *Exploiting Determinism in Logic Programming.* PhD thesis, University of Victoria, 1989.

[Sergot 1982] M. Sergot. A Query-The-User facility for logic programming. In P. Degano and E. Sandewall, editors, *Proceedings European Conference on Integrated Interactive Computing Systems.* North Holland, 1982.

[Shapiro 1983] Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[Shortliffe 1976] E.H. Shortliffe. *Computer-Based Medical Consultations: MYCIN.* Elsevier, 1976.

[Taylor 1989] Stephen Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, 1989.

[van Emden 1976] M.H. van Emden. A proposal for an imperative complement to prolog. Technical Report CS-76-39, University of Waterloo, 1976.

[van Emden 1977] M.H. van Emden. Computation and deductive information retrieval. In E. Neuhold, editor, *Formal Description of Programming Concepts*, pages 421–440. North Holland, 1977.

[van Emden *et al.* 1986] M.H. van Emden, M. Ohki, and A. Takeuchi. Spreadsheets with incremental queries as a user interface for logic programs. *New Generation Computing*, 4:287–304, 1986.

[Winograd 1972] T. Winograd. *Understanding Natural Language.* Edinburgh University Press, 1972.

[Winograd and Flores 1987] T. Winograd and F. Flores. *Understanding Computers and Cognition.* Addison-Wesley, 1987.

[Zloof 1977] M. Zloof. Query-By-Example: a database language. *IBM Systems Journal*, 16:324–343, 1977.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

An Integrated Knowledge Support System

B R Gaines, M Linster† and M L G Shaw

Knowledge Science Institute, University of Calgary
Calgary, Alberta, Canada T2N 1N4
†Expert Systems Research Group, GMD
PO Box 1240, D-5205 St. Augustin 1, Germany

Abstract

The overall aim of the studies reported is to explore the possibility of creating highly integrated *knowledge support systems* through the loose coupling of software tools developed independently at unrelated sites for different purposes. Three tools were integrated to form a combined knowledge acquisition and performance system. A hypermedia tool was used as a general-purpose knowledge acquisition tool for unstructured material in the form of text and diagrams. A knowledge acquisition tool was used to elicit knowledge more formally and structure it as a computable knowledge base. A knowledge representation and deduction tool was used to represent the elicited knowledge and perform inferences with it to generate advice in a performance situation. Strong synergy was created between the tools in that the annotation and explanation captured in the hypermedium system were available as context-sensitive help to the user of the expert system, and the expert system was used to validate the knowledge base generated by the knowledge acquisition tool and feed back anomalous cases as additional data for induction.

1 Introduction

This paper reports the results of a collaborative program of research between the Canadian Knowledge Science Institute (KSI) and the German National Research Centre for Computer Science (GMD) on the integration of knowledge acquisition and performance systems. Researchers at the GMD have developed a knowledge-based systems shell, BABYLON, which combines object-oriented knowledge representation with a number of powerful inference paradigms (Christaller, di Primio & Voss, 1989). Researchers at the KSI have developed a knowledge acquisition system, KSS0, which combines object-oriented knowledge representation with a number of powerful elicitation, visualization and induction paradigms (Shaw & Gaines, 1987). The Canadian researchers have also developed inter-program communication protocols enabling Apple's HyperCard to be used as an extension to KSS0 for the textual and graphic annotation of knowledge structures (Gaines, Rappaport & Shaw, 1989). The outcome of the collaborative program is a system, Hyper-KSE, combining hypermedia, knowledge acquisition and performance tools, to provide an environment supporting knowledge-based system development from acquisition, through application to maintenance.

The approach we have adopted to system implementation is the *heterogeneous integration* of existing tools, involving substantial redevelopment but not the design and implementation of a monolithic system. This is an important issue in its own right and our logic for heterogeneous integration is manyfold (Gaines, 1990a); notably: the rapid pace of change of all the base technologies making systems obsolescent as they are implemented; the need to incorporate subsystems optimized for particular roles, probably by different organizations; and the overarching requirement for continuous upgrading and enhancement without a total system rebuild. All our current system development is based on highly modular systems communicating through servers and implemented as class libraries in object-oriented languages (Gaines, 1991).

2 The Principles Involved

When combining tools in an integrated system it is important to have a clear systems architecture in terms of required functionality and how it is to be allocated among the tools. It is necessary to exclude some facilities provided by some tools because it is duplicated in others or inappropriate in the overall system. If this is not done, and the combined system does not project a clear model of its intended use, then users can become confused by excessive features. A functional model for the integrated system has been built at a high level of abstraction in terms of four dimensions of logical validation of a knowledge base, corresponding to similar dimensions in evaluating the truth of scientific theories (Rescher, 1979):-

- Coherence—the coherence of internal relationships between knowledge structures
- Consistency—the lack of logical contradiction between knowledge structures
- Correctness—the correctness of deductions from the knowledge structures checked against external data
- Completeness—the adequate coverage of an intended scope for deductions from the knowledge structures.

Figure 1 shows how the three tools relate to the four dimensions of validation.

At the center of Figure 1, each of the tools provides means for *visualizing* the conceptual structures they support. For example, the knowledge acquisition tool provides various clustering algorithms for presenting case data graphically, the hypermedia tool is essentially visual, and the representation system provides its own knowledge-base grapher. From a formal validation point of view such visualization supports the expert and knowledge engineer in evaluating the *coherence* truth of the knowledge structures—the internal relations between different representations. Such evaluation provides feedback through each of the tools to correct errors, improve expressiveness, and so on, that are made manifest through incoherence.

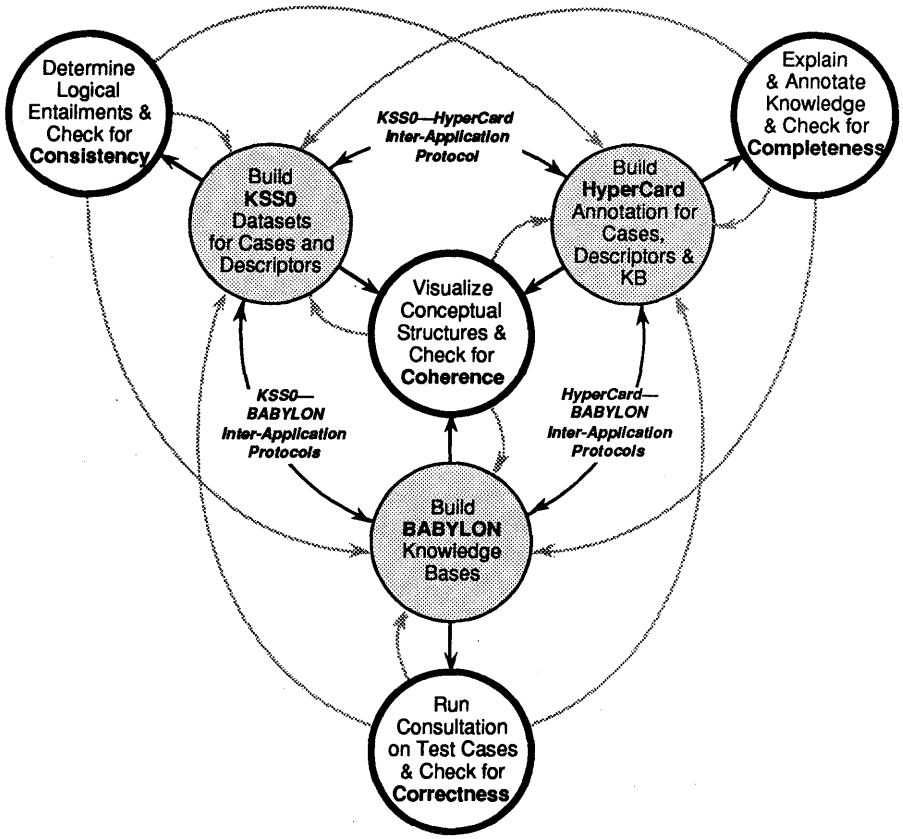


Fig. 1 Logical basis for modes of knowledge validation in the integrated system

At the top left of Figure 1, the induction module in the knowledge acquisition tool derives plausible constraints on the database of case data in the form of entailments that eventually become expressed as rules. It supports the evaluation of these entailments for mutual *consistency* (can two rules arrive at different conclusions on the same case) and for consistency with the case data. Such evaluation again provides feedback through all the tools—for example, one may amend cases in the knowledge acquisition tool, amend rules in the representation tool, or make a note of the potential problem in the hypermedia tool.

At the bottom of Figure 1, the running of new test cases, perhaps in routine system use, provides evaluation of the *correctness* of the knowledge base as a decision support tool. Again such evaluation provides feedback through all the tools, amendment of the case base and re-induction, direct editing of the knowledge structures, or commentary in hypertext.

Finally, at the top right of Figure 1, it is shown that the hypermedia tool provides a far more significant logical validation role that its annotation duties may suggest. Systems intrinsically cannot validate themselves for *completeness*, and clearly there can be no guarantees of completeness in an open universe. However, in terms of validating the formal knowledge base completeness, the informal knowledge base held in hypermedia form plays a very significant role. Anything mentioned informally that has no referent formally leads to a suspicion of incompleteness requiring further investigation.

3 The Integrated System Architecture and Operation

The knowledge acquisition tool KSS0, providing repertory grid, conceptual clustering, conceptual comparison, empirical induction, and knowledge base creation tools, is already configured as a set of modules around a specialist database. It focuses on mediating representations supporting the expert's modeling processes in moving from skilled performance to overt knowledge structures supporting its emulation in the computer. In recent years it has been extended to support the informal representations of knowledge that are prior to those within the tool, such as text, pictures, diagrams, semi-structured interviews, protocols, and so on. This has been done by providing an interapplication protocol allowing KSS0 to interact with HyperCard to provide the appearance of a seamless single application to users. KSS0-specific functionality in HyperCard is supported by scripts that allow conceptual structures on the KSS0 side to be linked to informal sources and annotation on the HyperCard side.

KSS0 exports knowledge bases to a range of existing knowledge-based system shells, and a number of collaborative studies have been reported in which these shells have been integrated directly (Gaines, Rappaport & Shaw, 1989; Gaines & Linster, 1990). Figure 2 shows the distributed knowledge base and inter-application protocols linking the hypermedia, knowledge acquisition and knowledge-based system shell in Hyper-KSE.

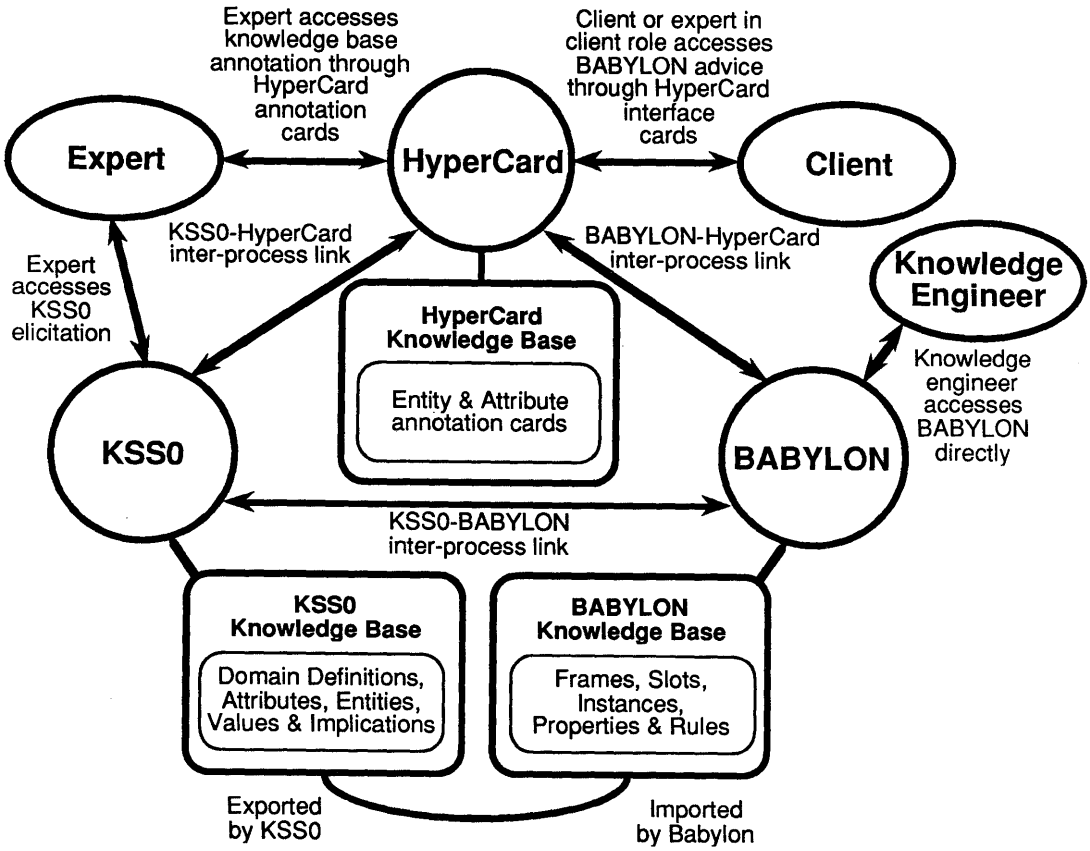


Fig. 2 Integration of hypermedia, knowledge acquisition and knowledge-based system shell

Figure 3 shows a typical sequence of activity in using the integrated system to develop a knowledge base. The acquisition commences with parallel creation by the knowledge engineer of a repertory grid in KSS0 and annotation stack in HyperCard. The elicitation tools are then used by the expert to enter critical cases characterizing the domain together with their relevant distinctive attributes and annotation about both cases and attributes. In particular, this annotation can include descriptions of the procedures by which the expert characterizes a case in terms of the attributes, and tutorial examples of such characterization. At any time during this process the expert or knowledge engineer can analyze the knowledge entered to date, visually clustering it, deriving underlying implications, and extending the knowledge structures and annotation in the light of such feedback.

When the expert feels that a reasonable amount of knowledge has been entered it is exported to BABYLON where it can be consulted on test cases, including access to the annotation in HyperCard. If a test case leads to an error then this case can be posted back to KSS0 with a corrected recommendation, re-analyzed and a revised knowledge base exported to BABYLON.

The following sequence of screen dumps is based on a simple tutorial example on mushroom toxicity provided for training with Hyper-KSE. Figure 4 shows the entity screen in KSS0 where a list of mushrooms has been entered, KSS0 is automatically evaluating entity and attribute matches and suggesting further elicitation, and the user has popped up a menu giving access to commands, annotation in HyperCard and consultation in BABYLON.

Figure 5 shows the way in which matching attributes are shown graphically in KSS0. The expert can change a rating just by dragging an entity along the scale. He or she may also respond to the prompt at the top to enter another, discriminating case. This graphic presentation has proved very effective in involving domain experts, and allowing them to enter knowledge directly into the computer without communicating it through a knowledge engineer.

Figure 6 shows a cluster analysis of the data entered in KSS0. This analysis is available interactively at any time, providing a different perspective on the cases which both motivates the expert and allows the coherence of the data in terms of meaningful relations between entities and between attributes to be evaluated.

Figure 7 shows the result of the user selecting HyperCard annotation on the popup menu over "Sweet scented Boletus" in Figure 5. The fields in the upper half are generated automatically from the case data, and those in the lower half are user-entered annotation, including a button to show further information.

Figure 8 shows the result of the user selecting the "Show" button in Figure 7. The additional annotation can be of any form, access to a videodisc, sound replay, database, simulation, and so on. This is not computational information, but it is an important part of the interface of the knowledge base to a user, for example, in explaining terminology.

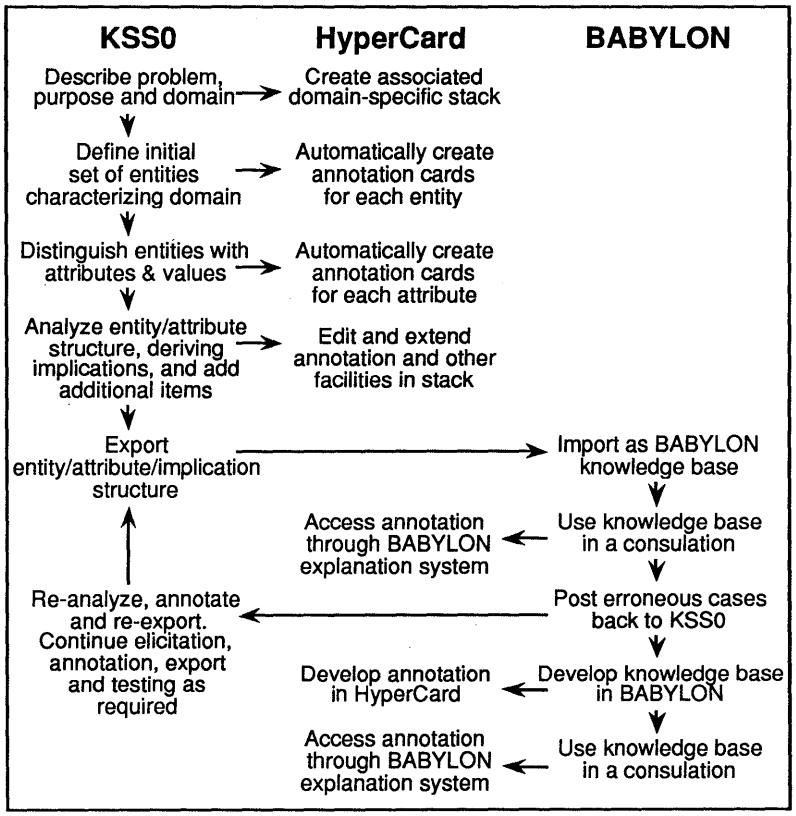


Fig. 3 Sequence of activities in using integrated system for knowledge base development

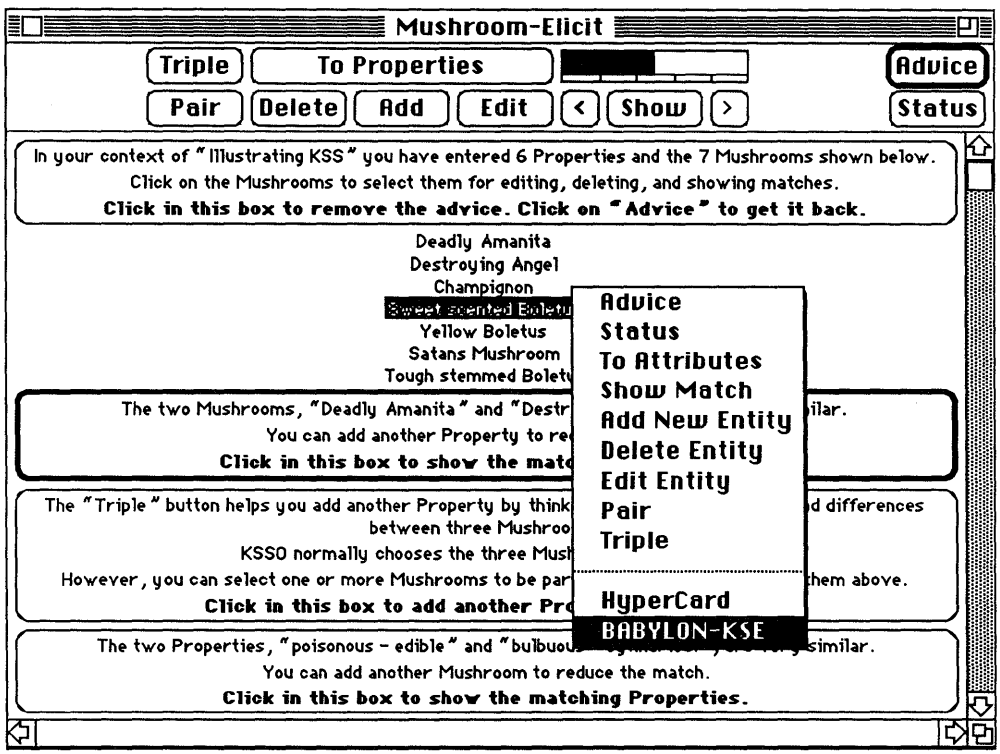


Fig. 4 Entity screen in KSS0 showing popup menu to access annotation and shell

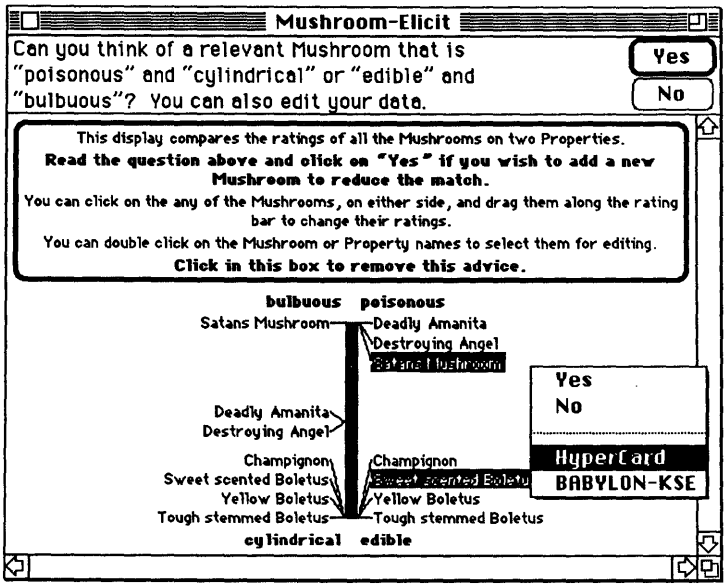


Fig. 5 Attribute match screen in KSS0 prompting entry of a new entity

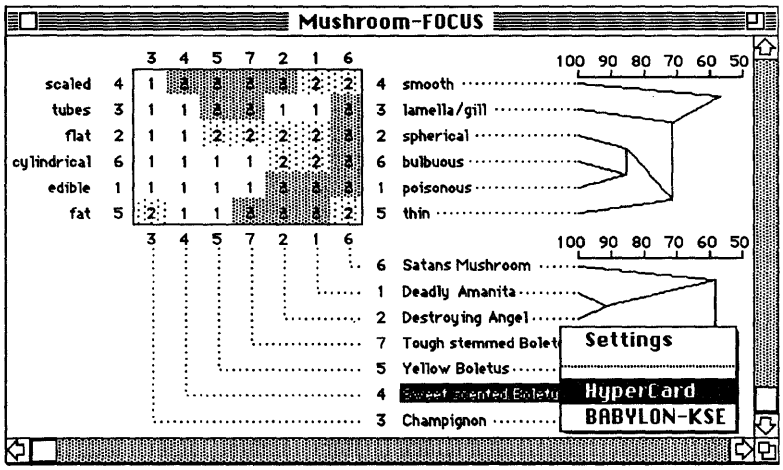


Fig. 6 FOCUS cluster analysis screen in KSS0

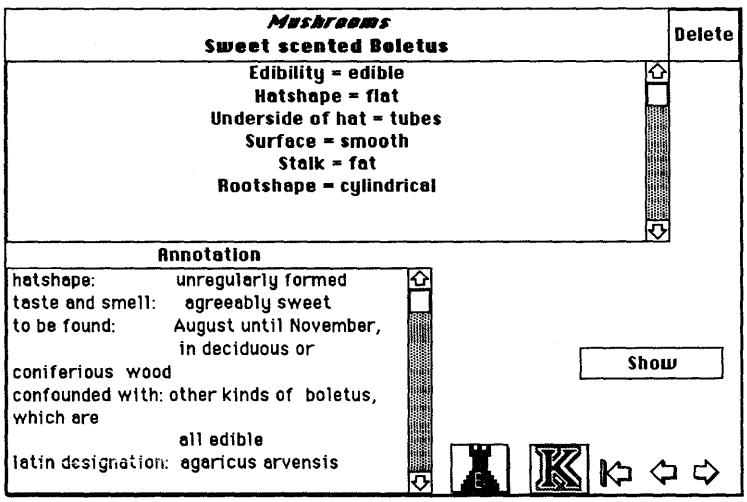


Fig. 7 Entity annotation card

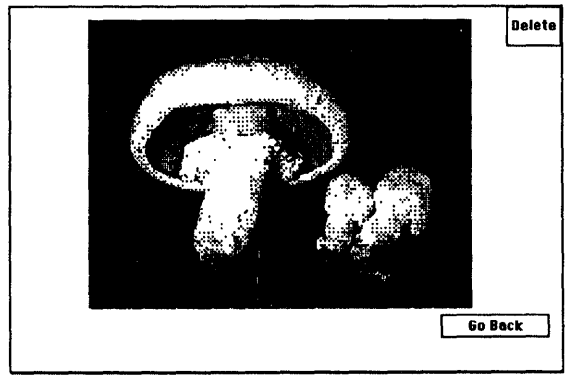


Fig. 8 Associated auxiliary annotation

Figure 9 shows a consultation running directly in BABYLON using the knowledge base entered in KSS0. The popup dialog box allows the user to select one of the possible values, or to go to the HyperCard annotation if, for example, he or she does not understand the question and needs some explanation of how this attribute should be evaluated.

Figure 10 shows the same consultation running in BABYLON with the requests for data being made through HyperCard using the attribute annotation cards for query purposes. This is the preferred mode for end users since HyperCard may be used to give a non-technical interface oriented to the particular domain with extensive help facilities. The availability of the direct mode as shown in Figure 9 is important to the knowledge engineer since it gives access to the extensive tracing and knowledge structure visualization facilities in BABYLON.

Figure 11 shows how the knowledge engineer can edit the results of a consultation on a test case in BABYLON and then post it back as a new entity to KSS0—similar facilities are available to the expert or client during the equivalent HyperCard-based consultation. These facilities put the application of the knowledge-based system into the knowledge acquisition process. They model what must be achieved throughout knowledge-based systems if we are to develop systems in which knowledge-base maintenance is an integral feature of ongoing knowledge acquisition.

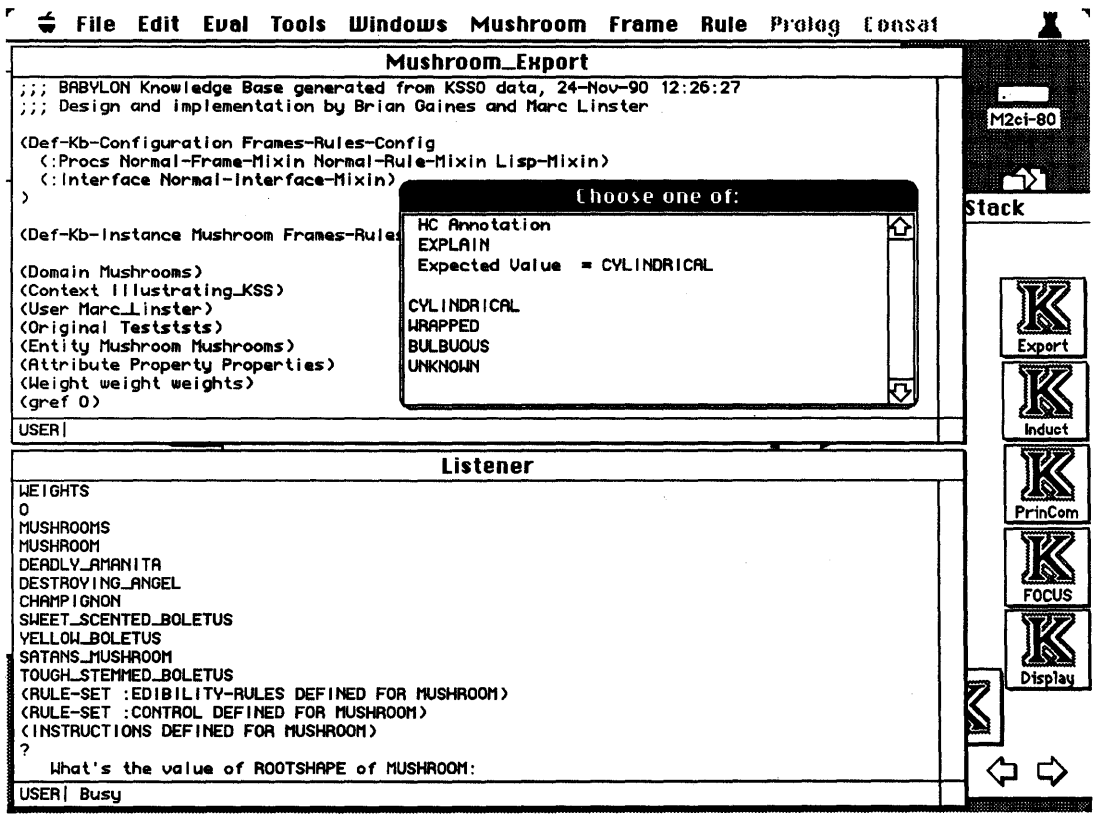


Fig. 9 Direct consultation with BABYLON

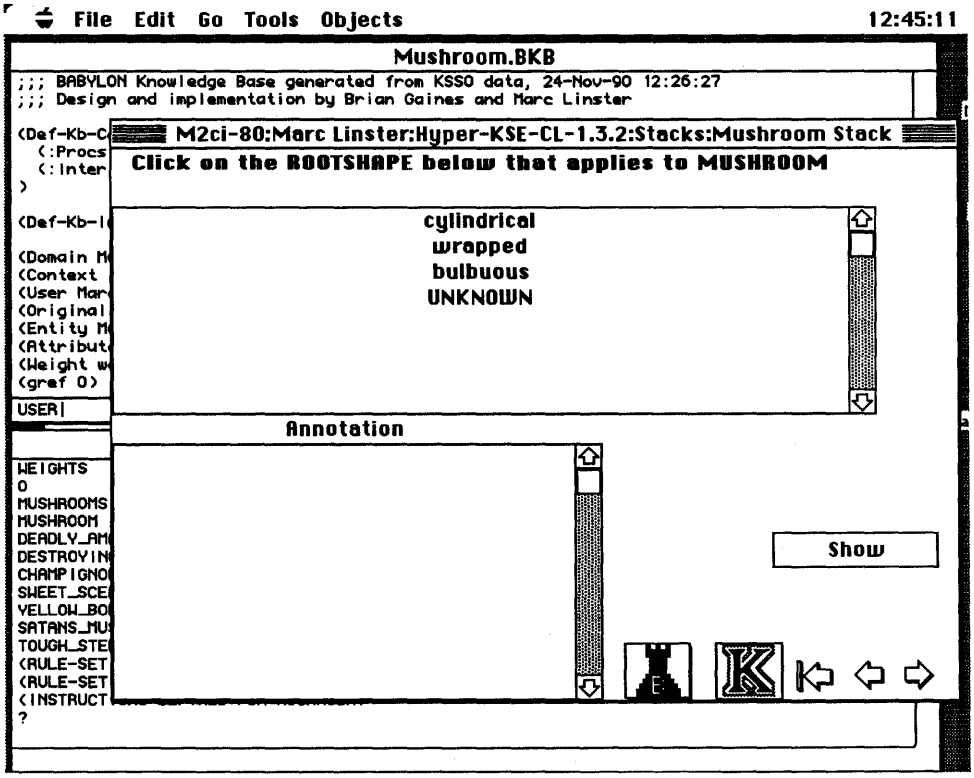


Fig. 10 Consultation with BABYLON using HyperCard as user interface

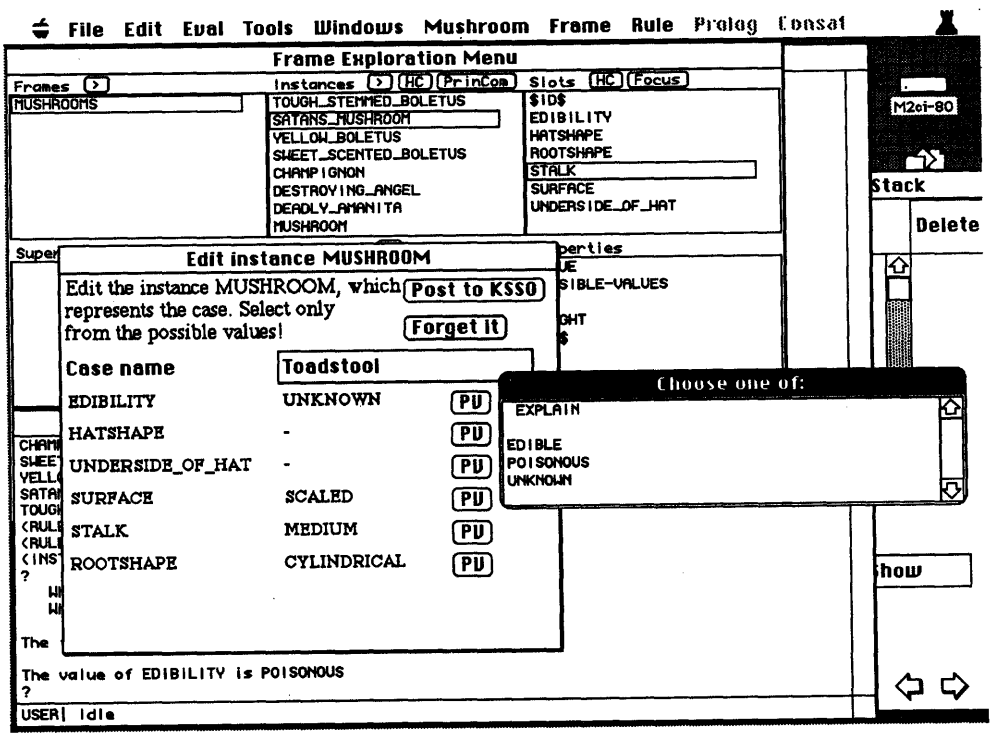


Fig. 11 Editing a case in BABYLON and posting it back to KSS0

4 Conclusions

This simple example serves to illustrate the main features of the integrated system. Each of the tools supports very large knowledge bases and a variety of applications has shown that the system scales effectively to significant applications. The graphic case elicitation tools in KSS0 operate effectively for some 20 to 50 cases which is enough to characterize the attributes of a coherent subdomain. The induction tool in KSS0 has been shown to be effective with databases exceeding 10,000 cases (Gaines, 1991d), inducing rules and evaluating edited knowledge bases rapidly enough for interactive use. HyperCard, with appropriate indexing tools, is capable of annotating databases of 10,000 cases without loss of interactivity. BABYLON has been used on a number of majorknowledge-based system developments, and has recently been reimplemented to support large scale industrial applications.

The main weakness of Hyper-KSE is in the difficulty of sustaining the functional integration of the knowledge acquisition tools in the development and application of complex applications. In a straightforward diagnostic application, soluble through heuristic classification, the major part of the knowledge base is a single, large but coherent, case base, and induced and manually entered rules. The representation and inference system has a simple task and its knowledge structures do not extend beyond those of the acquisition tools. In more complex system developments involving multiple subdomains, the acquisition tools may be used to characterize each subdomain, but the problem-solving, strategic knowledge that is involved in using the subdomain knowledge effectively has to be entered directly into the representation and inference tool. As the balance of the system changes such that this knowledge becomes increasingly important, the involvement of the acquisition tools in knowledge base maintenance is reduced.

This indicates the need for acquisition tools supporting problem-solving techniques, and knowledge-level integration based, for example, on generic problem-solving methodologies (Chandrasekaran, 1988). Some recent experiments have shown that multiple heuristic classification of subdomains may be used to solve complex procedural problems, such as sequential decision making in room allocation (Gaines, 1991e). As a wider range of structures for generic problem solving methodologies are developed it is becoming feasible to extend the type of system described here to include more meta-knowledge designed to manage the acquisition, validation, and maintenance phases systematically. Extensions to KSS0 to support such methodologies have been reported recently (Gaines, 1991a,b,c), and many researchers are working to develop the problem solving methodologies and test them in applications.

5 References

Chandrasekaran, B (1988) Generic tasks as building blocks for knowledge-based systems: the diagnosis and routine design examples. **The Knowledge Engineering Review, 3**(3), 183-211.

Christaller, T., di Primio, F. and Voß, A. (1989) **Die KI-Werkbank BABYLON**. Bonn: Addison Wesley.

Gaines, B.R. (1990). Organizational integration: modeling technology and management. Zunde, P. & Hocking, D. (Eds.) **Empirical Foundations of Information and Software Sciences II.** pp.51-64. New York: Plenum Press.

Gaines, B.R. (1991a). Empirical investigation of knowledge representation servers: design issues and applications experience with KRS. **AAAI Spring Symposium: Implemented Knowledge Representation and Reasoning Systems**. pp. 87-101. Stanford (March) and **SIGART Bulletin 2**(3), 45-56 (June).

Gaines, B.R. (1991b). Integrating rules in term subsumption knowledge representation servers. **AAAI'91: Proceedings of the Ninth National Conference on Artificial Intelligence**. pp.458-463. Menlo Park, California: AAAI Press.

Gaines, B.R. (1991c). An interactive visual language for term subsumption visual languages. **IJCAI'91: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence**. San Mateo, California: Morgan Kaufmann.

Gaines, B.R. (1991d) Induction and visualization of rules with exceptions. Boose, J.H. & Gaines, B.R. (Eds) **Proceedings of the Sixth AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop.** pp. 9-1-9-25. Banff (October).

Gaines, B.R. (1991e). Organizational modeling and problem solving using an object-oriented knowledge representation server and visual language. **COCS'91: Proceedings of Conference on Organizational Computing Systems**. ACM Press (November).

Gaines, B.R. & Linster, M. (1990). Development of second generation knowledge acquisition systems. Wielinga, B., Boose, J.H. & Gaines, B.R., Schreiber, G., van Someren, M., Eds. **Current Trends in Knowledge Acquisition.** pp. 143-160. Amsterdam, IOS.

Gaines, B.R., Rappaport, A. & Shaw, M.L.G. (1989). A heterogeneous knowledge support system. Boose, J.H. & Gaines, B.R., Eds. **Proceedings of the Fourth AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop.** pp.13-1-13-20. Banff (October).

Rescher, N. (1979) **Cognitive Systematization**. Oxford: Basil Blackwell.

Shaw, M.L.G. & Gaines, B.R. (1987). KITTEN: Knowledge Initiation & Transfer Tools for Experts & Novices. **International Journal of Man-Machine Studies, 27**, 251-280.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Modeling the Generational Infrastructure of Information Technology

B R Gaines

Knowledge Science Institute, University of Calgary
Calgary, Alberta, Canada T2N 1N4
gaines@cpsc.ucalgary.ca

Abstract

A socio-economic model of generational infrastructure of information technology is presented as a tiered progression of 'learning curves' in mutually supportive technologies. This model is used to analyze trends in research and product development, and the transition from 'computer science' to 'knowledge science' that characterizes the fifth generation. The achievements of fifth generation research are evaluated, and the expected directions of future generations research are projected.

1 Introduction

The Japanese Fifth Generation research program has had an important socio-economic impact internationally in raising government awareness of fundamental changes in the nature of information technology and its strategic role. It has been directly responsible for structural change in national computing policy such as the formation of the MCC within the US anti-trust ethos, and the ESPRIT program in Europe cutting across strongly entrenched national boundaries. One side-effect of this has been to bring into prominence what was previously seen only as a marketing/technical description of the evolution of information technology in terms of its generational infrastructure. As the fifth generation program comes to an end, this raises policy questions as to the nature and significance of the next generation, and as to the utility of conceptualizing computing research in terms of generational advances.

This paper analyzes the questions in terms of a general model of 'learning curves' whose time scale is largely determined by the medium term business cycle of capital replacement. It highlights an important difference between computing and other industrial technologies in that the pace of change in the base, vlsi, technology is so rapid that conventional 'substitution' effects are swamped by a tiered infrastructure of learning curves involving major qualitative differences in technologies. A detailed account of the underlying model and its fit to the past development of information technology has been given elsewhere (Gaines & Shaw, 1986; Gaines, 1990, 1991), and this paper focuses on fifth generation and later issues.

2 Electronic Device Technology

The initial breakthrough for computing was in electronic device technology, and a definition of computer generations in terms of hardware works well for the early machines. However, as Rosen (1983) notes it blurs thereafter and "we are faced with an anomalous situation in which the most powerful computers of 1979-1980, the CDC Cyber 176 and the CRAY 1, would have to be assigned to the second and third generations, respectively, while the most trivial of hobbyist computers would be a fifth-generation system." The reason for this anomaly is that the substitution effects of one form of technology for another are gradual and do not correspond to major structural changes. The enabling effects of changing technologies are far more dramatic: the change from mechanical to electronic devices made it possible to store programs as data and enabled the use of computers as a general-purpose tool and the development of language compilers; the transistor made reliable operation possible and enabled routine data processing and then interactive timesharing; integrated circuits reduced costs to the level where computers became commonplace and made possible the personal computer dedicated to a single user.

Modern microelectronics commenced in 1956 when silicon planar process was invented and enabled integrated circuit technology to be developed. As Figure 1 shows, the number of devices on a chip follows Moore's law in doubling each year through the 1960s, and has continued to double every eighteen months through the 1970s and 1980s (Robinson, 1984). The current projected limit is some 1,000,000,000 devices on a chip in the late 1990s when quantum mechanical effects will become a barrier to further packing density on silicon planar chips. However, three-dimensional packing, semiconducting peptides, optical devices, or, most likely, new materials not yet considered, are expected to extend the growth.

Microelectronics shows over 9 decades of performance increase in 40 years. Such exponential growth is common in many technologies, but never over more than 2 decades and then in periods of the order of 100 years. Computer technology is unique in being based on underlying devices whose performance has increased at a rate and over a range achieved by no other technology. Logarithmic plots, such as that of Figure 1, do not adequately project the impact such a long-term sustained growth, but this is apparent in the linear plot of the devices on a chip by computer generation as shown in Figure 2. During each generation, changes have taken place that correspond in magnitude to those of some hundred years in other industries. These quantitative changes have led to major qualitative effects that are analyzed in the following sections.

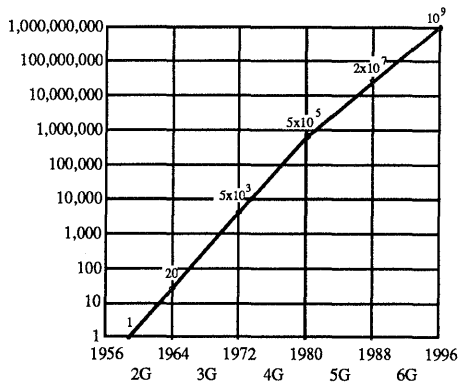


Figure 1 Growth of devices on a chip

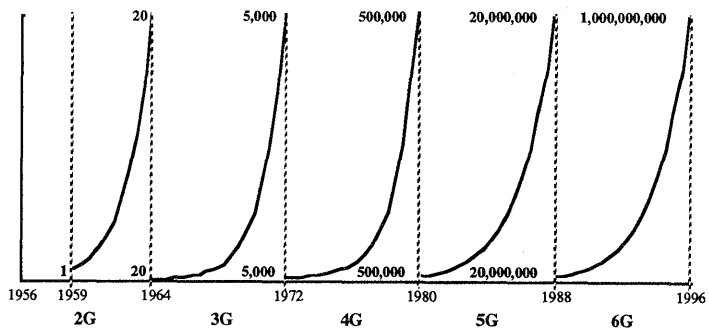


Figure 2 Devices on a chip during six generations of computers

3 Learning Curves in Scientific and Technological Development

There is a simple phenomenological model of developments in science technology as a logistic "learning curve" of knowledge acquisition (Ayres, 1968; Marchetti, 1981). It has been found to be a useful model of the introduction of new knowledge, technology or product in which growth takes off slowly, begins to climb rapidly and then slows down as whatever was introduced has been assimilated. Such curves arise in many different disciplines such as education, ecology, economics, marketing and technological forecasting (Van Dujin, 1983; Stoneman, 1983).

It has also been noted in many disciplines that the qualitative phenomena during the growth of the logistic curve vary from stage to stage (Crane, 1972; De Mey, 1982; Gaines & Shaw, 1986). The era before the learning curve takes off, when too little is known for planned progress, is that of the inventor having very little chance of success. When an inventor makes a *breakthrough*, very rapidly his or her work is *replicated* at research institutions world-wide. The experience gained in this way leads to *empirical* design rules with very little foundation except previous successes and failures. However, as enough empirical experience is gained it becomes possible to inductively model the basis of success and failure and develop *theories*. This transition from empiricism to theory corresponds to the maximum slope of the logistic learning curve. The theoretical models make it possible to *automate* the scientific data gathering and analysis and associated manufacturing processes. Once automation has been put in place effort can focus on cost reduction and quality improvements in what has become a *mature* technology.

4 The Infrastructure of the Information Sciences

The fast, sustained, learning curve for electronic devices, and the scope for positive feedback in the information sciences, together result in a tiered infrastructure for the information sciences and technologies which is fundamental to their nature. It involves a succession of learning curves as rapid advances in one level of technology trigger off invention in others as shown in Figure 3.

The breakthrough in *electronic device technology* leading to the zeroth generation is placed at 1940 about the time of the Atanasoff and Berry experiments with tube-based digital calculations. Automation by 1980 had reached the extreme level where silicon compilers allow a designer to implement ideas directly in devices with little further human intervention (Fields, 1983).

The first breakthrough generating a computing infrastructure was the introduction of the stored program *virtual machine architecture*. Mauchly (1973) recognized the significance of stored programs, noting that subroutines create "a new set of operations which might be said to form a calculus of instructions." This was the key conceptual breakthrough in computer architecture, that the limited functionality provided directly by the hardware could be increased by stored programs called as subroutines or procedures, and that the hardware and these routines together may be regarded as a new virtual machine. This is the foundation of the development of a variety of forms of virtual machine architectures (Weegenaar, 1978) that separates out computing science as a distinct discipline from other areas of electronic applications.

The next level of breakthrough was in software to bridge the gap between machine and task through the development of *problem-oriented languages*. Work on the design of FORTRAN in 1954 and its issue in 1957 marks the beginning of the second generation era with languages targeted to specific problem areas of business data processing, text processing, database access, machine tool control, and so on. A 1968 paper on the coming fourth generation notes that "programming today has no theoretical basis" and calls for a scientific basis in the next generation (Walter, Bohl & Walter, 1968). Sure enough the theory linking languages to the underlying virtual machines developed during the fourth generation era, for example, that of abstract data types and initial algebras (Goguen, Thatcher & Wagner, 1978). In the fifth generation era the application of experience, design rules and theory to the automation of software production became the top priority (Balzer, Cheatham & Green, 1983).

The next level of breakthrough was in *interactive activity systems* when continuous interaction becoming a significant possibility as the mean time between failures of computers began to be hours rather than minutes in the early 1960s. The move from batch-processing to direct human-computer interaction was made in 1963/1964 with the implementation of MIT MAC, RAND JOSS and Dartmouth BASIC systems. The study of such systems led to design rules for HCI in the 1970s (Hansen, 1971) and theoretical foundations started to emerge in the 1980s

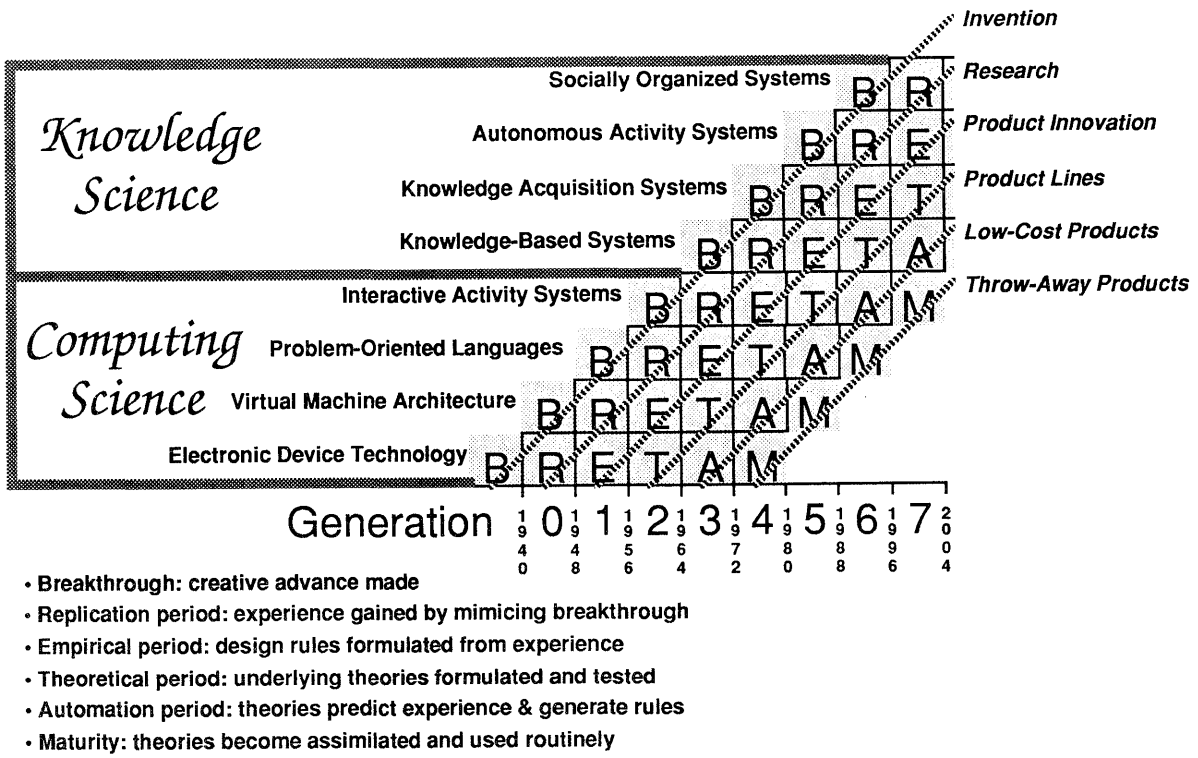


Figure 3 The infrastructure of the information sciences

(Alexander, 1987). The improvement of human-computer interaction was a major stated priority in the Japanese fifth generation development program (Karatsu, 1982). Other forms of interaction also became feasible as a result of improved reliability such as direct digital control, and various forms of digital communications systems.

The next level of breakthrough was one of *knowledge-based systems* supporting knowledge-processing, the human capability to store information through its inter-relations and make inferences about its consequences. The breakthrough in knowledge-based systems dates from the development of DENDRAL (Buchanan, Duffield & Robertson, 1971) for inferring chemical structures from mass-spectrometry data and MYCIN (Shortliffe, 1976) for the diagnosis of microbial infections in the early 1970s. It led to a spate of expert system development in the fourth generation era of the 1970s (Gevarter, 1983), and pragmatic design rules for knowledge engineering in the current fifth generation era (Hayes-Roth, 1984). The utilization of their vlsi production capability (Gaines, 1984; Galinski, 1983) for the support of knowledge-based systems through PROLOG machines (Kitsuregawa & Tanaka, 1988) has been the other major priority in the Japanese fifth generation development program (Moto-oka, 1982).

Defining the upper levels of the infrastructure becomes more and more speculative as we move into the immediate past of our own era and look for evidence of learning curves that are at their early stages. It is reasonable to suppose that the level above the representation and processing of knowledge in the computer is that of *knowledge acquisition systems*, breakthroughs in machine learning and expertise modeling. Two breakthroughs in this area have been Lenat's AM learning mathematics by discovery (Davis & Lenat, 1982) and Michalski's inductive inference of expert rules for plant disease diagnosis (Michalski & Chilausky, 1980). In the fifth generation era machine learning became a highly active research area in its replication phase (Michalski & Carbonell, 1983). The general field of knowledge acquisition has also seen a massive growth in research (Boose, 1989).

One may speculate that the growth of robotics will provide the next breakthroughs in which goal-directed, mobile computational systems will act autonomously to achieve their objectives. The breakthrough into the sixth generation era commencing in 1988 will probably be seen as one of *autonomous activity systems*. It is possible to see the nascent concepts for this breakthrough in the adoption of the goal-directed programming paradigms of logic programming languages such as PROLOG. When, in a robot, a goal specification is expanded by such a programming system into a sequence of actions upon the world dependent on conditions being satisfied in that world, then the behavior of such a system will deviate sufficiently from its top-level specification, yet be so clearly goal-directed, as to appear autonomous. However, to achieve significant results with such systems we need to add perceptual acts to the planning structures of a language such as SIPE (Wilkins, 1984) and develop logic programming languages that cope with the resulting temporal logic (Allen, 1984)—in these developments the sixth generation breakthrough will come to be recognized, possibly in the notion of "situated action" (Suchman, 1987) and its application in subsumption architectures for autonomous robots (Brooks, 1990; Connell, 1990).

One may speculate further that interaction between these systems will become increasingly important in enabling them to cooperate to achieve goals and that the seventh generation era commencing in 1996 will be one of *socially organized systems*. The Japanese "Sixth Generation" research program proposals emphasize emulation of

creativity and intuition and the development of inter-disciplinary *knowledge sciences* (STA, 1985; Gaines, 1986a). This recognizes the distinction between "computer science" and "knowledge science" as shown in Figure 3, and that cutting edge innovation in the information sciences involves human and social considerations intrinsic to the nature of knowledge.

It is also possible that building an adequate forecasting model based on the premises of this paper may undermine the very processes that we model. If we come to understand the dynamics of our progress into the future then we may be able to modify the underlying process—to make the next steps more rapidly when the territory is better mapped.

5 Using the BRETAM Model

The tiered infrastructure model of Figure 3 also shows the superimposed trajectories of invention, research, and so on. The intersection of these with the horizontal lines of the different information sciences may be used to model and predict the primary focus of different types of activity in each generation of computers:

- *Invention* is focused at the **BR** interface where new breakthrough attempts are being made based on experience with the replicated breakthroughs of the technology below.
- *Research* is focused at the **R E** interface where new recognized breakthroughs are being investigated using the empirical design rules of the technologies below.
- *Product Innovation* is focused at the **ET** interface where new products are being developed based on the empirical design rules of one technology and the theoretical foundations of those below.
- *Product Lines* are focused at the **T A** interface where product lines can rest on the solid theoretical foundations of one technology and the automation of the technologies below.
- *Low-cost Products* are focused at the **AM** interface where cost reduction can be based on the automated mass production of one technology and the mature technologies below.
- *Throw-away Products* are at the **MM** interface where cost reduction has become such that maintenance and repair costs exceed replacement costs.

For example, by the end of the fourth generation (1972-80):

- **BR**: recognition of the knowledge acquisition possibilities of knowledge-based systems led to the breakthrough to inductive-inference systems.
- **R E**: research focused on the natural representation of knowledge through the development of human-computer interaction, e.g. the Xerox Star direct manipulation of objects.
- **ET**: experience with the human-computer interaction using the problem-oriented language BASIC led to the innovative product of the Apple II personal computer.
- **TA**: the simplicity of the problem-oriented language RPG II led to the design of the IBM System/3 product line of small business computers.
- **AM**: special-purpose chips allowed the mass-production of low-cost, high-quality calculators.

By the end of the fifth generation (1980-88):

- **B R**: recognition of the goal-seeking possibilities of inductive inference systems led to breakthroughs in autonomous-activity systems.
- **R E**: research focused on knowledge acquisition for knowledge-based systems.
- **ET**: the advantages of the non-procedural representation of knowledge for human-computer interaction led to the innovative designs of Lisp and Prolog machines.
- **T A**: the ease of human-computer interaction through a direct manipulation problem-oriented language led to the Apple Macintosh product line of personal computers.
- **AM**: the design of highly-integrated language systems has allowed the mass-production of low-cost, high-quality software such as Turbo Pascal.
- **MM**: calculators have become so low in cost that replacement is preferable to repair.

By the end of the sixth generation (1988-96):

- **BR**: recognition of the cooperative possibilities of autonomous intelligent systems will lead to a breakthrough in socially organized systems.
- **RE**: research will be focused on autonomous intelligent behavior in systems such as neural networks and subsumption robots.
- **ET**: the advances in inductive systems will lead to new products for extracting knowledge from large datasets.
- **T A**: non-procedural problem-oriented languages will become routinely available on main-frame computers.
- **AM**: highly interactive personal workstations will drop in cost to a level where they become standard office equipment.
- **MM**: workstation replacement will have become more effective than maintenance and repair.

6 Significant Developments and Interactions

The BRETAM model can be used to highlight the significant developments in information technology for purposes of planning research, development and applications. Figure 4 left shows the cross section of Figure 3 that is relevant to the state of the art in information technologies during the previous, fifth generation of computers. The top three levels on the right of invention, research and innovation show why the fifth generation is generally recognized for its innovations in artificial intelligence. It was during this period that knowledge-based system products such as expert system shells first became available. However, in terms of reliance upon proven technology, it is the lower levels of product lines and below that are significant. The fifth generation was that in which human-computer interaction was dramatically improved through *graphic users interfaces, object oriented languages* brought control of complex system development in software engineering, and *networking* became ubiquitous. All these innovations took for granted advances in the underlying device technology that offered very fast powerful and reliable processors and large high-speed memories at low-cost.

Figure 4 right shows the equivalent picture of what is happening now as we progress through the sixth generation of computers. Hardware and networking have become almost negligible in cost and almost indefinitely powerful. Large-scale distributed systems are becoming readily available in terms of equipment and hardware architectures. Object oriented languages, and their associated application programming support environments (APSEs) and class libraries, are becoming routinely available at very low-cost. Graphic user interfaces (GUIs) are become standardized and portable across platforms as a routinely available technology. By the end of this generation the lowest level of knowledge-based system technology will have become available as well-supported product lines. These will support large-scale conceptual modeling at the enterprise level, the integration of heterogeneous information and

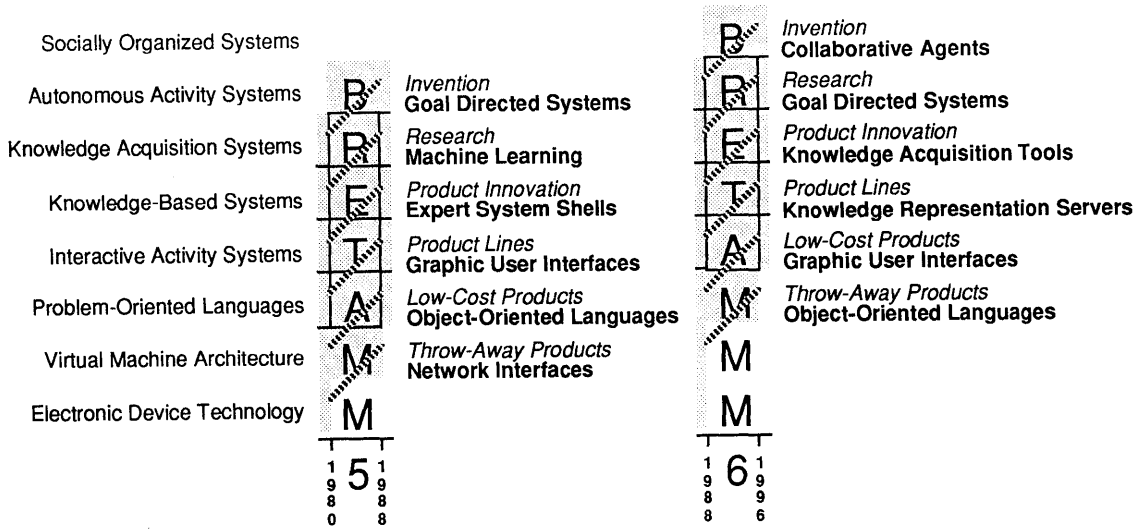


Figure 4 Significant technologies in the fifth and sixth generations

processes at lower levels, and the emulation of many aspects of human skilled behavior. Whether they are called "object oriented deductive databases," "second generation expert system shells," or "knowledge representation servers," or something completely different, is a matter for fashion, chaos theory, linguistics and marketing, to determine—we can already define their functionality and exhibit their application and that is enough.

If one focuses on a particular area of development or application, the BRETAM model may be used to examine the influences on it from technologies at different levels, and hence in differing states of maturity. For example, Figure 5 shows the influences on the development of Computer-Aided Software Engineering (CASE) tools during the fifth and sixth generations. CASE tools were developed as part of the *automation* phase for problem-oriented languages and, while commercial tools are designed to support many paradigms, the full impact is dependent on the development of formal *specification languages* subject to verification of correct implementation. The computer-support of CASE diagramming tools, group support, and theorem-proving verification methods has been dependent on the availability of low-cost high-power processors and displays from the electronic device technology level supporting workstation and networks at the virtual machine architecture level. The application of this low-level technology to CASE has been dependent on the development of graphic user interfaces at the interactive activity systems level. The complexity of reasoning required in operating a full CASE environment has made it a major target for applications of expert systems technology at the knowledge-based systems level, and we may expect an increasing application of machine learning techniques to support automatic programming as the knowledge acquisition systems level develops.

Thus, at any given level, there is dependency on the availability of the more mature technologies below, and support for further development from innovations in the less mature technologies above.

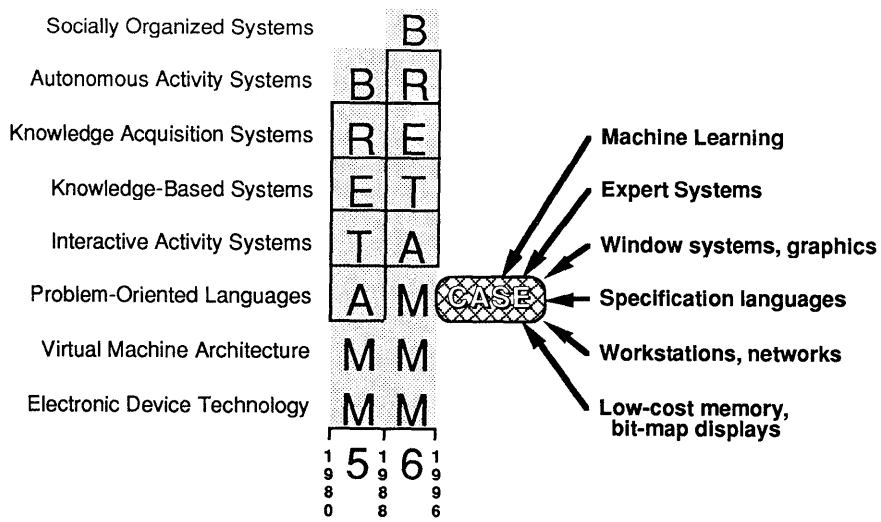


Figure 5 Influences between levels in development of Computer-Aided Software Engineering (CASE)

7 Current Issues at Each Level

The levels in the BRETAM model can be used as the basis for a check list of current issues in information technology.

In *electronic device technology* the packing densities are becoming so great that to sustain the line of growth in Figure 1 new approaches are needed. Electron beam lithography has been used experimentally to fabricate devices with gate lengths down to 65nm and current efforts are targeted on 20nm (Allee, Broers & Pease, 1991). There is also extensive exploration of alternative materials to silicon for the fabrication of computing devices. Nonlinear interactions between material and photon streams are capable of supporting computation and maintaining continuing research activities on optical computing technologies (Lebreton, 1991). Research also continues on organic semiconductors (Gunshor, 1988) where high fabrication densities are theoretically possible and genetic engineering techniques offer new manufacturing approaches. It is significant to note that the technology of DNA, its analysis and fabrication, is not only critically dependent on information technology but also is itself a parallel "information processing technology"—in organic material rather than silicon. The most innovative advances in materials during the fifth generation era have been those targeted on *nanotechnology*, of system fabrication at the molecular level. The molecular 'train sets' of today (Stoddart, 1991) are a fascinating curiosity that illustrate a breakthrough in both fabrication and instrumentation at the molecular level, and offer opportunities for new technologies which are currently at the limits of our imagination.

In *virtual machine architecture* the primary motivation has been to take advantage of the opportunities for parallel processing offered by modern vlsi technology. This has resulted in a very wide range of approaches from machines utilizing tens of thousands of conventional processor chips (Alder, 1988) to neural network technologies performing computations in radically different ways (Soucek, 1991). The essential tension between special-purpose computing and general-purpose computing that has existed since the earliest days of the digital computer underlies research activities and product innovations in this area. It is commonly stated as one of relation to the level above, of the provision of problem-oriented languages to interface computational requirements to computational resources, and it also a problem of integration, of combining special-purpose functionality with general-purpose support technology.

In *problem-oriented languages* the move towards a mature technology is apparent in many changing attitudes to the technology as well as in new developments. The application of a conceptual framework of manufacturing to software has become very fruitful and is a major thrust in Japan, Europe and the USA (Fernström, 1991; Humphrey, 1991; Basili, Caldiera & Cantone, 1992). However, the requirements for zero-defect, maintainable software manufactured from reusable parts has generated new requirements in the base technology. The development of Ada and C++ have been two practical extensions to previous technology addressing issues of reusability such as abstraction and encapsulation. However, there are also more fundamental trends towards theoretically well-founded languages such as PROLOG and ML that offer possibilities for combining verification with reasonable run-time performance. In the sixth generation era the combination of features and experience from logic programming, functional programming and object-oriented programming, functional programming and object-oriented programming to provide new generation languages will be a major area of research and development—Aït-Kaci's proposals for LIFE are an indication of what might be achieved (Aït-Kaci & Podelski, 1991).

In *interactive activity systems* the need to achieve interoperability between diverse equipment across high-speed local and wide-area networks has been a major developmental thrust in the fifth generation leading to widespread adoption of the ISO OSI standards (Day and Zimmerman, 1983), and to major standardization efforts at the upper levels concerned with data content and application integration (Modiri, 1991). There has been a parallel thrust in human-computer interaction where the need to develop products with platform-independent graphic user interfaces has led to the adoption of CMU's X-Windows and OSF's Motif as virtual standards under Unix, and the development of user interface management systems such as OIT and XVT that allow a single interface definition to be implemented automatically on virtually all personal computers and workstations. What is still missing in networking is the capability to integrate communities across wide area networks such that the system appears as a single entity rather than a local having one set of functions, loosely and heterogeneously coupled to a global network. What is missing in the user interface are the capabilities to recognize spoken language and to interact in natural language. These have been objectives since the first generation of computers and it is reasonable to expect that advances in all the lower level technologies supporting knowledge-based systems together with the development of more powerful knowledge bases make speech and language attractive targets for sixth generation development.

In *knowledge-based systems* the need to development secure theoretical foundations for knowledge representation may be seen as a major dynamic in all research. The acceptability of arbitrary heuristics in artificial intelligence systems declined in the fifth generation, and complexity analyses of basic knowledge representation systems began to identify the intrinsic problems of deductive knowledge representation systems (Brachman and Levesque, 1984; Nebel, 1988; Schmidt-Schauß, 1989). In the mid-1980s Aït-Kaci (1984, 1986) gave a lattice-theoretic model of knowledge base languages with operational semantics through term rewriting that resolved many of the issues of complexity and deduction algorithms for term subsumption knowledge representation systems. This ψ-calculus is particularly interesting because it provides foundations for complex object representation in deductive databases, for type computation in functional programming languages, and for knowledge representation in artificial intelligence. Developments based on sound theory are now targeted on the provision of knowledge representation services, and effort is becoming increasingly focused on the development of standardized modular systems with layers of well-defined and well-implemented functionality. This research is underway not only in the artificial intelligence community but also in the deductive database (Ohori, 1990), logic programming (Yardeni and Shapiro, 1991) and functional programming communities (Fuh and Mishra, 1990). This convergence of interests is to be expected as certain aspects of 'knowledge' and 'intelligence' are factored out to become realized by standard computational data structures and processes.

In *knowledge acquisition systems* the replication era of pragmatic copying of techniques and tools in the 1980s has given way to attempts to integrate methodologies and understand their underlying basis in the 1990s.

McDermott's (1988) *role limiting methods* are based on the abstraction of control knowledge from a family of related tasks, and the use of this to classify knowledge requirements and usage. As we identify role limiting methods we may come to rationalize them and develop alternative approaches that are more principled which is the basis of Chandrasekaran's (1988) analysis of *generic tasks*: The KADS research project (Akkermans, Harmelen, Shreiber and Wielinga, 1992) in the ESPRIT program has led to major advances in principled approaches to knowledge acquisition using a software engineering model. Apart from the development of more principled knowledge acquisition methodologies, we may also expect existing knowledge acquisition tools to be applied to the develop of large-scale knowledge bases in the sixth generation era. The Cyc project at MCC (Lenat & Guha, 1990) has been the outstanding example of such an attempt during the fifth generation, and the move towards knowledge interchange standards (Neches, Fikes, Finin, Gruber, Patil, Senator & Swartout, 1991) may be seen as supporting more widespread developments in the sixth generation.

In *autonomous activity systems* we are only at the beginning of an understanding of the nature of situated action and the interplay between the activities we interpret as 'planning' and 'representation' and the underlying neural processes which seem completely different in nature (Clancey, 1990). It is probable that for advances in this area we shall have to adopt a much wider perspective on knowledge-based systems in computing that analyzes their essential relationships to the cognitive and social knowledge processes of the human species (Gaines, 1991). Similar considerations apply to the expected breakthrough in *socially organized systems*. Current experiments in computer-supported cooperative work, intelligent agents surrogates, and distributed artificial intelligence are the necessary preliminaries to a major advance, and are reflected in the emphasis on system integration at lower levels. Socially organized will be a very active but relatively uncoordinated area of research throughout the sixth generation era.

8 Conclusions

It is interesting to examine the fifth generation program, and its world-wide emulation, in the light of the BRETAM model. In terms of timing, the Japanese announcement in 1982 came as knowledge-based system technology was moving into its empirical phase with expectations of significant product innovation by the late-1980s. The distinction between the lower four levels of 'computer science' and the qualitative change in the nature of information technology innovation in a new era of 'knowledge science' was also becoming apparent. The entire focus of the ICOT research and development program has been consistent with this rationale since it has concentrated on the enabling technologies for effective knowledge-based systems in terms of use of vlsi technology for new machine architectures supporting logic programming on interactive work stations. Thus the emphasis has been on the application of mature or maturing computer science technologies to provide a solid technological foundation for innovation in knowledge-based systems.

The parallel programs triggered off in the USA, UK and Europe, have not had such a specific focus but have tended to address the whole cross-section of fifth generation technology issues from chip encapsulation, through parallel architectures, specification languages, user interfaces, network protocols, knowledge representation systems, knowledge acquisition and planning. They have also addressed specific application domains significantly impacted by fifth generation developments, such as office automation and computer-integrated manufacturing. These other areas have also been addressed in Japan but not as part of the specific 'fifth generation' initiative.

As a final conclusion, it is reasonable to conjecture that the essence of sixth and seventh developments will that of treating 'knowledge' as the raw material to be processed rather than 'data.' This conclusion will not be surprising for anyone at this meeting, but it is one that may be given more quantitative substance and detail through the model presented in this paper. In particular, it is important to note that the shift towards a knowledge perspective in no way reduces our dependency on continuing advances in electronics, machine architectures, software engineering and improved human-computer interaction. These are the bedrock on which the foundations of knowledge-based systems are laid.

9 Acknowledgements

Financial assistance for this work has been made available by the Natural Sciences and Engineering Research Council of Canada.

10 References

Aït-Kaci, H. (1984). A lattice-theoretic approach to computation based on a calculus of partially-ordered types. University of Pennsylvania, Philadelphia.

Aït-Kaci, H. (1986). An algebraic semantics approach to the effective resolution of type equations. **Theoretical Computer Science 45** 293-351.

Aït-Kaci, H. and Podelski, A. (1991). Towards a meaning of LIFE. Paris Research Laboratory, Digital Equipment Corporation. 1-42.

Akkermans, H., Harmelen, F. van, Shreiber, G. and Wielinga, B. (1992). A formalisation of knowledge-level models for knowledge acquisition. **International Journal of Intelligent Systems** to appear.

Alder, B.J., Ed. (1988). **Special Purpose Computers.** Boston: Academic Press.

Alexander, H. (1987). **Formally-Based Tools and Techniques for Human-Computer Dialogues.** Chichester, UK: Ellis-Horwood.

Allee, D.R., Broers, A.N & Pease, R.F.W. (1991). Limits of nano-gate fabrication. **Proceedings IEEE, 79**(8), 1093-1105.

Allen, J.F. (1985). Towards a general theory of action and time. **Artificial Intelligence, 23**(2), 123-154 (July).

Ayres, R.U. (1968). Envelope curve forecasting. **Technological Forecasting for Industry and Government: Methods and Applications.** pp. 77-94. Englewood-Cliffs, New Jersey: Prentice-Hall.

Balzer, R., Cheatham, T.E. & Green, C. (1983). Software technology in the 1990s: using a new paradigm. **Computer, 16**(11), 39-45 (November).

Basili, V.R., Caldiera, G. & Cantone, G. (1992). A reference architecture for the component factory. **ACM Transaction Software Engineering 1**(1), 53-80.

Boose, J H. (1989) A survey of knowledge acquisition techniques and tools. **Knowledge Acquisition 1** (1), 39-58 (March).

Brachman, R.J. and Levesque, H.J. (1984). The tractability of subsumption in frame-based description languages. **Proceedings of AAAI-84.** San Mateo, California, Morgan Kauffman. 34-37.

Brooks, R.A. (1990). Elephants don't play chess. Maess, P., Ed. **Designing Autonomous Agents.** pp. 3-15. Cambridge, Massachusetts: MIT Press.

Buchanan, B.G., Duffield, A.M. & Robertson, A.V. (1971). An application of artificial intelligence to the interpretation of mass spectra. Milne, G.W.A., Ed. **Mass Spectrometry Techniques and Applications.** New York: John Wiley.

Chandrasekaran, B. (1988). Generic tasks as building blocks for knowledge-based systems: the diagnosis and routine design examples. **The Knowledge Engineering Review 3**(3) 183-211.

Clancey, W.J. (1990) The frame of reference problem in the design of intelligent machines. In K. van Lehn & A. Newell, **Architectures for Intelligence: The Twenty-Second Carnegie Symposium on Cognition.** Hillsdale: LEA.

Connell, J.H. (1990). **Minimalist Mobile Robots.** Boston: Academic Press.

Crane, D. (1972). **Invisible Colleges: Diffusion of Knowledge in Scientific Communities.** Chicago: University of Chicago Press.

Davis, R. & Lenat, D.B. (1982). **Knowledge-Based Systems in Artificial Intelligence.** New York: McGraw-Hill.

Day, J.D. and Zimmerman, H. (1983). The OSI reference model. **Proceedings IEEE 71**(12) 1334-1340.

De Mey, M. (1982). **The Cognitive Paradigm.** Dordrect: Reidel.

Fernström, C. (1991). The Eureka software factory: concepts and accomplishments. **ESEC'91: 3rd Eurpean Software Engineering Conference.** Berlin, Springer. 23-36.

Fields, S.W. (1983). Silicon compiler cuts time to market for DEC's MicroVAX I. **Electronics, 56**(21), 47-48 (October).

Fuh, Y.-C. and Mishra, P. (1990). Type inference with subtypes. **Theoretical Computer Science 73**(2) 155-175.

Gaines, B.R. (1984). Perspectives on fifth generation computing. **Oxford Surveys in Information Technology, 1,** 1-53.

Gaines, B.R. (1986). Sixth generation computing: a conspectus of the Japanese proposals. **ACM SIGART Newsletter,** No.95, 39-44 (January).

Gaines, B.R. (1990). Intelligent systems as a stage in the evolution of information technology. Ras, Z.W. & Zemankova, M., Eds. **Intelligent Systems: State of the Art and Future Directions.** pp.431-455. Chichester, UK: Ellis Horwood.

Gaines, B.R. (1991). Modeling and forecasting the information sciences. **Information Sciences 57-58** 3-22 (Sep-Dec).

Gaines, B.R. (1991) Between neuron, culture and logic: explicating the cognitive nexus. **ICO: Intelligence Artificielle et Sciences Cognitives au Québec 3**(2) 47-61.

Gaines, B.R. & Shaw, M.L.G. (1986). A learning model for forecasting the future of information technology. **Future Computing Systems, 1**(1), 31-69.

Galinski, C. (1983). VLSI in Japan: the big leap forward, 1980-1981. **Computer, 16**(3), 14-21 (March).

Gevarter, W.B. (1983). Expert systems: limited but powerful. **IEEE Spectrum, 18,** 39-45.

Goguen, J.A., Thatcher, J.W. & Wagner, E. (1978). An initial algebra approach to the specification, correctness and implementation of abstract data types. Yeh, R., Ed. **Current Trends in Programming Methodology.** pp. 80-149. Englewood Cliffs, New Jersey: Prentice-Hall.

Gunshor, R.L, Ed. (1988). **Multifunctional materials.** Bellingham, Washington: International Society for Optical Engineering.

Hansen, W.J. (1971). User engineering principles for interactive systems. **AFIPS Proceedings of the FJCC.** Vol. 39, pp. 523-532.

Hayes-Roth, F. (1984). The industrialization of knowledge engineering. Reitman, W., Ed. **Artificial Intelligence Applications for Business.** pp. 159-177. Norwood, New Jersey: Ablex.

Humphrey, W.S. (1991). Software and the factory paradigm. **Software Engineering Journal 6**(5), 370-376.

Karatsu, H. (1982). What is required of the fifth generation computer—social needs and impact. In Moto-oka (1982) pp. 93-106.

Kitsuregawa, M. & Tanaka, H., Eds. (1988). **Database Machines and Knowledge Base Machines.** Boston: Kluwer.

Lenat, D.B. & Guha, R.V. (1990). **Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project.** Reading, Massachusetts: Addison-Wesley.

Levreton, G.J., Ed. (1991). **Optics for Computers: Architectures and Technologies.** Bellingham, Washington: International Society for Optical Engineering.

Marchetti, C. (1981). Society as a learning system: discovery, invention and innovation cycles revisited. **Report RR-81-29** (November). Laxenburg, Austria: International Institute for Applied Systems Analysis.

Mauchly, J.W. (1973). Preparation of problems for EDVAC-type machines. Randell, B., Ed. **The Origins of Digital Computers: Selected Papers.** pp. 365-369. Berlin: Springer-Verlag.

McDermott, J. (1988). Preliminary steps toward a taxonomy of problem solving methods. **Automating Knowledge Acquisition for Expert Systems.** Boston, Kluwer. 225-256.

Michalski, R.S. & Chilausky, R.L. (1980). Knowledge acquisition by encoding expert rules versus computer induction from examples—A case study involving soyabean pathology. **International Journal of Man-Machine Studies, 12,** 63-87.

Michalski, R.S. & Carbonell, J.G., Eds. (1983). **Machine Learning: An Artificial Intelligence Approach.** Palo Alto, California: Tioga.

Modiri, N. (1991). The ISO reference model entities. **IEEE Network Magazine 5**(4) 24-33.

Moto-oka, T., Ed. (1982). **Fifth Generation Computer Systems.** Amsterdam: North-Holland.

Nebel, B. (1990). **Reasoning and Revision in Hybrid Representation Systems.** Berlin, Springer.

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T. and Swartout, W.R. (1991). Enabling technology for knowledge sharing. **AI Magazine 12**(3) 36-56.

Ohori, A. (1990). Semantics of types for database objects. **Theoretical Computer Science 76**(1) 53-91.

Robinson, A.L. (1984). One billion transistors on a chip?. **Science, 223,** 267-268 (January).

Rosen, S. (1983). Generations, computer. Ralston, A. & Reilly, E.D., Eds. **Encyclopedia of Computer Science and Engineering.** pp. 657-658.

Schmidt-Schauß, M. (1989). Subsumption in KL-ONE is undecidable. **Proceedings of KR'89: First International Conference on Principles of Knowledge Representation and Reasoning.** San Mateo, California, Morgan Kauffman. 421-431.

Shortliffe, E.H. (1976). **Computer-Based Medical Consultations: MYCIN.** New York: Elsevier.

Soucek, B., Ed. (1991). **Neural and Intelligent Systems Integration.** New York: Wiley.

STA (1985). Promotion of Research and Development on Electronics and Information Systems That May Complement or Substitute for Human Intelligence. Tokyo: Science and Technology Agency.

Stoneman, P. (1983). **The Economic Analysis of Technological Change.** Oxford: University Press.

Stoddart, F. (1981). Making molecules to order. **Chemistry in Britain,** 714-718 (August).

Suchman, L.A. (1987). **Plans and Situated Actions.** Cambridge: University Press.

Van Dujin, J.J. (1983). **The Long Wave in Economic Life.** London: George Allen & Unwin.

Walter, C.J., Bohl, M.J. & Walter, A.B. (1968). Fourth generation computer systems. **AFIPS Proceedings of Spring Joint Computer Conference, 32,** 423-441.

Weegenaar, H.J. (1978). Virtuality and other things like that. **Proceedings of IEEE COMCON, CH1388-8/78,** 287-293.

Wilkins, D.E.. (1984). Domain-independent planning: representation and plan generation. **Artificial Intelligence,** 269-301.

Yardeni, E. and Shapiro, E. (1991). A type system for logic programs. **Journal Logic Programming 10**(2) 125-153.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Co-HLEX: Co-operative Recursive LSI Layout Problem Solver on Japan's Fifth Generation Parallel Inference Machine

Toshinori Watanabe and Keiko Komatsu

Systems Development Laboratory, Hitachi, Ltd.
1099 OHZENJI, ASAO-KU, KAWASAKI-SHI, KANAGAWA, 215 JAPAN
Tel: (044) 966-9111, Telex: 3842-577, Fax: (044) 966-6862

Abstract

Co-HLEX is a co-operative hierarchical layout problem solver developed as an application program of parallel inference machines; Multi-PSI and PIM. The kernel of Co-HLEX is a hierarchical recursive concurrent theorem prover nicknamed HRCTL. Due to its recursive nature, HRCTL has a size of only O(1,000) lines in KL1; the kernel language of ICOT. Due to its stream-parallel and distributed-memory architecture, nearly linear time complexity could be attained. Moreover, shape and wire abutment among modules running in parallel could be made possible through message passing co-operation. In this paper, a brief overview of Co-HLEX is given with its application to bipolar-analog LSI layout.

1 Introduction

The main role of Co-HLEX development in Fifth Generation Computer System project was to find some answers to the following questions;
(Q1) Are there any real-world problems which require symbolic, parallel problem solving?
(Q2) If they exist, can we find any new parallel algorithms to solve them?
(Q3) Can we find elegant descriptions of these algorithms?
(Q4) Can we execute these descriptions effectively on Multi-PSI or PIM?
(Q5) What are the new break-throughs brought about?
(Q6) What are the new problems to be pursued further?
We picked up an LSI layout problem due to the following reasons;
(R1) It is and will be one of the gigantic real-world problems requiring massive computation power. At present, the number of rectangles contained in the layout of 1 cm^2 DRAM chip is almost equal to that of 100 m^2 rectangles covering Japan. New ideas including parallel computation are greatly required to cope with this complexity.
(R2) Both development and enhancement of a layout system, with more than 1 million-lined program codes, consume huge amount of programmers' unrewarded labour. The possibility of more elegant program descriptions should be investigated.
One of our ideas is the use of the recursion principle to reduce it [Kleene 1952]. The other is the use of a streamed parallel process network computation model to give an elegant description of mutually related layout objects. We nicknamed our algorithm HRCTL (Hierarchical Recursive Concurrent Theorem prover for Layout). The kernel language KL1, which runs on Multi-PSI and PIM, can be a powerful tool to implement them.
(R3) The classical divide and conquer - the hierarchical problem solving - works well as long as subproblems correlate weakly. In case of LSI layout, this premise cannot be sufficed. Neighbouring modules are not independent in that they should have abutted shapes and wires to avoid dead spaces. Our idea is the use of communication among modules to solve this AND-typed dependency.

In the following, Section 2, 3, 4, and 5 give basic concepts, an overview of Co-HLEX, a brief complexity consideration, and experiment results, respectively. Based on them, we hope to assert that parallel symbolic computation can contribute much to LSI design automation.

2 Basic Concepts

2.1 Layout Problem and Solution Representation

The original layout problem which Co-HLEX solves can be specified by a Prolog goal:

```
:- mode solve_a_layoutproblem(+,-,+,+).
?- solve_a_layoutproblem(CirNet, LPlan, Proc, Constr).
```

where arguments have the follwing meanings as shown in Figure 2.1.

CirNet::= A circuit network represented by modules and module connection nets.
LPlan::= [PQtree, Wires].
PQtree::= A quadtree [Samet 1984, Otten 1982, Luk et al. 1986] representing a placement by a slice hierarchy each node of which carries a module name placed in the slice and peripheral connector names placed on north-, west-, south-, and east-edge of the slice.
Wires::= [[Conns, Lines] | Wires].
Conns::= Set of peripheral- and inner-connectors of a net. Note that induced connectors are peripheral connectors of subnodes. Inner connectors include terminal points and vias. Vias are through-holes connecting different silicon layers on the chip. An induced connector is introduced at

each point where a wire crossses a slice edge.
Lines::= Set of line segments spanning two connectors of a net. It includes connector names on both ends, the run layer name, and the line width.
Proc::= LSI fabrication process name which affects geometrical shapes of modules, usable wiring layers, line width, minimum allowances among objects, and others.
Constr::= A list of constraints including a set of proximity conditions of modules, usually called "Pairs", the topmost PL (planned layout - a planned chip size; Width and Height, and a set of planned peripheral connector placements).

2.2 Recursive Problem Solving

The slicing structure representation of a layout permits us the following recursive problem solving.
(S1) If the module is an indivisible leaf cell, import its layout from a library. If the module is a divisible block, divide the original layout problem, composed of a circuit data and a PL (planned layout), into at most 4 subproblems, each having a homologous structure with the original problem.
(S2) Solve all the subproblems in parallel using the recursion principle. If much processing elements or PEs are available, fork these subproblems on different PEs.
(S3) Aggregate finished layouts of subproblems following the placement plan given in (S1) to generate the layout of the original problem.

2.3 Problems and Solutions

(SL1) Pre-compilation of circuit net into a quadtree
In (S1) above, the original planned area should be divided into at most four slices, the circuit should also be divided into relevant number of subcircuits, and their embedding plan into slices should be made. To avoid the untractable computational complexity caused by these two divisions and one embed, the CirNet is transformed into a quadtree-shaped process network named CMPN (Circuit Module Process Network) before layout generation. Each node in the CMPN is a process having message-passing streams among its upper and lower nodes. Each leaf node of CMPN represents a module in CirNet but a non-leaf node represents a block module newly defined in this transformation. Modules specified as a pair by Constr are compiled into an identical node near the leaf of CMPN to assure mutual proximity. The main role of CMPN is the layout generation, i.e., module placement and inter-module wire generation. If the module placement topology, i.e., in what quadrant each subcircuit should be placed, can be given in the stage of CMPN generation, later placement task is fairly simplified.
(SL2) Vertical co-ordination of module shapes
As subproblems are solved in parallel in (S2) above, non-abutment of their final shapes might happen. In that case, chip area will be enlarged due to many dead spaces among modules. To avoid this, the planned shape of the subslice in which a subcircuit is being placed is descended down to the

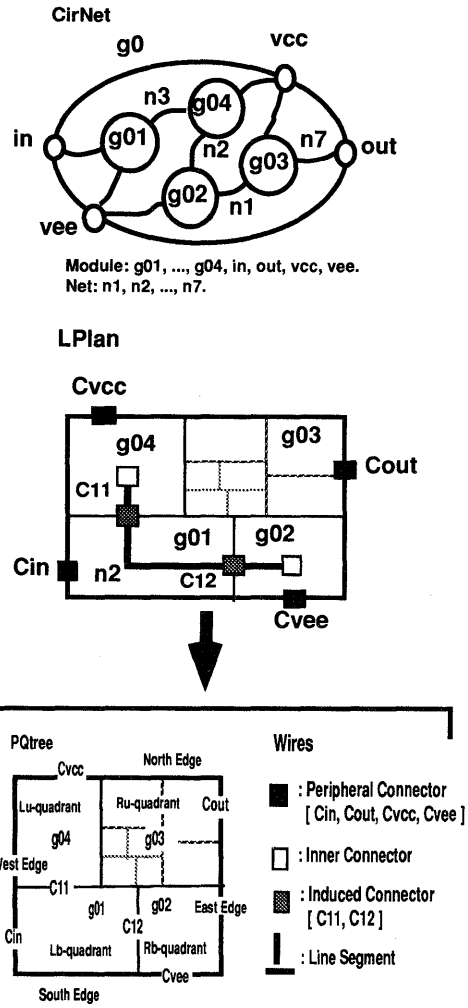


Figure 2.1 Circuit and Layout Representations

subcircuit as its planned shape. See Figure 2.2(1).
(SL3) Horizontal co-operation in wiring
Wire abutment among modules running in parallel had been an unsolved problem in LSI layout design automation. This problem is solved by way of runtime co-operation among nodes in CMPN. The induced connector processes (See 2.1) are used for this purpose. Each of them holds a CERW (Current Existence Range of a Wire on a slice edge). As the first task of wiring, each node process in CMPN tries to narrow CERWs of its peripheral connectors on the north-, west-, south-, and east-edges. If a CERW intersects with two internal subslices, then the CERW is narrowed to one of the two intersections. Among the two candidates, the one which has both enough wiring capacity and minimum wire length with inner connectors is selected. For feed-through nets, i.e., nets having no inner connectors, the CMPN node eagerly waits for a completion of narrowing action by some neighbour node to avoid useless wire bend generation. See

igure 2.2(2).
;L4) Wiring direction control to reduce co-operation loads
lthough the CERW narrowing co-operation is useful in arallel wiring, it is expensive due to the repeated peripheral onnector inspections. This is particulary true for cell wiring here tremendously large number of cells attend in the co-peration. To reduce useless co-operation, wiring direction is o-ordinated in cell wiring. First, in all the cells and for all ets, partial wires are generated that connect inner- and eripheral-connectors on south or east edges. Each CERW n these edges is narrowed into a point where the wire rrived (SE-wiring). Then, NW-wiring follows. Finallly, on-directional feed-through wiring (ND-wiring) is made.)ue to the CERW reductions in two previous steps, much ager-waiting co-operation in ND-wiring can be avoided. ee Figure 2.2 (3).

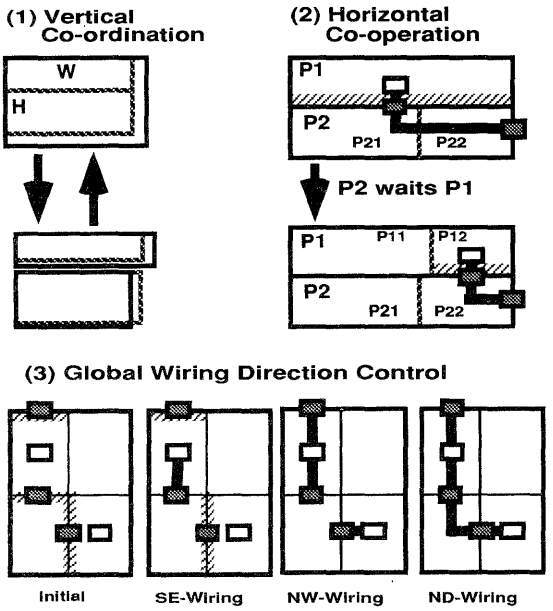


Figure 2.2 Problem Solving Heuristics

3 Overview of Co-HLEX

An overview of Co-HLEX is given in Figure 3.1. The main components of Co-HLEX include: a set of original data, I/O functions, a backup memory, a problem solving kernel based on CMPN, and a template library.

3.1 The Problem Solving Kernel

The problem solving kernel is a quadtree-shaped process network CMPN that generates a chip layout. Before the layout generation, each node of CMPN has only circuit data including a module name, the module property, a list of net names connecting this module to others, and a list of subcircuit names. After the layout, a set of layout data is added to each node including; a name of the layoutframe (parameterized slicing template) used to slice the node, an enveloping rectangle size, a list of slicing points in the rectangle, a list of submodule names in each slice, a list of adopted wiring pattern name for each net, and a list of peripheral- and induced-connector names.

3.1.1 Problem Solving Steps

The overall layout problem solving is performed by the follwoing steps.
(ST1) Placement: A placement message containing a PL - a list of planned shape and planned peripheral connector placements - is sent to the top node of CMPN from the top level co-ordination process. Then a set of placement actions based on HRCTL is performed by CMPN processes.
(ST2) Wiring preparation: Upon receiving the placement completion message from the top node of CMPN, the co-ordinator sends a wiring preparation message to it. Then a set of wiring preparation actions are made by CMPN.
(ST3) Wiring non-terminal power nets: Power supply nets - Vcc and Vee - have different width from other signal nets. As they offend the latter, they are wired first. The co-ordinator sends a message to the top node of CMPN to envoke recursive power wiring actions.
(ST4) Wiring non-terminal signal nets: The co-ordinator sends a message to the top node of CMPN to generate signal nets. Then a set of wiring actions based on HRCTL is performed by CMPN processes. Recursion terminates when it reaches to a cell node. At that time, the CERWs held by connector processes contract to the magnitude of cell size.
(ST5) Wiring nets in cells (SE-wiring): The co-ordinator sends a messages to the top node of CMPN to do SE-wiring. This message is passed down to cells. Cells which have inner connectors such as base-, emitter-, collector-contact, etc., that should be wired to peripheral connectors on south or east edges, wires all these nets. After this, each CERW on these edges reduces to a point where a wire reached. As layout rules such as wiring obstacle avoidance, minimun allowances between layout objects, etc., should be sufficed so the maze-router of Lee [Lee 1961] was used with some modifications.
(ST6) Wiring terminal nets in cells (NW-wiring): Similar to that of (ST5).
(ST7) Wiring terminal nets in cells (ND-wiring): The co-ordinator sends a message to the top node of CMPN to do ND-wiring. This is for feed-through wires which only pass above the cell without any inner connectors. All the cell nodes make feed-through wires in parallel co-operation.

3.1.2 Placement by HRCTL

(ST1) Termination of placement: When a terminal cell node in CMPN receives a placement message from above, it imports layout data from relevant layoutframe in the

template library.
(ST2) Subproblem generation: When a node - call it CN - in CMPN is a non-terminal module, it generates subproblems as follows. It sends its own PL and a list of its subcircuits with their estimated areas to a subproblem generation planframe (planning frame) in the library.
The planframe sends requests to all layoutframes to make and evaluate possible slicing of PL and embedding of subcircuits into derived slices. The evaluation is made in view of the estimated wire length among inner- and peripheral-connectors, the estimated layout area, and estimated distortions of realized layout from the PL. Receiving the best plan and the relevant layoutframe name from the planframe, CN memorizes them. Then inter-slice wiring is made by a wiring planframe relevant to the chosen layoutframe. In the first step of this wiring, CERWs of peripheral connectors are narrowed. Then inter-slice wires are planned for each net. Only abstract wiring plan is made as shown in Figure 2.1. Wiring patterns attatched to the chosen layoutframe is used. As the final wireability largely depends on the wire congestion on slice edges, so the wiring resource consumption on these edges should be balanced. To do this, the idea of wiring resource vector is introduced. It is a list of maximum possible wires through slice edges. In selecting a wiring pattern for each net, the resource vector consumption is analyzed for all the possible patterns and the best one is selected. New induced connectors are given, each having a CERW identical to the edge length on which it was defined. Induced connector processes are newly spawned, each having a message stream to CN. Finally, subproblem definition planframe is envoked to give all the PLs for all subcircuits. The planframe defines PLs by using derived subslices and their peripheral connectors and descends them to lower CMPN nodes. Streams to peripheral connectors are also descended to assure subsequent narrowing actions by subnodes. Notice that by this combination of placement and wiring, PLs of subcircuits homologous to that of parent CN could be generated.
(ST3) Recursion: All the subnodes of CN are invoked in parallel to solve their problems. When many PEs are available, they are spawned on different PEs.
(ST4) Placement aggregation: Layout aggregation planframe is invoked by CN. It waits for the completion message from all subnodes and after eceiving the message, it aggregates all the realized layouts of subnodes to generate the CN layout. The layoutframe chosen in (ST2) is reused here to give an aggregation scheme. But when it gives a large dead space in CN layout, layoutframe swapping is tried.

3.1.3 Wiring Preparation

To make dead spaces usable in wiring, they are compiled into CMPN as dummy modules. Module placement points are determined in world co-ordinate with its origin at noth-west corner of the chip. After the dead space compilation, connector processes generated in placement become unusable, so they are killed. A CWPN - Cell Wiring Process Network approximating the meshed cell - is newly generate under each cell sharing a communication stream with th cell. Wiring obstacles in each cell are examined by using th relevant layoutframe and written into CWPN.

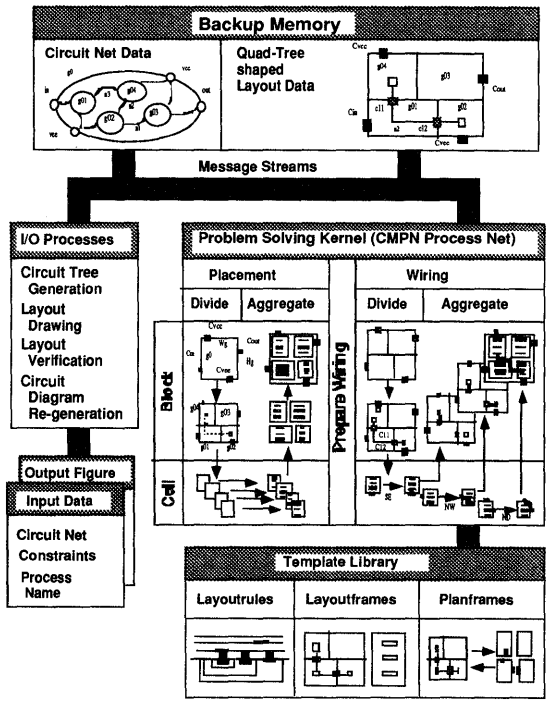


Figure 3.1 Overview of Co-HLEX

3.1.4 Non-terminal Power Net Wiring by HRCTL

(ST1) Termination of wiring: When CN is a terminal node having no inner power connectors, it only ascends a completion message to its upper node. Otherwise, a power wire termination planframe is invoked to generate power-wire connectors in the cell.
(ST2) Subproblem generation: When CN is a non-terminal block with inner power connectors, a planframe is invoked to extend power wires along slice edges reaching to subslices.
(ST3) Recursion: All the subnodes of CN run in parallel to make power wires.
(ST4) Wiring aggregation: Aggregation planframe for power wire is invoked by CN. It waits for the power wiring completion messages from subnodes and determines the width of CN power lines based on electrical considerations.

3.1.5 Non-terminal Signal Net Wiring by HRCTL

(ST1) Termination of wiring: When CN is a terminal node, it

nly ascends a completion message.
ST2) Subproblem generation: When CN is a non-terminal lock, it generates wiring subproblems by using the same ıethod as explained in 3.1.2 (ST2). As the module lacement is already given, only wiring action is repeated. :ERWs of peripheral connectors are narrowed first. Then vires are made giving new induced connector processes. heir names are descended down to subnodes for recursion.
ST3) Recursion: All the subnodes of CN run in parallel to olve their problems. When many PEs are available, they are pawned on different PEs.
ST4) Wiring aggregation: When CN receives completion nessages from all subnodes, it ascends its own completion nessage.

3.1.6 Wiring Terminal Nets

(ST1) Wirable net detection: Each terminal CN, in this case a terminal cell, finds a net that has at least one fixed peripheral- or inner- connector. If such a net is found, CN broadcasts the net name, its connectors, and usable wiring layers at these connectors to its CWPN.
(ST2) Pre-processing: Using the broadcasted information, CWPN changes the passage cost on its nodes. High costs are given to nodes in wire inhibition area.
(ST3) Wiring a net: Modified maze-routing is performed on CWPN to find wires among given connectors.
(ST4) Post-processing: Upon completion, the CWPN node on which a connector or a wire is placed is given a high passage cost. Finally a completion message is sent to CN. Then other net is tried from (ST1).
(ST5) Wiring other nets: After memolizing the reported wire, the cell repeats step (ST1) untill all nets are wired.

3.2 Template Library

3.2.1 Plan Frames

Planframes are a set of procedures used by CN as explained in 3.1. Many of them are layoutframe specific.
(1) Choose an appropriate layoutframe for subproblem generation.
(2) Evaluate a proposed plan for placement or wiring.
(3) Generate subproblems for placement or wiring.
(4) Descend subproblems down to subnodes.
(5) Aggregate subproblem solutions for placement or wiring.
(6) Other functions.

3.2.2 Layout Frames

Layoutframes are templates, or types in other words, for layout.
(1) Block level layoutframes: Slicing templates of arity 1, 2, 3, and 4 containing several slicing structure variants. Template specific wiring patterns are included.
(2) Cell level layoutframes: Parameterized configuration templates of transistors, resisters, capacitors, and connectors.

3.2.3 Layout Rules

(1) Cell size definitions (depends on fabrication process).
(2) Allowances (same, admissible gaps between objects).
(3) Wiring rules (same, wiring layers and their usabiliy by signal and power nets).

3.3 I/O Functions

3.3.1 CMPN Generator

(ST1) Input data: The original circuit net CirNet and the PL of the topmost chip.
(ST2) Process network generation: Circuit modules and nets are transformed into processes and their connections are replaced by streams giving a CMPN - Circuit Module Process Network.
(ST3) Module shape alignment: Align-shape message is broadcasted to CMPN from the top co-ordinator. Divisible modules in CMPN such as resisters divide themselves to give aligned heights to those of standard transistors. As a result, an enlarged CMPN is given.
(ST4) Hierarchy generation: The flat CMPN given by (ST3) is recursively partitioned to give a hierarchical CMPN.

3.3.2 Assignment of Processes on PEs

LAP (List of available processors) is given to the top node of CMPN. The top node divides LAP into the number of its subnodes in accordance with their computation loads. As an approximation of the load, total number of modules in the circuit is used. One of the PE is picked up from each subset and a subnode is spawned on the PE. This process is recurred until LAP becomes indivisible. After that, all subnodes in CMPN is spawned on the same PE.

3.3.3 Other Functions

Layout data in CMPN is written out to a display terminal.

4 Computational Complexity of HRCTL

Definitions.
Let PrBPT(R,N) denotes a balanced CMPN with R subnodes and height N. Let leaf(PrBPT(R,N)) denotes the number of leaf nodes of PrBPT(R,N). Let no(PEs) denotes the number of parallel processing elements on which the problem PrBPT(R,N) is solved by the HRCTL algorithm.

Suppositions.
All the nodes of PrBPT(R,N) consume the same computation power. Instantaneous communication among processes is possible without any computation load. The total elapsed time of processing on one PE is proportional to its total computation load.

Theorem.
For PrBPT, HRCTL has the time complexity of either
O(log(leaf(PrBPT(R,N))))
or O(log(no(PE))+leaf(PrBPT(R,N))/no(PE)).
The latter is the usual case where large problem is solved on limited PEs.

Proof.
Case1. no(PE) >= leaf(PrBPT(R,N)) : The president PE is the bottleneck processor which receives the topmost node of PrBPT(R,N). It processes maximum number of nodes among PEs. The maximum number is log(leaf(PrBPT(R,N))).
Case2. no(PE) =< leaf(PrBPT(R,N)) : The president PE is also the bottleneck processor. Until the depth of log(no(PE)) is reached on PrBPT(R,N), case1 applies. After it, each PE is obliged to solve all the unsolved nodes in pseudo parallel mode. Here, the number of unsolved nodes is
leaf(PrBPT(R,N))/no(PE). As the president PE faces the two situations sequentially, they should be added to give the log(no(PE))+leaf(PrBPT(R,N))/no(PE) complexity. **QED.**

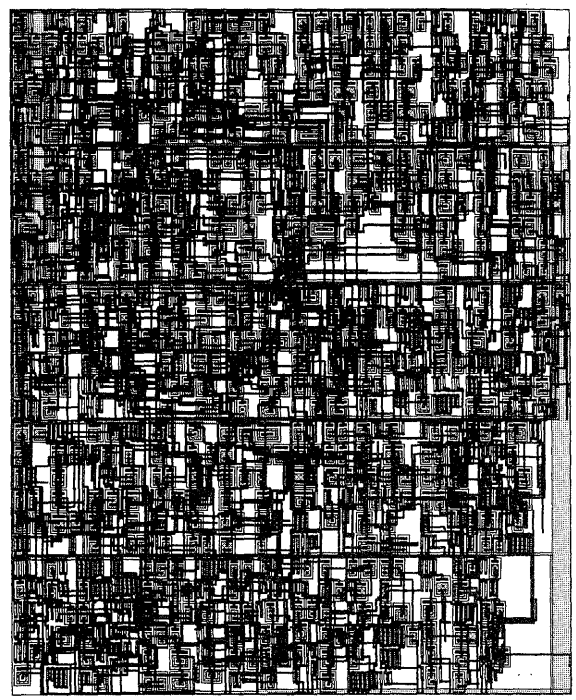
5 Experiments

5.1 Experiment Design

(1) Main objectives: The main objectives of the experiment are the verifications of;
OE1. Parallel placement and wiring capability,
OE2. Wire length and chip area reduction by vertical co-ordination and holizontal co-operation,
OE3. Enhanced computation speed,
OE4. The program size reduction and maintenability.
(2) Used circuit and fabrication process: A real bipolar analog circuit with 1019 modules and 683 nets are used in the experiment. 149 pairs were given as Constr. After module shape alignment, CMPN had 1299 modules and 901 nets. The height of generated CMPN was 14. From this original circuit, 254-, 489-, and 810-moduled subcircuits were extracted for computation speed measurement. A bipolar analog fabrication process with 3 wiring layers of AL1, AL2, and AL3 was assumed. The first two are for signal nets and the last one for power nets. For signal nets, as much AL1 should be used as possible to attain high electrical quality. Traditionally, time consuming maze-router is usually applied to this problem.

5.2 Experiment Results

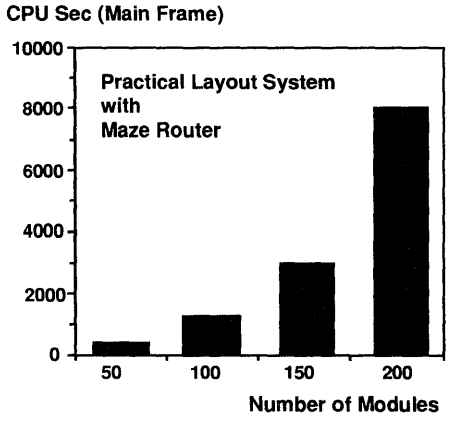
Figure 5.1: Example chip layout.
Figure 5.2: Computation speed.
Figure 5.3: PEs vs Speedup.
Table 5.1 : Scale of Co-HLEX.



1299 modules, 683 nets, 623 sec/Multi-PSI.64PEs
Figure 5.1 Bipolar Analog Circuit Layout by Co-HLEX

5.3 Considerations

(1) The possibility of parallel layout problem solving.
This has been proved through the experiment. As far as we know, Co-HLEX is the first system that can abut layouts - module shapes and wires - by runtime co-operation.
(2) Quality of Generated Layout: Wire length and chip area.
Through observations of Figure 5.1 we notice that both compact module placement and wires without useless bends could be generated. By the runtime wire abutment co-operation, traditional channel areas to patch inter-submodule wires could be diminished. This contributes to chip area reduction.
(3) Computational efficiency.
Figure 5.2 shows the performance of both Co-HLEX on Multi-PSI/64PE and a practical layout system on a main frame. Co-HLEX has a time complexity of nearly $O(N^{1.0})$. Here N is the number of modules in the circuit. For the 1299 moduled circuit, it took only 623 sec. This extraordinary outperforms the traditional system. Also, nearly linear peedup could be attained as shown in Figure 5.3.
(4) Program size and maintainability.
The 6,000-lined Co-HLEX remarkably outperforms the $O(10^5)$~$O(10^6)$-sized traditional implementations. See



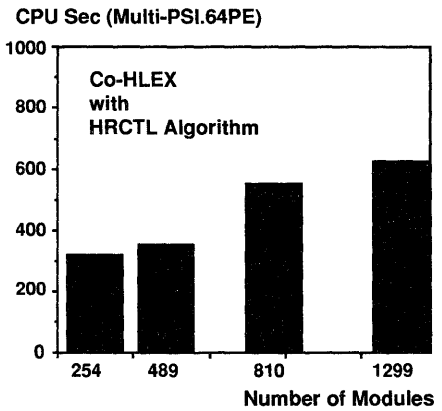


Figure 5.2 Problem Size vs Problem Solving Time

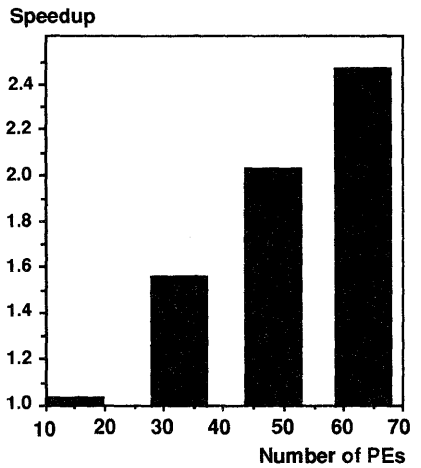


Figure 5.3 PEs vs Speedup

Table 5.1 Scale of Co-HLEX

Subsystems	KL1.Lines
Kernel	620
Planframes	2648
Layoutframes	1180
Layoutrules	684
Utilities	865
Total	5997

Table.5.1. This is due to the recursive HRCTL algorithm. Highly modularized program description was possible on streamed-parallel dataflow computation model offered by KL1.

6 Conclusions

6.1 Results

(1) A co-operative hierarchical layout problem solver named Co-HLEX was developed in FGCS project as an application program of Fifth Generation Parallel Inference Machines.
(2) The kernel algorithm of Co-HLEX is HRCTL which is a hierarchical recursive concurrent theorem prover for layout. Taditional wiring channels could be avoided due to its runtime co-operation to abut module shapes and wiring connectors.
(3) Due to the recursive nature of HRCTL, Co-HLEX is nearly 6,000-lined in KL1 which remarkably outperforms traditional LSI layout program implementations.
(4) Nearly $O(N^{1.0})$ time performance could be attained due to the streamed-parallel and distributed-memory architecture of Co-HLEX which greatly outperforms traditional methods.

6.2 New Problems

Programmers are in the well-known plan-do-see cycle. The non-repeatablity of parallel computation often destroyes the loop and deteriorates debugging efficiency. A programming environment for parallelism would be one of the most important issues to be studied further.

Acknowledgements

We acknowledge Dr. K.Furukawa, Dr. K.Nitta, Dr. K.Taki and other ICOT members for general supports, Prof. M.Iri of the University of Tokyo for advices on the problem solver design, Mr. M.Oda, Mr. H.Yamaguchi, Mr. K.Kawamura,

Mr. Y.Mori, and Y. Itoh of ISC Ltd. for assisting the program development, Mr. S.Domen and Dr.K.Ishihara; the general manager and the senior chief researcher of the Systems Development Lab. of Hitachi, for their supports of the research, Mr. K. Maio of the Semiconductor Design and Development Center, Hitachi Ltd., for assisting layout experiments, Dr. Y.Shiraishi of the Central Research Lab. of Hitachi for state of the art discussions, and Dr. S.Hayashi of Systems Development Lab. of Hitachi for discussions on analog circuit layout.

References

[Kleene 1952] S. C. Kleene. Introduction to Metamathematics, Van Nostrand,1952.

[Mandelbrot 1982] B. Mandelbrot. The Fractal Geometry of Nature, W. H. Freeman, 1982.

[Watanabe and Hayashi 1989] T. Watanabe, and S.Hayashi. Modelling Layout Problem Solving in Logic, in CAD Systems using AI Techniques, G. Odawara(ed.), pp.181-188, North-Holland, 1989.

[Samet 1984] H. Samet. The Quadtree and Related Hierarchical Data Structures, Computing Survey, Vol.16, No.2, June (1984), pp.187-260.

[Otten 1982] R. Otten. Automatic Floorplan Design, Proc. 19th DAC (1982), pp.261-267.

[Luk et al. 1986] W. K. Luk, D. T. Tang, and C. K. Wong. Hierarchical Global Wiring for Custom Chip Design, Proc. 23rd DAC (1986), pp.481-489.

[Lee 1961] C. Y. Lee. An Algorithm for Path Connections and its Applications, IRE TEC, September (1961), pp.346-365.

[Watanabe and Komatsu 1991] T. Watanabe and K. Komatsu. Co-operative Hierarchical Layout Problem Solver on Parallel Inference Machine, Proc. of the LPC'91, ICOT (1991) pp.9-24.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Cooperative Logic Design Expert System on a Multiprocessor

Yoriko Minoda, Shuho Sawada, Yuka Takizawa,
Fumihiro Maruyama, and Nobuaki Kawato

FUJITSU LIMITED
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan
pro114@flab.fujitsu.co.jp

Abstract

CAD systems that can quickly produce quality designs are needed for the expanding VLSI market. This paper presents a cooperative design mechanism in a cooperative logic design expert system on a multiprocessor, co-LODEX. co-LODEX accepts constraints on area and speed, and outputs a CMOS standard cell netlist that satisfies the constraints. The user can even get an optimal circuit for area or speed by iteratively strengthening the corresponding constraint. Short turnaround is expected through the combination of parallel processing by several processors and their cooperation.

The cooperative design mechanism is based on an evaluation-redesign mechanism using assumption-based reasoning within a single processor. Design alternatives are considered as assumptions and constraint violations as contradictions. Redesign is implemented as a contradiction resolution. The evaluate-redesign cycle repeats itself until the design satisfies the specified constraints. Global evaluation-redesign takes place by processors exchanging design results for subcircuits in terms of gate counts and delays (in case of success) or justifications for constraint violations (in case of failure).

Experimental results show that (1) co-LODEX can efficiently carry out global optimization. For example, a circuit with the minimum number of gates has been obtained while satisfying constraint on speed. (2) Linear speedup has also been observed.

1 Introduction

CAD systems that can produce quality circuits quickly are needed for the expanding VLSI market. One of the most pressing problems is the lack of a means to iterate the cycle of evaluation and redesign until the design satisfies all constraints. Without it, it would be impossible to design a quality circuit with the desired characteristics (area and speed) by looking at the design from a global point of view. There is also demand for CAD systems that can do global optimization for the whole circuit. With such systems, designers can get a circuit with the gate count minimized and the delays kept shorter than the given constraints or vice versa.

Turnaround time seems to be another key issue. Short turnaround allows designers to rapidly implement a variety of architectural choices and to choose the solution best suited for their specific situation by comparing area and speed characteristics. Designers can thus explore their options in a way that has not been practical before.

Since design decisions may be retracted after later evaluation, they can be thought of as assumptions. Assumption-based reasoning uses both facts and assumptions that can be retracted [de Kleer 1986]. Justification, originally introduced for truth maintenance [Doyle 1979], is the key concept to manipulating information containing assumptions. In de Kleer's Assumption-based Truth Maintenance System (ATMS), all assumptions are enumerated in advance and all combinations are examined. In design, however, we are not interested in all combinations. This is because a decision's significance depends on decisions made earlier. We can prune a considerable number of combinations.

A global optimization technique using as linear programming (LP) was proposed [Kageyama 1990]; however, we can not get the exact optimal circuit, because the solution does not always give 0's or 1's for variables that must take 0 or 1.

We proposed an evaluation-redesign mechanism using assumption-based reasoning [Maruyama 1988]. In our evaluation-redesign mechanism, design alternatives are considered as assumptions and constraint violations as contradictions. Redesign is implemented as contradiction resolution. Justifications for violations, called nogood justifications (NJs), play a central role in the mechanism. NJs enable us to drastically prune the search space for constraint satisfaction or optimization problems [Maruyama 1991].

In this paper, we present a cooperative logic design ex-

pert system on a multiprocessor, co-LODEX. co-LODEX divides the whole circuit to be designed into subcircuits in advance and designs each subcircuit on each processor to exploit parallel processing. Global evaluation-redesign takes place by processors exchanging design results (in case of success) or NJs (in case of failure). In our cooperative design mechanism, NJs received from other agents help narrow down the search space for an agent in the sense that NJs made out of the received ones enable the agent to prune the search space. That is the reason why we claim co-LODEX as "cooperative". Short turnaround is expected through the combination of parallel processing by several processors and their cooperation. co-LODEX also has the advantage of exact global optimization.

The next section gives an overview of co-LODEX. Section 3 describes its cooperative design mechanism. We give some experimental results in Section 4 and concluding remarks in Section 5.

```
UHDL;
interface_view: interface_example01;
 inputs: .xi(12), .yi(12), .dxi(12), .ui(12), .ai(12);
 outputs: .xo(12), .yo(12);

behavior_view: behavior_example01;
 define: const5 = 5, const3 = 3;
 terminal: u1(12), u2(12), u3(12), u4(12), u5(12), u6(12), y1(12), FF;
 operator: 2stage_pipelined_multiplier(x, y, z) = ( len = 2 ), z <- x * y; end_op;
 function: main: clk;
 while (FF) do
  2a: '2stage_pipelined_multiplier'(u, dx, u1);
  3a: '2stage_pipelined_multiplier'(x, const5, u2);
  4a: '2stage_pipelined_multiplier'(const3, y, u3);
  5a: '2stage_pipelined_multiplier'(u2, u1, u4),
      x <- x + dx;
  6a: '2stage_pipelined_multiplier'(u, dx, y1),
      FF <- x < a;
  7a: '2stage_pipelined_multiplier'(u3, dx, u5),
      u6 <- u - u4;
  8a: y <- y1 + y;
  9a: u <- u6 - u5, xo := x, yo := y;
 enddo;
 1a: stop(x<a), x <- xi, y <- yi, dx <- dxi, u <- ui, a <- ai;
endUHDL.
```

Figure 1. Example of behavioral specification

2 co-LODEX Overview

2.1 Inputs and Outputs

The user specifies a behavioral specification, a block diagram of the datapath, and constraints on area and speed. co-LODEX outputs a CMOS standard cell netlist that satisfies the constraints. The resulting netlist can be input to an automatic place-and-route system for CMOS standard cells.

The specification language for behavior used in co-LODEX is UHDL [Fujisawa 1989], an extension of DDL [Duley and Dietmeyer 1969]. Figure 1 shows the specification for a circuit that solves a second-order differential equation (DiffEQ). The program might be used to describe a subsystem of a controller or have a digital signal processing application. [Brewer 1987]

A block diagram of the datapath is shown in Figure 2. The boxes signify functional blocks. COMP, MULTI, ADD_SUB, MUX, REG, FF, and the others represent a comparator, a multiplier, an ALU(add/subtract), a multiplexer, a register, a flip-flop and input/output buffers.

Constraints on area are expressed as inequalities in the gate count, for example, "(Total gate count) ≤ 2000." The user can specify as an area constraint the maximum gate

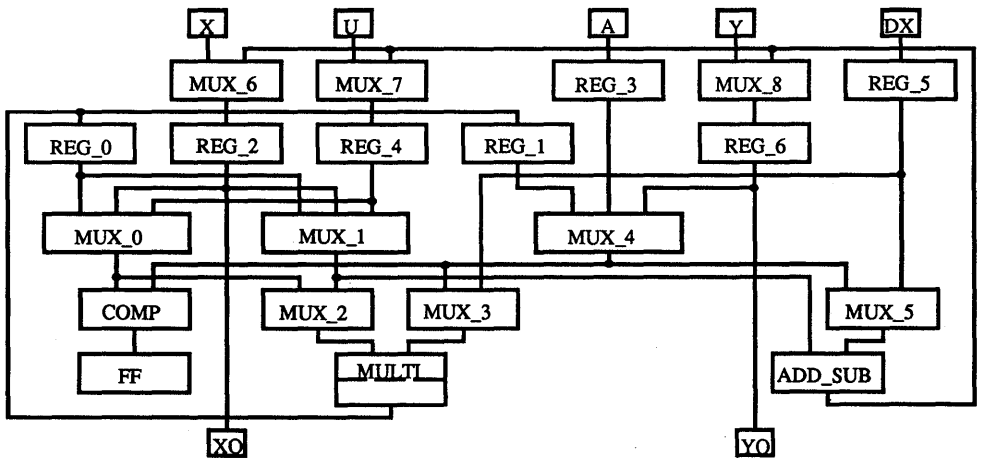


Figure 2. Block diagram

count that could be squeezed into a given LSI device. Constraints on speed are expressed as inequalities in the propagation delay, for example, "(Maximum delay) ≤ 120 ns." The user can specify as a timing constraint the clock cycle the LSI device should operate with.

2.2 Brief Overview

co-LODEX divides the whole circuit to be designed into subcircuits. Each subcircuit is designed by a design agent. Figure 3 shows the five subcircuits for the DiffEQ example and the agents in charge. It should be noted that the control circuit, CTRL, is included. co-LODEX establishes a finite-state machine from the behavioral specification and extracts the specifications for the control circuit in terms of logical expressions. It then divides the whole circuit so that the blocks along critical path candidates are distributed to as few agents as possible. It is likely that agents along a critical path candidate need a considerable amount of mutual communication since agents sharing a constraint must communicate with each other.

Each agent designs given functional blocks hierarchically using the top-down method. It keeps splitting up functional block and subblocks into sub-subblocks until all given blocks are implemented with CMOS standard cells. This is done by referring to the library that includes knowledge about functional block design, knowledge about technology mapping, and standard cells data. Then it counts the number of gates and estimates delays to evaluate the implemented circuit against constraints on area and time.

An agent usually designs its subcircuit independently and in parallel with the other agents. However, since the design results of the other agents are necessary for evaluation against global constraints, agents exchange their results every time they finish design/redesign. An agent redesigns when it detects a constraint violation for which it is responsible, for example, if a path passing through it is too slow. If it designs a standard cell netlist that satisfies all the local constraints, it notifies the resulting gate count and delays. If it cannot, it notifies information about constraint violation.

3 Cooperative Design Mechanism

We propose a cooperative design mechanism on a multiprocessor. It is based on the redesign mechanism within each agent. Moreover, (1) exchanging design results and NJs among agents and (2) combining the NJs received from other agents are necessary. Agents exchange the design results (gate counts and delays) of subcircuits when they succeed in design. They exchange the resulting NJs when they fail to design subcircuits without any stored NJ satisfied.

3.1 Redesign within Each Agent

The area a circuit requires and its delay are the sum of their constituent parts. The delay of a path, for example, can be attributed to that of the components along it. This fact lets us break a global condition into local conditions. A hierarchical structure is useful for this. We explain a redesign

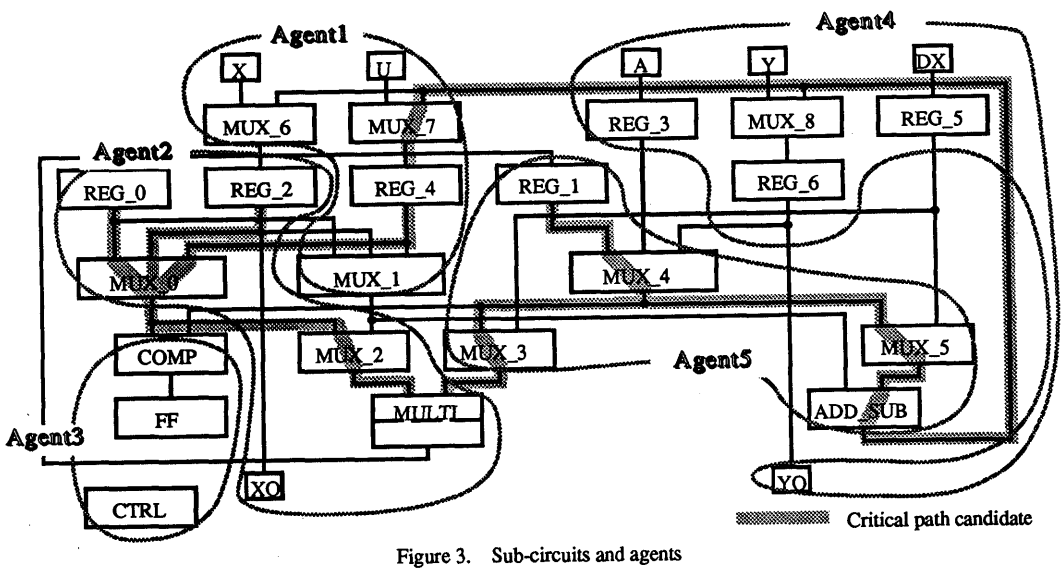


Figure 3. Sub-circuits and agents

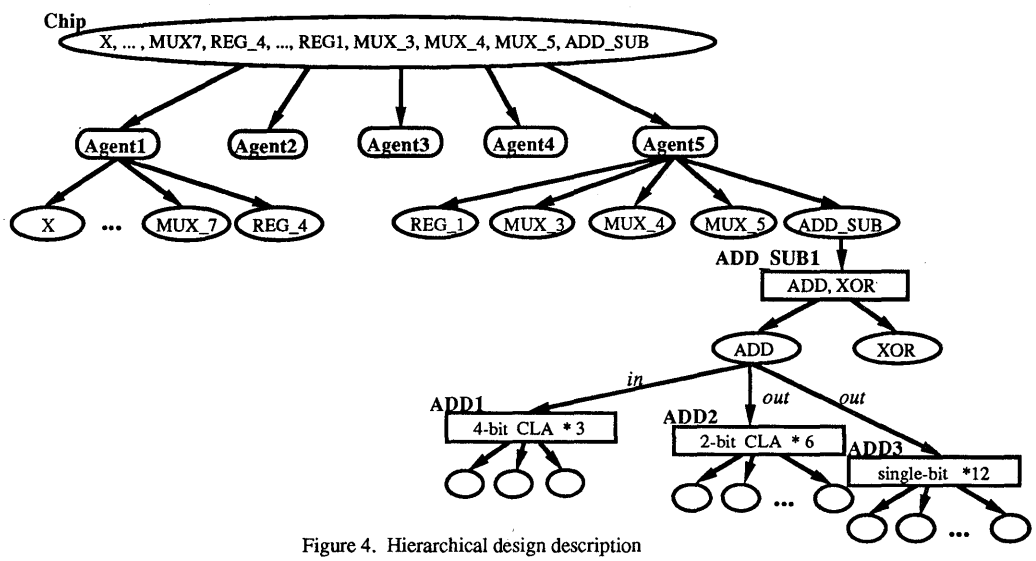


Figure 4. Hierarchical design description

mechanism using assumption-based reasoning, which operates on a hierarchical design description.

Hierarchical Design Description

Design objects are represented in a hierarchy. Figure 4 shows part of the hierarchy corresponding to Figure 3. There are three types of nodes; agent nodes (capsules), component nodes (ovals) and alternative nodes (rectangles). An agent node is responsible for one or more component nodes. A component node associates alternative nodes as possibilities of implementation. There is a special component node called the chip node that corresponds to the whole chip. An alternative node contains information about the connection between subcomponents and has the subcomponent nodes as children. An alternative is called either "*in*" or "*out*" based on whether it is adopted or discarded. Each component node has at most one *in* alternative node. Other alternative nodes are stored in the *out* alternative list to be recalled later if necessary.

Figure 4 shows the following:

- The whole chip (Chip) consists of an input buffer (X), registers (REG_1 and REG_4), multiplexers (MUX_3, MUX_4, MUX_5, and MUX_7), an add/subtract unit (ADD_SUB), and other parts.
- Agent5 is responsible for five components.
- ADD_SUB consists of an adder (ADD) and an exclusive or (XOR).
- ADD, the 12 bit adder, consists of three 4-bit CLA (carry-lookahead adder) cells connected serially. Current *out* alternatives might include a serial connection of six 2-bit CLA adder cells and 12 single-bit adder cells.

Justifications for Constraint Violations (NJs)

An NJ (nogood justification) is a logical expression that must not hold during design. Satisfying an NJ means a constraint violation and invokes the redesign mechanism.

The following default NJ at Chip (in Figure 4) is equivalent to the original constraint on gate count in that any design violating the constraint satisfies it.

$$X(a) + REG\_1(a) + REG\_4(a) + MUX\_3(a) + MUX\_4(a) + MUX\_5(a) + MUX\_6(a) + ADD\_SUB(a) + \ldots > CHIP \quad (1)$$

The form, "component ('a')", represents gate count of each component. This says that if the total gate count of the input buffer, the registers, the multiplexers, and so on, exceeds the value of variable CHIP, it means a constraint violation. CHIP is the variable that refers to the currently valid constraint value on gate count, for example 2000. co-LODEX transforms each constraint specified by the designer into default NJs.

A timing constraint in terms of the clock cycle is transformed into a set of default NJs, that is, an inequality representing that the sum of the delays of the components along a path from source to destination exceeds the constraint value. For example, one of the default NJs represents that the path

from REG_1 via MUX_4, MUX_5, ADD_SUB, MUX_7, to REG_4 is longer than the clock cycle. It is as follows:

REG_1(p2) + MUX_4(p1) + MUX_5(p2) + ADD_SUB(p2) + MUX_7(p2) + REG_4(p1) > CLOCK (2)

The form, "component ('p' number)", represents a path within each component. CLOCK is the variable that refers to the currently valid constraint value of the clock cycle, for example, 120.

Starting from default NJs, new NJs are added during redesign through NJ expansion and generation as described below. NJs save us doing direct evaluation against constraints. All we have to do is to check to see if any NJ is satisfied.

NJ Expansion

NJ expansion is used to narrow the scope and go down the hierarchy to resolve contradictions, or constraint violations. NJ expansion is formally defined in the following three steps. The NJ to be expanded is the one that is satisfied at the moment.

Step 1: Select a component appearing in the NJ to be expanded. Call it C.

Step 2: Replace C in the NJ with its *in* alternative's subcomponents. If the *in* alternative is at the leaf of the hierarchical structure (at the standard cell level), replace C with its actual gate count or its delay value.

Step 3: Go down the hierarchy to the alternative node and store the NJ obtained in Step 2.

(End)

NJ Generation

If every alternative of a component causes a constraint violation, NJ generation enables us to get a new NJ, the logical product of the NJs corresponding to each alternative. The generated NJ does not refer to that component. It is put at the alternative node one level up. This procedure is justified by resolution [Robinson 1965]. In general, the generated NJ is a logical product of NJs about gate count and NJs about delay.

Evaluation-Redesign Algorithm within Each Agent

The redesign algorithm within each agent uses NJ expansion and generation. Redesign is invoked when an NJ turns out to be true, since satisfying an NJ means a constraint violation.

Step 1: Set ALT to the agent node and proceed to Step 2.

Step 2: Check to see if there is any satisfied NJ at the ancestor alternative nodes (including itself) of ALT. If so, set ALT to the alternative node where the satisfied NJ is put, and proceed to Step 3. Otherwise, go to Step 7.

Step 3: If there is a subcomponent of ALT appearing in the NJ, proceed to Step 4. Otherwise, go to Step 5.

Step 4: Expand the NJ. Set ALT to the current alternative node and return to Step 3.

Step 5: Make ALT *out*. Select another alternative node that is not inhibited by an NJ, make it *in*, set ALT to it, and go to Step 2. If every alternative is inhibited by NJs, proceed to Step 6.

Step 6: Generate an NJ. Set ALT to the current alternative node and go to Step 3. If there is no alternative node one level up, output the generated NJ and exit (Fail!).

Step 7: If there is no component node whose alternative nodes are all *out*, exit. (Succeed!). Otherwise, select an alternative node that is not inhibited by NJs, make it *in*, set ALT to it, and go to Step 2.

(End)

In Step 5, selection is done either by recalling an *out* alternative or by generating a new implementation.

The above algorithm starts when an agent receives information from the other agents. Once the algorithm terminates in success or failure, the agent sends information to the other agents.

3.2 Cooperative Design Algorithm

We propose a cooperative design algorithm by describing the procedure for each agent.

Step 1: Design its subcircuit. Repeat redesign by the evaluation-redesign algorithm. The gate counts and delays of the other subcircuits are assumed to be 0. If any agent fails, the algorithm terminates in failure. Otherwise, proceed to Step 2.

Step 2: Exchange the design results, that is the gate counts and delays of the subcircuits, with the other agents. Proceed to Step 3.

Step 3: Set the gate counts and delays of the other subcircuits to the design results received in Step 2. If no stored NJ is satisfied, go to Step 9. If some of the stored NJs are satisfied and the design results of each agent are the same as in the previous cycle (caught in a loop), go to Step 7. Otherwise, proceed to Step 4.

Step 4: Redesign its subcircuit. If at least one agent succeeds in redesign without any stored NJ satisfied, go to Step 2. Otherwise (all agents fail), proceed to Step 5

Step 5: Exchange the generated NJs with the other agents. Proceed to Step 6.

Step 6: Combine the NJs received in Step 5. Go to Step 1.

Step 7: Set a temporary constraint and proceed to Step 8.

Step 8: Design its subcircuit. Repeat redesign by the evaluation-redesign algorithm until all the constraints including the temporary one are met. The gate counts and delays of the other subcircuits are assumed to be 0. If all the agents fail, the algorithm terminates in failure. Otherwise, go to Step 2.

Step 9: Put together all the subcircuits. The algorithm terminates in success.

(End)

Only default NJs are stored initially. As the algorithm proceeds, new generated NJs and combined NJs are added. In Step 7, select one of the violated constraints with the fewest agents related, and set the current value corresponding to that constraint as a temporary constraint.

Once the above algorithm terminates in success or failure (In Step 1, Step 8, and Step 9), the design run is finished, and the user can retry by changing the constraints. The user can look for a faster circuit by tightening the delay constraint, or can rerun by relaxing the constraints in case of failure. When the constraints are changed, the system updates them and re-evaluates by checking all the stored NJs. As more NJs are accumulated, the efficiency of the algorithm is further improved.

3.3 Combining NJs

When an agent fails in redesign with the evaluation-redesign algorithm described in the above section, it generates an NJ and sends it out to the agents that share it. Each agent "combines" the NJs received from other agents and makes a new NJ out of them. Considering an NJ from an agent as a condition where design is impossible for the agent, the combined NJ can be seen as a condition where design is impossible for agents other than the recipient agent. Agents are required to design without any combined NJ satisfied.

For example, suppose Agent5 received the following two generated NJs (3) and (4) originated from default NJ (1) and (2) from Agent1 and Agent4, respectively ("∧" signifies logical product):

$$192 + \text{Agent2(a)} + \text{Agent3(a)} + \text{Agent4(a)} + \text{Agent5(a)} > \text{CHIP}$$
$$\wedge\ 19.2 + \text{Agent5(p1)} > \text{CLOCK} \quad (3)$$

$$\text{Agent1(a)} + \text{Agent2(a)} + \text{Agent3(a)} + 96 + \text{Agent5(a)} > \text{CHIP} \quad (4)$$

Agent5 combines the above NJs and makes the following new NJs:

$$288 + \text{Agent2(a)} + \text{Agent3(a)} + \text{Agent5(a)} > \text{CHIP}$$
$$\wedge\ 19.2 + \text{Agent5(p1)} > \text{CLOCK} \quad (5)$$

$$96 + \text{Agent2(a)} + \text{Agent3(a)} + \text{Agent5(a)} > \text{CHIP} \quad (6)$$

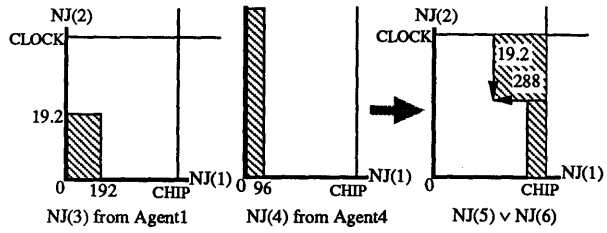


Figure 5. Example of combining NJs

(5) and (6) are added to Agent5.

Figure 5 illustrates the above. The two axes of each graph correspond to default NJs (1) and (2). NJ (3) means that any design by Agent1 is either 192 gates or more, or 19.2 ns or longer along the path for default NJ (2). The left graph shows that Agent1 cannot design inside the hatched part. Similarly, the middle graph shows that Agent4 cannot design inside the hatched part. Combining (3) and (4) gives a condition for the agents other than Agent1 and Agent4 to be unable to design without violating any constraint. If the agents other than Agent1 and Agent4 design inside the hatched part of the right graph, that will cause constraint violation. NJs (5) and (6) represent the hatched part.

4 Experimental Results

We implemented co-LODEX on Multi-PSI [Taki 1988] in KL1 [Ueda 1986] to evaluate the performance of the cooperative design mechanism, and tested as examples to design a specific circuit and usual circuits.

4.1 Optimization

Optimization using co-LODEX proceeds as follows: First, co-LODEX requests the user for area and speed constraints and produces a solution satisfying the constraints. The user then changes area or speed constraint value to the value for the solution just obtained minus 1, and iterates as long as the constraints are satisfied. If constraint satisfaction fails, the previous solution is used as the optimal solution.

Figure 6 shows some of the results for the MAG example. MAG approximates $(a^2 + b^2)^{1/2}$. At first, the area constraint was large enough, and the timing constraint was 130. We obtained the circuit shown at the right. As the area constraint was strengthened, different results were achieved. The smallest circuit, we find is shown at the left. Finally, the above optimization failed in constraint satisfaction with NJ, 1224>CHIP. This means that design is impossible if

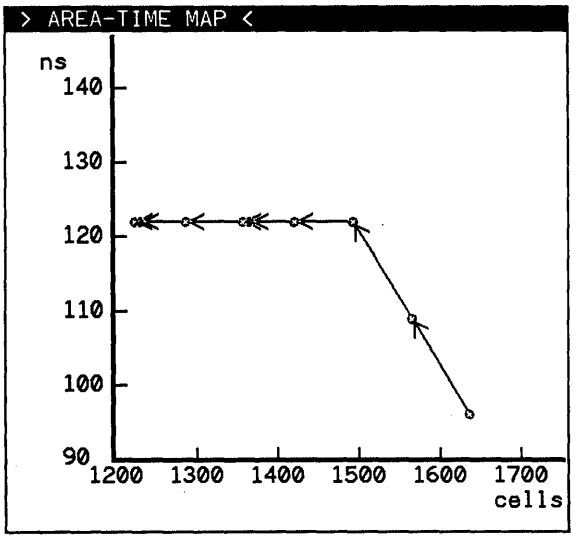


Figure 6. Experimental result for MAG circuit

the specified set of constraints satisfies the NJ. We must thus relax the constraints so that the above NJ is not true any more.

4.2 Speedup

Speedups were examined by increasing the number of agents from 1 to 15. Agents correspond to processors on a one-to-one basis. We had one extra processor for distributing the functional blocks to the other processors and taking statistics, so we used up to 16 processors altogether. We expected that speedups would increase in proportion to the number of agents.

Specific Circuit

The example presented here is to design a multi-argument adder (array adder). The function of this circuit is to calculate the sum of nine integers represented in two's complement format. This circuit is adopted in ALUs and multipliers in other example circuits described below. This circuit consists of 122 one-bit adders. The function of a one-bit adder is to calculate the sum and the carry-out of one-bit integers. Each one-bit adder has many design methods, so the whole circuit has over 50 million design combinations. Table 1 lists the number of design methods with the number of inputs and outputs. Each one-bit adder can be implemented with CMOS standard cells immediately. Thus, we have tested only the cooperative design mechanism of co-LODEX. We used 30 default NJs.

Figure 7 shows a part of this circuit. The boxes represent one-bit adders and the number inside them represent the number of input bits. The arrows represent default NJs. The upper and lower side or upper-left and lower-right side of the arrangement of neighboring blocks has a relationship to the same default NJ. Accordingly, co-LODEX divided the whole circuit and the boundary lines between subcircuits as vertical or slanting (from upper-left to lower-right).

We averaged design costs to agents in this test. We assumed that design costs depend on the total number of design methods for the agent in charge. Taking the number of design methods into account, co-LODEX divided the whole circuit into subcircuits as many as agents. The shaded areas in Figure 7. show two of the subcircuits, where the number of agents is 10. co-LODEX can easily divide this circuit with agents, since it is orderly.

The relation between the number of agents and the speedups is shown in Figure 8, which shows a change in

Table 1. The number of combinations for design method

inputs	sum	carry-out	number of design methods
1	1	0	1
2	1	1	1
3	1	1	1
4	2	1	12
5	2	2	30
6	2	2	15
7	3	2	105
8	3	3	420
9	3	3	84

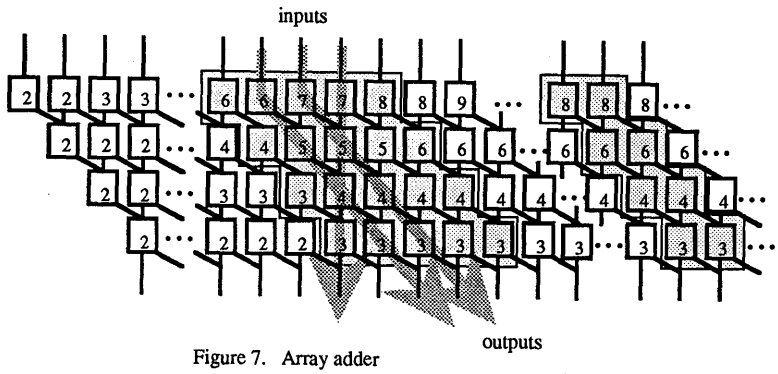


Figure 7. Array adder

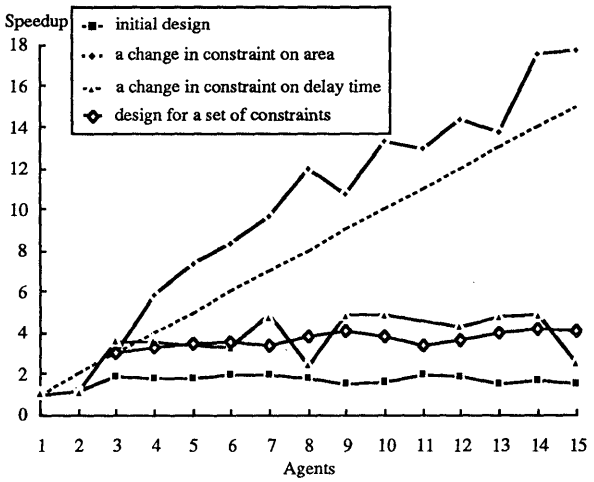


Figure 8. Relation between the number of agents and speedup

design time according to the number of agents. The slanting straight line represents the ideal line. All agents are active in consequence of a change in area constraint, while some agents are active and others are inactive in consequence of a change in delay time constraint. A change in area constraint thus increases speedups and the result surpasses the ideal line. The reason seems to be that our cooperative mechanism reduces the amount of computation by saving useless combinations of alternatives from each agent. Initial design time, time taken until evaluation-redesign occurs, is roughly constant, because the increase in distribution work of the entire specification to agents cancels out the decrease in each agent's design work due to an increase in the number of agents. Figure 8 also shows the speedups for a design, including initial design, when a set of constraints are given.

Usual Circuits

Table 2 lists the results of speedups for design of six usual circuits, including initial design, when a set of constraints are given, together with the optimal number of agents and the time.

A block diagram of the datapath includes various functional blocks. Some functional blocks such as ALU are complex, and others are simpler. We observed that one or two special agents work hard but that the other agents spend time waiting for messages from busy agents. Processing time depends on the busy agents which manage complex functional blocks.

To take advantage of our cooperative design mechanism on a multiprocessor, distribution strategy would need, in addition to focusing on critical path candidates, (1) to look ahead in the library when distributing the functional blocks, and (2) to set up sub-agents if necessary.

5 Conclusion

We presented a cooperative logic design expert system on a multiprocessor, co-LODEX. co-LODEX divides the whole circuit to be designed into subcircuits in advance and designs each subcircuit on each processor to take advantage of parallel processing. Global evaluation-redesign takes place by processors exchanging design results or NJs. A cooperative design algorithm based on assumption-based reasoning makes this possible. Short turnaround is expected through the combination of parallel processing by several processors and their cooperation.

co-LODEX can efficiently carry out global optimization. For example, a circuit with the minimum number of gates has been obtained while satisfying constraint on speed. By

Table 2. Results of experiments

Circuit	Number of Components	Main Components	Speedup	Optimal # of Agents	Time(sec)
Greatest common devisor	11	1 subtracter, 1 comparator	1.1	2	1.7
Differential equation $y'' + 5xy' + 3y = 0$	28	1 multiplier, 1 ALU(add/subtract) 1 comparator	1.3	3	72
MAG(1)	14	1 ALU(add/subtract), 1 comparator 1 two's complementer	1.7	4	3.3
MAG(2)	13	1 ALU(add/subtract/compare) 1 two's complementer	1.2	3	6.4
MAG(3)	16	1 adder, 1 subtracter, 1 comparator 2 two's complementers	5.0	15	3.7
Correlational function $y[i] = \sum_{j=1}^{N-1-j} x[j] * x[i+j]$	22	RAMs, 1 ALU(multiply/add) 1 adder, 1 comparator 1 decrementer, 1 incrementer	2.1	4	113

MAG: Approximation of $(a^2 + b^2)^{1/2}$

increasing the number of agents up to 15, the best linear speedup has been observed.

Our future plans include working on parallel processing of design, evaluation, and redesign within an agent. Distribution strategy is also important for load balancing among processors.

Acknowledgments

This work has been done as part of the Fifth Generation Computer Systems (FGCS) Project of Japan. We would like to thank Dr. Nitta, manager of the Seventh Laboratory of ICOT, for his support.

References

[de Kleer 1986] J. de Kleer: "An Assumption-Based Truth Maintenance System," Artificial Intelligence 28, pp.127-162 (1986).

[Doyle 1979] J. Doyle: "A Truth Maintenance System," Artificial Intelligence 24 (1986).

[Kageyama 1990] N. Kageyama et al.: "Logic Optimization Algorithm by Linear Programming Approach," Proc. of the 27th Design Automation Conference, pp.345-348 (1990).

[Maruyama 1988] F. Maruyama et al.: "co-LODEX: a cooperative expert system for logic design," Proc. of FGCS'88, pp.1299-1306 (1988).

[Maruyama 1991] F. Maruyama et al.: "Solving Combinatorial Constraint Satisfaction and Optimization Problems Using Sufficient Conditions for Constraint Violation," Proc. of the Fourth International Symposium on Artificial Intelligence (1991).

[Fujisawa 1989] H. Fujisawa et al.: "UHDL (Unified Hardware Description Language) and its support tools," Int. J. Computer Aided VLSI Design (1989).

[Duley and Dietmeyer 1969] J. R. Duley and D. L. Dietmeyer: "A digital system design language (DDL)," IEEE Trans. Computers, Vol.C-17, No. 19, pp.850-861 (1968).

[Brewer 1987] F. D. Brewer: "Knowledge Based Control in Micro-Architecture Design," Proc. of the 24th Design Automation Conference, pp.203-209 (1987).

[Robinson 1965] J. A. Robinson: "A Machine Oriented Logic Based on the Resolution Principle," Journal of the ACM, Vol.12, No.1, pp.23-41 (1965).

[Taki 1988] K. Taki: "The Parallel Software Research and Development Tool: Multi-PSI system," Programming of Future Generation Computers (1988).

[Ueda 1986] K. Ueda: "A Parallel Logic Programming Language with the Concept of a Guard," ICOT Technical Report, TR-208 (1986).

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

A Parallel Inductive Learning Algorithm for Adaptive Diagnosis

Yoichiro Nakakuki Yoshiyuki Koseki Midori Tanaka
C&C Systems Research Laboratories, NEC Corporation

4-1-1 Miyazaki, Miyamae-ku, Kawasaki 216, JAPAN
E-mail: nakakuki@btl.cl.nec.co.jp

Abstract

This paper describes a parallel inductive learning algorithm for adaptive model-based diagnosis. Although the model-based systems are more robust than the rule-based systems, they require more computation time. This is because they lack heuristic knowledge. On the other hand, human experts can learn and utilize such knowledge from experience. Therefore, in order to realize efficient model-based diagnosis, learning capability from experience is indispensable. We had proposed an inductive learning mechanism but unfortunately it took much computation time. In order to reduce the computation time, this paper proposes a parallel learning algorithm. The experiential knowledge is represented as a fault probability model and the proposed algorithm searches the most appropriate one out of all the possible models. In order to search effectively, a partial order is introduced into the search space. By using this ordering, two kinds of search control mechanisms, that are local pruning and global pruning, are developed. The algorithm is implemented in KL1 language on a parallel inference machine, Multi-PSI. The experimental results show the effectiveness of the mechanisms. It is also shown that the 16 PE implementation is about 11 times as fast as the sequential one.

1 Introduction

Since the creation of the MYCIN system[Shortliffe 1976], most of expert systems, have incorporated the idea of representing their knowledge in a form of symptom-failure association rules. Those expert systems that take rule-based approach have two major inherent disadvantages. First, those systems lack robustness because they cannot deal with unexpected cases which are not covered by rules in their knowledge bases. Second, their knowledge bases are expensive to be created and maintained.

There has been a series of research to tackle those problems. The most distinct ones are on model-based methods, i.e. first-principle methods. Model-based methods use design descriptions, such as structure and behavior descriptions [Davis 1984, de Kleer 1987, Genesereth 1984].

However, model-based diagnostic systems are generally not as efficient as rule-based ones since they require more complex computation. This is because they lack heuristic knowledge which human experts usually utilize. We have been working on a research to explore a general architecture to realize an adaptive diagnostic agent and introduced its basic architecture[Koseki 1989]. Moreover, an experimental system based on the architecture [Koseki et al. 1990a, Koseki et al. 1990b, Ohta et al. 1991a, Ohta et al. 1991b] have been developed. The system realizes adaptability with learning capability from its experience. The experiential knowledge is represented in a form of a fault probability model of target system components. With this experiential knowledge, it is able to diagnose a failing component faster with a fewer tests than pure model-based systems.

However, it takes much computation cost to learn experiential knowledge. This is because the hypothesis space to search grows rapidly with the size of the target problem. In order to reduce the computation time, we developed a parallel learning algorithm.

The algorithm utilizes two kinds of search control mechanism, that are local pruning and global pruning. The search space is divided and assigned to each processor so that the transmission of local pruning information does not require interprocess communication. The interprocess communication is restricted to the plausible global pruning information.

The algorithm is implemented in KL1 language on a parallel inference machine, Multi-PSI. The experimental results show that the implementation using 16 PEs is about 11 times as fast as the sequential one.

Section 2 presents the mechanism of the adaptive diagnostic system. In section 3, the probabilistic-model learning problem is described. A parallel learning algorithm is presented in section 4, and experimental results are shown in section 5.

2 Adaptive Diagnosis Mechanism

This section presents the architecture of an adaptive model-based diagnosis. We can observe two kinds of intelligent behavior in maintenance expert's diagnostic procedure. First, they can quickly identify a faulty component with a little information utilizing their experience. Second, even if a novel symptom arises, the expert can reach a conclusion, by consulting with other information sources, such as design description manuals. They can reason which component might have gone wrong and caused the symptom to appear, by knowing how the system is supposed to work.

To realize those kinds of intelligent behavior, the system consists of several modules as shown in Figure 2–1. The knowledge base consists of *design knowledge* and *experiential knowledge*. The design knowledge represents a correct model of the target device. It consists of structural description which expresses component interconnections and behavior description which expresses component behavior. The experiential knowledge is expressed as component failure probability for each component.

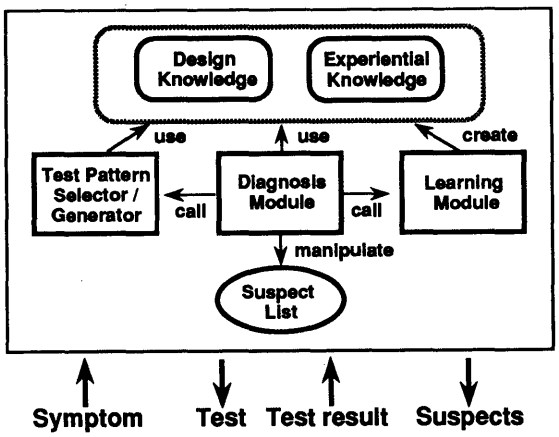


Fig. 2–1 Structure of the System

The general flow of the diagnostic system is shown in Figure 2–2. The system keeps a set of suspected components as a suspect-list. And it takes *eliminate-not-suspected* strategy[Tanaka et al. 1989] to reduce the number of the suspects in the suspect-list, repeating the test-and-eliminate cycle.

It starts with getting an initial symptom. It calculates an initial suspect list from the given initial symptom by performing a model-based reasoning. After obtaining the initial suspect-list, the system repeats a test-and-eliminate cycle, while the number of suspects is greater than one and an effective test exists. A set of tests is generated by the test pattern generator. Among the generated tests, the most cost effective one is selected as the next test to be performed. The selected test is suggested and fed into the target device. By feeding the test into the target device, another set of observation is obtained as a test result and is used to eliminate the non-failure components.

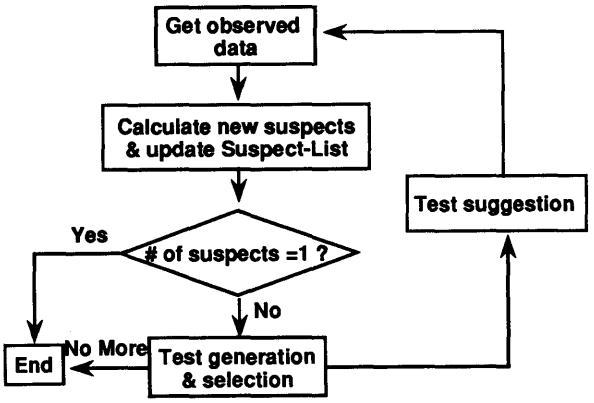


Fig. 2–2 Diagnosis Flow

In order to compute test effectiveness, the system uses fault probability distribution for each component. The mechanism employed in the system is basically same as the one found in the reference [de Kleer 1989]. It is so called *minimum entropy technique* where entropy is calculated from the fault probability for each suspected component. Here, an entropy $E(SL)$ of a suspect-list SL is defined in terms of the estimated probabilities of each component in the list. Let SL denote the set of suspected components,

$$SL = \{S_1, S_2, \ldots S_n\},$$

and let $p_1, p_2, \ldots p_n (\sum p_i = 1,\ p_i > 0)$ be failure probabilities of suspects $S_1, S_2, \ldots S_n$. Then an entropy $E(SL)$ is defined as

$$E(SL) = -\sum_{i=1}^{n} p_i \log p_i.$$

The system evaluates $gain(T)$ for all of the available tests. In addition to this value, the system considers the test execution cost to select a cost effective test. The system selects a test according to the following evaluation function.

$$gain(T)/cost(T)$$

At first, the diagnostic system does not know the probability distribution for a target device. Therefore, it should assume that the all of the components have the same fault probability. However, the system becomes efficient as it acquires information on the fault probability from its experience.This is because it can estimate more precise probability distribution and can generate more effective test sequence. In the next section, a learning mechanism is presented.

3 Learning Probabilistic Models

The performance of the diagnostic mechanism relies on the correctness of the presumed probability distribution of components. However, it is not easy to predict appropriate probability for each component from observed data, especially when the number of observed data is small.

For example, consider a diagnosis of a network system with 100 modems and 100 communication terminals. Here, we assume that 10 modems have broken down in the past (once for each). A simple estimate concludes that each of the 10 components has higher fault probability than any other component. However, human may presume that a modem has higher fault probability than a terminal because modems have broken 10 times in the past and terminals have never broken. Therefore, it is important to select an appropriate estimation method to derive a precise probability distribution (probabilistic model).

Here, we consider an example of a target device which consists of 16 components. The observed number of faults for each component is shown in Table 3–1. Several attributes for each component are also shown in the table.

Table 3–1 Example

Component	Attributes		No. of Obs. (Times)
	Type	Age	
1	a	new	1
2	a	old	0
3	b	new	13
4	b	old	9
5	c	new	1
6	c	old	1
7	d	new	0
8	d	old	0
9	e	new	0
10	e	old	0
11	f	new	1
12	f	old	0
13	g	new	0
14	g	old	5
15	h	new	1
16	h	old	0

First, we consider the relationship between the component type and the fault frequency. A type b component seems to have a very high fault probability. And it may be natural to conclude that type g component has also slightly higher probability than the other components. On the other hand, it is dangerous to conclude that each of the other components has different fault probability, e.g., the fault probability of type c component is about twice as large as type a component's. Because the difference between the number of observation may be due to an accident.

Next, we consider the relationship between component age and the fault probability. In the example, it seems that the component age does not affect the fault probability. Therefore, in order to estimate the fault probability distribution precisely, it is important to consider component type.

In general, some attributes are important to estimate the fault probability and the other attributes are not so important. Moreover, a combination of several attributes may be important. For instance, in the above example, we had better to consider component age, in the case of the component type is g.

In order to estimate the probability distribution precisely, we must find relevant attributes (and/or their combination) and consider how to estimate with those attributes.

Here we define the presumption problem. Consider a set of events $X = \{x_1, x_2, ..., x_m\}$ and attributes $a_1, a_2, ..., a_n$. Here, we assume that the events are exhaustive and mutually exclusive, and that the domain for each attribute a_j $(j = 1, 2, ..., n)$ is a finite set $Dom(a_j)$. As shown in Table 3–2, for each event, x_i, a value, v_{ij} $(\in Dom(a_j))$, for each attribute, a_j, is given. Also, n_i, the number of observations is given.

Table 3–2 Table of events

Event	Attributes				No. of Obs. (times)
	a_1	a_2	$\cdots$	a_n	
x_1	v_{11}	v_{12}	$\cdots$	v_{1n}	n_1
x_2	v_{21}	v_{22}	$\cdots$	v_{2n}	n_2
x_3	v_{31}	v_{32}	$\cdots$	v_{3n}	n_3
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
x_m	v_{m1}	v_{m2}	$\cdots$	v_{mn}	n_m

The problem is to presume the probability $\hat{p}_i$ for each event x_i, from the number of observations n_i. If enough amount of data are given, it seems to be easy to estimate the probability appropriately. However, if only a few observation data are given, we must consider the noise affection. Therefore, it is important to extract reliable information by avoiding the noise affection. In order to estimate the fault probability appropriately, we introduced an inductive learning mechanism [Nakakuki et al. 1990, Nakakuki et al. 1991b, Nakakuki et al. 1991c].

In the learning mechanism, a *presumption tree* is used to express a probabilistic model. Using a presumption tree, all the events are classified into several *groups*. Here, each event in a group is assumed to have the same fault probability. Therefore, the probabilities for individual events can be calculate from a presumption tree. The details are described below.

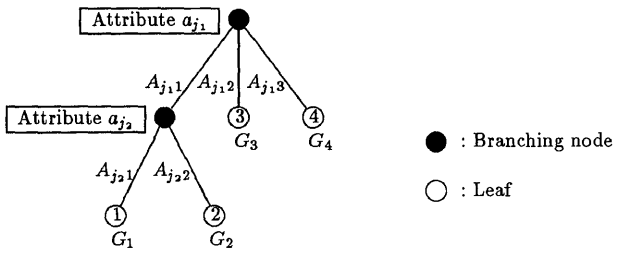


Fig. 3–1 Presumption tree

As shown in Fig. 3–1, a presumption tree consists of several branching nodes and leaves. An attribute a_j corresponds to each branching node, and subset A_{jk} of $Dom(a_j)$ corresponds to each branch. Here, each A_{jk} must satisfy the following conditions.

$$A_{ij} \cap A_{ik} = \phi \quad (j \neq k)$$

$$\bigcup_j A_{ij} = Dom(a_i)$$

A presumption tree classifies all possible events into several groups. For example, the tree in Fig. 3–1 has four leaves, therefore, the events are classified into four groups by using the tree as a decision tree[Quinlan 1986].

For example, G_1 is a group of events which corresponds to leaf 1. Each event in G_1 is considered to have the same fault probability. Here, for each leaf k, let its corresponding group of event be X_k, and for all event $x_i \in X_k$, let the sum of n_i be O_k. By using a presumption tree, the probability $\hat{p}_i$ for each event $x_i \in X_k$ can be calculated as follows:

$$\hat{p}_i = \frac{1}{|X_k|} \cdot \frac{\sum_{x_i \in X_k} n_i}{\sum_{x_i \in X} n_i}$$

As shown above, a presumption tree represents a probabilistic model. The problem is to find the most appropriate presumption tree for given data.

As a criterion for the selection, we introduced the minimum description length (MDL) criterion[Rissanen 1978, Rissanen 1983, Rissanen 1986]. Rissanen argued that the least description length model is expected to fit for presuming the future events better than any other models. Here, description length for a model is defined as the sum of the model-complexity and model-fitness for the given data. The description length of a presumption tree is the sum of:

(1) Code length of a tree, and

(2) Log-likelihood(distance) between the tree and observed data.

The code length(model complexity), L1, for a presumption tree is defined as follows[Nakakuki et al. 1991b].

$$L1 = \sum_{x \in P \cup Q} \log(n - d_x + 1) + \sum_{x \in Q} \frac{1}{2} \log O_x + \sum_{x \in P} \{\log(k_x - 1) + \log \mathrm{c1}(k_x, l_x)\}$$

Here, P is a set of all the branching nodes and Q is a set of all the leaves. For each branching node x, l_x is the number of branches, d_x is the depth of the node, $k_x = |Dom(a_i)|$ (a_i is a corresponding attribute for node x), n is the number of attributes, and $\mathrm{c1}(k_x, l_x) = \binom{k_x}{l_x - 1}$ if $l_x < k_x$, otherwise 1. On the other hand, log-likelihood (model fitness), L2, between a model and observed data is defined as follows.

$$L2 = -\sum_i n_i \log \hat{p}_i$$

Here, $\hat{p}_i$ is the presumed probability that is derived by using the model. The total code length is the sum of L1 and L2.

4 A Parallel Learning Algorithm

4.1 Local Pruning Mechanism

As described in the previous section, the problem is to search the least description length tree out of all the possible presumption trees. A heuristic algorithm for the problem was implemented[Nakakuki et al. 1991b] for a sequential machine by using branch-and-bound technique. The following summarizes the algorithm and then proposes a parallel version of the algorithm.

Here, let the length of a presumption tree T be denoted by $L(T)$. It is the sum of model complexity($L1(T)$) and the model fitness ($L2(T)$). Intuitively, a large tree has large model-complexity, and a small tree has large(bad) model-fitness[Nakakuki et al. 1991d]. In order to discuss such characteristics more precisely, we introduce a partial order "$\succ$" among the possible presumption trees. The order is defined as:

$T_2 \succ T_1 \underset{\text{def}}{\iff}$ Presumption tree T_2 can be obtained by replacing some leaves in presumption tree T_1 with branching nodes.

For example, presumption tree T_b in Fig. 4–1 can be obtained by replacing leaf z in T_a, therefore,

$$T_b \succ T_a.$$

Similarly,

$$T_c \succ T_a \ \textit{and} \ T_c \succ T_b.$$

Intuitively, $T_2 \succ T_1$ means T_2 is strictly larger than T_1.

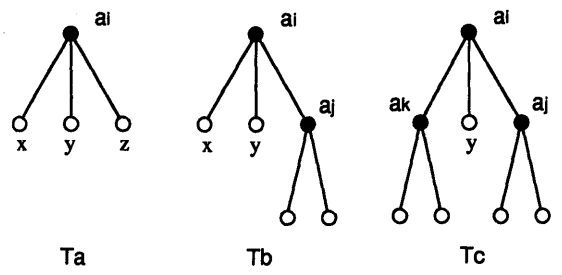


Fig. 4–1 Example

If $T_2 \succ T_1$, then the following inequalities hold by the definition:

$$L1(T_1) \leq L1(T_2)$$
$$L2(T_1) \geq L2(T_2)$$

Therefore, for a certain presumption problem, if a presumption tree T is a maximal one under the ordering, then $L2(T)$ will take the least value, say $L2_{MIN}$. $L2_{MIN}$ can be easily calculated in advance. By using these characteristics, we can effectively find a least description length tree.

The proposed algorithm searches the space of possible presumption trees. It tests simpler tree before testing more complex ones. That is, if there are two presumption trees T and T' such that $T' \succ T$, the system calculates the length of T before trying T'.

Here, consider that the length of a tree T has been tested. Then, the system considers the necessity of testing T' which is more complex than T (i.e. $T' \succ T$). If it turned out to be unnecessary(i.e., there is no possibility that T' has shorter length than T), then all the trees which are more complex than T' also turns out to be unnecessary to examine. The details of this technique are as follows.

In order to decide the necessity, the algorithm tests the following pruning condition:

$$\log\,(n - d_x + 1) + \log(k_x - 1) + \log\; c1(k_x, l_x)$$
$$+L2_{MIN} - L2(T) > 0$$

Here, x is one of the leaves in T and its corresponding node in T' is a branching node. If the inequality holds, it is not necessary to calculate the length for T'.

proof First, it is clear that the following inequality holds by the definition of L1:

$$L1(T') - L1(T)$$
$$\geq \log\,(n - d_x + 1) + \log(k_x - 1) + \log\; c1(k_x, l_x).$$

Second, the following inequality holds obviously:

$$L2(T') - L2(T) \geq L2_{MIN} - L2(T).$$

Here, if the sum of the right hand sides of the above two inequalities is positive(i.e., the pruning condition holds), then the sum of the left hand sides will be positive. Hence,

$$L1(T') + L2(T') > L1(T) + L2(T).$$
$$\text{i.e. } L(T') > L(T).$$

Therefore, it is not necessary to test T'. □

Here we consider to implement a parallel version of the algorithm. It is natural to divide the search space and to assign each sub-space to individual processor. However, we must be careful when we divide the search space because the performance of the system is greatly affected by the dividing method. For example, in Fig. 4–2(a), the search space is divided into four parts and each of them are assigned to processor p_1 to p_4. Here, we assume that p_2 found that the hatched area in the figure can be eliminated from the search space. Then p_2 must transmit that information to other processors. On the other hand, if we divide the search space as shown in Fig. 4–2(b), then p_2 can reduce the search space without communicating with other processors. Therefore, it is better to divide the search space so that the reduction can be done locally in a processor.

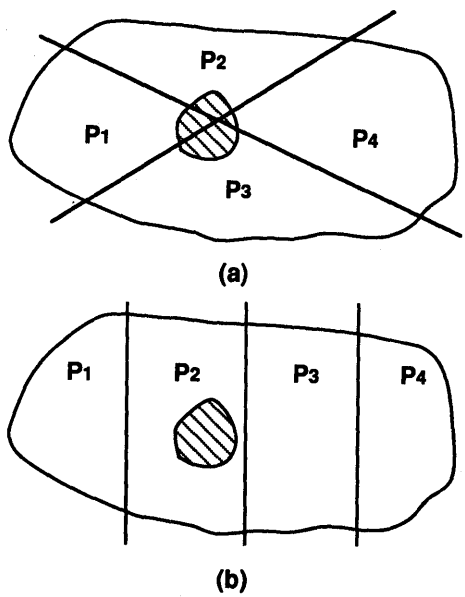


Fig. 4–2 Search Space Division

In the presumption problem, the search space has a tree structure. Each node in the search tree corresponds

to a possible presumption tree. Moreover, for a internal node of the search tree, each of its child node corresponds to a presumption tree which has longer description length than the parent node's corresponding one. Therefore, for example, the root node of the search tree corresponds to the simplest presumption tree.

If a search process examined node T, and the pruning condition for a child node of T is satisfied, then the subtree below the child node can be pruned (Fig. 4–3(a)). This means that the pruned area is included in a subtree which has node T as a root. In other word, parallel search for multiple disjoint subtrees can be performed independently. The algorithm we propose divides the search tree into several disjoint subtrees and searches each of them with individual processor (Fig. 4–3(b)).

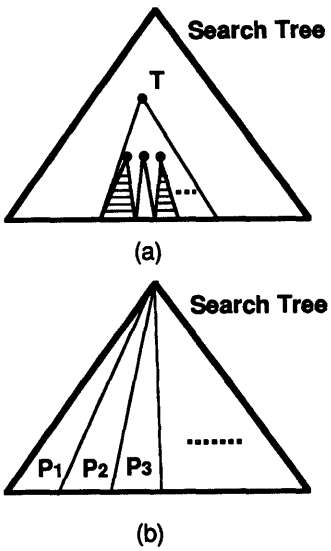


Fig. 4–3 Local Pruning

4.2 Global Pruning Mechanism

There is another kind of search tree pruning mechanism. If a certain process finds that a presumption tree T_0 has less description length than ever known, then each processor need not to test a tree that seems to have longer description length than T_0. The rest of this section describes details of this technique.

Here we consider two presumption trees T and T' such that $T' \succ T$. Then

$$\begin{aligned} & L1(T') + L2(T') \\ & \geq L1(T') + L2_{MIN} \\ & > L1(T) + L2_{MIN}. \end{aligned}$$

Here, if newly found tree T_0, which has shorter description length than ever known, satisfies the pruning condition :

$$L1(T) + L2_{MIN} \geq L(T_0)$$

then, from the above inequalities, we can conclude:

$$L1(T') + L2(T') > L(T_0)$$

$$\text{i.e. } L(T') > L(T_0).$$

Therefore, it is not necessary to examine T'. Therefore if we find a presumption tree which has shorter length than ever known, then some portion of the search space will be able to be eliminated.

However, reducible part of the search tree may be distributed widely throughout the search space. In other words, the pruning information should be announced to all of the other processors. Therefore, it is important to consider the trade-off between the increase of communication cost and the reduction of computation cost. That is, in a searching process, if a presumption tree is found to have shorter length than ever known, then the length of the tree should not always be announced to the other processors.

In order to solve the problem, a simple mechanism is incorporated. That is, the newly found length is transmitted only if it is over x bits smaller than the previously known least length. Here, x is a threshold value.

5 Implementation and Results

The learning algorithm was implemented in KL1 language on Multi-PSI, a distributed-memory multi processor machine. First, we implemented the algorithm with the local pruning mechanism. The experiments were performed by using up to 16 PEs in parallel. As a sample data, a fault history which comprised about 100 fault examples was given. The computation time was measured 5 times, and we took the average. The speedup curve of the example is shown in Fig. 5–1.

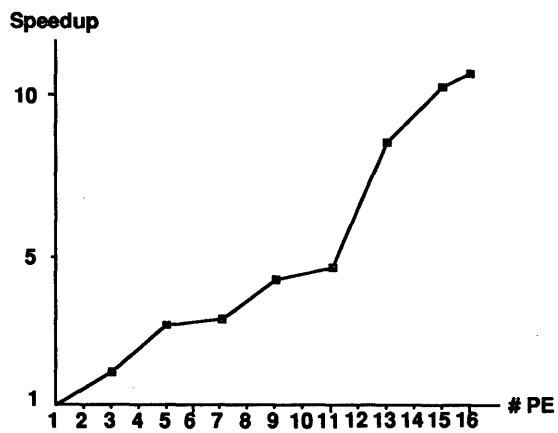


Fig. 5–1 Speedup of the Algorithm

The implementation using 16 PEs is about 11 times as fast as the sequential implementation (1 PE). There is a

possibility of further speedup by equalizing the load of each PE. An example of the overall load distribution is illustrated in Fig. 5–2. The difference of the load among the PEs may be improved by adding a dynamic load balancing mechanism into the system. Development of this mechanism is under investigation.

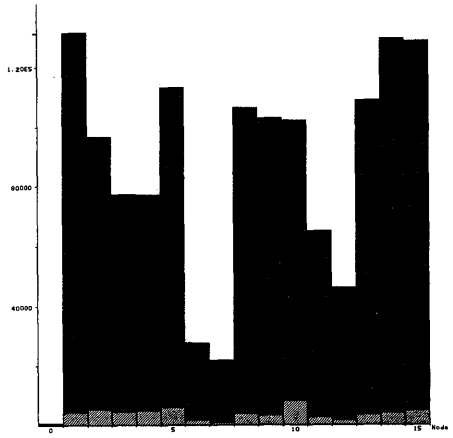


Fig. 5–2 Load Distribution

Next, we implemented the global pruning technique in addition to the local pruning mechanism. The threshold value for transmission is set to 2. This value was acquired empirically.

The performance of the algorithm with both the local and global pruning mechanism is shown in Table 5–1.

Table 5–1 Performance of the Algorithms

(a)

	No. of Reductions	
	Local	Local+Global
Example 1	870716	558661
(ratio)	1.00	0.64
Example 2	3602255	2588851
(ratio)	1.00	0.71
Example 3	30773602	23342853
(ratio)	1.00	0.76

(b)

	Execution Time (msec)	
	Local	Local+Global
Example 1	4522	3378
(ratio)	1.00	0.75
Example 2	16050	11282
(ratio)	1.00	0.70
Example 3	109892	89549
(ratio)	1.00	0.81

Each experiment is performed with three randomly generated examples. The number of reductions and the execution time are measured for the two versions of the algorithm. One is an algorithm with local pruning mechanism (Local), and another version incorporates both local and global pruning mechanism (Local+Global). Both of them are executed with 16 PEs.

The results show that the global pruning mechanism improved both of the number of reductions and execution time about 20% to 30% in comparison with the local pruning version. Fig. 5–3 shows an example of acquired presumption tree.

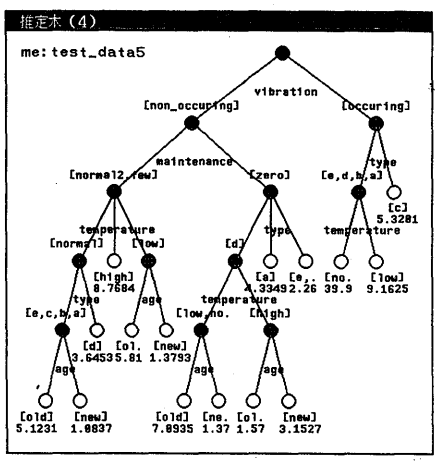


Fig. 5–3 Example of Acquired Tree

6 Conclusion

This paper has described a parallel learning algorithm for adaptive model-based diagnosis. The algorithm is based on branch-and-bound technique, and local and global pruning mechanisms are incorporated into the algorithm. The 16 PE implementation with local pruning mechanism is shown to be about 11 times as fast as the sequential one. Moreover, the global pruning mechanism is shown to have an ability to accelerate the parallel search.

Future work is to improve the heuristics used in the pruning process. If we can find more effective global pruning criterion which can be computed with low time complexity, it seems to be possible to perform superlinearly.

Acknowledgment

This research has been carried out as a part of the Fifth Generation Computer Project. The authors would like to thank Katsumi Nitta of the Institute for New Generation Computer Technology for his support, and to express their appreciation to Masahiro Yamamoto and Takeshi Yoshimura of NEC for their encouragement in this work.

References

[Davis 1984] Davis, R., "Diagnostic reasoning based on structure and behavior," *Artificial Intelligence,* Vol. 24, pp. 347–410, 1984.

[de Kleer 1987] de Kleer, J. and Williams, B. C., "Diagnosing multiple faults," *Artificial Intelligence,* Vol. 32, pp. 97–130, 1987.

[de Kleer 1989] de Kleer, J. and Williams, B. C., "Diagnosis with behavioral modes," *Proc. IJCAI-89,* Vol. 2, pp. 1324–1330, 1989.

[Genesereth 1984] Genesereth, M. R., "The use of design descriptions in automated diagnosis," *Artificial Intelligence,* Vol. 24, pp. 411–436, 1984.

[Koseki et al. 1990a] Koseki, Y., Nakakuki, Y., and Tanaka, M., "An adaptive model-Based diagnostic system," *Proc. PRICAI'90,* Vol. 1, pp. 104–109, 1990.

[Koseki et al. 1990b] Koseki, Y., Nakakuki, Y., and Tanaka, M., "An adaptive model-based diagnostic system and its learning method," *Proc. 4th Annual Conference of Japanese Society for Artificial Intelligence (in Japanese),* pp. 503–506 1990.

[Koseki 1989] Koseki, Y., "Experience learning in model-based diagnostic systems," *Proc. IJCAI-89,* Vol. 2, pp. 1356–1361, 1989.

[Nakakuki et al. 1990] Nakakuki, Y., Koseki, Y., and Tanaka, M., "Inductive learning in probabilistic domain," *Proc. AAAI-90,* Vol. 2, pp. 809–814, 1990.

[Nakakuki et al. 1991a] Nakakuki, Y., Koseki, Y., and Tanaka, M., "An adaptive model-based diagnostic strategy," *Proc. 5th Annual Conference of Japanese Society for Artificial Intelligence (in Japanese),* 1991.

[Nakakuki et al. 1991b] Nakakuki, Y., Koseki, Y., and Tanaka, M., "An inductive learning method and its application to diagnostic systems," *IPSJ SUG Reports 91-AI-74 (in Japanese),* Vol. 91 (3), pp. 19–28, 1991.

[Nakakuki et al. 1991c] Nakakuki, Y., Koseki, Y., and Tanaka, M., "Inductive learning of probabilistic knowledge," *Proc. 42nd Annual Convention IPS Japan (in Japanese),* 1991.

[Nakakuki et al. 1991d] Nakakuki, Y., Koseki, Y., and Tanaka, M., "A parallel algorithm for learning presumption tree," *Proc. 43rd Annual Convention IPS Japan (in Japanese),* 1991.

[Ohta et al. 1991a] Ohta, Y. et al., "A parallel processing method for an adaptive model-based diagnostic system," *Proc. 5th Annual Conference of Japanese Society for Artificial Intelligence (in Japanese),* 1991.

[Ohta et al. 1991b] Ohta, Y. et al., "A parallel processing method for an model-based diagnosis," *Proc. KL1 Programming Workshop (in Japanese),,* 1991.

[Quinlan 1986] Quinlan, J. R., "Induction of decision trees," *Machine Learning,* Vol. 1 (1), pp. 81–106, 1986.

[Rissanen 1978] Rissanen, J., "Modeling by shortest data description," *Automatica,* Vol. 14, pp. 465–471, 1978.

[Rissanen 1983] Rissanen, J., "A universal prior for integers and estimation by minimum description length," *Ann. of Statist.,* Vol. 11, pp. 416–431, 1983.

[Rissanen 1986] Rissanen, J., "Complexity of stings in the class of Markov sources," *IEEE Trans. on Information Theory,* Vol. 32 (4), pp. 526–531, 1986.

[Shortliffe 1976] Shortliffe, E. J., *Computer Based Medical Consultations: ba1MYCIN,* American Elsevier, New York 1976.

[Tanaka et al. 1989] Tanaka, M., Koseki, Y., and Nakakuki, Y., "A method of narrowing down suspects using experiential knowledge in model-based diagnostic systems," In *Proc. 39th Annual Convention IPS Japan (in Japanese),* pp. 243–244 1989.

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Logic Simulator based on Time Warp and its Evaluation

Yukinori Matsumoto and Kazuo Taki

Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, Japan

yumatumo@icot.or.jp

Abstract

This paper focuses on parallel logic simulation. An efficient logic simulator on a large-scale multiprocessor is targeted. The Time Warp mechanism, an optimistic method for time-keeping, was experimented and evaluated.

Synchronous mechanisms and conservative mechanisms for time-keeping have been examined already, and their inefficiency on large-scale distributed memory machines has been noted. There have been few reports, however, on evaluation of the Time Warp mechanism although rollback processes have been presumed to be heavy. We aim at evaluating the efficiency of this mechanism. Several devices, such as a local message scheduler, an antimessage reduction mechanism and a load distribution scheme, are added in order to reduce rollback overhead.

The simulator is implemented on the Multi-PSI, a distributed memory multiprocessor. The simulator is written in concurrent logic language KL1. KL1 is expected to be suitable for parallel programming because it supports data-flow synchronization and global name space across the processor boundary.

Experiments were done so that the speedup, performance and influences of various overheads could be measured. Using 64 processors, 48-fold speedup and 99K events/sec performance were obtained. The overhead measurements revealed that rollback processes slightly affected performance. These results showed that the simulator had fairly good performance as a full-software logic simulator and that the Time Warp mechanism worked efficiently.

1 Introduction

Logic simulation is used in order to verify not only the functions of designed circuits but also the timing of signal propagation. Since logic simulation is currently one of the most time-consuming stages in LSI design, faster simulators are urgently needed. A parallel logic simulator is one likely way of producing quick simulation.

Parallel logic simulation is usually treated as a typical application of parallel discrete event simulation (PDES). In PDES, performance essentially depends on the time-keeping mechanism.

The mechanisms broadly fall into three categories: synchronous, conservative and optimistic mechanisms. Since synchronous mechanisms require global synchronization, they, apparently, do not work efficiently on distributed memory multiprocessors[Soulé and Blank 1988]. Furthermore, conservative mechanisms tend to deadlock when circuits have feedback loops. A lot of computation power is needed to avoid this[Lubachevsky 1989, Misra 1986, Shimogori and Kage 1989]. On the contrary, optimistic mechanisms cannot deadlock, however, they do expend some computation power on rollback processes [Fujimoto 1990, Jefferson 1985]. Only a few experiments with the optimistic mechanism were reported [Chung 1989, Briner *et al.* 1991] but the details have not been evaluated yet.

We are targeting an efficient logic simulator on large-scale MIMD multiprocessors, most of which will be distributed memory machines. We adopted the Time Warp mechanism, an optimistic mechanism, because the overheads of the mechanism were considered to be reduced using some devices such as a local message scheduler, an antimessage reduction mechanism and a load distribution scheme. By adding them to the Time Warp mechanism, we expected that it would become suitable for logic simulation on large-scale MIMD machines.

We have implemented a parallel logic simulator on the Multi-PSI[Taki 1988] — an experimental parallel inference machine, a distributed memory multiprocessor. The simulator was written in concurrent logic language KL1. KL1 provides several advantages for quickly programming parallel applications. Data-flow synchronization, global name space and dynamic memory allocation are expected to remove the causes of many bugs.

Several benchmark circuits have been simulated on the

simulator in order to evaluate the efficiency of the Time Warp mechanism. Performance, speedup, rollback overhead and inter-PE (processor element) communication overhead have been measured.

This paper firstly overviews our system. Remarkable devices to enhance efficiency, such as a load distribution scheme, a local message scheduler and an antimessage reduction mechanism are mentioned. Secondly, KL1 and the Multi-PSI are briefly introduced. Then, fairly good performance and speedup in actual execution are reported. Finally, we refer to the examination that revealed the main causes affecting performance in our simulator.

2 The Time Warp Mechanism

Event simulation can be modeled so that several objects change their states by communicating with each other. An object is a state-automaton. A message has information of an event whose occurrence time is stamped on the message (time-stamp).

Jefferson proposed the Virtual Time paradigm and its implementation, the Time Warp mechanism[Jefferson 1985]. He suggested that the Time Warp mechanism would be useful as the time-keeping mechanism for PDES.

In the Time Warp mechanism, each object usually acts according to received messages and also records the history of messages and states, assuming that messages arrive chronologically. But when a message arrives at an object out of time-stamp order, the object rewinds its history (this process is called rollback), and makes adjustments as if the message had arrived in correct time-stamp order. After rollback, ordinary computation is resumed. If there are messages which should not have been sent, the object also sends antimessages in order to cancel those messages.

In addition to the above, a global control mechanism sometimes works to update GVT (global virtual time) which is used for memory management. GVT must satisfy the following two conditions.

1. GVT is not greater than the minimum simulation time at any object.

2. GVT is not greater than the minimum time-stamp values in the messages that have been sent but not yet received.

After the global control mechanism updates GVT, it notifies all objects of the new GVT. As no objects rewind their histories before GVT, the memory area occupied by histories before GVT can be released.

3 System Overview

3.1 System Specification

The system simulates combinatorial circuits and sequential circuits that have feedback loops. It handles three values: Hi, Lo, and X (unknown). A different delay time can be assigned to each gate (non-unit delay model). Since this system only treats gates, flip-flops and other functional blocks should be completely decomposed into gates.

The supported functions are the minimum set for experiments, but they can be easily expanded (e.g. to handle more signal values).

3.2 Load Distribution Scheme

For efficient execution of parallel logic simulation on a distributed memory machine, the scheme of load distribution is important at the following three points: load balancing, keeping inter-PE communication frequency low and deriving a lot of parallelism.

In our simulator, target circuits are partitioned statically in the preprocessing phase. We propose a new partitioning strategy called "Cascading-Oriented Partitioning" (COP, for short) for high-quality load distribution.

COP makes several clusters by grouping gates that are connected to each other in a cascade form. A grouping operation starts from the primary input of the circuit. By tracing a path of the gate connection straightforward, subsequent gates are included in a cluster. If there are several candidates to be included, only one gate is selected and the others are left to be the starting points for other grouping operations.

After partitioning, small clusters that contain very few gates are merged into adjacent large clusters. Conversely, extremely long cascade-formed clusters are cut into several smaller clusters so that they do not cause load imbalancing.

Finally, clusters are assigned to PEs randomly; the only constraint is that each processor should contain a roughly equal number of gates.

COP intends to exploit parallelism of the multiple fanouts. COP also guarantees that a gate has at least one adjacent gate in the same cluster. So, COP is effective in keeping the communication locality, that is, reducing inter-PE communication. The random distribution of clusters attains load balancing.

In COP, the smaller each cluster, the better load balancing but the higher inter-PE communication frequency. There is a performance trade-off between good load balancing and the low frequency of inter-PE communication. It is necessary to decide the appropriate size of a cluster according to the number of gates in the

target circuit and the number of PEs[1].

COP may look similar to Agrawal's algorithm, however, COP is different from it in the next two points.

- Agrawal's algorithm basically assumes simulation using the synchronous time-keeping mechanism. According to the gate delay value, the algorithm estimates the number of messages generated in each cluster at each tick for the purpose of load balancing.

  Conversely, an estimation such as the above is slightly beneficial to our simulator because messages with different time-stamps can be evaluated simultaneously in the Time Warp mechanism.

- In Agrawal's algorithm, once a cluster is generated, it will never be decomposed into smaller ones. Therefore, the load is sometimes imbalanced.

  In COP, however, clusters that are too large are cut into several adequately sized clusters. This enables the system not only to be flexible for various numbers of PEs but also it to exploit more parallelism (i.e. pipeline parallelism).

3.3 Local Message Scheduler

In the simulation, there are usually several messages to be evaluated in a PE. When the Time Warp mechanism is used, the bigger time-stamp a message has, the more likely the message is to be rewound. For this reason, proper message scheduling in each PE is expected to reduce rollback effectively.

In our system, a message scheduler resides in each PE. When a message is spawned, it is first registered in the scheduler in which the destination object belongs. The scheduler picks up the messages with the smallest time-stamps and sends them to destination objects at the appropriate moment.

This scheduler ensures that rollback never happens as long as an object is receiving messages from other objects in the same PE. Only messages sent from other PEs may cause rollback.

3.4 Reduction of Antimessages

In Jefferson's original Time Warp mechanism, when rollback occurs, as many antimessages must be generated as the number of messages that need to be canceled (Figure 1). However, the number of antimessages can be reduced when we assume the next condition: for any objects A and B, messages transmitted from A to B are received by B in the same order as they are sent by A (order-preserved condition)[Fukui 1989].

[1]For reference, clusters with 12 to 32 gates are generated for a circuit consisting of 12,000 gates in the simulation using 64 PEs.

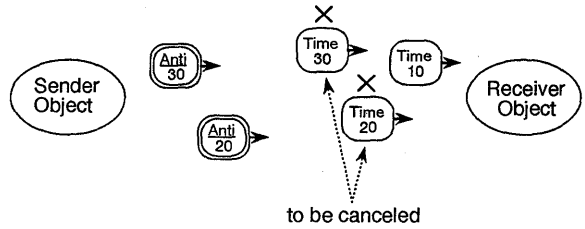


Figure 1: Cancellation with **several** antimessages

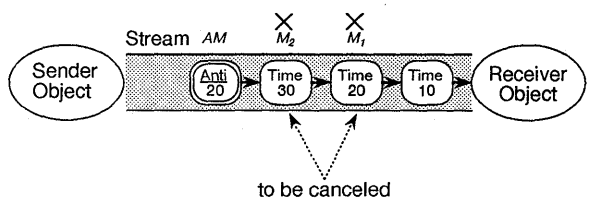


Figure 2: Cancellation with **one** antimessage

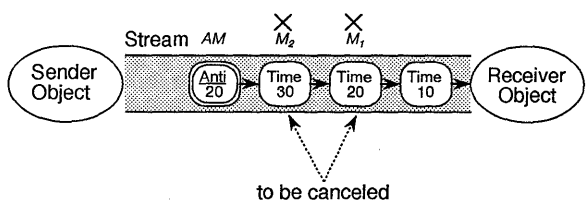


Figure 3: Cancellation with **no** antimessages

Assume that $M_1, M_2, .., M_n$ are messages and AM is an antimessage. Also assume $M_1, M_2, .., M_n$ all satisfy the following three conditions:

- $M_1, M_2, .., M_n$ were sent before AM,
- $M_1, M_2, .., M_n$ were sent along the same channel that AM is sent along,
- $M_1, M_2, .., M_n$ have time-stamps greater than or equal to AM.

Then it is clear that $M_1, M_2, .., M_n$ must be canceled. No other messages must be canceled. Only one antimessage that corresponds to the canceled message with the smallest time-stamp need be sent (Figure 2).

We advanced this idea one step further. Assume that a sender has to cancel messages $M_1, M_2, .., M_n$ which have already been sent in this order, and at the same time the sender knows that a new message M_{new} will be sent whose time-stamp is equal to or less than that of M_1.

In this case, the sender sends M_{new} but no antimessage. When a receiver receives M_{new} with a smaller time-stamp than the M_n that the destination object received just before, the receiver can easily notice that an invalid situation has occurred, and can cancel $M_1, M_2, .., M_n$ immediately (Figure 3).

In our system, the message streams of KL1 are used for communication between objects. Since KL1 keeps the order of messages in the stream, the order-preserved

condition is satisfied. So, we adopted the above optimization for reducing antimessages.

4 Hardware and Language

4.1 Hardware

This simulator is implemented on the Multi-PSI[Taki 1988], a distributed memory MIMD machine. The Multi-PSI consists of 64 processing elements (PEs) connected to each other by a 2-dimensional mesh network. A PE is a 40-bit (8 bits for tag and 32 bits for data) CISC processor controlled by horizontal micro-instruction. The cycle time is 200 nsec.

A network controller is paired with each PE, supporting message passing communication between PEs. The bandwidth of the controller is 5M bytes/sec. The network has wormhole routing functionality.

Since the Multi-PSI is a distributed memory machine, communication latency between objects in different PEs takes approximately twenty times longer than latency in the same PE. However, the distributed memory architecture can be scaled up easily.

4.2 Language and Implementation

This simulator is written in concurrent logic language KL1.

KL1 is a language without destructive value assignment to variables, that is a single assignment language. Due to its nature, data-flow synchronization is realized without significant overheads in the language implementation. Therefore, KL1 never compels programmers to describe synchronization explicitly at a primitive level.

On the other hand, a single assignment language tends to consume storage rapidly. A dynamic memory allocation mechanism and several garbage collection mechanisms are supported in the KL1 implementation. So, programmers are free from writing memory management.

The KL1 language assumes a system-wide (global) name space even on a distributed memory machine. In KL1 programming, first, a programmer writes only the logical concurrency, relations among concurrent objects or data-flow. Then, the programmer adds the "pragma" to specify object allocation to a certain processor, as below.

$$\ldots, B@processor(PE), \ldots$$

where B is a "goal" of KL1, which represents an object. Inter-PE reference pointers among objects or variables are maintained automatically by the KL1 language system at run time. Thus a programmer need not worry at all about the programming of inter-PE communication.

Since the characteristics described above eliminate the causes of many bugs, KL1 enables us to develop parallel programs much more easily than with the conventional languages (e.g. FORTRAN and C). In fact, it took one person just three months to complete the primary version of the simulator, including the circuit partitioning module! Moreover, because of KL1, several different experiments, which needed program modification, could be performed in a short period.

4.3 Avoiding Asynchronous Copying GC

As mentioned above, garbage collection (GC) is indispensable for KL1. Two kinds of garbage collection (GC) mechanisms, a copying GC[Baker 1978] and the MRB GC[Chikayama and Kimura 1987], are implemented for intra-PE memory management on the Multi-PSI.

For the Time Warp mechanism, the most important point in obtaining good performance is to keep the pace of simulation in each PE equal. However, the copying GC starts at different times in different PEs and disturbs the pace of simulation.

Fortunately, since the MRB GC collects single reference data area without stopping KL1 execution, it is expected to stabilize the pace of simulation. We took great care to keep the data reference single so that all data areas can be collected by the MRB GC. Hence we succeeded in preventing the copying GC[2].

5 Measurements and Discussions

Four sequential circuits, presented in ISCAS'89, were simulated on the Multi-PSI. The number of gates, average fan-ins and average fan-outs of the circuits are shown in Table 1. We measured system performance, speedup and overheads, such as rollback and inter-PE communication, in the experiments.

Table 2 shows the system performance for various numbers of PEs. Figure 4 indicates speedup. Table 3 shows the percentage of each process cost[3]. Table 4 shows the percentage of actual events[4] and rewound messages.

Table 5 shows the frequency of rollback f_r, the average depth of rollback d_r (i.e. the average number of rewound messages per rollback) and the frequency of inter-PE communication f_c. f_r is defined as R/E, d_r as H_r/R, and f_c as M_c/M_{all}, where R is the total number of rollback occurrences, E is the total number of actual events, H_r is the total number of rewound messages, M_c

[2]Consequently, when a certain circuit was simulated using 64 PEs, an improvement in performance of approximately 37% was attained compared to the case where the copying GC occurred(see appendix).

[3]These are the average values for 64 PEs.

[4]Actual events are the messages that are not rewound.

Table 1: Target circuits

Circuits	s1494	s5378	s9234	s13207
No. of gates	683	3,853	6,965	11,965
Avg. fan-ins	2.15	1.70	1.57	1.66
Avg. fan-outs	2.08	1.61	1.50	1.55

Table 2: Performance (events/sec)

PEs\Circuits	s1494	s5378	s9234	s13207
1	2,572	2,410	2,326	2,051
4	5,662	8,401	7,709	9,092
16	10,413	26,141	19,003	33,793
64	10,943	64,013	35,118	99,299

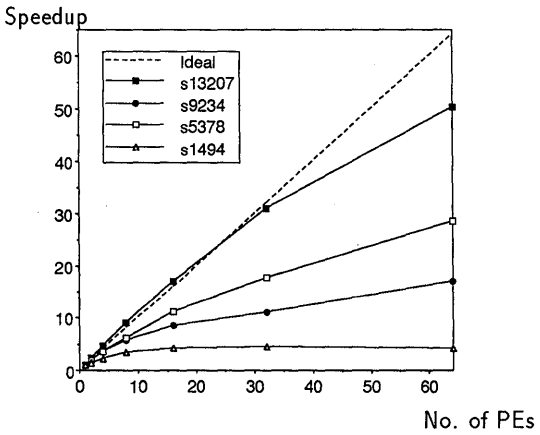


Figure 4: Speedup

is the total number of messages that are sent across PE boundaries and M_{all} is the total number of messages.

5.1 Performance and Speedup

As shown in Table 2 and Figure 4, the simulator attained approximately 99K events/sec performance and 48-fold speedup in the best case using 64 processors. This performance is fairly good for a full-software logic simulator. This good speedup shows that the Time Warp mechanism works efficiently.

In some cases, however, comparatively poor speedup was measured. In order to ascertain the cause of limited speedup, we will discuss the inter-PE communication overhead, the rollback overhead and parallelism in the following subsections.

Table 3: Percentage of time for each process (64PEs)

Process\Circuits	s1494	s5378	s9234	s13207
Evaluating and scheduling messages†, etc.	72.28	80.28	58.69	85.79
Rollback	5.32	2.50	1.61	1.53
Inter-PE communication	13.13	8.24	4.38	2.12
GVT updating	1.21	0.48	0.62	0.64
History releasing	0.43	2.41	1.57	3.86
Idling	7.63	6.09	33.13	6.06

† This process is not only for actual events but also for messages rewound.

Table 4: Percentage of actual events and rewound messages (64PEs)

Circuits	s1494	s5378	s9234	s13207
Actual events	29.34	81.84	65.23	86.54
Rewound msgs.	70.66	18.16	34.77	13.46

Table 5: Frequency and depth of rollback, Frequency of inter-PE communication(64PEs)

Circuits	s1494	s5378	s9234	s13207
f_r	0.938	0.0703	0.0510	0.0227
d_r	2.57	3.16	10.46	6.84
f_c	0.340	0.110	0.0761	0.0203

5.2 Inter-PE Communication Overhead

The cost per message for inter-PE communication was measured to be 0.503 msec[5]. However, the average cost of the essential work, that is, evaluating and scheduling a message, was 0.362 msec. So, the inter-PE communication cost was not negligible[6].

Tables 3 and 5, however, show that both the frequency and the percentage of inter-PE communication processes were low in the cases of s13207 and s9234. This means that our strategy for partitioning circuits worked effectively and that inter-PE communication had only a slight effect on performance in these cases. Conversely, in the case of s1494, both the inter-PE communication frequency and the percentage of its process were high compared to other cases. s1494 has, on average, more fan-ins and fan-outs than the others (Table 1), and it tends to

[5]A message is 25 bytes of data

[6]However, the relative cost of inter-PE communication is far lower than systems where inter-PE communication is supported by the operating system.

Table 6: Modified speedup and parallelism

Circuits	s1494	s5378	s9234	s13207
Modified speedup	3.96	23.62	12.51	35.66
Parallelism	18.88	35.52	17.95	43.24

cause the high inter-PE communication overhead.

5.3 Rollback Overhead

Rollback frequency and its cost greatly attracted our interest. Table 5 shows that the rollback frequency is not as high as we formerly suspected, except for s1494.

The average cost per rollback, even for the highest case, s9234[7], amounted to 0.578 msec by our measurement. Since the time for essential processes was 0.362 msec, the rollback procedure is not extremely time-consuming compared to the essential works, and consequently the percentage of total rollback cost is not seriously high in itself, as shown in Table 3.

5.4 Parallelism

Parallelism suggests the upper limit of speedup. In practice, however, the actual speedup is usually different from the parallelism because of several overheads.

We estimated the parallelism of each problem, as below. We made another simulator to measure parallelism. In that simulator, all PEs work according to the global synchronization. Here, we call an interval between global synchronizations a "*time slot*". A PE evaluates only one message in a time slot. All output messages, if any, are also sent and registered to their destination schedulers within the time slot. When there is no message to be evaluated in a PE at a certain time slot, the PE simply idles. Assume that the simulation finishes after N synchronization, and that M actual events, which are the messages that are not rewound, are measured in the simulation. Here, we define the parallelism of the problem as M/N. The parallelism means the speedup in such an environment where the cost for non-essential processes, such as rollback and inter-PE communication, can be ignored.

On the other hand, to make clear the effect of the inter-PE communication overhead and the rollback overhead, we measured the cost of releasing an unnecessary history area, which causes super-linear speedup[Matsumoto and Taki 1991]. Then we removed its effect from the measured speedup and recalculated the speedup. We named the recalculated speedup "*modified speedup*".

Table 6 compares the modified speedup and the parallelism of each problem using 64 PEs. The gap between the modified speedup and the parallelism is caused by the inter-PE communication overhead and the rollback overhead. For s5378, s9234 and s13207, the parallelism is close to the modified speedup. This means that the limited speedup was caused by a lack of parallelism. We conclude that our system could show good speedup as long as target problems have sufficient parallelism. With respect to the exceptional case, s1494, as Table 4 shows, a considerable percentage of the messages are rewound. It is considered that the high percentage was caused indirectly by the high inter-PE communication overhead or high rollback overhead, and resulted in further suppression of speedup.

[7]The cost is considered roughly proportional to the depth of rollback, and s9234 has the largest depth.

6 Further Experiments

Since neither the inter-PE communication cost nor the rollback cost are negligibly small, both of these costs are considered to affect performance not only directly but also indirectly. However, it is difficult to separate their influences clearly.

In this section, we report on the experiments that aim at clarifying which affects performance more, the inter-PE communication cost or the rollback cost. We assumed the model described below and made a system for the experiments.

6.1 Model

We assume that the only processes that need costs are the rollback, the inter-PE communication and an essential process. Here, an essential process consists of a message evaluation work and a scheduling work. Any other processes, such as GVT updating and releasing unnecessary history area, do not need any costs at all. It is also assumed that the essential process cost is equal for any gates.

In our model, the inter-PE communication cost C_c decomposes into three factors as follows.

$$C_c = C_{p_s} + C_l + C_{p_r} \quad (1)$$

where C_{p_s} is the time consumed in the sender PE for composing a message packet, C_l is the time from when the message leaves the sender until it arrives at the receiver (latency), and C_{p_r} is the time taken by the receiver to decompose the message packet.

As the rollback cost, t_r, is roughly proportional to the number of rewound messages, t_r is represented by the next equation.

$$t_r = k_r h_r + C_r \quad (2)$$

where h_r is the number of messages rewound in the history and C_r is a constant.

In practice, Equations 1 and 2 give a fairly precise representation of these costs. With regard to C_{p_s} and C_{p_r}, they are approximately equal[Nakajima and Ichiyoshi 1990] on the Multi-PSI, while the latency is negligible even if messages are transmitted between the most distant PEs.

6.2 Experimental System

The experimental system is based on the simulator presented in the previous sections. By adding several dummy loads to the original simulator, the actual costs for rollback and inter-PE communication become negligible. Thus this system maintains its fidelity to the model as much as is possible.

In the system, the cost for an essential process is fixed to be heavy, whereas the rollback cost and the inter-PE communication cost are changeable.

6.3 Results

We performed two kinds of comparative examination, as below.

1. The inter-PE communication cost is fixed so that its relative value to the essential process cost can be kept the same as in the actual simulation. With respect to rollback, k_r and C_r in Equation (2) are varied but C_r/k_r is kept equal to that in the actual simulation.

2. The rollback cost is fixed so that its relative value to the essential process cost can be kept the same as in the actual simulation. The inter-PE communication cost is varied but the equality between C_{p_s} and C_{p_r} in Equation (1) is kept because they were approximately equal in the actual simulation.

We simulated s9234 and s1494. They involved approximately the same parallelism, whereas both the inter-PE communication frequency and the rollback frequency were very different.

Figures 5 and 6 show the results. In Figure 5, the X axis shows the relative value of $C_{p_s}+C_{p_r}$ compared to the essential cost. In Figure 6, the X axis shows the relative value of C_r. The Y axis shows the relative performance (by solid lines) and the relative amount of rewound messages (by broken lines) compared to those when both the inter-PE communication cost and the rollback cost are set to zero. The arrows indicate the the actual proportion points between these costs.

For both circuits, the higher the inter-PE communication cost, the worse the performance. This apparently shows that the inter-PE communication cost affected performance adversely. Interestingly, the relative amount of rewound messages increased with the higher inter-PE communication cost for s1494, while the curve of the amount is approximately flat for s9234. The difference in declination of the performance curves was, therefore, caused not only by the difference in the inter-PE communication frequency but also by the difference in the amount of rewound messages.

On the contrary, neither performance nor the amount of rewound messages varied remarkably even though the

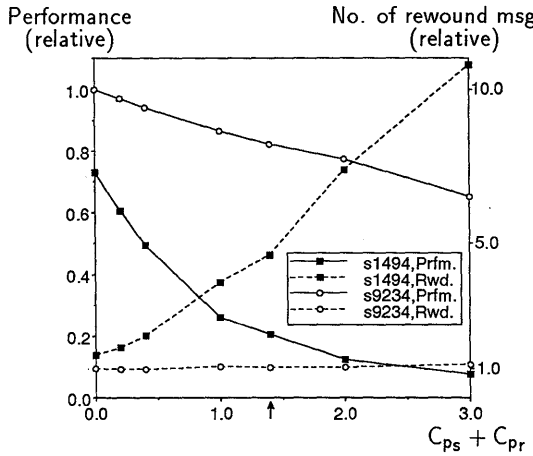


Figure 5: Factors affecting performance(1)

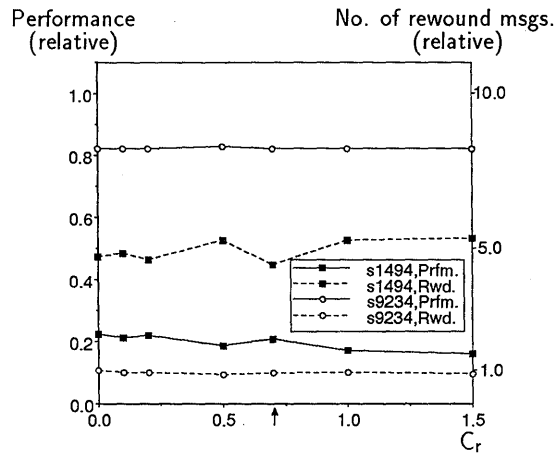


Figure 6: Factors affecting performance(2)

cost of rollback increased. This means that rollback cost did not have a dominant effect on performance in our system.

7 Summary and Conclusion

We constructed a parallel logic simulator on the Multi-PSI, a distributed memory multiprocessor. The simulator was programmed in concurrent logic language KL1. Since the causes of many bugs are essentially reduced by KL1, the simulator was able to be programmed in only three months by one person.

The Time Warp mechanism was adopted for time-keeping in the simulator. Since rollback overhead in a naive Time Warp mechanism was considered heavy, we added several devices such as a local message scheduler, an antimessage reduction mechanism and a load distribution scheme to reduce the overhead.

Several benchmark circuits were simulated on our system. Approximately 99K events/sec performance and 48-fold speedup were attained using 64 PEs. The per-

formance is fairly good for a software logic simulator. The good speedup shows that the Time Warp mechanism worked efficiently in the simulator.

We also examined the factors that are considered to affect performance adversely. The experiments revealed that the rollback overhead did not affect performance seriously in our system, while the inter-PE communication overhead decreased performance.

Acknowledgement

Valuable advice and suggestions were given by the members of PIC-WG, an ICOT working group, discussing parallel LSI-CAD. The authors gratefully thank them. Data for the evaluation of our system were recommended and given by Fujitsu Ltd. and Keio Univ. We also thank them.

References

[Agrawal 1986] P. Agrawal. Concurrency and Communication on Hardware Simulators. *IEEE Trans. on Computer-Aided Design*, Vol.CAD-5, No. 4 (1986), pp. 617–623.

[Baker 1978] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, Vol. 21, No. 4 (1978), pp. 280–294.

[Briner *et al.* 1991] J. V .Briner *et al.* Parallel Mixed-level Simulation using Virtual Time. *CAD accelerators*, North-Holland, 1991. pp. 273–285.

[Chung 1989] M. J. Chung and Y. Chung. Data Parallel Simulation using Time-Warp on the Connection Machine. In *Proc. 26th ACM/IEEE Design Automation Conf.*, 1989. pp. 98–103.

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. Fourth Int. Conf. on Logic Programming*, 1987. pp. 276–293.

[Fujimoto 1990] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, Vol.33, No.10 (1990), pp. 30–53.

[Fukui 1989] A. Fukui. Improvement of the Virtual Time Algorithm. *Trans. of Information Processing Society of Japan*, Vol.30, No.12 (1989), pp. 1547–1554. (in Japanese)

[Jefferson 1985] D. R. Jefferson. Virtual Time. *ACM Trans. on Programming Languages and Systems*, Vol.7, No.3 (1985), pp. 404–425.

[Kudoh *et al.* 1991] T. Kudoh *et al.*. Parallel Logic Simulator for Shared Memory Multiprocessors. *IEICE Technical Report*, CPSY91-23 (1991), pp. 151–131. (in Japanese)

[Lubachevsky 1989] B. D. Lubachevsky. Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks. *Communications of the ACM*, Vol.32, No.1 (1989), pp. 111–131.

[Matsumoto and Taki 1991] Y. Matsumoto and K. Taki. Parallel Logic Simulation based on Virtual Time. In *Proc. Joint Symposium on Parallel Processing '91*, 1991. pp. 365–372. (in Japanese)

[Misra 1986] J. Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, Vol.18, No.1 (1986), pp. 39–64.

[Nakajima *et al.* 1989] K. Nakajima *et al.*. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. 1989 Int. Conf. on Logic Programming* 1989. pp. 436–451.

[Nakajima and Ichiyoshi 1990] K. Nakajima and N. Ichiyoshi. Evaluation of Inter-processor Communication in the KL1 Implementation on the Multi-PSI. *ICOT Technical Report*, TR-531 (1990).

[Shimogori and Kage 1989] S. Shimogori and T. Kage. Parallel Logic Simulation using A Message-Driven Approach. *IEICE Technical Report*, CAS88-110 (1989), pp. 23–30. (in Japanese)

[Soulé and Blank 1988] L. Soulé and T. Blank. Parallel Logic Simulation on General Purpose Machines. In *Proc. 25th ACM/IEEE Design Automation Conf.*, 1988, pp. 166–170.

[Taki 1988] K. Taki. The parallel software research and development tool: Multi-PSI system. *Programming of Future Generation Computers*, North-Holland, 1988. pp. 411–426.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol.33, No.6 (1990), pp. 494–500.

Appendix

For the purpose of ascertaining the influence of the asynchronous copying GC, we made another simulator and compared it to the original. The difference between the comparative simulator and the original is as follows.

Original simulator :
Only the MRB GC works for collecting garbage.

Comparative simulator :
The copying GC happens asynchronously in each PE.

Table 7 compares the simulators when s13207 was simulated using 64 PEs. The result shows that asynchronous outbreaks of the copying GC in each PE increased both rollback frequency and rollback depth. It certainly caused the poor performance of the simulator.

Table 7: Influence of asynchronous copying GC

	Original simulator	Comparative simulator
Performance (K events/sec)	99.299	72.895
Frequency of rollback	0.0243	0.0261
Depth of rollback	7.96	11.684

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Applications of Machine Learning: Towards Knowledge Synthesis

Ivan Bratko

Faculty of Electr. Eng. and Computer Sc., Ljubljana University
and J. Stefan Institute, Slovenia

Abstract

This paper shows, by presenting a number of Machine Learning (ML) applications, that the existing ML techniques can be effectively applied in knowledge acquisition for expert systems, thereby alleviating the known knowledge acquisition bottleneck. Analysis in domains of practical interest indicates that the performance accuracy of knowledge induced through learning from examples compares very favourably with the accuracy of best human experts. Also, in addition to accuracy, there are encouraging examples regarding the clarity and meaningfulness of induced knowledge. This points towards automated knowledge synthesis, although much further research is needed in this direction. The state of the art of some approaches to Machine Learning is assessed relative to their practical applicability and the characteristics of a problem domain.

1 Introduction

Machine Learning is one of the most active areas of Artificial Intelligence. In the view of the technical results of this area, and the well known knowledge acquisition bottleneck in expert systems, sometimes known as the Feigenbaum bottleneck, it is surprizing that Machine Learning has not had a stronger impact on the practice of knowledge acquisition for expert systems. Even some known authorities on expert systems occasionally express a reserved view regarding automatic knowledge acquisition through machine learning. For example, Chandrasekaran (1991) in a recent discussion posed the question: "It is often proposed that a way to avoid teasing expertise from experts is to automatically learn from examples. Have you found this a useful strategy?" The answer from a leading practitioner from the commercial side of expert systems technology was: "... I have yet to see a situation where that is an effective way to go forward, especially when you're starting with somebody who knows something. ..."

The practice of AI applications in some laboratories and companies shows, however, that this expresses an overly pessimistic view. This paper presents examples of ML applications in which *existing* techniques were effectively applied.

The paper does not aspire to be in any way a complete survey of the state-of-the-art ML techniques and their applications. However, the example applications and programs discussed are generally illustrative of the practically oriented ML research done at many AI laboratories.

An early demonstration of the usefulness of Machine Learning from examples in knowledge acquisition was induction-assisted knowledge base construction for diagnosing soybean diseases (Chilausky and Michalski 1976; Michalski and Chilausky 1980). A comparison between a manually constructed knowledge base and one constructed with the assistance of an inductive learning program showed the advantages of the latter approach.

Michie (1989) describes another early interesting experience concerning the construction of a small expert system to decide whether a Space Shuttle pilot should lend manually or automatically. The decision depends on the current information about the stability, altitude and velocity estimates of the vehicle etc. This project was an early demonstration of the experts' difficulty in explicitly formulating decision rules although all the relevant information was in their heads. Experience shows that experts' difficulty of this kind is a rather typical phenomenon.

Michie (1989) writes:

"Early in 1984, to address a NASA requirement, the autolander's chief designer, Mr Roger Burke, with engineering colleagues, attempted to construct a computer program to map the real-time values of monitored variables to the alternative decisions *useauto* and *noauto*. Such a program running on an on-board computer was needed to display continually updated advice to the pilot. After some months of (noninductive) programming they concluded that further effort would not be rewarding. The trouble was later shown to have stemmed not from any intrinsic difficulty of the decision task but from the disability from which every expert suffers in articulating what he or she knows, whether about plant pathology, about medical diagnosis, about process control, about how to play lightening chess or about the movements of the stockmarket.

Mr Burke and his colleagues then attended a course in inductive programing given by Radian Corporation in Austin, Texas, based on the commercial induction software RuleMaster (Michie et. al. 1984). Relieved from the struggle to read the needed rules directly from inside their own heads, they were able (...) to construct the solution ..."

Although a not very sophisticated tool was applied to a not very difficult problem, the NASA experience is very instructive. It illustrates the phenomenon concerning the difficulty of eliciting explicit rules about a domain even if (1) there are experts that can solve concrete problems in the domain quite well, and (2) there is nothing inherently difficult about the domain. Even so, extracting explicit rules from the user turns out to be difficult. When the knowledge elicitation process is aided by a learning tool, the process suddenly appears trivial. Finally, when the actual simple looking solution becomes obvious, there is typically a somewhat embarassing impression that "clearly, the problem should have been possible to solve without the use of machine learning". However, experience confirms that often only when a machine learning tool is eventually applied, the problem solution emerges as obvious.

Another early and similar example of this phenomenon is W. Leech's (1986) application of ML to the synthesis of control rules for process control at a Westinghouse nuclear fuel processing plant. Control rules synthesised from examples using another early ML tool ExpertEase improved the yield drammatically. When analysing the project that led to this innovation, the company officially confirmed that the discovery of the new control rules only occurred when ML was used and the discovery would have been highly unlikely without it.

A review (Urbančič, Kononenko and Križman 1991) of AI applications done by my laboratory in Ljubljana also contains many applications with similar scenario. Among over sixty AI applications included in the review, almost half of them critically rely on the use of ML techniques. One more or less randomly chosen example among these applications, illustrating the same point as the NASA and Westinghouse experience, is from the Jesenice Steel Mill, Slovenia. Their problem was the control of the quality of the rolling emulsion for the Sendzimir rolling mill. The quality of rolling critically depends on the properties of emulsion. An expert therefore daily measured various parameters of emulsion in the rolling mill (concentration of iron, ashes, presence of bacteria, etc.) and decided on the appropriate action (e.g. change emulsion, add antibacteria oil, no action, etc.). When the expert was expected to leave the company they attempted to construct an expert system, extracting his decision knowledge from him in the dialogue fashion. Only when after half a year there was no clear progress, they were prepared to apply a ML tool (Assistant Professional in this case; Cestnik et al. 1987) using example decisions from the expert's practice as learning examples. The resulting decision tree, implemented as an expert system, is now used regularly and completely substitutes the decisions that were previously entirely made by the expert.

The most practically successful form of learning has been *attribute-based* learning exemplified by the TDIDT approach (top-down induction of decision trees, e.g. Quinlan 1986). The next section presents results of applications of attribute-based learning in various domains of medical diagnosis and prognosis. These results are interesting also in that they enable a quantitative comparison of the performance of human experts and ML-based diagnostic systems. Although very effective in many domains of practical interest, attribute-based learning has some clear limitations, pointed out in Section 3. These limitations are being overcome by the development of another generation of learning systems, implementing *relational learning*, such as ILP (Inductive Logic Programming, Muggleton 1991). Section 4 presents an example application where the ability of relational learning is essential. ILP, although less mature than attribute-based learning, shows great potentials in application problems that are hard to tackle with attribute-based learning. Section 5 discusses the fu-

ture of ML with respect to knowledge synthesis.

2 Applications in medical domains

Along with the development of various learning methods in the Ljubljana AI Laboratories, these methods were applied to a number of medical diagnosis/prognosis problems. These applications also served as a source of useful new ideas for further improvements of the learning methods. Some of our medical data (in particular the diagnosis in lymphography, location of primary tumor, and prognosis in breast cancer) were made available to other researchers and were used by many for experimentation and direct comparison of various learning algorithms.

This section presents some results obtained in Ljubljana with various learning systems in several medical domains. Most of this work in medical applications was done with the Assistant system although other programs were also used, including GINESYS (Gams 1988) and LogArt (Cestnik and Bratko 1988). Assistant belongs to the TDIDT family of learning programs (top down induction of decision trees, Quinlan 1986). Assistant is a successor of Quinlan's ID3 (Quinlan 1979) with a number of addidional mechanisms. Early experiments with a version of ID3 in learning of diagnostic rules for lymphatic cancers (Bratko and Mulec 1979) provided encouragement that led us to further exloration and substantial refinements of this approach that were implemented in Assistant. The new mechanisms, motivated and discovered through experiments in medical domains, include: automatic selection of good examples for learning, handling partially specified objects (missing data), forward pruning of decision trees, post pruning (Niblett and Bratko 1986, Cestnik and Bratko 1991), binarisation of attributes (Kononenko et. al. 1985; Bratko and Kononenko 1987). It should be noted that these techniques among some other important improvements to the basic TDIDT learning were contributed or independently discovered by other researchers, for example in the C4 program (Quinlan et al. 1989) and the CART system (Breiman et al. 1984). Mingers (1989a; 1989b) reviews various related techniques and makes an attempt at their comparison.

TDIDT programs belong to attribute-based learning. They accept learning examples in the form of attribute-value vectors. Similarly, both GINESYS and LogArt are attribute-based learning programs. GINESYS generates if-then rules. The innovation of GINESYS was *confirmation rules* that accompany the "main" rule and enable the system to exploit *redundancy* in the attribute-value data. Redundancy is in principle useful in noisy domains, such as medicine, as a means for filtering out errors. The idea of exploiting redundancy (rule bases with redundancy) was later accepted as generally useful in learning in noisy domains, but GINESYS (Gams 1989) was probably the first to build explicitly on this principle in Machine Learning.

Unlike most other systems, LogArt (Cestnik and Bratko 1988) generates *elimination rules*. The rules are ordered according to their statistical credibility. In diagnosis, rules are applied in this order to eliminate all but one of the diagnostic possibilities. When this is not possible and there are more than one residual diagnostic possibilities, the Bayes classifier is employed as a tie-breaker. The credibility of induced rules is measured simply as the number of confirming observations in the learning data. These rules are extremely simple and thus also useful for straightforward explanation of the diagnostic decision. Despite this almost unbelievable simplicity, LogArt compares extremely well with other learning systems in respect of diagnostic accuracy. The key to LogArt's performance lies in high number of simple elimination rules for each application which, similar to GINESYS, facilitates the use of redundancy. This makes LogArt very robust with respect to noise in the learning data and also enables it to cope easily with missing data, that is unspecified attribute values. On the theoretical side, it was shown that LogArt's classification procedure can be viewed as a special strategy of evaluating the Bayes classification rule without the attribute independence assumption (Cestnik and Bratko 1988). LogArt's classification procedure tends to use those conditional probabilities for whose estimation the learning data provides most evidence.

Table 1 summarises the properties of eight medical domains in which these learning systems have been applied. The domains are characterised by: the number of known examples (patients), the number of classes (that is: possible diagnoses), the number of attributes, the average number of possible attribute values per attribute. More detailed description of these applications can be found for example in (Bratko and Kononenko 1987), (Pirnat et al. 1989) and (Roškar et al. 1986).

Table 2 shows results of these applications in terms of diagnostic accuracy of learned diagnostic

domain	examples	classes	majority class	attributes	average no. values	entropy (bit)
lymphography 1	148	4	55 %	18	3.3	1.23
lymphography 2	150	7	46 %	18	3.3	2.11
primary tumor	339	22	25 %	17	2.2	3.64
breast cancer	288	2	80 %	10	2.7	0.72
hepatitis	155	2	79 %	19	3.6	0.74
thyroid	884	4	56 %	15	15.7	1.59
rheumathology	355	6	66 %	32	9.1	1.70
urinary tract m	1843	9	21 %	44	3.8	2.91
urinary tract f	3580	9	25 %	45	6.5	2.59

Table 1: Properties of the medical application domains.

rules by the three systems. The performance of medical experts is also included for comparison in the cases when their performance has been experimentally estimated on the same data as used by the systems. In one case (lymphography), the physicians' performance is an expert's own estimate and was not experimentally confirmed. It is probably an optimistic over-estimate. Systems' accuracy on new data was estimated in the usual way: 70% of the available data was randomly chosen for learning, and the remaining 30% was diagnosed by the learned rule. The system's diagnoses on the "new" data were then compared with the known physician's diagnoses. This was repeated several times (usually ten times, to reduce statistical fluctuation) and the figures in Table 2 are the average of these repeated experiments. For comparison, the performance of "naive Bayes" (that is Bayes classification under the assumption of attribute independence) is also included. It should be noted that this straightforward application of Bayes has the disadvantage that it does not support the usual style of explanation in expert systems. It is therefore avoided in expert systems, although Michie (1990) describes a way to overcome this difficulty.

Some accuracy results in Table 2 are surprizing as in some cases the system's or expert's accuracy are lower than the percentage of the majority class. For example, in the breast-cancer domain the specialists' performance is 64% and Assistant's performance is 77%. These performances are both below the 80% percent likelihood of the majority class, so an almost uninformed clasifier, always just predicting the majority class, would score better than both human experts and the learning programs. This reflects a drawback of simple accuracy measure as the criterion of success of a classifier. The accuracy criterion does not take into account the relative difficulty of predicting particular classes and is therefore misleading, particularly in domains where the probabilities are extremely unequally distributed between classes, as in the breast cancer domain. This problem with accuracy as a performance measure is discussed in (Kononenko and Bratko 1991), and an information-based criterion is proposed. Therefore classifiers' *information scores* (in bits) are also given wherever they were available. The information scores are in all cases positive, indicating that the clasifiers are in fact always doing better than an uninformed classifier (which would, always classifying into the majority class, by definition of the information-based performance measure score zero).

One conclusion indicated by Table 2 is that the knowledge bases induced from no more than a few hundreds of examples of patients in some narrow diagnostic domain, perform better than medical doctors, including best specialists. Such a conclusion has been empirically confirmed by several other studies. This result should, of course, be taken with some qualifications. Namely, the criterion of performance here is only in terms of classification acuracy (or information score) under the condition that both the human expert and the induced classifier are given the same information. In practice, the human expert might be able to use extra information. Also, the medical doctor would typically have a much better global understanding of the problem and be capable of deeper explanation of the particular cases.

3 Attribute-based learning *vs.* relational learning

Applications of Machine Learning described above all rely on the use of attribute-based learning. Both learning examples and induced concept descriptions employ global attributes of objects and not rela-

domain	doctors nonspec.	doctors specialists	Assistant	GINESYS	LogArt	naive Bayes
lymphography 1			76%		84%	79%
lymphography 2	60% (D)	85% (D)	65% (A)	70% (C)		67%
primary tumor	32% , 0.95 bit	42% , 1.22 bit	44% , 1.38 bit	52% (C)	44% (B)	49% , 1.59 bit
breast cancer	64% , 0.03 bit	64% , 0.05 bit	77% , 0.07 bit	74% (C)	78% (B)	79% , 0.06 bit
hepatitis			83%		85%	84%
thyroid		64% , 0.59 bit	73% , 0.86 bit			68% , 0.70 bit
rheumathology		56% , 0.26 bit	61% , 0.46 bit			57% , 0.28 bit
urinary tract m			70% (A)			67%
urinary tract f			80% (A)			79%

Table 2: Performance in terms of classification accuracy and information score (in bits) on new data of the three learning systems, physicians (specialists and non-specialists), and the Bayes classifier evaluated under the assumption of attribute independence. Labels A, B, C, D in the table mean: A - old implementation of Assistant on DEC-10; B - in the case that more than one class remain un-eliminated by rules, naive Bayes is applied as tie-break; C - original data preprocessed so that unknown attribute values in data are replaced by the most likely value; D - expert physician's estimate (not measured experimentally).

tions among their parts. Well known families of such learning programs are TDIDT (e.g. Quinlan 1986), AQ (e.g. Michalski 1983), CN2 (Clark and Niblett 1989). Attribute-based learning is a relatively simple approach to learning and is therefore most widespread and widely used. The following advantages of attributional learning contribute to its success in practical applications:

- Computational efficiency
- Attributional learning is relatively well understood
- Attributional learning process is easy to understand by the users and it is straightforward to apply
- The attribute-value language is natural in many domains and many users are used to this representation
- It is well understood how to handle noisy and incomplete data in attributional learning; there are methods that handle noise very well

However, attribute-based learning also has strong limitations:

- Background knowledge can be expressed in rather limited form
- Lack of relational descriptions makes the concept description language inappropriate for some domains

Attribute-based descriptions are essentially equivalent to propositional logic. This is not sufficiently expressive for describing concepts in some application areas. An example of such a problem area is the finite-element mesh design which is described in detail in the next section.

The realization of the limitations of attribute-based learning led to a number of recent developments towards learning at the level of first-order predicate logic, including programs CIGOL (Muggleton and Buntine 1988), FOIL (Quinlan 1990), GOLEM (Muggleton and Feng 1990) and LINUS (Lavrač, Džeroski and Grobelnik 1991). This led to the establishment of a special area of Machine Learning, named by Muggleton (1990) *Inductive Logic Programming* (ILP; see also Muggleton 1992). The learning problem in ILP is formalised as: given some background knowledge B expressed as a set of predicates, some examples E and some negative examples N, find a logic formula H, such that:

$$B \wedge H \vdash E$$

and

$$B \wedge H \nvdash N$$

The following section describes an application that illustrates the suitability of this approach.

4 Application of ILP to finite-element mesh design

Dolšak and Muggleton (1991) describe an application of ILP to a problem for which the attribute-

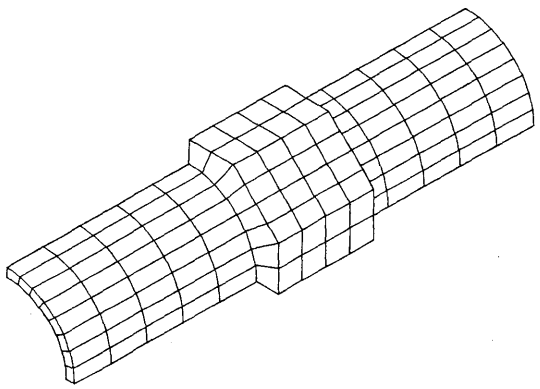


Figure 1: A cylindrical object partitioned by mesh suitable for the finite element computation. (Dolšak 1991)

based learning is unsuitable, and the relational representation appears natural. Here we illustrate this application with more recent results reported in (Dolšak 1991).

The problem of finite-element mesh design arises in numerical computation. Given, for example, a machine part and forces acting on it, the problem is to compute the pressure and deformations throughout the object. The finite-element method involves the partitioning of the given object into finite elements. Figure 1 shows an example. The resulting partition is called a finite element mesh.

For each element of the mesh, constraints in the form of equations are stated. The constraints approximately state the physical laws modelling the behaviour of the individual elements. These approximations are sufficiently accurate if the elements are sufficiently small. Generally, the finer the mesh, the smaller the error. However, a dense mesh results in a large number of equations, leading to a lengthy computation when solving the corresponding system of equations. The complexity of computation is often measured in days or weeks of CPU time and can easily become prohibitive. The problem, then, is to find a suitable compromise between the density and coarseness of the mesh.

Normally some regions of the object require denser mesh whereas in other regions a coarser mesh still suffices for good approximation. There is no known general method that would enable automatic determination of optimal, or reasonably good meshes. However, expert users of finite element methods are capable of making good guesses about proper density of the mesh in various regions of the objects. Unfortunately, the experts have difficulties in forming general rules that would enable the automation of such guesses.

In general, the mesh depends on the geometric properties of the object and forces acting on it. As pressure is transmitted between adjacent elements, the mesh density in a region of the object depends also on the adjacent regions. These general considerations were captured in Dolšak's application as background knowledge for the ILP learning in the form of properties and relations, such as:

```
short( Edge)
usual_length( Edge)
loaded( Edge)
not_loaded( Edge)
two_side_fixed( Edge)
neighbour_xy( Edge1, Edge2)
neighbour_xz( Edge1, Edge2)
...
```

The meaning of these relations is straightforward. For example, an edge is "two_side_fixed" if it is fixed at both ends. `neighbour_xy( Edge1, Edge2)` means that the edges are adjacent and they are in the xy-plane.

In an experiment to learn a characterisation of the density of a mesh in terms of these relations, five meshes known to work well were used as sources of examples for learning (Figure 2). The relation to be learned was:

```
mesh( Edge, N)
```

where `Edge` is the name of an edge in the structure, and N is the recommended number of finite elements along this edge. The target definition of this relation is to be learned in terms of the properties and relations in the given structure. All the five meshes used comprised altogether 278 edges, that is 278 positive examples for learning. The number of finite elements along the edges varied between 1 and 17. In edges with high partition, say 10, it was assumed that a similar partition would still make a good mesh, so 10 ± 1 was considered acceptable and sometimes used as another positive example. Negative examples were generated according to the closed-world assumption: if the given partitioning of an edge was 3, say, then partitionings such as 4, 5, etc. were taken as negative examples. This finally gives the following number of facts for learning in

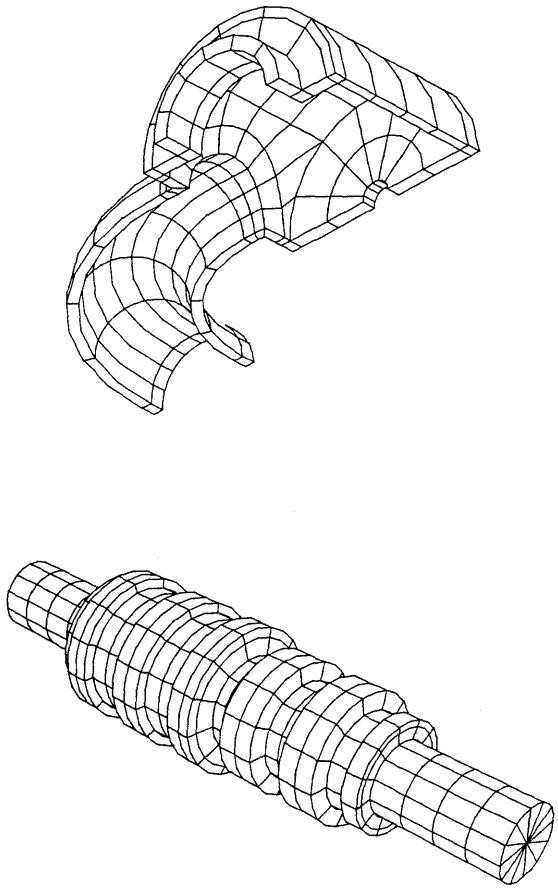


Figure 2: Two of the five meshes used for learning (Dolšak).

this experiment (Dolšak 1991):

357 positive examples
2840 negative examples
2132 background facts

Several relational learning algorithms were tried on this data: GOLEM (Muggleton and Feng 1990), LINUS (Lavrač, Džeroski and Grobelnik 1991) and FOIL (Quinlan 1990). The results obtained with GOLEM were judged to be the most satisfactory. GOLEM generated a large number of rules, some of them being practically irrelevant. For example, although logically correct, they were computationally useless when applied to classifying new edges. On the other hand, some rules appeared useful. Fortunately it was possible to formalise the criteria for distinguishing useful rules from the others. These criteria were implemented as a short Prolog program (Dolšak 1991) for postprocessing the rules generated by GOLEM.

The so resulting set of rules were of interest to expert users of the finite element methods. According to their comments, these rules reveal interesting relational dependences. The following is an example of such a generated rule (the generated syntax is that of Prolog clauses):

```
mesh( Edge, 7)  :-
  usual_length( Edge),
  neighbour_xy( Edge, EdgeY),
  two_side_fixed( EdgeY),
  neighbour_zx( EdgeZ, Edge),
  not_loaded( EdgeZ).
```

This rule says that an appropriate partitioning of `Edge` is 7 if `Edge` has a neighbour `EdgeY` in the xy-plane so that `EdgeY` is fixed at both ends, and `Edge` has another neighbour `EdgeZ` in the xz-plane so that `EdgeZ` is not loaded.

The following is a recursive rule also generated by GOLEM:

```
mesh( Edge, N)  :-
  equal( Edge, Edge2),
  mesh( Edge2, N).
```

This observes that an edge's partition can be determined by looking for an edge of the same length and shape in the same object. Of course, for this rule to be computationally useful, at least some of such equivalent edges has to have its partition determined by its own properties and those of its neighbours.

The accuracy of the induce knowledge base was estimated by a cross-validation mehod. Thereby a subset of 10 % of the example edges was effectively removed from the training set. The remaining 90 % of the data was used for rule induction, and the so induced rules were applied to the removed 10 % of the data now used as a test set. This was repeated ten times.

The results can be summarised as follows. On the average, the classification on the test set was correct in 78 % of the tested edges, incorrect in 2 % of the edges, and the edge remained unclassified (partition unknown) in 20 % of the test edges. An edge remains unclassified if there is no induced rule covering the edge.

In another, more practically realistic evaluation attempt, the generated knowledge base was applied to determining a mesh for a completely new structure, one not used for learning (shown in Figure 1). In this case, 67 % of the edges were classified correctly, 22 % incorrectly, and 11 % remained unclassified.

These results were input into a commercial automatic mesh generator as a partial specification of the mesh. The partial mesh was then completed automatically by the mesh generator, resulting in the mesh shown in Figure 3a. This mesh is close to the known good mesh of Figure 1, but unfortunately not quite acceptable with respect to the resulting numerical errors. Figure 3b shows the mesh generated by the commercial generator without any guidance from the user. This mesh is certainly fine enough with respect to the numerical erros, but completely unacceptable with respect to the computational complexity it requires. Figure 3c is again generated by the commercial generator, only this time guided by the user's advice regarding the "global" size of the elements in the mesh. This is again a deficient mesh which illustrates the generator's inability to adjust the density of the mesh in various regions of the object according to the criticality of the region. Comparing the meshes in Figures 3a-c it becomes clear that the induced knowledge base does "understand" the criticality of varios regions of the object and tries to adjust the density accordingly.

The mesh resulting from the induced knowledge base can actually be easily improved. There is a well known rule of thumb in mesh design that in a rectangular mesh the ratio between the length and width of elements should not exceed 2. Applying this rule to mending the mesh of Figure 3a in fact results in the very good mesh of Figure 1.

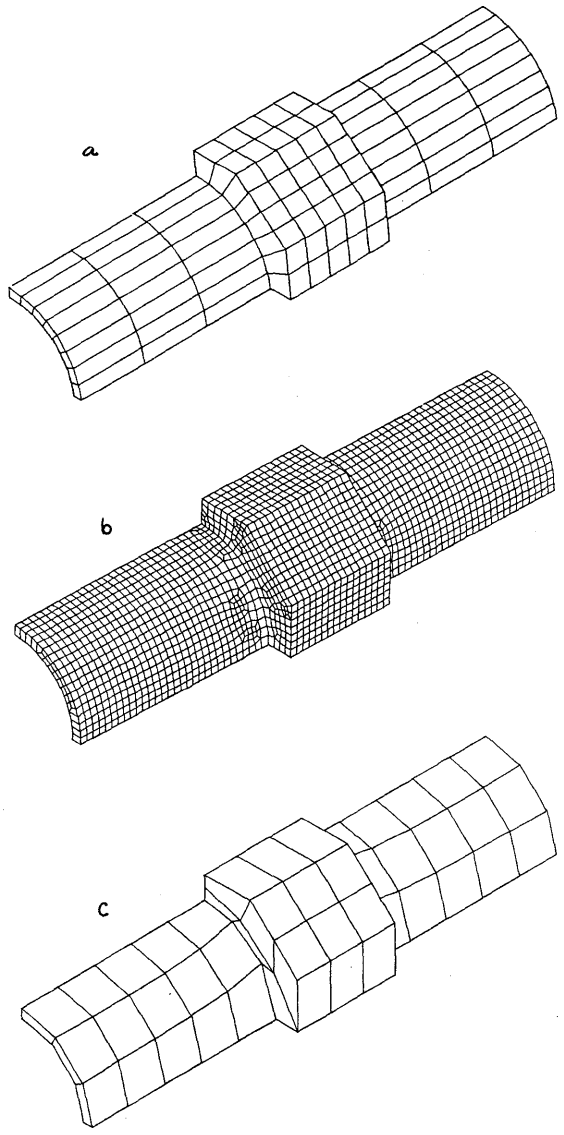


Figure 3: (a) A mesh generated by the induced knowledge base and completed by a commercial generator. (b) The mesh generated by the automatic mesh generator completely autonomously, without any guidance from the user. (c) A mesh, generated by the mesh generator, guided by the user's advice recommending the "global granularity" of 150 mm. (Dolšak 1991)

5 Towards knowledge synthesis

As illustrated by the applications described in this paper, and concluding from many other applications, ML techniques have proved to be a useful tool for efficient construction of expert systems for tasks like classification, prediction, decision making etc. In our experience, for example in the medical domains, employing ML it was possible to inductively construct competent diagnostic systems in the matter of months, weeks or even days (including time for defining the problem, choice of attributes, preparation of learning data, etc.) when it would take much longer without learning.

Muggleton (1991) and Clark et al. (1991) describe another comparison between dialogue-based and induction-based knowledge acquisition for large expert systems with thousands or tens of thousands of rules. That comparison showed that in projects employing ML the knowledge acquisition effort in man years (relative to the number of rules) was one or two orders of magnitude lower than in dialogue-based acquisition. It should be admitted that the basis for comparison was simply the number of rules in the knowldge-base per man-year invested. The quality of rules was not considered. Although the inductively constructed knowledge bases perform accurately, the question still remains whether automatically synthesised knowledge represents symbolically meaningful information. That is, does it tell the humans something about the problem domain in a transparent way that also fits nicely into the human's normal understanding of the domain. In other words, whatever has been induced from examples, does it deserve to be called *knowledge*?

In ML there has been strong awareness of the importance of this comprehensibility criterion (for example Michie 1986 and 1988). There exist some standard techniques that help in this respect. For example, tree pruning in induction of decision trees, in addition to supressing noise, often improves the transparency of induced trees enormously by simply reducing the tree size to, say, 10% of its original size. It should be admitted, however, that compactness is only one measure that is usually correlated with meaningfulness. Human experts often prefer less compact, possibly redundant descriptions, because thay better correspond to the way the problem domain is structured in their heads, or to the way that the knowledge is to be used. The use of knowledge may require not only classification, but for example the achievement of certain goals, explanation, planning, or making decision on the basis of incomplete information. Criteria to decide whether given information deserves to be called *knowledge* are intricate. Of course, these criteria do not exactly correspond to simple measures of accuracy or compactness of induced rules. Identifying and formalising these criteria is an important research topic. Still, there has already been some success in the direction of automatically inducing meaningful information. Knowledge has been generated through ML that was of interest and revealing to human experts.

I will illustrate this by an example from the KARDIO project (Bratko, Mozetič and Lavrač, 1989). In KARDIO, a deep qualitative model of the heart was compiled for efficiency reasons into a large shallow diagnostic knowledge base. This was then compressed, using ML techniques, into a small number of equivalent prediction and diagnostic rules. It was interesting to compare these mechanically synthesised descriptions with human-synthesised descriptions that can be found in the medical literature.

Here is an example of a synthesised prediction rule which tells what are the characteristic features in the ECG signal in the case of the disorder called AV block of the third degree (avb3 for short, possibly combined with any number of other defects in the heart):

```
[av_conduct = avb3] is characterised by
   [rhythm_QRS = regular] and
   [relation_P_QRS = independent_P_QRS]
```

This rule is in the VL1 formalism, normally used in the AQ family of programs (Michalski 1983). The propositions have the form `[attribute = value]`. Figure 4 illustrates what essentially happens in the case of the avb3 defect.

For comparison, one of the classical books on ECG (Goldman 1976) describes this arrhythmia as follows: "In this condition the atria and ventricles beat entirely **independently** of one another. ... The ventricular **rhythm** is usually quite **regular** but at a much slower rate (20-60)." Some words here are in bold face to help the comparison between Goldman's description and the machine synthesised description. It is easy to notice strong similarities between both descriptions. It is nice that even the same qualitative descriptors, such as **independent** or **regular** appear in both descriptions. Goldman notices that the ventricular rate is usually much lower (20-60) which is not mentioned in the machine generated description. This is in fact

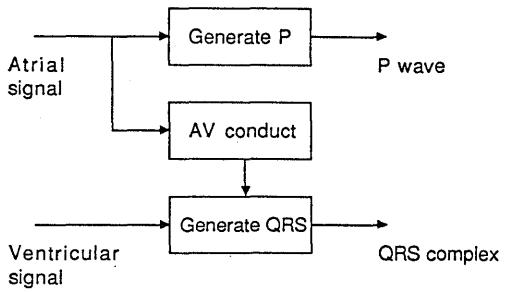


Figure 4: The mechanism of the heart disorder called av-block of the third degree. In the normal heart, the atrial signal reaches the ventricles through the AV-conductance and affects the QRS complex. In the case of the av-block, the atrial signal cannot propagate to the ventricles and has no effect on the QRS complex.

the only essential difference between both descriptions. The reason that the ventricular rate is not mentioned in the machine-generated description is that it is redundant with respect to distinguishing between those conditions of the heart in which avb3 appears, and those in which it does not. Another authority on ECG, Phibbs (1973) describes avb3 as: "(1) The atrial and ventricular **rates are different**: the atrial rate is faster; the ventricular rate is slow and **regular**. (2) There is **no consistent relation between P waves and QRS complexes**." Again, some descriptors are in bold face to facilitate comparison with the machine-generated description. The comparison is rather straightforward in this case as well.

The example above shows how well some of the synthesised descriptions correspond to those in the standard medical literature. On the other hand, some of the synthesised descriptions are considerably more complex than those in the literature. Machine-generated descriptions in such cases give much more detail that may not be necessary for an intelligent reader with a physiological background. Such a reader can usually infer the missing detail from the background knowledge. Making induced descriptions appealing to humans requires adding some redundancy or leaving out some information that can be usually recovered from background knowledge. How to add and leave out just the right amount is an open research problem.

6 Conclusions

A large number of ML applications confirm the practical importance of this technology. Experience shows that inductive knowledge-acquisition is typically an iterative process whereby the representation, background knowledge and example sets are gradually refined through experiments and feedback obtained from the domain expert. ML tools are repeatedly applied. Induction from examples can be viewed as a way of compiling a high level specification where the specification consists of examples and background knowledge. The practical advantage of this approach lies in the fact that it is often easier to obtain examples (e.g. from the domain expert) than to extract from the expert explicit general laws about the domain.

Until now, attribute-based learning has enjoyed most success in practice. However, the recent important developments in inductive logic programming (ILP) go beyond the limitations of the attribute-based learning. Recent applications of ILP include, in addition to the mesh design described in this paper, the prediction of protein secondary structure (Muggleton et al. 1992). Another exciting area facilitated by ILP is automated construction of qualitative models from observed behaviours. Work that has been done in this direction includes (Mozetič 1987a,b; also described in Bratko et al. 1989), (Coiera 1989), (Bratko et al. 1991) and (Krann et al. 1991)

Acknowledgements

A number of medical doctors helped in the applications described in this paper. I would like to thank S. Hojker, G. Klanjšček, J.Lamovec, V. Pirnat, M. Soklič and M. Zwitter of the University Medical Center, Ljubljana, P. Abrams and M. Torrens of the Clinical Investigation Unit, Ham Green Hospital, Bristol, UK, and G. Gong of Carnegie-Mellon University, Pittsburgh, who helped in the medical applications of induction either by the data for learning or by advice. M. Horvat, B. Čerček, A. Grad and P. Rode assisted with the cardiological expertise in the KARDIO project. I would also like to acknowledge the cooperation of my AI colleagues whose names appear in the references. Special thanks to Igor Kononenko for his work in medical domains, Bo-

jan Dolšak for lending the mesh design figures, and Sašo Džeroski who helped in the preparation of this paper.

This paper was written while the author was visiting the Strathclyde University and Turing Institute, Glasgow, Scotland

References

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J. (1984) *Classification and Regression Trees.* Belmont, CA: Wadsworth.

Bratko, I., Kononenko, I. (1987) Learning diagnostic rules from incomplete and noisy data. In: *Interactions in AI and Statistics* (ed. B. Phelps) London: Gower Technical Press.

Bratko, I., Mozetič, I, Lavrač, N. (1989) *KARDIO: A Study in Deep and Qualitative Knowledge for Expert Systems.* Cambridge, Massachusetts: The MIT Press.

Bratko,I., Muggleton, S., Varšek, A. (1991) Learning qualitative models of dynamic systems, *Proc. Inductive Logic Programming ILP-91*, Viana do Castelo, Portugal, March 1991. (An abbreviated version also in *Machine Learning: Proc. Eighth Int. Workshop*, eds. L.A. Birnbaum and G. Collins, San Mateo, CA: Morgan Kaufmann).

Bratko, I., Mulec, P. (1979) An experiment in automatic learning of diagnostic rules. *Informatica*, Vol. 4, No. 4, 353-359.

Cestnik, B., Bratko, I. (1988) Learning redundant rules in noisy domains, *Proc. ECAI 88, European Conf. on Artificial Intelligence*, Muenchen, August 1988

Cestnik, B., Kononenko, I., Bratko, I. (1987) ASSISTANT 86: a knowledge elicitation tool for sophisticated users. In *Progress in Machine Learning* (eds. I. Bratko, N. Lavrač; Proc. Second European Working Session on Learning. Bled, Slovenia, 1987) Wilmslow, England: Sigma Press, 1987.

Chandrasekaran, B. (1991) Interview with Frederick Hayes-Roth and Richards Fikes, *IEEE Expert*, October 1991, pp. 3-14.

Chilausky, R., Jacobsen, B., Michalski, R.S. (1976) An application of variable-valued logic to inductive learning of plant disease diagnostic rules. *Proceedings of the sixth international symposium on multiple-valued logic.* May 25-28, 1976, Logan, Utah.

Clark, P., Cestnik, B., Sammut, C., Stender, J. (1991) Applications of machine learning: notes from the panel members. *Machine Learning - EWSL-91; Proc. European Working Session on Learning* (Porto, Portugal, March 1991) Springer-Verlag.

Clark, P., Niblett, T. (1989) The CN2 induction algorithm. *Machine Learning*, Vol. 3, no. 4, 261-284.

Coiera, E. (1989) Generating qualitative models from example behaviours. DCS Report No. 8901, School of Electr. Eng. and Computer Sc., Univ. of New South Wales, Sydney, Australia.

Dolšak, B., Muggleton, S. (1991), The application of Inductive Logic Programming to Finite Element Mesh Design. Proc. ILP-91, Viana do Castelo, Portugal.

Dolšak, B. (1991) Determining the geometric model of finite element meshes using AI methods. University of Maribor: CAD Center (M.Sc. thesis) Maribor, Slovenia.

Gams, M. (1989) New measurements highlight the importance of redundant knowledge, *Proc. 4th European Working Session on Learning* (ed. K. Morik; Montpellier, France) Pitman and Morgan Kaufmann.

Kononenko, I., Bratko, I. (1991) Information based evaluation criterion for classifier's performance. *Machine Learning*, Vol. 6, 67-80.

Krann, I., Richards, B., Kuipers, B. (1991) Automatic abduction of qualitative models, *Proc. QR-91*, Austin, Texas, May 1991.

Lavrač, N., Džeroski, S, Grobelnik, M. (1991) Learning nonrecursive definitions of relations with LINUS. *EWSL '91: Machine Learning: Proceedings of the European Working Session on Learning*, Porto, Portugal (ed. Y. Kodratoff) Springer-Verlag, Lecture Notes in Artificial Intelligence.

Leech, W.J. (1986) A rule based process control method with feedback. In: Advances in Instrumentation, Vol 41, Part 1, 69 - 175.

Michalski, R.S. (1983) A theory and methodology of inductive learning. In: *Machine Learning: an Artificial Intelligence Approach* (eds. R.S. Michalski, J.G. Carbonell and T.M. Mitchell) Palo Alto, CA: Tioga.

Michalski, R.S., Chilausky, R.L. (1980) Learning by being told and learning from examples: an experimental comparison of the two methods of knowl-

edge acquisition in the context of developing an expert system for soybean disease diagnosis. *International Journal of Policy Analysis and Information Systems*, Vol. 4, 125-161.

Michie, D. (1986), The superarticulacy phenomenon in the context of software manufacture. *Proceedings of the Royal Society A*, London, Vol. 405, 185 - 212.

Michie, D. (1988) Machine learning in the next five years. *EWSL-88 – Proc. 3rd European Working Session on Learning*, Glasgow, 1988. London: Pitman.

Michie, D. (1989) Problems of computer-aided concept formation. In *Applications of Expert Systems, Vol. 2* (ed. J.R. Quinlan) Turing Institute Press in association with Addison-Wesley)

Michie, D. (1990) Personal models of rationality. *Journal of Statistical Planning and Inference*, Vol. 25, 381-399.

Michie, D., Muggleton, S., Riese, C., Zubrick, S. (1984) RuleMaster: a second generation knowledge engineering tool, *Proc. 1st Conf. Artificial Intelligence Applications, IEEE Computer Society.*

Mingers, J. (1989a) An empirical comparison of selection measures for decision tree-induction. *Machine Learning*, Vol. 3, No. 4, 319-342.

Mingers, J. (1989b) An empiricial comparison of pruning methods for decision-tree induction. *Machine Learning*, Vol. 4, No. 2.

Mozetič, I. (1987a) Learning of qualitative models. In *Progress in Machine Learning* (eds. I. Bratko, N. Lavrač; Proc. 2nd European Working Session on Learning, Bled, Slovenia) Wilmslow: Sigma Press.

Mozetič, I. (1987b) The role of abstractions in learning qualitative models. *Proc. Fourth Int. Workshop on Machine Learning*, Irvine, CA, June 1987. Morgan Kaufmann.

Muggleton, S. (1991), Inductive logic programming, *New generation computing*, Vol. 8, 295 - 318.

Muggleton, S. (1992, ed.) *Inductive Logic Programming*, Academic Press.

Muggleton, S., Buntine, W. (1988) Machine invention of first order predicates by inverting resolution. *Proc. 5th Int. Conf. Machine Learning*, Ann Arbor, MI, 1988.

Muggleton, S., Feng, C. (1990) Efficient induction of logic programs. *Proc. First Conf. Inductive Learning Theory*, Tokyo: Japanese Society for Artificial Intelligence, 1990.

Muggleton, S., King, R.D., Sternberg, M.J.E. (1992) Protein secondary structure using Logic (submitted for publication).

Niblett, T., Bratko, I. (1986) Learning decision trees in noisy domains. In: *Expert Systems 86: Proc. Expert Systems 86 Conf.* (ed. M. Bramer) Cambridge Univ. Press.

Pirnat, V., Kononenko, Janc, T., Bratko, I., Medical analysis of automatically induced diagnostic rules, *Proc. Third Int. Conf. AI in Medicine*, London, August 1989.

Quinlan, J.R. (1979), Discovering rules by induction from large collections of examples. In *Expert systems in the Microelectronic Age* (ed. D. Michie) Edinburgh University Press.

Quinlan, J.R. (1986) Induction of decision trees. *Machine Learning Journal*, Vol. 1, 81-106.

Quinlan, J.R. (1990) Learning logical definitions from relations. *Machine Learning Vol. 5*, pp 239-266.

Quinlan, J.R., Compton, P., Horn, K.A., Lazarus, L. (1989) Inductive knowledge acquisition: A case study. In *Applications of Expert Systems, Vol. 2* (ed. J.R. Quinlan) Turing Institute Press in association with Addison-Wesley.

Roškar, E., Abrams, P., Bratko, I., Kononenko, I., Varšek, A., MCUDS - an expert system for the diagnostics of lower urinary tract, *Journal of Biomedical Measurement Informatics and Control*, Vol. 1, No. 4, 201-204 (1986).

Urbančič, T., Kononenko, I., Križman, V. (1991) Review of AI applications at Ljubljana AI Labs. Ljubljana, J. Stefan Institute: Technical Report DP-6183.

Author Index

Vol.1 1– 460
Vol.2 461–1218