Egon Börger and Robert Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

March 11, 2003

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Preface

Quelli che s'innamoran di pratica senza scienzia
sono come 'l nocchieri ch'entra in navilio sanza timone o bussola,
che mai ha certezza dove si vada.[1]
— Leonardo da Vinci

Ich habe oft bemerkt, dass wir uns durch allzuvieles Symbolisieren
die Sprache für die Wirklichkeit untüchtig machen.[2]
— Christian Morgenstern

This is the place to express our thanks. First of all we thank all those who over the years have actively contributed to shaping the novel software design and analysis method explained in this book. They are too numerous to be mentioned here. They all appear in some way or the other on the following pages, in particular in the bibliographical and historical Chap. 9 which can be read independently of the book. We then thank those who have helped with detailed critical comments on the draft chapters to shape the way our arguments are presented in this book: M. Börger (Diron Münster), I. Craggs (IBM Hursley), G. Del Castillo (Siemens München), U. Glässer (Simon Fraser University, Vancouver, Canada), J. Huggins (Kettering University, Michigan, USA), B. Koblinger (IBM Heidelberg), P. Päppinghaus (Siemens München), A. Preller (Université de Montpellier, France), M.-L. Potet (INP de Grenoble, France), W. Reisig (Humboldt-Universität zu Berlin, Germany), H. Rust (Universität Cottbus, Germany), G. Schellhorn (Universität Augsburg, Germany), B. Thalheim (Universität Cottbus, Germany) and a dozen student generations at Università di Pisa. We thank M. Barmet (ETH Zürich) for her solutions of the exercises in Chap. 8. We also thank L. Logrippo (University of Ottawa, Canada) and D. Beecher (Carleton University, Canada) for their help with translating the above observation by Leonardo, and F. Capocci and I. Mulvany from Springer-Verlag for their careful copyediting of the typoscript.

Egon Börger, *Robert Stärk*
Pisa and Zürich, Christmas 2002

[1] Those who fall in love with practice without scientific knowledge or method are like the helmsman who enters a ship without rudder or compass, who is never certain which way it might go.

[2] I have often observed that by over-symbolizing we make the language inefficient to use in the real world.

Contents

1 Introduction

The method. This book introduces a systems engineering method which guides the development of software and embedded hardware–software systems seamlessly from requirements capture to their implementation. It helps the designer to cope with the three stumbling-blocks of building modern software based systems: size, complexity and trustworthiness. The method bridges the gap between the human understanding and formulation of real-world problems and the deployment of their algorithmic solutions by code-executing machines on changing platforms. It covers within a single conceptual framework both *design and analysis*, for procedural single-agent and for asynchronous multiple-agent distributed systems. The means of analysis comprise as methods to support and justify the reliability of software both *verification*, by reasoning techniques, and experimental *validation*, through simulation and testing.

The method *improves current industrial practice* in two directions:

- On the one hand by accurate high-level modeling at the level of abstraction determined by the application domain. This raises the level of abstraction in requirements engineering and improves upon the loose character of human-centric UML descriptions.
- On the other hand by linking the descriptions at the successive stages of the system development cycle in an organic and effectively maintainable chain of rigorous and coherent system models at stepwise refined abstraction levels. This fills a widely felt gap in UML-based techniques.

Contrary to UML, the method has a *simple scientific foundation*, which adds precision to the method's practicality. Within the *uniform conceptual framework* offered by the method one can consistently relate standard notions, techniques and notations currently in use to express specific system features or views, each focussed on a particular system aspect, such as its structure, environment, time model, dynamics, deployment, etc. (see Sect. 7.1). Thereby the method supports a *rigorous integration of common design, analysis and documentation techniques* for model reuse (by instantiating or modifying the abstractions), validation (by simulation and high-level testing), verification (by human or machine-supported reasoning), implementation and maintenance (by structured documentation). This improves upon the loose ties between different system design concepts as they are offered by the UML framework.

Target audience. This book combines the features of a handbook and of a textbook and thus is addressed to hardware–software system engineers (architects, designers, program managers and implementers) and researchers as well as to students. As a handbook it is conceived as a Modeling Handbook for the Working Software Engineer who needs a practical high-precision design instrument for his daily work, and as a Compendium for Abstract State Machines (ASMs). As a textbook it supports both self-study (providing numerous exercises) and teaching (coming with detailed lecture slides in ppt and/or pdf format on the accompanying CD and website `http://www.di.unipi.it/AsmBook/`). We expect the reader to have some experience in design or programming of algorithms or systems and some elementary knowledge of basic notions of discrete mathematics, e.g. as taught in introductory computer science courses. Although we have made an effort to proceed from simple examples in the earlier chapters to more complex ones in the later chapters, all chapters can be read independently of each other and unless otherwise stated presuppose only an understanding of a rather intuitive form of abstract pseudo-code, which is rigorously defined as ASM in Sect. 2.2.2. We have written the text to enable readers who are more interested in the modeling and less in the verification aspects to skip the proof sections.[1] The hurried reader may skip the numerous footnotes where we refer to interesting side issues or to related arguments and approaches in the literature.

There is another book through which the reader can learn the ASM method explained in this book, namely [406], which contains the up-to-now most comprehensive non-proprietary real-life ASM case study, covering in every detail ground modeling, refinement, structuring, implementation, verification and validation of ASMs. The focus of that book however is an analysis of Java and its implementation on the Java Virtual Machine (including a detailed definition and analysis of a compiler and a bytecode verifier), as a consequence it uses only basic and turbo ASMs (see Sect. 2.2, 4.1). The present book is an *introduction* to practical applications of the ASM method via small or medium-size yet characteristic *examples from various domains*: programming languages, architectures, embedded systems, components, protocols, business processes. It covers also real-time and asynchronous ASMs. In addition it provides the *historical and* the *theoretical background* of the method. We hope this book stimulates further technological and research developments, ranging from industrial applications to theoretical achievements.

[1] Mentioning this possibility does not mean that we consider system verification as an optional. It reflects the support the method presented in this book provides to systematically *separate different concerns within a well-defined single framework* so that one can ultimately tie the different threads together to achieve a design which via its analysis is certifiable as trusted (see Sect. 2.1).

In various places we state some problems whose solution we expect to contribute to the further advancement of the ASM method. They are collected in a list at the end of the book.

In the rest of this introduction we state in more detail the practical and theoretical goals of the book and survey its technical contents.

1.1 Goals of the Book and Contours of its Method

Through this book we want to introduce the reader into a hierarchical modeling technique which

– makes accurate virtual machine models amenable to mathematical and experimental analysis,
– links requirements capture to detailed design and coding,
– provides on the fly a documentation which can be used for inspection, reuse and maintenance.

The open secret of the method is to use abstraction and stepwise refinement, which often are erroneously understood as intrinsically "declarative" or syntax-based concepts, on a *semantical* basis and to combine them with the *operational* nature of machines. Such a combination (Sect. 1.1.1) can be obtained by exploiting the notion of Abstract State Machines (Sect. 1.1.2) – which gave the name to the method – and results in considerable practical benefits for building trustworthy systems (Sect. 1.1.3). We also shortly describe here what is new in the ASM method with respect to the established method of stepwise refining pseudo-code (Sect. 1.1.4) and what it adds to UML based techniques (Sect. 1.1.5).

1.1.1 Stepwise Refinable Abstract Operational Modeling

The hardware and software system engineering method that this book introduces is based upon semantical abstraction and structuring concepts which resolve the tension deriving from the simultaneous need for *heterogeneity*, to capture the richness and diversity of application domain concepts and methods, and for *unification*, to guarantee a consistent seamless development throughout. In fact it allows one to efficiently relate the two distant ends of each complex system development effort, namely the initial problem description for humans and the code running on machines to solve the problem. More precisely, using this method the system engineer can

– derive from the application-domain-governed understanding of a given problem, gained through requirements analysis, a correct and complete human-centric task formulation, called the *ground model*, which is the result of the requirements capture process, is expressed in application-domain

terms, is easy to understand and to modify, and constitutes a binding contract between the application domain expert (in short: the customer) and the system designer,[2]
- refine the ground model by more detailed descriptions which result from the relevant design decisions, taken on the way to the executable code and documented by the intermediate models which typically constitute a *hierarchy of refined models*,
- link the most detailed specification to *generated code*, to be run on various platforms and implementing computers and to be shown to correctly solve the problem as formulated in the ground model (the contract with the customer).

The conceptual key for crossing these rather different and for complex software systems numerous levels of abstraction is to maintain, all the way from the ground model to the code, a *uniform algorithmic view*, based upon an *abstract notion of run*, whether of agents reacting to events or of a virtual machine executing sequences of abstract commands. Having to deal in general with sets of "local actions" of multiple agents, an encompassing concept of basic "actions" is defined as taking place in well-defined abstract local states (which may depend on environmently determined items) and producing well-defined next states (including updates of items in the environment). We interpret simultaneous basic local actions as a generalized form of Dijkstra's guarded commands [185]: under explicitly stated conditions they perform updates of finitely many "locations", which play the role of abstract containers for values from given domains of objects, considered at whatever given level of abstraction. Those objects residing in locations, together with the functions and relations defined on them, determine the abstract states the computation is about.[3] The simple and intuitive mathematical form we adopt to represent this idea of transformations of abstract states for the system engineering method explained in this book is the notion of Abstract State Machines (ASM).[4]

[2] This does not preclude evolution of the ground model during the development process. Ground models need to be developed "for change", but at each development stage *one* version of a well-defined ground model is maintained; see below.

[3] Object-oriented methods in general and UML in particular share this first-order logic "view of the world" as made up of "things" ("abstractions that are first-class citizens in a model" [69]) and their "relationships".

[4] The idea of using "abstract" state transformations for specification purposes is not new. It underlies the event driven version of the B method [5, 6] with its characteristic separation of individual assignments from their scheduling. It underlies numerous state machine based specification languages like the language of statecharts [271] or Lampson's SPEC [316], which besides parallelism (in the case of SPEC including the use of quantifiers in expressions) and non-determinism offer constructs for non-atomic (sequential or submachine) execution, see Chap. 4. It underlies the wide-spectrum high-level design language COLD (see Sect. 7.1.2). It also underlies rule-based programming, often characterized as repeated local-

1.1.2 Abstract Virtual Machine Notation

This book explains the three constituents of the *ASM method*: the notion of ASM, the ground model technique and the refinement principle. The concept of ASMs (read: pseudo-code or Virtual Machine programs working on abstract data as defined in Sects. 2.2, 2.4) offers what for short we call *freedom of abstraction*, namely the unconstrained possibility of expressing appropriate abstractions directly, without any encoding detour, to

– build ground models satisfying the two parties involved in the system contract, tailoring each model to the needs of the problem as determined by the particular application, which may belong to any of a great variety of conceptually different domains (see Sect. 2.1.1), and keeping the models simple, small and flexible (easily adaptable to changing requirements),
– allow the designer to keep control of the design process by appropriate refinement steps which are fine-tuned to the implementation ideas (see Sect. 2.1.2).

Most importantly ASMs support the practitioner in exploiting their power of *abstraction in terms of an operational system view* which faithfully reflects the natural intuition of system behavior,[5] at the desired level of detail and with the necessary degree of exactitude. The underlying simple mathematical model of both synchronous and asynchronous computation allows one to view a system as a set of cooperating idealized mathematical machines which step by step – where the chosen level of abstraction determines the power of a step – perform local transformations of abstract global states. Essentially each single machine (driven by an agent, also called a thread) can be viewed as executing pseudo-code on arbitrary data structures, coming with a clear notion of state and state transition. This empowers the designer to work at every level of development with an accurate yet transparent concept of system *runs* for modeling the dynamic system behavior, whether the execution is done mentally (for the sake of high-level analysis) or by real machines (for the sake of high-level testing of scenarios). The availability of the concept of a run at each level of abstraction provides the possibility of also modeling non-functional features, like performance or reliability, or run time inspection of metadata associated with components as offered by CORBA and COM. Due to the mathematical nature of the concepts involved, established structuring, validation and verification techniques can be applied to ASM models, supporting architectural structuring principles and providing platform and

ized transformations of shared data objects (like terms, trees, graphs), where the transformations are described by rules which separate the description of the objects from the calculations performed on them and on whose execution various constraints and strategies may be imposed.

[5] See the observation in [399, Sect. 2.4] that even the knowledge base of experts has an operational character and guarded command form: "in *this* situation do *that*", which is also the form of the ASM transition rules defined in Sect. 2.2.2.

programming language-independent executable models which are focussed on the application-domain-relevant problem aspects and lend themselves to reuse in a design-for-change context.

1.1.3 Practical Benefits

The need to improve current industrial software engineering practice is widely felt. To mention only a few striking examples: too many software projects fail and are canceled before completion or are not delivered on time or exceed their budget,[6] the energy spent on testing code is ever increasing and tends to represent more than half of the entire development cost, the number of errors found in complex software is often rather high, there is almost no software warranty whatsoever, but again and again the world is surprised by Trojan horses and security holes, etc.

The major benefit the ASM method offers to practitioners for their daily work is that it provides a simple precise *framework to communicate and document design ideas* and a *support for an accurate and checkable overall understanding* of complex systems. Using a precise, process-oriented, intuitive semantics for pseudo-code on arbitrary data structures, the developer can bind together the appropriate levels of abstraction throughout the entire design and analysis effort. This implies various concrete benefits we are going to shortly mention now.

First of all the ASM method supports *quality from the beginning* using hierarchical modeling, based on ground modeling and stepwise refinement coupled to analysis. The method establishes a discipline of development which allows structuring (into appropriate abstractions), verification and validation to become criteria for good design and good documentation.[7] By its conceptual simplicity and the ease of its use, the ASM method makes the quality of the models depend only on the expertise in the application or design domain and on the problem understanding, not on the ASM notation. This is the reason why we expect to contribute with this book to making the method become part of every development activity which aims at ensuring that the produced model has the desired properties (e.g. to satisfy the needs of the future users), instead of waiting until the end of the development process to let another team (or the users) remove bugs in the code.[8]

[6] Some figures from the Standish Group 1998: 9% out of the 175 000 surveyed software projects are delivered on time and under budget, 52% go over budget by an average of 18%, 31% are canceled before completion.

[7] A quote from the recommendations of the UK Defense Standard 00–54, 1999 (Requirements for Safety-Related Electronic Hardware in Defense Equipment): "A formally-defined language which supports mathematically-based reasoning and the proof of safety properties shall be used to specify a custom design." See http://www.dstan.mod.uk/data/00/054/02000100.pdf.

[8] We believe it to be mistaken to relegate the specification and verification work, if done at all, to separate so-called "formal methods" teams. The work of such

The freedom ASMs offer to model arbitrarily complex objects and operations directly, abstracting away from inessential details (e.g. of encoding of data or control structures), allows one to *isolate the hard part of a system* and to turn it into a precise model which exposes the difficulties of the system but is simple enough to be understood and satisfactorily analyzed by humans.

As a by-product such core models and their refinements yield valuable *system documentation*: (a) for the customers, allowing them to check the fulfillment of the software contract by following the justification of the design correctness, provided in the form of verified properties or of validated behavior (testing for missing cases, for unexpected situations, for the interaction of to-be-developed components within a given environment, etc.), (b) for the designers, allowing them to explore the design space by experiments with alternative models and to record the design rationale and structure for a checkable communication of design ideas to peers and for later reuse (when the requirements are changed or extended, but the design diagrams on the whiteboard are already erased or the developers are gone),[9] (c) for the users, allowing them to get a general understanding of what the system does, which supports an effective system operator training and is sufficiently exact to prevent as much as possible a faulty system use,[10] and (d) for the maintainers, allowing them to analyse faulty run-time behavior in the abstract model.

Despite the abstract character of ASM models, which characterizes them as specifications, they can and have been refined to machine executable versions in various natural ways (see Sect. 8.3). Due to their abstract character they *support generic programming, hardware–software co-design as well as portability* of code between platforms and programming languages – the major goal of the "model driven architecture" approach to software development. ASM models are in particular compatible with cross-language interoperable implementations as in .NET. Since they are tunable to the desired level of abstraction, they support information hiding for the management of software development and the formulation of well-defined interfaces for component-based system development. The general refinement notion supports a method of stepwise development with traceable links between different system views

teams may contribute to a better high-level understanding of the system under development, but if the program managers and the implementers do not understand the resulting formalizations, this does not solve the fundamental problem of keeping the models at the different abstraction levels in sync with the final code – the only way to make sure the code does what the customer expects from the agreed upon ground model. The ASM method is not a formal method in the narrow understanding of the term, but supports any form of rigorous design justification. This includes in particular mathematical proofs, which represent the most successful justification technique our civilization has developed for mental constructions – the type of device to which models and programs belong. See the footnote to the "language and communication problem" at the beginning of Sect. 2.1.1.

[9] In this way a design does not remain only in the head of its creator and can play a role in later phases of the software life cycle.

and levels. In particular this allows one to localize the appropriate design level where changing requirements can be taken into account and where one can check that the design change is not in conflict with other already realized system features.

1.1.4 Harness Pseudo-Code by Abstraction and Refinement

As is well-known, pseudo-code and the abstraction-refinement pair are by no means new in computer science, often they are even looked at with scepticism – by theoreticians who complain about the lack of an accurate semantics for pseudo-code, by practitioners who complain about the difficulty of understanding the mathematics behind formalisms like abstract data types, algebraic specifications, formal methods refinement schemes, etc. So what value does the ASM method add to these omnipresent ideas and how does it avoid the difficulty to apply them in practical software engineering tasks?

The ASM method makes the *computational meaning of abstraction and refinement* available explicitly, in a mathematically precise but simple, easily understandable and easily implementable pseudo-code-like setting, including the crucial notion of runs. The formulation uses only *standard mathematical and algorithmic terms*, circumventing the unnecessary formalistic logico-algebraic complications that practitioners so often rightly complain about, so that the method comes as a set of familiar intuitive concepts which naturally support the practitioners' daily development work. To read and write ASMs no knowledge of the underlying theory is needed, though it is the mathematical underpinning which makes the method work. This is analogous to the role of axiomatic set theory, which provides the precise setting in which mathematicians work without knowing about the logical foundation.

Looking back into the history of computing reveals that the ingredients of the concept of the Abstract State Machine were there for decades before they appeared combined in the definition discovered in [248], triggered by a purely foundational concern:[11] (a) pseudo-code, (b) IBM's concept of virtual machines [305] and Dijkstra's related concept of abstract machines [183] (both born as operating system abstractions), and (c) Tarski structures as the most general concept of abstract states [325, 265, 359, 210]. It is the mathematical character one can attach through the notion of ASM to the semantically open-ended loose notions of pseudo-code and virtual machines which turns these concepts into elements of a scientifically well-founded method; it is the natural expression of fundamental intuitions of computing through ASMs and the simplicity of their definition which make the ASM method comprehensible for the practitioner and feasible for large-scale industrial applications.

[10] An important example of an erroneous use of a system whose control is shared by humans and computers, e.g. in modern aircrafts, is known as *mode confusion*. See [322] and the notion of control state ASM in Sect. 2.2.6, which provides a way to make the overall mode structure of a system transparent.

[11] See Chap. 9 for details.

The ASM method does nothing else than putting the elementary definition of local updates of abstract states together with Wirth's original stepwise refinement approach [429] and with the concept of ground model [71, 72, 76] to link requirements capture to code generation in a coherent framework. Historically speaking the ASM method "complete(s) the longstanding structural programming endeavour (see [164]) by lifting it from particular machine or programming notation to truly abstract programming on arbitrary structures" [86, Sect. 3.1]. It also appears as a natural completion of the evolution during the last century from programming to generic programming and high-level platform-independent modeling: leading from programming-any-which-way in the 1950s to programming-in-the-small in the 1960s to programming-in-the-large in the 1970s to programming-in-the-world since the 1990s[12] where, due to the evergrowing hardware performance, security, robustness and reusability play a larger role than time or space efficiency.

1.1.5 Adding Abstraction and Rigor to UML Models

UML exercises a strong attraction by the multitude it offers for radically different interpretations of crucial semantical issues. On the tool side this is reflected by the numerous "UML+…-systems", e.g. UML+RUP, UML+XP, UML+IBM Global Services Method, etc. However, this conceptual multitude is not mediated (except for being simply declared to constitute so-called "semantical variation points"), in addition it is represented by a limited graphical notation and prevents UML from supporting precise practical refinement schemes (see Sect. 2.1.2, 3.2). Furthermore, the drive in UML to structure models right from the beginning at the class level imposes a rather low level of abstraction, typically close to procedural code, besides leading to the risk of a conceptual explosion of the class hierarchy. It also makes it difficult to model features which relate to multiple classes and which are often spread in the class hierarchy (e.g. safety, security, logging), or to describe crosscutting concerns relating to one feature at different class levels. Furthermore, it has no clear semantical model of "atomic" versus "durative" actions and of asynchronous computation of multiple threads.

The ASM method we explain in this book provides a means to handle such architectural features in an accurate yet transparent way and at a higher level of abstraction than UML, providing support for a truly human centric yet precise algorithmic design and analysis, which is *completely* freed from the shackles of programming language constructs and of specific typing disciplines. For example, it provides a simple accurate semantics for standard diagram techniques (see the notion of control state ASMs in Sect. 2.2.6 and the ASM definition of UML activity diagrams in Sect. 6.5.1) and for use cases and their refinements to rigorous behavioral models (see Chap. 3), it

[12] The wording is taken from Garlan's lectures on *Software Architecture* held at the Lipari Summer School on *Software Engineering*, July 2002.

supports component techniques (see Sect. 3.1.2 and [406]), it uniformly relates sequential and asynchronous computations capturing the data exchange of interacting objects (see Sect. 6), it provides a clear definition of "atomic" and "composed" computation step (see Chap. 4), etc. To realize further design steps, high-level ASM models can be *refined* by object-oriented mappings to classes, by introducing type disciplines where useful, by restricting runs when needed to satisfy specific scheduling principles, etc.

1.2 Synopsis of the Book

This book was conceived to serve the double purpose of (a) a modeling handbook and textbook, teaching how to practically apply the ASM method for industrial system design and analysis (including its management and its documentation), and of (b) an ASM compendium, providing the underlying theory and a detailed account of ASM research. The domains of application cover sequential systems (e.g. programming languages and their implementation), synchronous parallel systems (e.g. general and special-purpose architectures), asynchronous distributed systems and real-time systems (network and communication and database protocols, control systems, embedded systems). This also determines the structure of the book, which leads from the definition of *basic* single-agent ASMs in Chap. 2 with an illustration of the principles of hierarchical system design by ground model construction and stepwise refinements in Chap. 3 to *structured ASMs* in Chap. 4, *synchronous* multi-agent ASMs in Chap. 5, and *asynchronous* multi-agent ASMs in Chap. 6. This is followed by Chap. 7 on the universality of ASMs, Chap. 8 on ASM tool support (on computer-supported verification of ASMs and on ASM execution and validation techniques). It concludes with Chap. 9, which surveys the ASM research, together with its applications and industrial exploitations, from its beginning to today and comes with an, as we hope, complete annotated bibliography of ASM related papers from 1984–2002.

A detailed index (including also the major ASMs defined in this book) and lists of figures and tables aid navigation through the text. The use of the book for teaching is supported by numerous exercises, most of them coming with solutions on the accompanying CD, and by pdf and powerpoint format slides on the CD, covering most of the chapters or sections of the book. Additional material (including lecture slides) and corrections are solicited and will be made available on the ASM book web site at `http://www.di.unipi.it/AsmBook/`. This includes the set of LaTeX macros we have used to write the ASMs in this book. They come with a tutorial explaining to the reader how to write his own ASMs in a strikingly simple and elegant way using this set of macros, which by its very nature can be extended and tailored to specific needs.

Central themes of the chapters. In Chap. 2 we introduce the three constituents of the ASM approach to system design and analysis: the concept of *abstract state machines*, the *ground model method* for requirements capture, and the *refinement method* for turning ground models by incremental steps into executable code. The notion of *basic ASMs* is defined which captures the fundamental concept of "pseudo-code over abstract data", supporting its intuitive understanding by a precise semantics defined in terms of abstract state and state transition. Finite State Machines (FSMs) are extended by *control state ASMs*.

In Chap. 3 we illustrate the *ground model method* for reliable requirements capture (formulating six fundamental categories of guideline questions) and the *refinement method* for crossing levels of abstraction to link the models through well-documented incremental development steps. The examples are control state ASMs for some simple devices (ATM, Password Change, Telephone Exchange), a command-line debugger control model, a database recovery algorithm, a shortest path algorithm and a proven-to-be-correct pipelined microprocessor model. Sect. 3.2.3 presents Schellhorn's scheme for modularizing and implementing ASM refinement correctness proofs.

In Chap. 4 some standard refinements for structuring ASMs are defined and their applications illustrated. The building blocks of *turbo ASMs* are sequential composition, iteration, parameterized (possibly recursive) submachines; they permit us to integrate common syntactical forms of encapsulation and state hiding, like the notion of a *local state* and a mechanism for *returning values* and *error handling*. We characterize turbo ASM subcomputations as SEQ/PAR-tree computations. *Abstract State Processes* realize the above constructs in a white-box view where interleaving permits us within a context of parallel execution to also follow the single steps of a component computation. As an illustration we provide succinct turbo ASMs for standard programming constructs, including the celebrated Structured Programming Theorem and forms of recursion which are common in functional programming.

In Chap. 5 multi-agent synchronous ASMs are defined which support modularity for the design of large systems. They are illustrated by *sync ASMs* for solving a typical industrial plant control problem (Production Cell) and the Generalized Railroad Crossing problem (verified real-time gate controller).

In Chap. 6 asynchronous multi-agent ASMs (*async ASMs*) are defined and illustrated by modeling and analyzing characteristic distributed network algorithms (for consensus, master–slave agreement, leader election, phase synchronization, load balance, broadcast acknowledgement), a position-based routing protocol for mobile ad hoc networks, a requirements capture case study for a small embedded (Light Control) system, two time-constrained algorithms which support fault tolerance for a distributed service (Kermit and a Group Membership protocol), Lamport's mutual exclusion algorithm *Bakery* with atomic or with durative actions, and the event-driven UML activity diagrams.

In the foundational Chap. 7 we investigate the universality properties of ASMs. We show that ASMs capture the principal models of computation and specification in the literature, including the principal UML concepts. We explain the ASM thesis, which extends Church's and Turing's thesis, and prove its sequential version from a small number of postulates.

Chapter 8 is dedicated to tool support for ASMs. In Sect. 8.1 we deal with techniques for mechanically verifying ASM properties, using theorem proving systems or model checkers. We present a logic tailored for ASMs and the transformation from ASMs to FSMs which is needed for model-checking ASMs. In Sect. 8.3 we survey various methods and tools which have been developed for executing ASMs for simulation and testing purposes. The history of these developments is presented in Sect. 9.4.3, which is part of Chap. 9, where we survey the rich ASM literature and the salient steps of the development of the *ASM method* from the epistemological origins of the *notion of ASM*.

2 ASM Design and Analysis Method

In this chapter[1] we introduce the three constituents of the ASM method for system design and analysis: the concept of *abstract state machines*, the *ground model method* for requirements capture, and the *refinement method* for turning ground models by incremental steps into executable code. We focus on motivating and defining the fundamental notions underlying the ASM approach; therefore the examples are tailored to illustrate the outstanding single features of *basic ASMs*. In Chap. 3 both ground model construction and ASM refinement are explained in more detail and these are illustrated by less elementary examples of some interest in their own right. In Chap. 4, 5, 6 the basic ASMs are extended to structured ASMs, synchronous multi-agent ASMs, and asynchronous multi-agent ASMs, and these are illustrated by more involved case studies.

The notion of ASMs captures some fundamental operational concepts of computing in a notation which is familiar from programming practice and mathematical standards. In fact it is correct to view basic ASMs as "pseudocode over abstract data", since their (simple) semantics supports this intuitive understanding by a precise notion of a tunable abstract state and state transition, as expressed by the working definition in Sect. 2.2. This definition lays a rigorous foundation for using ASMs as ground models and for stepwise refinement (Sect. 2.1). We use the popular LIFT example to illustrate the particularly important subclass of *control state ASMs*, which add to the mode-control mechanism of finite state machines (FSMs), synchronous parallelism and the manipulation of data structures (Sect. 2.3). In the last two sections we provide a more detailed formal definition of basic ASMs and survey our notation.

2.1 Principles of Hierarchical System Design

The ASM method which is explained in this book is a systems engineering technique which supports the integration of problem-domain-oriented modeling and analysis into the development cycle. Its goal is to improve industrial system development by accurate high-level modeling which is linked

[1] For lecture slides see `AsmMethod` (⇝ CD), `RefinemtMeth` (⇝ CD), `Falko` (⇝ CD), `AsmDefinition` (⇝ CD).

seamlessly, in a way the *practitioner* can verify and validate, to executable code. Two conceptually and technically different tasks of system development, known as requirements capture (or elicitation) and system design proper, have to be brought together in a coherent way by such a hierarchical approach to system design, as we are going to shortly describe here and to explain in more detail in the two subsections below. When we speak about systems, we mean both hardware and software systems, given that by its machine and programming-language-independent nature the ASM method works for descriptions of both hardware and software systems and in fact supports hardware/software co-design techniques.

The ASM method offers a uniform conceptual framework to fulfill these two tasks: the modeling of their algorithmic content as distinct from (though relatable to) the description of technological, managerial, financial and similar so-called non-functional system features. In fact the requirements can be captured by constructing *ground model* ASMs, in which non-functional features can be formulated as constraints or assumptions. Ground model ASMs are system "blueprints" whose role is to "ground designs in the reality". They represent succinct process-oriented models of the to-be-implemented piece of "real world", transparent for both the customer and the software designer so that they can serve as the basis for the software *contract*, a document which binds the two parties involved. Ground models come with a sufficiently precise yet abstract, unambiguous meaning to carry out an implementation-independent, application-oriented requirements analysis (i.e. both verification and validation) prior to coding. In particular, the requirements validation one can perform on ground model ASMs allows one to explore the problem space and the viability of different solutions before embarking on any particular one; it also enhances the traditional test activities by starting test reasoning and test executions right at the beginning of the project. In fact the operational character of ground model ASMs allows one to define the system test plan, and to already perform tests for (prototypical executable versions of) the ground model, using it as a playground for simulation experiments with and debugging of the design long before its expensive coding begins. Starting from such ground model ASMs, a hierarchy of intermediate models can be constructed by *stepwise refining ASMs*, leading to efficiently executable code, where each step can be justified (i.e. verified and validated) as the correct implementation of some explicitly stated design decision. In this way not only can a ground model be linked to its implementation via traceable requirements, but also a documentation of the entire design is provided which supports design reuse and code maintenance, namely through reflecting orthogonal design decisions in intermediate models, a fundamental feature for a method which supports *design-for-change*.

The key strategy for developing such a hierarchy of models is an example of the so-called *divide-and-conquer* technique, consisting of a systematic *separation of different concerns* with the ultimate goal of bringing the different threads together in the appropriate place. Major concerns to be separated include orthogonal design decisions, as well as design and analysis, as follows.

- *Separating orthogonal design decisions.* From the system engineering point of view, this principle supports the wisdom of separating system design from its implementation and is motivated mainly by two reasons. One is to *keep the design space open* as much as possible, to explore different possible software structures, avoiding premature design decisions, including whether a component should be realized in software or in hardware. The other reason is to *structure the design space*, defining precise interfaces for a system decomposition (called *system architecture*) which supports "specifying-for-change" and explicit documentation of design decisions, thus enhancing practical software management and maintenance procedures. From the analysis point of view, adherence to this principle provides the means to split the overall validation and verification tasks into manageable subtasks for orthogonal system components. In particular it opens the way to unit tests and to the use of mathematical verification techniques which are based upon reasoning from assumptions.
- *Separating design from analysis.* This principle is twofold, namely first to separate experimental *validation*, which is based upon simulations and provides the possibility to reason from the results of laboratory test executions and to detect incomplete and erroneous specifications at the stage of requirements capture, from mathematical *verification* of the blueprint (ground model), and second to separate the characteristic concerns for distinct levels of verification. The *verification layers* to be distinguished come with established degrees of to-be-provided detail,[2] whether by reasoning for human inspection (mathematical design justification) or by using rule-based reasoning systems (mechanical design justification). Such systems can come as inference calculi operated by humans or as computerized systems, either interactive systems or automatic tools, where within the latter one has to distinguish model checkers and theorem provers. Each verification or validation technique comes with its characteristic implications for the degree of detail needed for the underlying specification and for the cost of the verification effort.

This systematic separation of design and analysis concerns distinguishes the ASM method from most other approaches in the literature, which instead establish a priori determined intimate links between, on the one hand, the language structures offered for modeling and, on the other hand, corresponding validation and verification techniques, with the well-known resulting advan-

[2] An inherent difficulty of system design is to decide upon how much detail and consequently degree of formality is appropriate for the intended system level. As more details are given, it becomes more difficult to understand and formulate the checkable correctness conditions. On the other hand, omitting details often hides a misunderstanding of some relevant system feature.

tages and disadvantages. Arguably, the most general[3] abstraction mechanism associated with the concept of ASMs allows one to work with a general design language and to commit to a connection to specific platforms or language features or to particular analysis tools (simulators, provers, etc.) only where this provides a benefit.[4] The technical support for the needed crossing of abstraction levels is the ASM refinement method, whose flexibility and applicability to complex systems meets that of the ASM abstraction method. It enables the designer to adopt for the validation of his ASM models any appropriate simulation technique, whether mental simulation or testing of scenarios or computer execution, etc. Similarly, for verifying model properties any appropriate method can be chosen, whether mathematical proof or model checking or mechanically supported automatic or interactive theorem proving. Because this is a book, for the analysis of the ASMs in this text we will use mental model simulation and mathematical verification. In Chap. 8.1, 8.3 we present the various execution and validation mechanisms which have been built for ASMs and the links of ASMs to model checkers and to theorem provers.[5]

Before defining basic ASMs, in the next two sections we characterize in more detail how the ASM method supports hierarchical system design by ground model construction and stepwise refinement of models.

2.1.1 Ground Model Construction (Requirements Capture)

In this section we characterize in general terms how building ground model ASMs helps to solve three major problems of requirements capture. For a concrete illustration of the ASM ground model method see the Daemon Game example in Fig. 2.13, the LIFT example in Sect. 2.3, further introductory examples in Sect. 3.1 and more advanced examples in Chap. 3–6.

Elicitation of requirements is a notoriously difficult and most error prone part of the system development activities. In [287] it is reported that software developers in the information technology, production and service sectors consistently ranked requirements specification and managing customer requirements as the most important problem they faced, and that more than half of the respondents rated it as a major problem. Requirements capture is largely a *formalization task*, namely to realize the transition from natural-language problem descriptions – which are often incomplete or interspersed with misleading details, partly ambiguous or even inconsistent – to a sufficiently precise, unambiguous, consistent, complete and minimal description,

[3] See the discussion in Chap. 7.

[4] In fact the ASM method offers a specification language which fits the needs of the so-called “model driven architecture” approach to platform-independent software development (`http://www.omg.org/mda/`). The goal there is to separate abstract and model-driven descriptions of the functionality of components or software systems from their later mapping to multiple specific target platforms, operating systems, programming languages or middleware techniques.

[5] Their applications are also surveyed in Chap. 9, which can be read independently of the other chapters.

which can serve as a basis for the contract between the customer or domain expert and the software designer. We use the term *ground model* for such an accurate description resulting from the requirements elicitation (and possible extensions which may be recognized as necessary during later design phases). The formalization task requires the solution of three problems which relate to the support that ground models provide for software quality assurance through model inspection, verification and testing.

The first one is a *language and communication problem*, implied by the needed mediation between the application domain, where the task originates which is to be accomplished by the system to be built, and the world of mathematical (often inappropriately called formal[6]) models, where the relevant piece of reality has to be represented. The language in which the ground model is formulated must be appropriate in order to naturally yet accurately express the relevant features of the given application domain and to be easily understandable by the two parties involved in establishing the software contract,[7] i.e. the application domain expert (the contractor) and the system designer. This means in particular that it must be possible to calibrate the degree of formality of the language to the given problem domain, so as to support the concentration on domain issues instead of issues of formal notation. For example, the language should be tunable to naturally express data-oriented applications (as does for example the entity relationship model), but also to naturally express function-oriented applications (as do flow diagrams) and control-oriented applications (as do automata, whether sequential or distributed). Therefore, the modeling language has to provide a general (conceptual and application-oriented) data model together with a function model (for defining the system dynamics by rule-executing agents) and an appropriate interface to the environment (the users or neighboring systems or applications).

The second formalization problem is a *verification-method problem* which stems from the fact that there are no mathematical means to prove the correctness of the passage from an informal to a precise description. Nevertheless, means must be provided to establish that the precise requirements

[6] The use of the term *formal* in this context is misleading. Standard *mathematical* rigor which supports content-oriented precise intuitive reasoning has to be distinguished from the degree of precision of methods which are based upon formalizations in the syntax of some fixed logic and characterized by rule-based *mechanical* or mechanizable reasoning schemes. The ASM method is not a "formal method" in this restricted understanding of the term, although it supports mathematical verification.

[7] This notion of ground model as *software contract* is more general than the one known from Eiffel [333], which is formulated in terms of pre/postconditions for executable (in fact Eiffel) code. Ground models are specifications. Their *raison d'être* precedes that of the final code, to which they may be linked not directly, but through a hierarchy of stepwise-refined models which bridges the gap between the abstraction levels of the ground models – the software contract – and the code.

model is complete and consistent, that it reflects the original intentions and that these are correctly conveyed – together with the necessary underlying application-domain knowledge – to the designer. Therefore, an inspection of ground models by the application-domain expert must be possible,[8] but also forms of reasoning must be available to support the designer in formally checking the completeness and internal consistency of the model, as well as the consistency of different system views. These two complementary forms of ground model verification are crucial for a realistic requirements-capture method, since changes to be made in the final code, either to repair erroneous design decisions, or to add missing ones, due to a wrong or incomplete specification which is discovered only during the coding phase, are known to be difficult to handle and to result in prohibitive costs. A way to solve this problem is to use a language like the broad-spectrum algorithmic language of ASMs, which allows one to tailor the ground model to resemble the structure of the real-world problem, to make its correctness checkable by inspection and its completeness analyzable with respect to the problem to be solved.

The third problem is a *validation problem*. It must be possible to simulate the ground model for running relevant scenarios (use cases; see Sect. 3.1.2), which often are part of the requirements, and to define – prior to coding – a precise system-acceptance test plan. The operational character of ground model ASMs supports defining in abstract run-time terms the expected system effect on samples – the so-called *oracle definition* which can be used for static testing, where the code is inspected and compared to the specification, but also for dynamic testing where the execution results are compared. Furthermore, ASM ground models can be used to guide the user in the *application-domain-driven selection of test cases*, exhibiting in the specification the relevant environment parts and the properties to be checked, showing how to derive test cases from use cases. Last but not least, by appropriately refining the oracle, one can also *specify and implement a comparator* by determining for runs of the ground model and the code what are the states of interest to be related (spied), the locations of interest to be watched, and when their comparison is considered successful (the test equivalence relation). These features for specifying a comparator using the knowledge about how the oracle is refined reflect the ingredients of the general notion of ASM refinements described in the next section.[9] Concerning simulations of ground models, they are possible due to the executability of ASMs (by mental simulation or using the tools discussed in Section 8.3). This allows one to use a

[8] Providing a precise ground against which questions can be formulated, ground models support the rather Socratic method of asking "ignorant questions" [48] to check whether the semantic interpretation of the informal problem description is correctly captured by the mapping to the terms in the formal model.

[9] The idea expressed in [87] to relate, for testing purposes, runs of ASMs to those of the implementing code has been successfully exploited in [31] for dynamic testing, by monitoring simultaneously the execution of components and of their specifications.

ground model ASM seamlessly in two roles: (1) as an accurate requirements specification (to be matched by the application-domain expert against the given requirements) and (2) as a test model (to be matched by the tester against executions of the final code), thus realizing one of the suggestions of Extreme Programming.[10]

Although by the pragmatic nature of the concept there is no general mathematical definition of the notion of ground models, they appear to be characterized by the following intrinsic properties. They have to be

– *precise* at the appropriate level of detailing yet *flexible*, to satisfy the required accuracy and to be easily modifiable or extendable for reuse and to be adaptable to different application domains,
– *simple and concise* to be understandable by both domain experts and system designers and to be manageable for an analysis of model consistency, completeness and minimality. To permit a reliable analysis of such non-formalizable properties, given that they relate real-world features with model elements, the simplicity of ground models requires that they avoid as much as possible any extraneous encoding and through their abstractions "directly" reflect the structure of the real-world problem. It is for a good reason that object-oriented design approaches share this fundamental pragmatic concern,[11]
– *abstract (minimal) yet complete*. This notion of completeness cannot have a mathematical definition, but it has a meaning, namely that every semantically relevant feature is present, that all contract benefits and obligations are mentioned and that there are no hidden clauses (including those related to general laws, standards, regulations and current practice). In particular, a ground model must contain as interface all semantically relevant parameters concerning the interaction with the environment, and where appropriate also the basic architectural system structure. The completeness property of ground model ASMs forces the requirements engineer to produce a model which is "closed" modulo some "holes", which are however explicitly delineated, including a statement of the assumptions made for them at the abstract level and to be realized through the detailed specifica-

[10] The idea of executable specifications is not new. It has been heatedly discussed in the literature; a good source is [204]. So-called "controlled" subsets of natural languages proposed for software specifications tend to map into logic languages and thus inherit their limitations; see, for example, the subset of English introduced in [205], which is mapped to Prolog and so not surprisingly essentially restricted to writing functional requirements specifications. In contrast, the language of ASMs is as general as a scientific discourse of an algorithmic nature does allow.

[11] For example, S. McConnell writes in *Code Complete* (Microsoft Press, Redmond 1993): "The Object-Oriented paradigm is based upon the proposition that the more closely a program's structure resembles the real-world problem it is to solve, the better the program will be." ASMs allow one to write such programs in an application-oriented *abstract* form which is mappable to *any* platform or programming language.

tion left for later refinements.[12] Minimality means that the model abstracts from details that are relevant either only for the further design or only for a portion of the application domain which does not influence the system to be built,

- *validatable* and thus possibly falsifiable by experiment. A useful prerequisite is the largely *operational* character of ground models, which supports the process-oriented understanding and the mental or machine simulation. As a consequence, executable versions of ground models can also serve as prototypes,
- equipped with a *precise semantical foundation* as a prerequisite for analysis and as a basis for reliable tool development and prototyping.

One of the goals of this book is to explain how one can exploit ASMs as a class of models serving as satisfactory ground models for requirements capture. We want the reader to learn how to use the abstraction mechanism inherent in ASMs for tailoring system specifications to the characteristic conceptual frame of the underlying application-domain problem and to the desired level of detail, or, stating it the other way round, to the intended level of looseness, which represents the freedom the implementor is offered to exploit the design space. One can learn the skill to overcome by satisfactory ground models the tension between the simultaneous request for abstraction and for accuracy. The needed validation of ground model ASMs is supported by the operational character of abstract machines and by their precise semantical foundation, which provide a natural basis for making ASMs executable in various ways (see Section 8.3). Not only is there no inherent contradiction in being both operational and abstract, contrary to a widely held belief,[13] but the possibilities that the concept of ASMs offers to combine these two properties in a single model is one of the reasons for the success of numerous ground model ASMs. Important examples are ground model ASMs developed to define standards, e.g. the ASMs defining the ISO standard for PROLOG [131], the IEEE standard for VHDL'93 [111], and the ITU standard for SDL'2000 [292]. Consult Chap. 9 for many other real-life ground model ASMs.

2.1.2 Stepwise Refinement (Incremental Design)

In this section we characterize in general terms the second building block of the ASM system design and analysis method, namely stepwise refinement. For a concrete illustration of the ASM refinement method see the introductory examples in Sect. 3.2 and more advanced examples in Chap. 3–6.

[12] See [320] for methods to check a set of criteria which identify missing (as well as incorrect or ambiguous) requirements and have been related to process-control systems by the Requirements State Machine model, an instance of MealyAsms defined on p. 287.

[13] See the discussion of logico-algebraic design approaches at the end of Sect. 7.1.2.

The idea of using various forms of refinement for incremental system design is long established and in fact characterizes the structured programming approach [429, 184]. One of the established guiding principles of refinement notions in the literature is expressed as follows:

> *Principle of substitutivity*: it is acceptable to replace one program by another, *provided* it is impossible for a user of the programs to observe that the substitution has taken place. [176, p. 47]

Many refinement concepts in the literature are tailored to match this a priori epistemological principle, and as a result are restricted in various ways which limit their range of applicability. Among the restrictions of this sort are the following ones:

- Restriction to *certain forms of programs*, e.g. viewed as sequences of operations (straight-line programs). As a consequence, the refined programs are even structurally equivalent to their abstract counterpart, i.e. with corresponding operations occurring in the same places, thus precluding the analysis of the role of other forms of control for refinement, e.g. parallelism or iteration.
- Restriction to programs with *only monolithic state operations*, expressed by global functions of the state without the possibility of modifying elements of the state. This makes it difficult to exploit combinations of local effects for overall refinements and leads to the well-known frame problem, which typically makes formal specifications of programs more difficult to write and to read than the programs that they describe.
- Restriction to *observations interpreted as pairs of input/output sequences or of pre-post-states*, typically with the same input/output representation at the abstract and the refined level. This focus on the functional input/output behavior of terminating runs or on the pre-post-states of data-refined operations is implied by the declarative dictate to “forget the state” with its “internal” actions, so that what remains from computations are “traces” (sequences of labels for external actions) and refinement becomes a subset relation of traces. This precludes us from relating arbitrary segments of abstract and refined computations, using an equivalence notion for state elements (locations) which is fine-tuned to the problem under investigation (think about an interface whose natural description involves some state-related items). As a consequence, the invariants to be compared of the abstract and refined programs are viewed in terms of pre- or post-condition strengthenings or weakenings, which restricts a more general analysis of the effect of invariants as a retrenchment of the class of possible models. The fact that often no change of input/output representation is permitted also precludes the possibility of refining “abstract input”, e.g. in the form of monitored data, by “controlled data” which are computed through concrete computation steps (see Sect. 2.2.3).

- Restriction to *logic or proof-rule-oriented* refinement schemes [5, 167]. Tailoring refinement schemes to fit a priori fixed proof principles quickly leads to severe restrictions of the design space.[14]

Another restriction that many (but not all) approaches to refinement come with and which makes applications difficult is to allow only abstraction *functions* instead of relations.

In contrast, the ASM refinement method offers an open framework which integrates numerous well-known more specific notions of refinement (see [24, 334, 337, 25, 167, 176]), similarly to the way the notion of ASMs covers other models of computation and approaches to system design (see Sect. 7.1). The guiding principle of the refinement method is its problem-orientation. In fact its development during the last decade[15] was driven by practical refinement tasks and geared to support divide-and-conquer techniques for both design and verification, without privileging one to the detriment of the other. The "freedom of abstraction" offered by ASMs, i.e. the availability in ASMs of arbitrary structures to reflect the underlying notion of state, provides the necessary instrument to fine tune the mapping of a given (the "abstract") machine to a more concrete (the "refined") one, with its observable (typically more detailed) state and its observable (typically more involved) computation, in such a way that the intended "equivalence" between corresponding run segments of the two ASMs becomes observable (can be explicitly defined and proved to hold under precisely stated boundary conditions). The focus is not on generic notions of refinements which can be proved to work in every context and to provide only effects which can never be detected by any user of the new program. Instead the concern is to support a disciplined use of refinements which correctly reflect and explicitly document an intended design decision, adding more details to a more abstract design description, e.g. for making an abstract program executable, for improving a program by additional features or by restricting it through precise boundary conditions which exclude certain undesired behaviors. There is no a priori commitment to particular notions of state, program, run, equivalence, or to any particular method to establish the correctness of the refinement step. The major and usually difficult task is to first listen to the subject, to find the right granularity and to formulate an appropriate refinement – or abstraction in the case of a re-engineering project – that faithfully reflects the

[14] An illustrative example is the restrictive use of assignments in CSP where two processes P, Q can be put in parallel $P \mid\mid Q$ only if the write variables of P are disjoint from the variables of Q, see [281, p. 188]. A related restriction applies to the multiple substitution operator in the B method, e.g. parallel substitutions in a B machine need to modify disjoint variables (but may share variables for reading). For another example from the B method see Exercise 7.1.6. Similarly, when the temporal-logic-based *live sequence charts* of [165] are extended by assignments [270] to make them more appropriate to real-life scenario descriptions, a restriction to local variables is imposed.

[15] See Chap. 9 for a detailed survey.

underlying design decision or re-engineering idea, and only then to look for appropriate means to justify that under the precisely stated conditions the refinement correctly implements the given model, or that the re-engineered abstract model correctly abstracts from the given code. Whatever feasible method is available can – indeed should – be adopted, whether for verification (by reasoning) or for validation (e.g. testing model-based run-time assertions through a simulation), to establish that the intended design assumptions hold in the implementation and that refined runs correctly translate the effect of abstract ones.

As result of this openness of the ASM refinement method to any concrete refinement scheme and to a variety of analysis tools, ASM refinements on the one side can capture the various more restricted refinement notions studied in the literature and on the other side scale to industrial-size systems. In fact they support the splitting of a complex design task into simpler, piecemeal-verifiable and validatable steps, which are linked together into a hierarchy and as a result contribute to effective procedures for maintenance and reuse of system developments. In particular, for system design, ASM refinements permit us to explicitly capture orthogonalities by modular machines (components),[16] supporting the well-known software engineering principles of "design-for-change" and "design-for-reuse",[17] also by enhancing the very communication and exchange of designs. For system verification, ASM refinements support divide-and-conquer and proof reuse[18] techniques which are more widely applicable than so-called compositional, mostly syntax-oriented, proof methods in the literature. Above all they can be used by practitioners without the need of extensive special training. In fact to

(0) show that an implementation S^* satisfies a desired property P^*

the ASM method allows the designer to

(1) build an abstract model S,
(2) prove a possibly abstract form P of the property in question to hold under appropriate assumptions for S,

[16] For a real-life example, which we cannot show in this book, see [406], where ASM components have been used to construct a hierarchically decomposed model for Java and its implementation on the Java Virtual Machine. Horizontal refinements define piecemeal natural extensions of the language, from imperative to object-oriented, exception handling and concurrency features; vertical stepwise detailing of models decomposes the JVM into its loader, verifier, preparator, interpreter components, and transforms in a proven-to-be-correct way the Java model to its JVM implementation.

[17] Complex examples are to be found in the numerous extensions and adaptations of ASM models discussed in Sect. 9.2.

[18] An industrial-strength example is the reuse of the Prolog-to-WAM (Warren Abstract Machine) compilation correctness proof [132] for the compilation of CLP(R)-programs to CLAM (Constraint Logic Arithmetical Machine) code [133] and of Protos-L programs to PAM (Protos Abstract Machine) code [42, 41], two machines developed at IBM. See details in Sect. 9.2.

Fig. 2.1 The ASM refinement scheme

$$\begin{tikzcd}
\text{State } S \arrow[r, "\overbrace{\tau_1 \cdots \tau_m}^{m \text{ steps of } M}"] \arrow[d, leftrightarrow, "\equiv"'] & S' \arrow[d, leftrightarrow, "\equiv"] \\
\text{State } S^* \arrow[r, "\underbrace{\sigma_1 \cdots \sigma_n}_{n \text{ steps of } M^*}"'] & S^{*\prime}
\end{tikzcd}$$

With an equivalence notion ≡ between data in locations of interest in corresponding states.

(3) show S to be correctly refined by S^* and the assumptions to hold in S^*.

The practice of system design shows that the overall task (0), which for real-life systems is usually too complex to be tackled at a single blow, can be accomplished by splitting it into a series of manageable subtasks (1)–(3), each step reflecting a part of the design. Last but not least, through the analysis reports the ASM refinements provide for system users a reliable system documentation, which can be put to further use for system maintenance (exploiting the accurate and precise detailed information in the refinement documentation), e.g. to examine the model for fault analysis or to recognize which parts of the code are affected to correct bugs which have been reported.[19] One can exploit such an improved system documentation also to support practical reuse techniques (exploiting orthogonalities and hierarchical layers, including the reuse of proof techniques, e.g. for versioning). In this sense the ASM refinement method pushes Wirth's [429] and Dijkstra's [184] refinement program to its most general consequences.

Without entering into technical details, which are explained in Sect. 3.2, we illustrate here the scheme for an ASM refinement step, which generalizes the more restricted refinement notions in the literature. The scheme can also

[19] For an illustration of this maintenance feature see the use of ASMs in the industrial re-engineering project reported in [121]. The report is available also as a powerpoint slide show in `Falko` (⇝ CD).

be viewed as describing an abstraction step if it is used for a high-level model of an implementation, as happens in re-engineering projects.[20]

Figure 2.1, which enriches the traditional commutative refinement diagrams, shows that to refine an ASM M to an ASM M^*, one has the freedom (and the task) to define the following items:

- the notion of a *refined state*,
- the notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest, i.e. the pairs of states in the runs one wants to relate through the refinement, including usually the correspondence of initial and (if there are any) of final states,
- the notion of abstract *computation segments* $\tau_1, \ldots, \tau_m$, where each τ_i represents a single M-step, and of corresponding refined computation segments $\sigma_1, \ldots, \sigma_n$, of single M^*-steps σ_j, which in given runs lead from corresponding states of interest to (usually the next) corresponding states of interest (the resulting diagrams will be called (m, n)-diagrams and the refinements (m, n)-refinements),
- the notion of *locations of interest* and of *corresponding locations*, i.e. pairs of (possibly sets of) locations one wants to relate in corresponding states, where locations represent abstract containers for data (see the definition of ASM locations below),
- the notion the of *equivalence* $\equiv$ of the data in the locations of interest; these local data equivalences usually accumulate to the notion of the equivalence of corresponding states of interest.

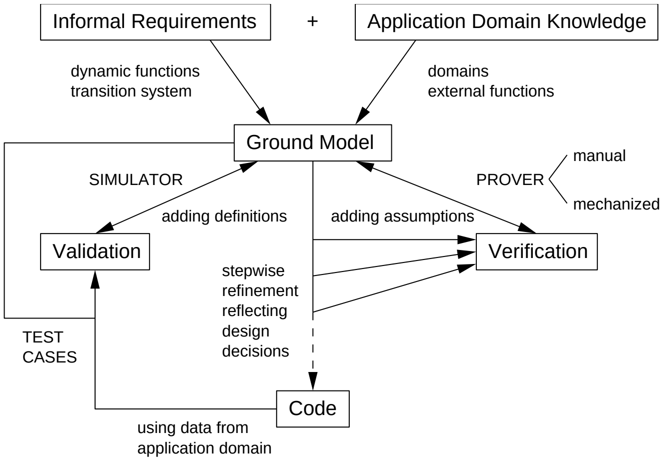
The scheme shows that an ASM refinement allows one to combine in a natural way a change of the signature (through the definition of the correspondence of states, of corresponding locations and of the equivalence of data) with a change of the control (defining the “flow of operations” appearing in the corresponding computation segments). These are two features that many notions of refinement in the literature can deal with at most separately. Notably the scheme includes the case of optimizations where in the optimized model the computation segments may be shorter than their corresponding abstract counterpart, due to an $(m, 0)$-refinement with $m > 0$, or where some abstract state locations may have been eliminated.[21] It also includes the symmetric case where the implementation may have longer run segments than the specification, due to $(0, n)$-refinements with $n > 0$ discussed in Sect. 3.2.[22]

[20] See [26] for an illustration by an industrial case study, which is available also as a powerpoint slide show in `Debugger` (⇝ CD).

[21] This covers the extension of the traditional trace-based notion of abstraction function to abstraction relations, or equivalently adding to the refined model so-called *history variables* to keep track of the abstract state locations which have been optimized away.

[22] This covers the extension of the traditional trace-based notion of abstractions of type $(1, 1)$ by so-called *prophecy variables*. For an illustrative example see the machine EarlyChoice and its implementation LateChoice on p. 116.

Fig. 2.2 Models and methods in the development process



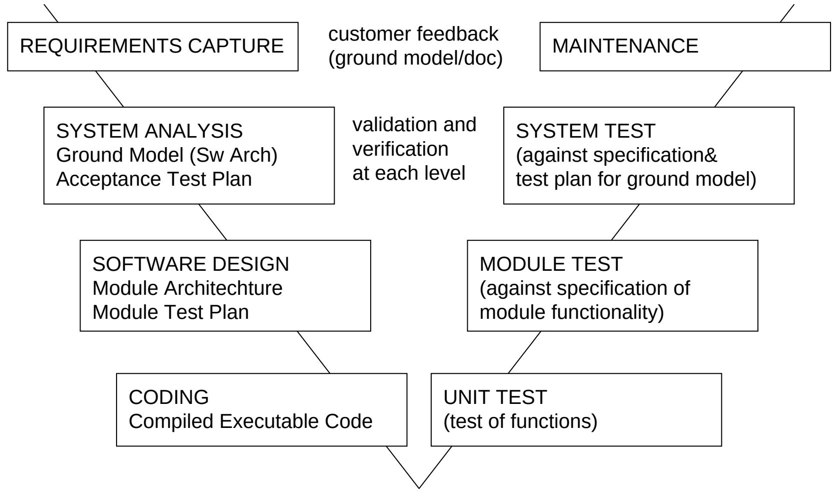
Furthermore, the scheme covers the refinement involved in the specification of comparators used in testing code against abstract models, as explained on p. 18. Once the notions of corresponding states and of their equivalence have been determined, one can define that M^* is a correct refinement of M if and only if every (infinite) refined run simulates an (infinite) abstract run with equivalent corresponding states (see Sect. 3.2 for a precise definition). By this definition, refinement correctness implies for the special case of terminating deterministic runs the equivalence of the input/output behavior of the abstract and the refined machine, a special feature on which numerous refinement notions in the literature are focussed.

To make the relation between functions in corresponding states easy to see, we often use in the two ASMs the same names for the corresponding functions. When there seems to be a danger of misunderstanding, we index them by the name of the ASM they belong to.

2.1.3 Integration into Software Practice

We summarize the discussion in this section by two diagrams which illustrate the role a smooth combination of stepwise refinements of ground model ASMs with established design and analysis techniques can play for considerable improvements in the various phases of the software development cycle. Figure 2.2 pictorially summarizes the different verification and validation techniques which can be applied to ASM models in the design hierarchy leading from a ground model to executable code. A practical combination of these different techniques requires a transparent conceptual continuity of the

Fig. 2.3 Integrating ASMs into the V-scheme



kind ASMs offer from the ground model through the intermediate levels to the implementation.

Figure 2.3 presents the levels in the well-known V-scheme, where ASMs can be integrated in a uniform way, so that the ground model can be kept in agreement with the ASMs describing the further refinements of the system by modules and units. Established testing procedures can be enhanced in this way by performing them for (executable versions of) the refined ASMs at each level, side by side with the other development activities mentioned in the V-model, namely requirements capture, analysis, design, and coding. Formally verified development of components may make unit testing spurious, as has been reported for the Meteor project [37] based upon the use of the B method.

2.2 Working Definition

In this section we define *basic ASMs* in a form which justifies their intuitive understanding as pseudo-code over abstract data. We define the ASM *function classification*, which incorporates a powerful semantical abstraction, modularization and information-hiding mechanism, and can be used besides the usual purely syntactical module notation. We illustrate the two definitions by simple examples in Sect. 2.2.5. We then define the particularly important subclass of *control state ASMs* which represent a normal form of UML activity diagrams and naturally extend finite state machines by synchronous parallelism and by the possibility to also manipulate data. In Sect. 2.3 we illustrate the definitions, as well as the ASM ground model and analysis method, by the celebrated LIFT example. Section 2.4 provides a more detailed recursive definition of the syntax and the semantics of basic ASMs.

In this section we use two fundamental concepts of computation theory and logic, namely of the transition system and of the interpretation of predicate logic terms and formulae. They are explained in standard textbooks (e.g. see [70]) and are reviewed in Sect. 2.4.

2.2.1 Basic ASMs

Historically the notion of ASMs moved from a definition which formalizes simultaneous parallel actions of a single agent to a generalization where multiple agents act and interact in an asynchronous manner.[23] Also, some extensions by particular though for applications rather useful features were introduced, dealing with forms of non-determinism ("choice" or existential quantification) and of unrestricted synchronous parallelism (universal quantification "forall"). The search for postulates from which to prove the ASM thesis (see Sect. 7.2) led to a distinction between so-called sequential ASMs (with only fixed amount of computation power per step and only bounded synchronous parallelism),[24] and synchronous parallel and distributed ASMs. Instead of such a classification in terms of the underlying logic, we follow practical system design criteria, where for a specification the distinctive features are whether the system to be described has one or more agents. In the former case the question is how the agent interacts with its environment, in the latter case whether the multiple agents act in a synchronous or in an asynchronous manner, and furthermore whether they are distributed or not. We also consider the orthogonal classification whether (and how) the programs executed by the agents are structured, giving rise to submachine concepts.

This leads us to define in this chapter *basic ASMs* as single-agent machines which may dispose of potentially unrestricted non-determinism and parallelism (appearing in the form of the "choose" and "forall" rules defined below) and to distinguish a version with flat programs from structured versions (Chap. 4). This class of single-agent ASMs is then extended to synchronous (Chap. 5) and to asynchronous (Chap. 6) multi-agent ASMs.

2.2.2 Definition

Basic ASMs are finite sets of so-called *transition rules* of the form

if *Condition* **then** *Updates*

[23] See Chap. 9 for the historical details.

[24] This class of machines is intimately related to the class of quantifier-free interpretations in logic, as observed in [54, Sect. 2].

which transform abstract states. (Two more forms are introduced below.) The *Condition* (also called *guard*) under which a rule is applied is an arbitrary predicate logic formula without free variables, whose interpretation evaluates to *true* or *false*. *Updates* is a finite set of assignments of the form

$$f(t_1, \ldots, t_n) := t$$

whose execution is to be understood as *changing* (or defining, if there was none) in parallel the value of the occurring functions f at the indicated arguments to the indicated value. More precisely, in the given state (see below) first all parameters t_i, t are evaluated to their values, say v_i, v, then the value of $f(v_1, \ldots, v_n)$ is updated to v, which represents the value of $f(v_1, \ldots, v_n)$ in the next state. Such pairs of a function name f, which is fixed by the signature, and an optional argument $(v_1, \ldots, v_n)$, which is formed by a list of dynamic parameter values v_i of whatever type, are called *locations*. They represent the abstract ASM concept of basic object containers (memory units), which abstracts from particular memory addressing and object referencing mechanisms.[25] Location-value pairs (loc, v) are called *updates* and represent the basic units of state change.

This abstract understanding of memory and memory update allows the designer to combine the operational nature of the concepts of location and update with the freedom of tailoring them to the level of abstraction which is appropriate for the given design or analysis task, namely when defining the machine state. The notion of ASM *states* is the classical notion of mathematical *structures* where data come as abstract objects, i.e. as elements of sets (also called *domains* or *universes*, one for each category of data) which are equipped with basic operations (partial *functions* in the mathematical sense) and predicates (attributes or relations). The instantiation of a relation or function to an object o can be described by the process of parameterization of, say, f to the function $o.f$, which to each x assigns the value $f(o, x)$.[26] For the evaluation of terms and formulae in an ASM state, the standard interpretation of function symbols by the corresponding functions in that state is used. Without loss of generality we usually treat predicates as characteristic functions and constants as 0-ary functions. Partial functions are turned into total functions by interpreting $f(x) = undef$ with a fixed special value $undef$ as $f(x)$ being undefined. The reader who is not familiar with this notion of structure may view a state as a "database of functions" (read: a set of function tables) instead of predicates.

[25] One may imagine functions as represented by tables. Then a location is a table entry and an update describes an update of the value residing in the table entry. In fact ASMs provide a precise and simple foundation for the different forms of Parnas tables; see Sect. 7.1. A particular form of such a table notation for a class of basic ASMs is reported in [149] to have been introduced successfully into an industrial software process. See also the graphical notation on p. 31.

[26] This simple logical framework covers the object-oriented understanding of the states of an object as (paraphrasing G. Booch) "encompassing all of the prop-

The notion of the ASM *run* is an instance of the classical notion of the computation of transition systems. An ASM computation step in a given state consists in executing *simultaneously* all updates of all transition rules whose guard is true in the state, if these updates are consistent, in which case the result of their execution yields the *next* state.[27] In the case of inconsistency the computation does not yield a next state, a situation which typically is reported by executing engines with an error message. A set of updates is called *consistent* if it contains no pair of updates with the same location, i.e. no two elements $(loc, v), (loc, v')$ with $v \neq v'$. An ASM step resembles a database transaction; it is performed as an atomic action with no side effects.[28]

In general, ASMs are reactive systems which iterate their computation step, but for the special case of terminating runs one can choose among various natural termination criteria, namely that no rule is applicable any more (see Definition 2.4.22) or that the machine yields an empty update set (see Definition 4.1.2), or that the state does not change any more (the criterion apparently adopted by AsmL [201]).

When analyzing runs $S_0, S_1, \ldots$ of an ASM, we call S_n the n-th state or state n and denote the value of a term t in $S = S_n$ by t_S or t_n. By the interval (S_n, S_m) we denote the states between S_n and S_m. We say that S_n is before S_m (and write $S_n < S_m$) if $n < m$.

Simultaneous execution provides a rather useful instrument for high-level design to *locally describe a global state change*, namely as obtained in one step through executing a set of updates. The only limitation – imposed by the need of uniquely identifying objects residing in locations – is the consistency of the set of updates to be executed. The local description of global state change also implies that by definition the next state differs from the previous state only at locations which appear in the update set. This avoids the rightly criticized length increase of numerous forms of specifications which have to express as the effect of an operation not only what is changed but also what remains unchanged (the so-called *frame problem* of specification approaches, which work with a global notion of state or with purely axiomatic descriptions).[29]

Simultaneous execution also provides a convenient way to *abstract from sequentiality* where it is irrelevant for an intended design. This synchronous parallelism in the ASM execution model directly supports refinements to parallel or distributed implementations. It is enhanced by the following notation to express the simultaneous execution of a rule R for each x satisfying a given condition φ (where typically x will have some free occurrences in R which are bound by the quantifier):

erties of the object plus the current values of each of these properties", formally represented by a set of locations for each property.

[27] More precisely, it yields the *next internal state*, see below Sect. 2.2.3.

[28] It is characteristic of the ASM method to abstract away every effect one considers as irrelevant, so that all the visible effects are principal ones.

[29] For some examples see Sect. 7.1.

```
forall $x$ with $\varphi$
  $R$
```

Similarly, non-determinism as a convenient way to abstract from details of scheduling of rule executions can be expressed by rules of the form

```
choose $x$ with $\varphi$
  $R$
```

where φ is a Boolean-valued expression and R is a rule. The meaning of such an ASM rule is to execute rule R with an arbitrary x chosen among those satisfying the selection property φ. If there exists no such x, nothing is done.[30] For rules of such forms we sometimes use graphical notations as follows:

$$\begin{array}{|c|}\hline \textbf{forall } x \textbf{ with } \varphi \\ \hline R \\ \hline \end{array} \qquad \begin{array}{|c|}\hline \textbf{choose } x \textbf{ with } \varphi \\ \hline R \\ \hline \end{array}$$

or a linear notation (with an additional keyword **do** to ease the parsing)

```
forall $x$ with $\varphi$ do $R$        choose $x$ with $\varphi$ do $R$
```

or R_1 **par** ... **par** R_n for an ASM consisting of the set $\{R_1, \dots, R_n\}$ of rules R_i. This is why we consider a set of rules and the **par**-composition of all these rules as the same machine. We freely use common abbreviations and standard variations of notations where convenient and without risk of misunderstanding. For example we often express the range of the quantifiers by usual set notation or by a mixture of set and property notation, where X stands for a set:

```
choose $x \in X$ with $\varphi$
  $R$
```

Similarly, we freely use combinations of **where**, **let**, **if-then-else**, etc. which are easily reducible to the above basic definitions. Instead of **let** $x = s$ **in** (**let** $y = t$ **in** R) we also use the shorthand **let** $\{x = s, y = t\}$ **in** R and the same with a successive vertical displacement of $x = s, y = t$. Sometimes we also use the table-like **case** notation with pattern matching, in which case we try out the cases in the order of writing, from top to bottom. We also use rule schemes, namely rules with variables, and named parameterized rules, mainly as an abbreviational device to enhance the readability or as macro allowing us to reuse machines and to display a global machine structure. For example

```
if $\dots a = (x, y) \dots$ then $\dots x \dots y \dots$
```

abbreviates

[30] In [379] the application of **choose** to empty sets is forbidden for the sake of some algebraic properties of the operator.

$\mathbf{if} \ldots \mathit{ispair}(a) \ldots \mathbf{then} \ldots \mathit{fst}(a) \ldots \mathit{snd}(a) \ldots$

sparing us the need to write explicitly the recognizers and the selectors. Similarly, an occurrence of

$r(x_1, \ldots, x_n)$

where a rule is expected stands for the corresponding rule R (which is supposed to be defined somewhere else, with $r(x_1, \ldots, x_n) = R$ appearing in the declaration part (see below) of the ASM where $r(x_1, \ldots, x_n)$ is used). Such a "rule call" $r(x_1, \ldots, x_n)$ is used only when the parameters are instantiated by legal values (objects, functions, rules, whatever) so that the resulting rule has a well-defined semantical meaning on the basis of the explanations given above.[31] The use of submachines and of macros supports the modularization and stepwise refinement of large machines.

For purposes of separation of concerns it is often convenient to impose for a given ASM additional constraints on its runs to circumscribe those one wants to consider as *legal*. Logically speaking this means restricting the class of models satisfying the given specification (read: the design space). Such restrictions are particularly useful if the constraints express reasoning assumptions for a high-level machine which are easily shown to hold in a refined target machine. The constraint mechanism (which frequently is used also to impose desired properties on the functions appearing in the signature of the machine) allows the designer to smoothly combine in the specification so-called declarative and axiomatic features with operational ones without having to pay the price for the above-mentioned frame problem. As part of the run constraints a set of final states may be defined, although usually we rely upon the standard notion of termination that a state is final for M if none of the rules of M can be fired.

In summary, to define an ASM M one has to indicate its *signature*, the set of *declarations* of functions and rules (including the function classification explained in Sect. 2.2.3 and constraints on signature and runs, thus determining the class of possible states of the machine), the set of its *initial states*, and the (unique) *main rule* which is often identified with the machine M. Often we only indicate the rules with the understanding that the signature is defined by what appears in the rules. For the use of a module notation to structure declarations into manageable groups see Sect. 2.2.4.

ASMs as defined here circumscribe non-determinism to appear through **choose**-rules (or external or shared functions; see below). The definitions of non-deterministic machines in the literature, including the early definition of ASMs in [245], allow the "user" of a machine to choose among rules to be applied, hiding an implicit top-level **choose**-construct ranging on a set of rules (see the investigation in Sect. 4.2). An ASM as defined here and in [248] fires in every state all of its rules; every rule of the machine produces a (possibly empty) update set, the union of all of which (if consistent) determines

[31] For a precise semantical definition of such submachine calls see Sect. 4.1.2.

the resulting next state, *uniquely* modulo the non-determinism which is circumscribed by the occurrences of **choose**-rules (and by the values that the monitored and shared functions described in the next section happen to have in the given state).

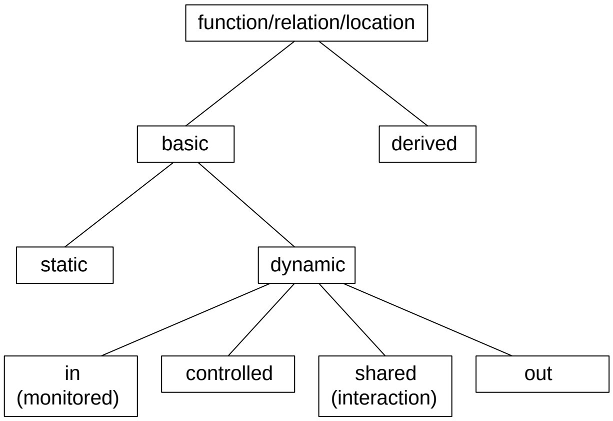
The preceding definitions are completed in the following section by the classification of the functions which may occur in ASM rules and are then illustrated by simple examples in Sect. 2.2.5. The reader interested in a mathematically detailed recursive definition of the syntax and semantics of ASM rules may consult Sect. 2.4.

Problem 1 (Abstract performance evaluation models). Exploit the abstract notion of ASM runs to formulate interesting performance evaluation models at different levels of performance analysis and to relate these levels in a methodologically fruitful way.

2.2.3 Classification of Locations and Updates

A priori no restriction is imposed either on the abstraction level or on the complexity or on the means of definition of the functions used to compute the arguments and the new value denoted by t_i, t in function updates. In support of the principles of separation of concerns, information hiding, data abstraction, modularization and stepwise refinement, the ASM method exploits, however, the following distinctions reflecting the different roles these functions (and more generally locations) can assume in a given machine, as illustrated by Fig. 2.4. For a purely syntactical splitting of large ASMs into manageable modules see Sect. 2.2.4

Fig. 2.4 Classification of ASM functions, relations, locations



The major distinction for a given ASM M is between its *static* functions – which never change during any run of M so that their values for given arguments do not depend on the states of M – and *dynamic* ones, which may change as a consequence of updates by M or by the environment (read: by some other – say an unknown – agent representing the context in which M computes), so that their values for given arguments may depend on the states of M. By definition static functions can be thought of as given by the initial state, so that, where appropriate, handling them can be clearly separated from the description of the system dynamics. Whether the meaning of these functions is determined by a mere signature ("interface") description, or by axiomatic constraints, or by an abstract specification, or by an explicit or recursive definition, or by a program module, depends on the degree of information-hiding the specifier wants to realize. Static 0-ary functions represent constants, whereas with dynamic 0-ary functions one can model variables of programming (not to be confused with logical variables). Dynamic functions can be thought of as a generalization of array variables or hash tables.

The dynamic functions are further divided into four subclasses. *Controlled* functions (for M) are dynamic functions which are directly updatable by and only by the rules of M, i.e. functions f which appear in at least one rule of M as the leftmost function (namely in an update $f(s) := t$ for some s, t) and are not updatable by the environment (or more generally by another agent in the case of a multi-agent machine). These functions are the ones which constitute the internally controlled part of the dynamic state of M.

Monitored functions, also called *in* functions, are dynamic functions which are read but not updated by M and directly updatable only by the environment (or more generally by other agents). They appear in updates of M, but not as the leftmost function of an update. These monitored functions constitute the externally controlled part of the dynamic state of M. To describe combinations of internal and external control of functions, one can use *interaction* functions, also called *shared* functions, defined as dynamic functions which are directly updatable by the rules of M and by the environment and can be read by both (so that typically a protocol is needed to guarantee consistency of updates). The concepts of monitored and shared functions allow one to separate in a specification the computation concerns from the communication concerns. In fact, the definition does not commit to any particular mechanism (e.g. message passing via channels) to describe the exchange of information between an agent and its environment (and similarly between arbitrary agents in the case of a multi-agent machine). As with static functions the specification of monitored functions is open to any appropriate method. This feature helps the system designer to control the amount of information which he wants to give to the programmer. The only (but crucial) assumption made is that in a given state the values of all monitored functions are determined.

Out functions are dynamic functions which are updated but not read by M and are monitored (read but not updated) by the environment or in general by other agents. Formally, such output functions do appear in some rules of M, but only as the leftmost function of an assignment.[32]

We call functions *external* for M if for M they are either static or monitored.

An orthogonal, pragmatically important classification comes through the distinction of *basic* and of *derived* functions. Basic functions are functions which are taken for granted (declared as "given", typically those forming the basic signature); derived functions are functions which even if dynamic are not updatable either by M or by the environment, but may be read by both and yield values which are defined by a fixed scheme in terms of other (static or dynamic) functions (and as a consequence may sometimes not be counted as part of the basic signature). Thus derived functions are a kind of auxiliary function coming with a specification or computation mechanism which is given separately from the main machine; they may be thought of as a global method with read-only variables.

The classification principle explained above for functions is applied in the same way to (sets of) locations or updates.

A frequently encountered kind of function are choice functions, used in particular to abstract from details of static or dynamic scheduling strategies. Rules of the form

$$\textbf{choose } x \in X \textbf{ do } R(x)$$

can be interpreted as an abbreviation for $R(\mathit{select}(X))$ where *select* is a selection function which applied to states with non-empty X yields an element of X.[33]

A widely used special notation is

$$\textbf{let } x = \mathit{new}(X) \textbf{ in } R$$

which is intended to provide a completely fresh (i.e. previously not used) element and to put it into X. We use it also in the simultaneous form

$$\textbf{let } x_1, \ldots, x_n = \mathit{new}(X) \textbf{ in } R$$

[32] Whereas in [245, 248] the output was disregarded, for the foundational analysis in [61] of the interaction between multi-agent parallel synchronous computational systems the output is introduced via a mechanism of sending information between agents, separated from updating by special "rules" **Output**(t) (which collect the values of terms t in the current state into an output multiset). We prefer to keep the simpler basic framework where the output functions are particular controlled functions; their values may well be multisets should one need to keep track of multiple instances of the same output value, issued by one agent in one basic computation step or simultaneously by different agents.

[33] There is a price to pay in terms of the underlying logic when moving the nondeterminism from selection functions into **choose**-rules; see Sect. 8.1.1.

with the understanding that pairwise different fresh elements are provided. The new elements come from a set *Reserve* whose role is to provide new elements whenever needed. Usually it is supposed to be infinite and to be part of the state, but without any structure. Nevertheless, when specifying ASMs we use freshly imported elements without further definition as arguments of standard operations, such as operations over sets, lists, trees, etc., abstracting from the definitions of such derived functions.[34] For a detailed technical justification and discussion of choice functions and of the *new* construct see Sect. 2.4.

Via the classification of locations, machine states are divided into an internal part, consisting of all the controlled and the internally updated shared locations with their values, and an external part, consisting of all the external and the externally updated shared locations with their values. Correspondingly we sharpen the definition of the ASM run given in Sect. 2.2.2 by stipulating that the set of updates an ASM M yields when applied in state S is a set of *internal updates* of M which determines the *next internal state* S', with unchanged values of the external and non-updated shared locations of M. The *next state* in which M may be applied is defined as the state resulting from S' by a set of *external updates* and possibly updates of some shared locations of M, as provided by the environment changes of monitored and shared locations of S.[35] When there are no updates made by the environment or when no confusion is to be expected, we identify the next internal state with the next state (formally one may consider this as assuming that the external updates provided by the environment are executed simultaneously with the updates computed by the machine).

2.2.4 ASM Modules

In this section we outline a standard module concept to syntactically structure large ASMs. The module interface for the communication with other modules is described by import and export clauses. The import clause specifies the names which are imported from other modules, and the export clause lists the names which can be imported by other modules. Obviously the transitive closure of the import clauses is not allowed to be cyclic. Every module is allowed to use only identifiers which are defined in the module or imported from other modules.

[34] From the logical point of view this means assuming on the set *Reserve* some "external" structure to be given without further definition, such as powersets, cartesian products etc. together with their standard operations involving reserve elements. For a formalization of such a *background* structure for the *Reserve* set and an analysis of its foundational implications for the concept of choice see [57].

[35] If a shared location in one "step" is updated both internally and externally, the external update wins, unless a protocol for updating the location specifies such conflict situations differently.

An ASM module is defined as a pair consisting of *Header* and *Body*. A module header consists of the name of the module, its import and its export clause, and its signature:

MODULE m
IMPORT $m_1(id_{11}, \ldots, id_{1l_1}), \ldots, m_k(id_{k1}, \ldots, id_{kl_k})$
EXPORT $id_1, \ldots, id_e$
SIGNATUREs

where $id_{i1}, \ldots, id_{il_i}$ are names for functions or rules which are imported from another module m_i, and $id_1, \ldots, id_e$ are the names for functions or rules which can be exported from module m. The signature s of a module, which determines its notion of state, contains all the basic functions occurring in the module and all the functions which appear in the parameters of any of the imported modules. We assume that there are no name clashes in these signatures.

The body of an ASM module consists of declarations (definitions) of functions and rules

$decl_1 \ldots decl_n \, axioms$

and may include also axioms expressing constraints one wants to assume for some of these functions or rules.

An ASM is then a module together with an optional characterization of the class of initial states and with a compulsory additional (the main) rule. We write ASMs in the same way as modules with *MODULE* replaced by *ASM*; the name of the ASM is used also as the name of the main rule. Executing an ASM means executing its main rule.

Every ASM M becomes an ASM module if its main rule is added (with name, say, M) to the declarations and the name M to the export list.

2.2.5 Illustration by Small Examples

We are now going to illustrate the function classification and the ASM constructs for parallelism and non-determinism by simple examples.

Clock. The following real-time clock illustrates the function classification.

```
$\textsc{Clock} = \textbf{if } DisplayTime + Delta = CurrTime \textbf{ then}$
  $DisplayTime := CurrTime$
```

CurrTime is a 0-ary monitored function which is supposed to be strictly increasing in a domain of real values determined by the desired precision. *DisplayTime* is a 0-ary controlled function whose values can be determined as belonging to some (not furthermore specified) set *Time*, and which is type compatible (or made so using some static conversion function) with the values of *CurrTime*. *Delta* is a 0-ary static real-valued function which determines the system dependent time granularity (and in this sense may be dynamic as part of another machine where it is controlled by an agent playing the role of the system operator); + is a static function representing the addition of reals for the needed precision.

One can separate the description of the rule guard computation from the rule itself by defining *ClockTick* as a derived 0-ary Boolean-valued function, e.g. by the specification

$$ClockTick = (DisplayTime + Delta = CurrTime)$$

or by an independent machine which computes *ClockTick*. Formulated this way the example exhibits the *pattern of sustaining signals* (or, more generally, events) which is supported in synchronous programming languages by the special construct *sustain S*; e.g. in Esterel it is described as an infinite loop to *emit S* at each clock tick [266].[36] Here the signal is *CurrTime*, and EMIT(*CurrTime*) means to update the output channel *DisplayTime* by *DisplayTime* := *CurrTime*.

$$\textsc{Sustain}(signal) = \textbf{if } ClockTick \textbf{ then } \textsc{Emit}(signal)$$

Experience shows that the use of derived functions is crucial for obtaining a manageable well-structured specification. The figures in the industrial project survey [121] report that in the ground model ASM of 120 rules developed there (which led to a final program of 9000 lines of generated C++ code), out of 315 functions only 71 were controlled against 116 derived, 59 static and 69 monitored ones. Numerous ASM models in the literature demonstrate the considerable modularization effect obtained by using static, derived and monitored functions.[37]

Resolving conflicting writes to shared variables. Hardware design languages provide constructs to cope with the problem that for given variables v, multiple and possibly conflicting update requests for $val(v)$ by independent processes may occur concurrently. In the IEEE standard for VHDL'93, for example, such conflicts are resolved by an implementation-defined resolution mechanism which can be represented by an external function *resolve*, selecting one value out of a set $competingVal(v)$ of values currently offered to

[36] We disregard in this example the peculiar handling of termination and timeout in Esterel, which is based upon the Watchdog construct.

[37] For example, the derived function *procdef* [71] dynamically determining the alternatives for a goal in a logic program abstractly represents a chunk of code which is responsible for much of the complexity of the WAM implementation of PROLOG [132], whereas in the same model the static function *unify* hides the details of a unification algorithm. The functions *DrivingVal* and *EffectiveVal*, derived from signal sources respectively from the port-signal association of a VHDL program [112], modularize a complex signal propagation scheme. The monitored *event* function in the ASM model for the Parallel Virtual Machine [107] encapsulates an asynchronous message-passing system underlying the interaction between PVM tasks and daemons; see Sect. 6.5.

update v. *competingVal* is a dynamic function which collects every update request to v by any of the involved processes which share v. These two functions determine the interface for the following variable assignment rule taken from [112, Sect. 4.2], where a static function *kind* is used to distinguish shared from local variables (whose values are stored in the environment of the process they belong to). The rule does not depend on the interface specification and determines the range of possible implementations modulo the interface. For example, it covers also the case of composite variables in VHDL'93 with possibly interleaved assignments to the component variables, simply by applying *resolve* componentwise.

```
VhdlSelectedAssign =if $Process$ executes $v = exp$ then
  if $kind(v) = local$ then $val(Process, v) := value(exp)$
    else $val(v) := resolve(competingVal(v))$
```

Bounded synchronous parallelism. We illustrate here the bounded synchronous parallelism of ASMs, i.e. the application of **forall** to finite sets of cardinality bounded by a fixed n. Consider cycling through a finite number n of given machines. The following ASM describing this has as static functions mod, the successor function and the equality over natural numbers, and might be considered as parameterized by a sequence of rules of length $n + 1$. A frequent special case is Alternate(R,S), which alternates two machines R and S.

```
CycleThru$(R_0, \ldots, R_n)$ = forall $i \leq n$ do
  if $cycle = i$ then
    $R_i$
    $cycle := (cycle + 1) \bmod (n + 1)$
```

Another example is taken from operating systems where an interrupt controller schedules the CPU access of a hardware-determined number of independent devices $device_i (i \leq n)$. The scheduler operates on a dynamic interrupt request array which records every $socket_i$ when it has been set to *high* by $device_i$, namely to signal the need of a driver process to get executed using the CPU. This is expressed by the following rule taken from [186], where the additional guard expresses that, upon termination of the *currDevice*, updating its *requestFrom*(*currDevice*) has priority over storing a new request.

```
InterruptStorage = if not $terminating(currDevice)$ then
  forall $i \leq n$ do if $socket_i = high$ and not $requestFrom(device_i)$
    then $\{socket_i := low, requestFrom(device_i) := true\}$
```

Conway's game of life. We use Conway's game of life to illustrate the unbounded synchronous parallelism of ASMs. Imagine a grid of square cells, elements of an abstract domain *Cell*, which can be alive or dead. The rule of survival describing the behavior of a single cell states that a cell with 3 alive neighbors gets (or remains) alive, whereas a cell with less than 2 or more than

3 alive neighbors dies. We represent this rule using an abstract predicate *alive* on *Cell* together with a derived function *aliveNeighb*: *Cell* → ℕ (not specified further here) which indicates for each cell the number of its alive neighbors. Then the rule of the game for a cell c is expressed by the following ASM:

```
CONWAY($c$) =
  if $aliveNeighb(c) = 3$ then $alive(c) := true$
  if $aliveNeighb(c) < 2$ or $aliveNeighb(c) > 3$ then $alive(c) := false$
```

Now, in every state of their life, all the cells are supposed to execute their life rule simultaneously, all in the same state, never mind the topology and the finite cardinality of *Cell* (which may be imagined also as a dynamic set). The following ASM expresses this behavior.

```
GAMEOFLIFE = forall $c \in Cell$ do CONWAY($c$)
```

In the next section we illustrate with Turner's Daemon Game another use of the synchronous ASM parallelism, namely to describe independent actions of multiple players without committing to any particular scheduling.

Swapping elements. The meaning of non-determinism as expressed by **choose**-rules is often explained by the following ASM which provides a specification for sorting of an array say a, namely by iterating a local swap (in a way not furthermore determined here).[38] In this example the synchronous parallelism avoids the use of intermediate storage.

```
SWAPSORT = choose $i, j \in dom(a)$ with $i < j$ and $a(i) > a(j)$
  SWAP($a(i), a(j)$)

SWAP($x, y$) = $\{x := y, y := x\}$
```

Choosing variable assignments. In the high-level *C*ommon *O*bject-oriented *L*anguage for *D*esign[39] the following construct allows us to express a non-deterministic choice of a subset of variables in a set *Var* and a subset of values in a set *Value* for updating the chosen variables to the chosen values.

```
COLDMODIFY($Var$) =
  choose $n \in \mathbb{N}$, choose $x_1, \ldots, x_n \in Var$, choose $v_1, \ldots, v_n \in Value$
    forall $1 \leq i \leq n$ do $val(x_i) := v_i$
```

[38] The turbo ASMs QUICKSORT on p. 172 and MERGESORT on p. 173 correctly refine this specification.

[39] For more details on COLD see Sect. 7.1.2.

Ambiguous grammars. The following ASM illustrates the power of non-determinism which is provided by the **choose** construct. It generates for a given alphabet A and a given natural number $n > 0$ exactly the set of all pairs vw of different words v, w over A, both of length n. It does it in the sense that if all possible choices are realized, the set of reachable states vw of this ASM (where v, w represent the only two controlled functions), started with say $vw = ab$ for some letters $a \neq b$ in A, is the desired set.

```
DifferentWords($A, n$) = choose $i$ with $1 \leq i \leq n$
  choose $a, b \in A$ with $a \neq b$
    $v(i) := a$
    $w(i) := b$
  forall $j$ with $1 \leq j \leq n$ and $j \neq i$
    choose $c, d \in A$
      $v(j) := c$
      $w(j) := d$
```

The language generated by this ASM is accepted by some non-deterministic finite automaton with $O(n^2)$ states, but every unambiguous automaton that accepts it needs at least 2^n states [304]. A similar example from op.cit., for arbitrary fixed n, is the set $\{0,1\}^{n-1}1\{0,1\}^*$ of words over alphabet $0, 1$ with a 1 in the n-th place. There is a non-deterministic FSM with $O(n)$ states which accepts the set, but every deterministic FSM accepting this set has at least 2^n states. It is generated as a set of all possible values of *out* by the following ASM:

choose $v \in \{0,1\}^{n-1}, w \in \{0,1\}^*$ **in** $out := v1w$

Problem 2 (Alternating choose/forall classification). Classify ASMs by alternations of **choose**, **forall** and relate the resulting classes of machines to known quantifier hierarchies in logic and complexity theory.

Scheduling non-deterministic rule execution. For the special case of non-deterministic choice among rules $R(i)$ we use the following abbreviation (in the literature often [] is used instead of **or**):

$R(0)$ **or** $\ldots$ **or** $R(n-1)$ = **choose** $i < n$ **do** $R(i)$

The same rule is also expressed by $R(select)$, where *select* is a 0-ary monitored choice function taking rules with index $< n$. The refinement of this rule to Round Robin scheduling can be obtained by adding in parallel the scheduler as follows:

RoundRobin $= \{R(select), select := select + 1 \bmod n\}$

This is a special case of the following general scheme for scheduling by a *scheduler* the non-deterministic execution $R(select)$ of rules *select*ed from an often dynamic set S. The use of such a possibly dynamic *scheduler* function allows one to restrict the unconstrained non-determinism (so-called *interleaving*) by conditions which may still leave some freedom to choose the next rule to be executed, maybe also in dependence of the state where *scheduler* is used. One can either specify *select* as derived function, e.g. by an equational definition, or include an update to this purpose in the scheduled machine as follows.

```
SCHEDULING($S$, $scheduler$) =
  $R(select)$
  $select := scheduler(S, select)$
```

New-instruction in Java. We illustrate the use of the special form **let** $x = new(X)$ by a description of the creation of new class instances in Java. If in the current program (read: in the abstract syntax tree described by the unary function *pgm*) at the current position (described by the 0-ary function *pos*) an instruction *new c* occurs for execution, then (in case class c is initialized) a new reference to an object of that class is placed on the heap with all its instance fields initialized to the default value of the class. To formalize that the new reference is returned, it is substituted in the abstract syntax tree for the executed instruction *new c*. The test whether c is initialized is due to the language manual requirement that Java classes have to be initialized before being accessed. The predicate *initialized* which we consider in the rule below as monitored can be refined independently of its use here, together with the corresponding submachine $initialize(c)$ for initializing a class; see [406]. This description is concisely and completely expressed by the following ASM rule taken from [406]:

```
JAVAINSTANCECREATION = if $pgm(pos) = new\ c$ then
  if $initialized(c)$ then
    let $ref = new(Reference)$ in
      $heap(ref) := Object(c, InstFieldsDefaultVal(c))$
      $pgm(pos) := ref$
  else $initialize(c)$
```

Creating new Occam processes. Another example illustrating the use of **let** $x = new(X)$ is an ASM formalization of the semantics of instructions which handle the parallelism in the programming language OCCAM. When an agent (in OCCAM *parlance* a daemon process) a in running *mode*, upon walking through an abstract syntax tree, in its current *pos*ition has to execute an instruction $par(a, k)$, it spawns k new child daemons, activates them – i.e. equips each of them with an instance of the current variable environment $a.env$, places it at the start position $pos(a, i)$ of its respective code and puts it to running *mode* – , leaves its trace as parent process to whom to report, and goes itself to its next position in idle mode. This is succinctly expressed by the following rule taken from [104] (and refined there into a model for its Transputer implementation), which is instantiated to the subprocess spawning rule AltTmSpawn for alternating Turing machines on p. 290 and is similar to the UML activity diagram rule UmlFork on p. 279. If you wonder how after having fired this rule a daemon process returns to running *mode*, see Exercises 2.2.1, 2.2.2.

```
OccamParSpawn =
  if $a.mode = running$ and $instr(a.pos) = par(a, k)$ then
    forall $1 \le i \le k$ let $b = new(Agent)$ in
      Activate$(b, a, i)$
      $parent(b) := a$
    $a.mode := idle$
    $a.pos \quad := next(a.pos)$
  where Activate$(b, a, i)$ =
    $\{b.env := a.env,\ b.pos := pos(a, i),\ b.mode := running\}$
```

Function classification to support modularity. Judicious selection of which functions or locations in an ASM specification are external and which ones are internal helps to achieve modular descriptions and supports information hiding, enhanced by an appropriate mix of (explicit or inductive) declarative and of operational definition elements. For example, the complex signal assignment $S \leftarrow$ `INERTIAL`... with inertial delay in the hardware design language VHDL'93 can be defined operationally by the simple rule below taken from [111, Sect. 3.2.2], using separately defined explicit as well as recursive external functions. The meaning of the inertial delay instruction $S \leftarrow$ `INERTIAL` exp_1 `AFTER` $time_1, \dots, exp_n$ `AFTER` $time_n$ extends the simpler `TRANSPORT` delay which preemptively schedules for signal S on the $driver(P, S)$ of process P each value $val(exp_i)$ for time $currTime + time_i$. Pre-emption means that values which were scheduled on the driver for time points $\geq currTime + time_1$ are deleted; it can be defined explicitly by a function $|_<$, which for given driver and time t retains precisely the driver elements ('transactions') with time component $< t$. This results in the `TRANSPORT` driver update by $driver(P, S) \mid_{< currTime + time_1} * Waveform$ where

$$Waveform =< (val(exp_1), time'_1), \dots, (val(exp_n), time'_n) >$$

and $time'_i = currTime + time_i$, describing that after pre-emption of the current driver the waveform constituted by the schedule for the new values is appended. Note that by the discrete VHDL time model the sequence of time values $time_i$ is strictly increasing. In addition to this, the inertial delay manipulates the driver also for elements with time $< time'_1$, namely by an algorithm which keeps the first driver element and *rejects* all transactions whose value is not equal to the value $val(exp_1)$ of the first new transaction. The *reject* procedure is defined in the IEEE language reference manual for VHDL. This yields the following definition of inertial delay signal assignments:

```
InertialSignalAssign =if $Process$ executes
  $S \leftarrow$ INERTIAL $exp_1$ AFTER $time_1, \dots, exp_n$ AFTER $time_n$ then
```

```
if $time_1 = 0$ then $driver(Process, S) := Waveform$
  else $driver(Process, S) :=$
    $fst(driver(Process, S)) * reject(driver', time'_1) * Waveform$
where
  $driver' = tail(driver(Process, S)) \mid_{<time_1 + currTime}$
  $Waveform =< (val(exp_1), time'_1), \ldots, (val(exp_n), time'_n) >$
  forall $i \leq n$ $time'_i = currTime + time_i$
```

reject can be defined recursively on transaction lists *Trans* as follows:

```
$reject(Trans, Val) =$
  if $Trans = empty$ or $val(last(Trans)) \neq Val$ then $empty$
  else $reject(front(Trans, Val)) * last(Trans)$
```

A further variation of inertial signal assignments by an explicit pulse rejection limit, which may be different from the first driver element, is captured by a simple modification of *driver′* defined in [111]. This book is full of further examples which illustrate this flexibility in exploiting the advantages of different specification styles within the uniform semantical ASM framework.

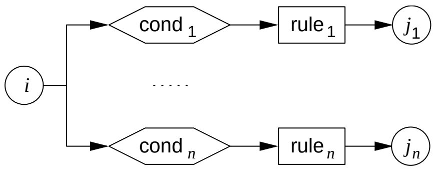
2.2.6 Control State ASMs

In this section we define a particularly frequent class of ASMs which represent a normal form for UML activity diagrams and allow the designer to define machines which below the main control structure of finite state machines provide synchronous parallelism and the possibility of manipulating data structures. We illustrate the definition here by small machines describing the state control structure of pipe statements in the system-level extension SpecC of C, of multi-threaded Java with an abstract scheduler, of the Java Virtual Machine, and of Turner's Daemon Game (which we use also as a first example to discuss the use of ASMs for ground model construction). The enrichment of the FSM control structure by parallelism and/or data structure manipulation is used also in numerous FSM extensions, e.g. to StreamProcessingFsms (p. 287), TimedAutomata (p. 288), co-design FSMs, etc., and provides also a uniform scheme for the generalization to stronger machine concepts like PushDownAutomaton and TuringLikeMachines (see Sect. 7.1). It also reflects the control function which *places* often play in Petri nets; e.g. see the machines in Sect. 6.1.

Definition 2.2.1. A *control state ASM* is an ASM whose rules are all of the form defined and pictorially depicted in Fig. 2.5. Note that in a given control state i, these machines do nothing when no condition $cond_j$ is satisfied.

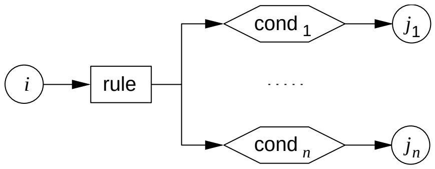
The finitely many control states $ctl\_state \in \{1, \ldots, m\}$ resemble the so-called "internal" states of Finite State Machines and can be used to describe different system *modes*. A particularly frequent form of control state ASM

Fig. 2.5 Control state ASMs



```
if $ctl\_state = i$ then
  if $cond_1$ then
    $rule_1$
    $ctl\_state := j_1$
  ...
  if $cond_n$ then
    $rule_n$
    $ctl\_state := j_n$
```

Fig. 2.6 Control state ASMs: alternative definition



```
if $ctl\_state = i$ then
  $rule$
  if $cond_1$ then
    $ctl\_state := j_1$
  ...
  if $cond_n$ then
    $ctl\_state := j_n$
```

is described in Fig. 2.6, which is equivalent to the original definition (see Exercise 2.2.5).

A typical example coming from language standardization is the machine SpecCPipe in Fig. 2.7, taken from [344] where an ASM definition is proposed for the semantics of the SpecC language, an extension of C by system-level features which are used in industrial hardware design. The machine is a control state ASM which defines the top-level sequential structure of the execution semantics of so-called pipe statements. These statements are parameterized by an *Init*ialization statement, a *cond*ition which guards an iterative process, by an *Incr*ementing statement used to advance the iteration, and by finitely many subprocesses which are spawned and deleted in a synchronized manner to fill, run and eventually flush the pipe. The corresponding submachines which appear in Fig. 2.7 can be defined as independent modules; see [344].

As with FSMs, one could consider deterministic and non-deterministic control state ASMs, using non-determinism as a mechanism to resolve possibly conflicting updates of *ctl_state*. For the reasons explained in Sect. 2.2.2 we prefer also for control state ASMs the parallel synchronous understanding of ASMs as firing in each step every rule. The designer can control possible conflicts, e.g. by taking care that the rule guards $cond_k$ of rules fireable in control state i are disjoint.

For the graphical representation of control states we will use in this book both their inscription into circles, as in Fig. 2.5, and the usual flowchart

Fig. 2.7 Control state ASM for SpecC pipe statements

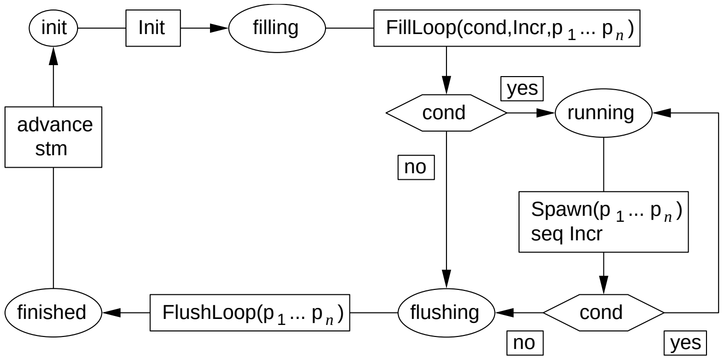
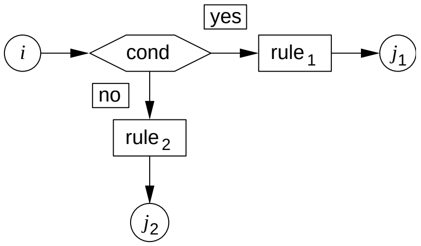


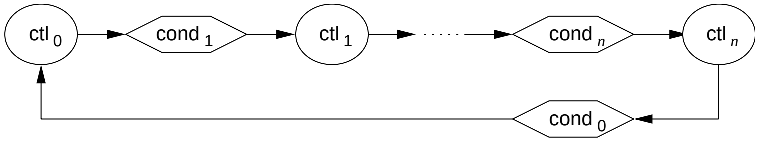
Fig. 2.8 Opposite conditions in control state ASMs



```
if $ctl\_state = i$ then
   if $cond$ then
      $rule_1$
      $ctl\_state := j_1$
   else
      $rule_2$
      $ctl\_state := j_2$
```

or UML notation where the control states appear as named directed arcs (arrows) or as unnamed arcs. The former notation, which is common in automata theory, helps to visually distinguish the role of control states – to "pass control" – from that of ASM rules, which describe the update "actions" concerning the underlying data structure and are inscribed into rectangles, often separated from the rule guards which are written into rhombs or hexagons labeling the arcs outgoing the control states as in Fig. 2.5 or ingoing as in Fig. 2.6, following the practice of UML activity diagrams. The most common cases are those of diagrams with one or with two opposite conditions in the rhombs or hexagons ($n = 1, 2$); in the latter case usually the condition is written into the rhomb and the two exiting arcs are labeled with "yes" and "no", respectively (Fig. 2.8), which are sometimes colored in grey to let them stand out better. As an example see the $\textsc{Switch}(cond_i, ctl_i)_i$ machine in Fig. 2.9, which under condition $cond_i$ switches to control state ctl_i. When using graphical notation we allow ourselves sometimes some self-explaining variations of the layout, which can always be reduced to the official definition explained above.

Fig. 2.9 Switch machine



Definition 2.2.2. When it is convenient to have also a textual representation besides the graphical one, we use the following translation.

$$\begin{aligned}&\textsc{Fsm}(i, \textbf{if } \mathit{cond} \textbf{ then } \mathit{rule}, j) =\\ &\quad \textbf{if } \mathit{ctl\_state} = i \textbf{ and } \mathit{cond} \textbf{ then } \{\mathit{rule}, \mathit{ctl\_state} := j\}\end{aligned}$$

Using this notation the textual representation of the rule in Fig. 2.5 becomes the set of rules $\textsc{Fsm}(i, \textbf{if } \mathit{cond}_k \textbf{ then } \mathit{rule}_k, j_k)$ for $k = 1, \ldots, n$. If in $\mathit{ctl\_state} = i$ the *cond*ition is not satisfied (and if there is no other rule for this control state), then what is often called a *persistent if-then* is realized: the machine remains in $\mathit{ctl\_state} = i$ until *cond* becomes true, in which case the machine proceeds to $\mathit{ctl\_state} = j$. An example is the special case of an alternating $\textsc{Switch}((\mathit{high}, 1), (\mathit{low}, 0))$, known as FlipFlop.

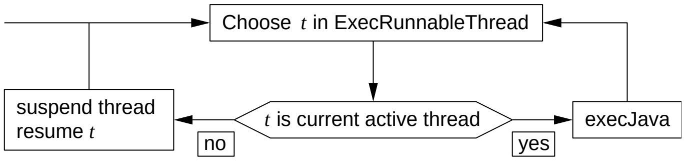
$$\begin{aligned}&\textsc{FlipFlop} =\\ &\quad \{\textsc{Fsm}(0, \textbf{if } \mathit{high} \textbf{ then skip}, 1), \textsc{Fsm}(1, \textbf{if } \mathit{low} \textbf{ then skip}, 0)\}\end{aligned}$$

Sometimes we will have ASMs which are built up from control state ASMs as submachines whereas the main machine has only one control state, see for example Fig. 2.13. In this case we omit in the textual representation mentioning the unique control state and its trivial updates; e.g. see the textual definition of DaemonGame in Sect. 2.1.1.

Control state ASMs represent a normal form of (synchronous) UML activity diagrams. In fact UML activity diagrams are defined as graphs connecting by labeled arcs so-called *action nodes* (rectangles, describing atomic actions to be performed and control to proceed) and *branching nodes* (rhombs, describing a case distinction resulting in the control to proceed to one among finitely many directions). Such diagrams can obviously be constructed by appropriately combining control state ASM rules as in Fig. 2.5, where the extreme cases are allowed to have rules without guard (read: with an always true guard, expressing that the rule is executed unconditionally) or rhombs not followed by a rectangle (as known from automata theory such successive rhombs can be compressed to one). Thus the normal form claim follows if one accepts the interpretation of the (intentionally undefined) UML notion of *action* as the application of an ASM rule.[40]

[40] In [98, 99, 153] the semantics of UML activity diagrams and state machines is formalized based upon that interpretation of actions. See also Sect. 6.5.

Fig. 2.10 Multiple thread Java machine execJavaThread



The proviso of synchrony made in the preceding argument stems from the fact that the concurrent nodes of UML activity diagrams may behave asynchronously. Concurrent nodes of UML in the synchronous understanding are covered by the action nodes considered above, since an ASM rule can consist of a finite number of subrules to be fired simultaneously. Concurrent nodes of UML in the asynchronous understanding, however, are different. As we will see in Chap. 6, they can be understood as calls of asynchronous multi-agent ASMs which work with a priori unrelated clocks, but are (expected to be) synchronized after each of them has returned a result, as described in Exercise 2.2.1 for OccamParSpawn.

The **multiple thread Java control structure**, characterized by an abstract scheduler (not detailed here) in the way implied by the language reference manual, illustrates the graphical notation we use for control state ASMs. In this example there is only one main control state in which a thread t among the executable runnable ones is chosen to execute the underlying single-threaded Java interpreter *execJava*, where in case the newly chosen t is different from the currently active *thread*, it must first be resumed and the currently active thread be suspended. The resulting machine ExecJavaThread expresses the separation of the semantics of thread execution from thread scheduling and is formalized by Fig. 2.10, taken from [406, p. 7] where the macros and *execJava* are further refined, which in the diagram appear as abstract submachines.

The **Java VM and bytecode verifier interaction** in Fig. 2.11 has one main control state in which a decision is taken as to which of the two submachines is executed: *trustfulVM* for the trustful execution of JVM code or *verifyVM* for the verification of all the methods of a (newly loaded) class. The almost self-explanatory definition of RefineDiligentVM in Fig. 2.11 has been exploited in [406] (where also the definition of the macros can be found) to modularize the specification and the verification of the JVM at each of four levels of the two submachines determined by language layers.

Figure 2.12 exhibits the main control state of the verifyVM machine governing the instantiation of its three submachines by an instruction to be verified. The submachines check the conditions to be verified and in case no failure has to be reported propagate the result to all successor instructions of the instruction which had been chosen for the current verification step.

Fig. 2.11 Decomposing JVM into trustfulVM and verifyVM

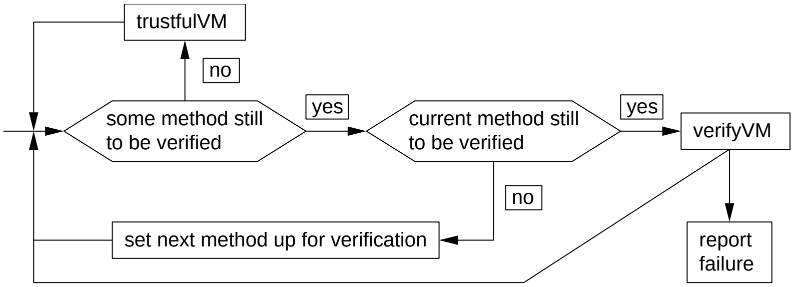
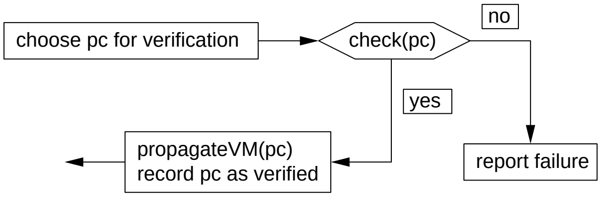

trustfulVM
no
yes
some method still to be verified
current method still to be verified
yes
verifyVM
no
set next method up for verification
report failure

Fig. 2.12 Decomposing verifyVM into propagateVMs and checks



(Each submachine is layered into further submachines corresponding to language levels; see [406].) This machine illustrates again the frequent use of the **choose** construct for implementation-independent abstract scheduling.

Turner's daemon game. We illustrate by this example the use of the control state ASM notation and of synchronous parallelism in defining a ground model ASM, which allows one to answer at an abstract level the questions posed in [416] concerning implementations of the game. For the sake of brevity we slightly rephrase the original problem formulation.

> Design a system for the following multi-player game. A daemon generates *bump* signals at random. Players in the environment of the system have to guess whether the number of generated bump signals is odd or even, by sending a *Probe* signal. The system replies by sending the signal *Win* or *Lose* if the number of the generated bump signals is odd or even, respectively. The system keeps track of the score of each player. The score is initially 0. It is increased (decreased) by 1 for each unsuccessful/successful guess. A player can ask for the current value of the score by the signal *Result*, which is answered by the system with the signal *Score*. Before a player can start playing, the player must log in by the signal *NewGame*. A player logs out by the signal *EndGame*. The system allocates a player a unique ID on logging in, and de-allocates it on logging out.

> The system cannot tell whether different IDs are being used by the same player.

One can recognize three categories of agents: users, players and the system. In the problem description users and players are not distinguished consistently. A player (or better a play, given that every user is allowed to have simultaneously different plays open) is created and initialized upon a user's (not a player's) *NewGame* input. This is expressed by the following rule, where the parameter *usr* is an element of the domain of all users (*USER*) and *in* is its input function, which we assume to be monitored by the system and consumed (e.g. by switching to value *undef*) when reading its value upon firing the rule:

```
NewGame($usr$) =
  if $usr.in = NewGame$ and $usr \in USER$ then
    let $p = new(PLAY)$ in Initialize($p$)
```

The macro Initialize(p) certainly includes the initialization of the score function $p.score := 0$ and communicating the created play to *usr*, say by $usr.out := p$. (See also the refinement below.) By formalizing logging in with the *new* construct, applied to a set *PLAY* (not furthermore specified), we avoid any particular mechanism of allocation of IDs as part of Initialize(p).[41] The corresponding log out rule (for a play, not a user) is as follows:

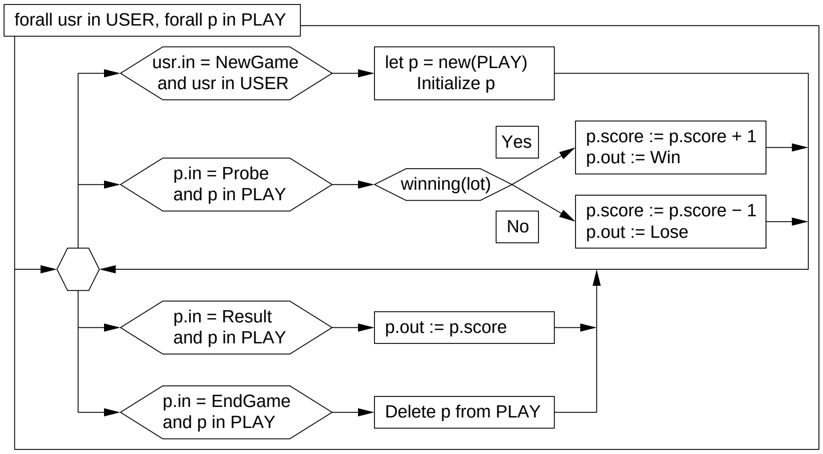
```
EndGame($p$) =
  if $p.in = EndGame$ and $p \in PLAY$ then Delete($p, PLAY$)
```

Delete($p, PLAY$) is supposed to have the effect that p is deleted from the dynamic set *PLAY*, e.g. by setting $PLAY(p) := false$ (or maybe with additional information to a garbage collector that p can be reused as an element of *Reserve*). This is all one needs to specify the "de-allocation" of a play.

To formalize the effect of moves in a play, we have to be clear about what the bump signals are. Apparently the only property which determines the meaning of the game is that the response of the system to an input $p.in = Probe$ is to answer to p either *Win* or *Lose* and to internally update $p.score$ accordingly. The nature of the *bump* does not matter really, so that we abstract from its implementation-oriented description in the problem formulation and formalize its effect by a predicate $winning(lot)$, with a parameter

[41] This answers the questions which are discussed in [416] about the "identification of players and games": "Presumably some identifiers are needed, but how should they be allocated and what should they distinguish?" "The players should be an anonymous part of the environment of the system." The answer is: each play is added upon creation as a fresh element to the abstract set *PLAY* from where it inherits its attributes and within which it has its "identity" as an element of the set, e.g. entitling it to send inputs *Probe*, *Result*, *EndGame* to the system.

Fig. 2.13 Daemon game ASM


forall usr in USER, forall p in PLAY
usr.in = NewGame and usr in USER
let p = new(PLAY) Initialize p
p.in = Probe and p in PLAY
winning(lot)
Yes
No
p.score := p.score + 1
p.out := Win
p.score := p.score − 1
p.out := Lose
p.in = Result and p in PLAY
p.out := p.score
p.in = EndGame and p in PLAY
Delete p from PLAY

lot representing a monitored function which will allow us to speak about various implementations of this notion of randomly issuing bump signals. This results in the following rule:

```
$\textsc{Probe}(p) = \textbf{if } p.in = Probe \textbf{ and } p \in PLAY \textbf{ then}$
  $\textbf{if } winning(lot) \textbf{ then } \{p.score := p.score + 1, p.out := Win\}$
    $\textbf{else } \{p.score := p.score - 1, p.out := Lose\}$
```

The request for the current result is formalized by sending the internal score as output to the play which required the information:

$$\textsc{Result}(p) = \ (\textbf{if } p.in = Result \textbf{ and } p \in PLAY \textbf{ then } p.out := p.score)$$

It remains for us to decide how the system handles simultaneous requests from users or plays. In order not to compromise the range of possible schedulings which can be used in an implementation, we exploit the synchronous parallelism of ASMs to make the system react in every step to every request which presents itself, serving every request independently of each other and immediately, in an atomic way, i.e. we construct a DaemonGame ASM which is quantified over all users and plays.[42] See the definition in Fig. 2.13.

```
$\textsc{DaemonGame} =$
  $\textbf{forall } usr \in USER \textbf{ do } \textsc{NewGame}(usr)$
  $\textbf{forall } p \in PLAY \textbf{ do } \{\textsc{Probe}(p), \textsc{Result}(p), \textsc{EndGame}(p)\}$
```

[42] As explained in Sect. 2.2.6, in the textual definition we skip mentioning the unique control state and its trivial update.

The problem solution formulated in SDL in [416, Chap. 5.5] is accompanied by an extensive discussion of typical ground model issues which can be clarified on the basis of the ASM model above. One question is about the nature of the bump-issuing daemon, namely whether it is an "integral part of the description" or an "artefact of the informal explanation". The implementation-oriented bump details in the problem description seem to be an artefact, because for the functionality of the game it does not matter whether winning or loosing is determined by an oracle or a random 0-1 function or an alternating 0-1 function, etc. The integral part is to declare bump (or *lot*, as we have preferred to say to render its random character) as monitored for the system and thereby to isolate all further questions about it as belonging to a separate part of the further implementation of the system. For example, the question whether the bump count is global ("since the system started") or per play is the question whether all plays read a unique bump location, *bump*, or whether one wants to derive the bump values local to a play from such a globally updated location, *bump*, say by

$$p.bump = bump - p.bump_{init}$$

with an update $p.bump_{init} := bump$ to be added to the macro Initialize(p).

Another question is about the interruption of *Probe* or *Result*: "Should it be allowable for another signal to be processed by the system between *Probe* and *Win*/*Lose*, or between *Result* and *Score*?" The expected answer "*Probe* and *Result* should be followed by their respective responses before any other signal is processed" is realized for the ASM model automatically, namely by the atomicity of single steps (rule execution).

Also, robustness conditions are asked for, such as: "What should happen if a player who is already logged in tries to issue *NewGame* again?"; "*NewGame* should be allowed to happen in a current game, but should be ignored."; "What should happen if a player issues any signal other than *NewGame* before logging into a game?"; "The intention was to allow *Probe*, *Result*, or *EndGame* when a game is not current, but to ignore these signals." All this is guaranteed by the fact that in the considered cases, no ASM rule is applicable.

Other questions are about excluding behavior: "What should happen if the player issues *Win*, *Lose*, or *Score* signals?"; "The intention was to disallow such behavior: it simply must not happen, as opposed to happening but be ignored." In fact it is excluded in the ASM model by the signature condition for the input functions $x.in \in \{NewGame, Probe, Result, EndGame\}$.

Reference. There are numerous formalizations for the Daemon Game in the literature. Compare the DaemonGame ASM, for example, with the Petri net in [50, Fig. 7, p. 314] or the process algebra (LOTOS) specification in [68, Sect. 6].

2.2.7 Exercises

Exercise 2.2.1. ($\rightsquigarrow$ CD) Complete OccamParSpawn by a rule which wakes up an Occam process after all its children have reported their termination.

Exercise 2.2.2. ($\rightsquigarrow$ CD) Refine the reporting of subprocesses to a parent process in Exercise 2.2.1 by introducing a children *count* at parent nodes. As a consequence of this children count, if two or more children want to report their termination simultaneously, their subtraction has to be taken with cumulative effect, e.g. by introducing multi-sets of items to be subtracted.[43]

Exercise 2.2.3 (Doubly linked lists). ($\rightsquigarrow$ CD) Write an ASM which provides a rule for each of the following operations and test predicates, respectively, and definitions for derived functions on double-linked lists over a set $VALUE$ of values. Assume a set $NODE$ of nodes with a special constant $null$ which is not in $NODE$. The unary dynamic functions $prev$, $next$ and $cont$ return the previous node, the next node and the content of a node. A double linked list L is represented by $head(L)$ and $tail(L)$, which contain the first and the last node of L.

Implement the following operations as ASMs:

CreateList(L): create an empty double-linked list L
Append(L, Val): append at the end of L the new element Val
Insert(L, Val, i): insert before the ith element of L the new element Val
Delete(L, i): delete the ith element from L
Update(L, i, Val): update the value of the ith element of L to Val
Cat(L_1, L_2, L): concatenate lists L_1, L_2 in the given order into list L
Split(L, n, L_1, L_2): split L into L_1, containing the first n elements of L, and L_2 containing the rest list of L

Define the following derived functions:

$get(L, i)$: return the value of the ith element of L
$index(L, Val)$: return the index of the first element of L with value Val
$length(L)$: return the length of L
$occurs(L, Val)$: return $true$, if Val is an element of L, and $false$ otherwise.

Prove the following properties to hold for the operations:

- If the next-link of x points to y, then the previous-link of y points to x.
- A list L is empty iff the next-link of its head points to its tail.

[43] Note that the vast majority of sequential as well as of distributed algorithms in a natural way do not lead to multiple occurrences of items, so that except for a few cases ASMs could be satisfactorily explained and used in terms of sets instead of multi-sets. In [262, 263] basic ASMs are extended by cumulative updates covering some relevant common data structures.

- After applying $\textsc{Append}(L, \mathit{Val})$, the list is not empty.
- A newly created linked list is empty and its length is 0.
- By Append/Delete the list length increases/decreases by 1.
- Executing $\textsc{Insert}(L, n, \mathit{Val})$ followed by $\textsc{Delete}(L, n)$ is the identity for L.

Exercise 2.2.4. ($\rightsquigarrow$ CD) Define an ASM which computes a non-recursive function. For an analysis of universal computation systems in terms of their functions for input, output, 1-step transition and termination criterion see [146] or the textbook elaboration in [70, pp. 129–136].

Exercise 2.2.5. ($\rightsquigarrow$ CD) Show that the notions of control state ASMs defined by Figs. 2.5, 2.6 are equivalent.

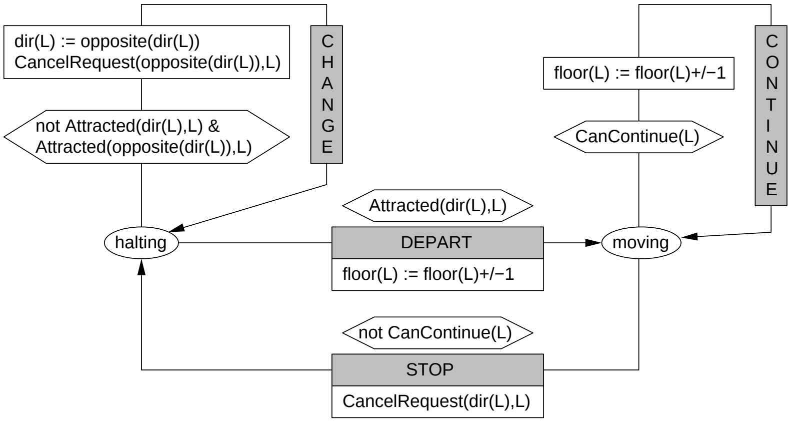
Exercise 2.2.6. Define $\textsc{Switch}(\mathit{cond}_i, \mathit{ctl}_i)_{i \leq n}$ by CycleThru.

2.3 Explanation by Example: Correct Lift Control

In this section we illustrate the definition of (control state) ASMs by constructing for the popular lift example a ground model for which we prove the desired correctness properties. The model can easily be refined to executable code. For the sake of brevity we slightly rephrase the original 1984 problem formulation by N. Davis, which is reprinted in [5].

> Design the logic to move n lifts between m floors satisfying the following requirements:
> 1. Each lift has for each floor one button which, if pressed, illuminates and causes the lift to visit (read: move to and stop at) that floor. The illumination is cancelled when the floor is visited by the lift.
> 2. Each floor (except ground and top) has two buttons to request an up-lift and a down-lift. They are cancelled when a lift visits the floor and is either traveling in the desired direction, or visits the floor with no requests outstanding. In the latter case, if both floor request buttons are illuminated, only one should be cancelled.
> 3. A lift without requests should remain in its final destination and await further requests.
> 4. Each lift has an emergency button which, if pressed, causes a warning to be sent to the site manager. The lift is then deemed “out of service”. Each lift has a mechanism to cancel its “out of service” status.
>
> Prove its correctness (well functioning) in the following sense:
> 1. All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel.
> 2. All requests for lifts from floors must be serviced eventually, with all floors given equal priority.

Fig. 2.14 Lift ground model



Lift FSMs. Looking at the relation between lifts and floors, a lift does nothing else than *moving* between or *halting* at floors. To pass from *halting* to *moving*, a lift has to DEPART, and vice versa to STOP. In its *moving* state a lift may also CONTINUE, typically[44] when arriving at a floor without request; when *halting*[45] it has also to be able to CHANGE its direction. This consideration brings us to a first view of a single lift as an FSM with the two control states *moving* and *halting* and four transitions ("symbolic" rules) not furthermore specified (and thereby non-deterministic). This is depicted in Fig. 2.14 (the specification of the macros resolve the non-determinism).

This level of abstraction allows one to prove (see Exercise 2.3.1) the following lemma, which will be useful for the correctness proof. To have a definite start condition we assume for the initialization that every lift is *halting* at the *ground* floor and directed *up*. Unless otherwise stated, we consider in the following only runs which are started in the initial state.

Lemma 2.3.1. For every lift the non-empty runs have the regular form
$(\text{DEPART CONTINUE}^* \text{ STOP})^+ \ (\text{CHANGE } (\text{DEPART CONTINUE}^* \text{ STOP})^*)^*$.

Enriching lift FSM transitions by floor manipulations. We now refine this FSM by assigning to the abstract transitions a more detailed meaning concerning the visited floors. This refinement is a $(1,1)$-refinement in which each abstract lift "operation" is replaced by a more detailed one; we interpret here an ASM rule (which may contain parallel actions) as an operation. For the sake of modularity we add a lift parameter L to the transition names to

[44] This becomes an additional requirement that a lift does not stop at floors without request.

[45] Clearly a lift is different from a paternoster and thus we assume as an additional requirement that it will never change its direction when moving.

indicate their uniform dependence upon the currently considered element in the abstract set *Lift*.

We assume a static ordered set of floors with functions $+1, -1$, *ground*, *top* to be given; the controlled function *floor*(L) takes its values in this set *Floor*. The direction *dir*(L) of a Lift is either *up* or *down*. Since the requested "logic to move" abstracts from matters like the duration of moves, closing or opening doors, etc., the immediate (atomic) effect of MoveLift is to update *floor*(L) by adding or subtracting 1 from it, depending on *dir*(L).

```
MoveLift($L$) =
  if $dir(L) = up$ then $floor(L) := floor(L) + 1$
  if $dir(L) = down$ then $floor(L) := floor(L) - 1$
```

DEPART. Requirements 1, 2 (given at the beginning of this section) demand that a lift L leaves its halting state when it is Attracted. We use Attracted in this intermediate model as a monitored function, constrained to reflect the abstract lift operation of cancelling requests when stopping or changing direction. Requirement 2 together with correctness property 2 imply that any L departs only in its direction *dir*(L) of travel, so that a *halting* L will Depart if it is Attracted$(dir(L), L)$, i.e. attracted in its direction of travel. This analysis leads to the following refinement of Depart:

```
Depart($L$) = if Attracted($dir(L), L$) then MoveLift($L$)
```

CONTINUE. Similarly to Depart, Continue also updates *floor*(L) if L CanContinue, so that Continue is refined as follows:

```
Continue($L$) = if CanContinue($L$) then MoveLift($L$)
```

STOP. Requirements 1, 2 imply that when a lift stops at a floor, the requests pending there in the lift's direction of travel have to be cancelled. Therefore the refinement of Stop is as follows:

```
Stop($L$) = if not CanContinue($L$) then
  CancelRequest($dir(L), L$)
```

CHANGE. Correctness property 2 (together with the obvious minimality assumption that a lift should not change its direction without being attracted in the opposite direction of its direction of travel) implies that L should change its direction of travel when and only when it is not attracted in its current direction *dir*(L) but attracted in the *opposite* direction. Requirement 2 implies that upon changing direction, L also has to cancel a request at *floor*(L) for the opposite direction. Thus we arrive at the following refinement of Change:

```
Change($L$) = let $d = dir(L)$ and $d' = opposite(dir(L))$ in
  if not Attracted($d, L$) and Attracted($d', L$) then
    {$dir(L) := d'$, CancelRequest($d', L$)}
```

Thus we arrive at the following refinement for the Lift ASM.

$$\begin{array}{l}\textsc{Lift}(\mathit{this}) = \\ \quad \textsc{Fsm}(\mathit{halting}, \textsc{Depart}(\mathit{this}), \mathit{moving}) \\ \quad \textsc{Fsm}(\mathit{moving}, \textsc{Continue}(\mathit{this}), \mathit{moving}) \\ \quad \textsc{Fsm}(\mathit{moving}, \textsc{Stop}(\mathit{this}), \mathit{halting}) \\ \quad \textsc{Fsm}(\mathit{halting}, \textsc{Change}(\mathit{this}), \mathit{halting})\end{array}$$

Clearly the parameterization with *this* suggests and paves the way for (but does not depend upon) an object-oriented implementation of a LIFT class with methods DEPART, CONTINUE, STOP, CHANGE, so that *Lift* appears as the set of current LIFT instances. Often, instead of *this*, also *self* is used.

At the level of abstraction at which we have defined LIFT we can add further information to the analysis of Lift FSM runs as regular expressions given in Lemma 2.3.1, namely on the floors traversed by a computation segment in DEPART CONTINUE* STOP and on the floors in which a CHANGE takes place. This is expressed by the following lemma which is proved in Exercise 2.3.2.

Lemma 2.3.2. Running from any state which is reachable from the initial state, LIFT(*this*) moves floor by floor in its direction of travel to the dynamically farthest point of attraction in that direction where, after at most $2 \cdot |\mathit{Floor}|$ steps, it STOPs and then either waits – namely iff it is not attracted in any direction – or it CHANGEs direction and moves in the new direction.

Corollary 2.3.1. Requirement 3 is satisfied by LIFT(*this*).

Refinement by request manipulations. The next refinement step is again a $(1, 1)$-refinement, this time a mixture of a pure data refinement and an operation refinement. It consists in more detailed definitions for the rule guards (a pure data refinement which replaces an abstract function by a derived one) and for the CANCELREQUEST macro appearing in Fig. 2.14. The rule guards are derived from an internal request function $\mathit{hasToDeliverAt}(L, \mathit{floor})$ (reflecting requirement 1 when inside the lift a button is pressed) and an external request function $\mathit{existsCallFromTo}(\mathit{floor}, \mathit{dir})$ (reflecting requirement 2 when on a floor outside the lift the *up* or *down* button is pressed). These two functions are shared between the lift user (who sets them, being part of the environment) and the lift control (which has to reset them in CANCELREQUEST to satisfy requirements 1, 2). The shared function $\mathit{existsCallFromTo}(\mathit{floor}, \mathit{dir})$ is supposed to be initially everywhere false. By requirement 2 it is constrained to be always false for the extreme cases $(\mathit{ground}, \mathit{down})$ and $(\mathit{top}, \mathit{up})$. The additional constraint that it is false as well for $(\mathit{floor}(L), \mathit{dir}(L))$ when L is *halting* formalizes that where a lift is halting no further call can be made for going in its direction of travel. Similarly, $\mathit{hasToDeliverAt}(L, \mathit{floor})$ is assumed to be initially everywhere false. The constraint that it is always false for $(L, \mathit{floor}(L))$ when L is *halting* formalizes that no further delivery can be requested for a floor where the lift is halting.

Definition 2.3.1. A lift has to visit a floor, if inside the lift the button for that floor is pressed, or outside the lift the button at that floor is pressed due to a pending request on that floor.

$$\textsc{HasToVisit}(L, \mathit{floor}) \iff \mathit{hasToDeliverAt}(L, \mathit{floor}) \textbf{ or}$$
$$\exists \mathit{dir}\colon \mathit{existsCallFromTo}(\mathit{floor}, \mathit{dir})$$

A lift is attracted in a direction d if it has to visit a floor in that direction.

$$\textsc{Attracted}(d, L) \iff$$
$$d = \mathit{up} \textbf{ and } \exists f > \mathit{floor}(L)\colon \textsc{HasToVisit}(L, f) \textbf{ or}$$
$$d = \mathit{down} \textbf{ and } \exists f < \mathit{floor}(L)\colon \textsc{HasToVisit}(L, f)$$

The definition of CanContinue realizes the priority given in requirement 2 to keeping the direction of travel of a lift.

$$\textsc{CanContinue}(L) \iff \textsc{Attracted}(\mathit{dir}(L), L) \textbf{ and}$$
$$\textbf{not } \mathit{hasToDeliverAt}(L, \mathit{floor}(L)) \textbf{ and}$$
$$\textbf{not } \mathit{existsCallFromTo}(\mathit{floor}(L), \mathit{dir}(L))$$

The macro for canceling a request in a lift for a floor and canceling a request from a floor for a direction is refined by the following ASM:[46]

$$\textsc{CancelRequest}(\mathit{dir}, L) =$$
$$\mathit{hasToDeliverAt}(L, \mathit{floor}(L)) := \mathit{false}$$
$$\mathit{existsCallFromTo}(\mathit{floor}(L), \mathit{dir}) := \mathit{false}$$

For the ASM with these three definitions, which we call again Lift, one can add and prove the information missing in Lemma 2.3.2 about turning off requests which have been served, as is proved in Exercise 2.3.3

Lemma 2.3.3. In runs of Lift, when moving to the farthest point of attraction in its direction of travel, it Stops at each floor where it is attracted, with respect to its direction of travel, and turns off the (internal) delivery request and the (external) call from that floor to go into the current direction of travel. When it Changes, it turns off the (external) call from its current floor to go into the new direction of travel.[47]

[46] A modularization effect can be obtained by adding a third parameter *floor* to the macro so that a scheduler can use it to cancel requests issued to a lift independently of its current floor.

[47] Without contradicting the requested well functioning properties, it may happen that Lift(L) stops at a floor f with an external request in the opposite direction d' of its direction of travel d, but with no other request pending (imagine a customer left after having pushed the down-button setting $\mathit{existsCallFromTo}(f, d')$ to *true*). Upon arrival at f, L cancels the external d-request and then remains *halting* without Change because there is no other floor left it HasToVisit. $\mathit{existsCallFromTo}(f, d')$ will be reset only should a request for another (higher or lower) floor arrive to make L again attracted.

Proposition 2.3.1. The machine LIFT satisfies the well functioning properties 1, 2.

Proof. The claim follows from Lemmas 2.3.1, 2.3.2, 2.3.3 by induction on LIFT runs, since every internal request from within a lift, and every not-yet-serviced external request from a floor, cause the lift to be eventually attracted in the requested direction. □

Remark 2.3.1. If a lift is requested on floor f in direction d, then the person at that floor has to wait at most $1 + 2 \cdot \mathit{dist}(L, f, d)$ steps until the lift L stops at floor f and either arrives in direction d or waits at f to change its direction. The distance is defined as follows. First, by $|f|_d$ we denote the maximum distance that a lift can travel in direction d starting at floor f.

$$|f|_d = \begin{cases} m - f, & \text{if } d = \mathit{up}; \\ f, & \text{if } d = \mathit{down}. \end{cases}$$

Then the distance to the lift L can be computed as follows, where $e = \mathit{dir}(L)$ and $\ell = \mathit{floor}(L)$:

$$\mathit{dist}(L, f, d) = \begin{cases} |\ell|_e - |f|_d, & \text{if } e = d \text{ and } |f|_d \leq |\ell|_e; \\ |\ell|_e + 2m - |f|_d + 2, & \text{if } e = d \text{ and } |\ell|_e < |f|_d; \\ |\ell|_e + m - |f|_d + 1, & \text{if } e \neq d. \end{cases}$$

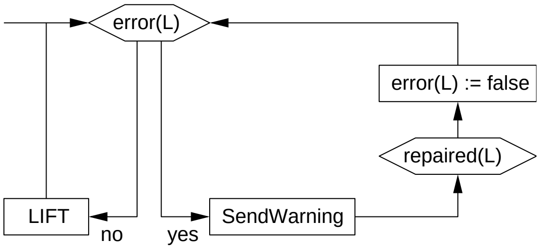
Remark 2.3.2. The proof does not exclude real-life situations with crowded lifts, where requests may be satisfied logically, but the lack of capacity prevents users from entering the lift. This problem has no logical solution, and should be solved by providing more capacity (larger bandwidth) on the basis of a performance analysis of the model.

Adding exception handling. Part of the requirements capture task is to check that all the requirements have been analyzed. In large applications this is a difficult task, whereas for the simple example here it is easy to check that our LIFT machine constructed so far fulfills all the requirements except number 4. We capture this requirement by a frequently occurring refinement step, called incremental or *conservative extension*, adding an exception handling machine to a machine which describes the functionality of faultless behavior.

Conservative extensions are typically used to incrementally introduce new behavior in a modular and controllable fashion, e.g. robustness conditions. To define a conservative extension of a given machine, one has to do the following:

- Define the *condition for the "new case"* in which the new machine should execute and the given machine either has no well-defined behavior or should be prevented from executing. In the example this condition is expressed by a shared Boolean-valued *error* function which is supposed to become true when an emergency has happened.

Fig. 2.15 Lift exception handling model



- Define the *new machine* with the desired additional behavior, e.g. an exception handling machine which is executed in case an error has been thrown. The return from the "new" to the "old" machine, if there is any, is typically a result of the computation of the new machine. For the lift the new machine consists of two rules, an out-of-service entry rule with guard $error(L) = true$ to SendWarning to the site manager, and an exit rule with guard, say $repaired(L) = true$, to cancel the "out-of service" status (reset *error*) for returning in the normal case machine. No requirement has been formulated for the service action proper, so that there is no further rule to be defined here.
- Add the new machine and *restrict the given machine to the "old case"* by guarding it by the negation of the "new case" condition, in the example $error(L) \neq true$.

The resulting machine is defined in Fig. 2.15 (where we omit depicting the two control states *normal*, in which *error(L)* is checked, and *out-of-service* in which *repaired(L)* is checked). It satisfies requirement 4 (see Exercise 2.3.4). A real-life example which follows the rather typical pattern explained here for the simple lift control is the refinement of the Java machine by an exception handling mechanism that has been proven to be correct in [406, Chap. 6].

Scheduling refinement. If one wants to separate the investigation of the correctness conditions for single lifts from an analysis – in a dedicated refinement step – of the possible schedulings for a set of multiple lifts, one can synchronize all lift instances in parallel, guaranteeing the independence of their actions and the preservation of the properties shown above for every instance of Lift. In the following definition we denote by Lift(L) the result of replacing *this* by L in *Lift*.

$$\textsc{ParLift} = \textbf{forall } L \in \mathit{Lift} \textbf{ do } \textsc{Lift}(L)$$

Such a parallel lift machine makes all lifts attracted by all external calls, i.e. where $existsCallFromTo(floor, d)$ is true for some direction d so that it can happen that the same request is serviced by every lift. This can be avoided by introducing a scheduler *scheduledTo* which assigns exactly one lift to each external call. A modular way to achieve this for our LIFT machine is to refine the interface predicates HASTOVISIT, CANCONTINUE and the CANCELREQUEST macro defined in Def. 2.3.1, strengthening[48] the listening to every external request from a floor F by the more specific listening to a request from the scheduler, namely $L = \mathit{scheduledTo}(\mathit{floor}, \mathit{dir})$. Thus we obtain a lift PARLIFT with scheduler by replacing Def. 2.3.1 with the following one.

Definition 2.3.2. A Lift has to visit a floor, if inside the lift the button for that floor is pressed, or on that floor a request has been made and the lift has been selected by the scheduler.[49]

```
HasToVisit$(L, \mathit{floor}) \iff \mathit{hasToDeliverAt}(L, \mathit{floor})$ or
  $\exists \mathit{dir}\colon \mathit{existsCallFromTo}(\mathit{floor}, \mathit{dir})$ and $L = \mathit{scheduledTo}(\mathit{floor}, \mathit{dir})$

CanContinue$(L) \iff$ Attracted$(\mathit{dir}(L), L)$ and
  not $\mathit{hasToDeliverAt}(L, \mathit{floor}(L))$ and
  not $(\mathit{existsCallFromTo}(\mathit{floor}(L), \mathit{dir}(L))$ and
      $L = \mathit{scheduledTo}(\mathit{floor}(L), \mathit{dir}(L)))$
```

To refine CANCELREQUEST we guard the canceling of an external request from a floor for a direction by $L = \mathit{scheduledTo}(\mathit{floor}(L), \mathit{dir})$. Under appropriate assumptions, *scheduledTo* can be proved to preserve the correctness properties for PARLIFT, though this may lead to longer waiting times for the lift users (see Exercise 2.3.5).

```
CancelRequest$(\mathit{dir}, L) =$
  $\mathit{hasToDeliverAt}(L, \mathit{floor}(L)) := \mathit{false}$
  if $L = \mathit{scheduledTo}(\mathit{floor}(L), \mathit{dir})$ then
    $\mathit{existsCallFromTo}(\mathit{floor}(L), \mathit{dir}) := \mathit{false}$
```

Refinement to requirement changes. We illustrate with two small examples how to exploit modeling with ASMs for "design-for-change", namely by refinements which cope with additional requirements brought in after the original design task has been accomplished. Consider the additional request to schedule only non-crowded lifts. This can be accommodated by constraining the scheduler *scheduledTo* to choose only lifts which are *nonCrowded*, but at a price: such a simple minded refinement of the scheduling affects the

[48] One could think of simply replacing the predicate $\mathit{existsCallFromTo}(F, \mathit{Dir})$ by $L = \mathit{scheduledTo}(F, \mathit{Dir})$, but that would lead to a less modular refinement. The modularity of our refinement implies also that we leave it to the scheduler and not to the lift's request canceling to invalidate the scheduling $L = \mathit{scheduledTo}(F, \mathit{Dir})$ once this order has been executed.

[49] For purposes of modularization one could assume that $\mathit{scheduledTo}(\mathit{floor}, \mathit{dir})$ implies $\mathit{existsCallFromTo}(\mathit{floor}, \mathit{dir})$, so that listening to external calls in the macro HASTOVISIT becomes replacable by listening to the scheduler.

correctness property in Proposition 2.3.1, namely in case of a continuously crowded lift. We leave it as an exercise to construct a counter-example.

Consider the additional request to reserve a lift $L_{n,m}$ for a floor section, say $[n, m]$, and floor 1 with $1 < n < m$. It suffices to

- constrain the internal call function $\mathit{hasToDeliverAt}(L_{n,m})$ to those floors,
- restrict the external call function *existsCallFromTo* for those floors to calls within that section – a "hardware" requirement,
- restrict the scheduler *scheduledTo* to take into account the requested target section.

The correctness property is relativized to the served section (see Exercise 2.3.6).

References. In the literature there are numerous formalizations for the lift control. See in particular the Petri net in [371, Sect. 4, Fig. 26] and the B machine in [5, Sect. 8.3] (which has inspired our Lift ASM).

2.3.1 Exercises

Exercise 2.3.1. ($\rightsquigarrow$ CD) Prove Lemma 2.3.1.

Exercise 2.3.2. ($\rightsquigarrow$ CD) Prove Lemma 2.3.2 by an induction on runs of an arbitrary $L \in \mathit{Lift}$.

Exercise 2.3.3. ($\rightsquigarrow$ CD) Prove Lemma 2.3.3.

Exercise 2.3.4. Show the machine in Fig. 2.15 to satisfy requirement 4.

Exercise 2.3.5. Formulate assumptions on the scheduler *scheduledTo* under which it can be proved to preserve the correctness properties for ParLift. Show by an example that this refinement may lead to longer waiting times for the lift users.

Exercise 2.3.6. Prove the correctness property for $L_{n,m}$.

Exercise 2.3.7. ($\rightsquigarrow$ CD) Refine the above Lift ASM to an executable program of a programming language of your choice.

Exercise 2.3.8. ($\rightsquigarrow$ CD) Extend the Lift ASM by introducing opening and closing doors first as atomic action, then as durative action, then together with error handling for cases where doors do not open or close.

2.4 Detailed Definition (Math. Foundation)

In this section we provide a detailed mathematical definition for the syntax and semantics of ASMs. We first introduce the notion of the abstract state and summarize some elementary definitions from mathematical logic. Then we proceed to the update set semantics of ASM transition rules, including already the extension by sequencing and calling turbo submachines, which is explained in Chap. 4. Finally, we extend it to ASMs with a reserve, mathematically clarifying what the introduction of fresh elements means in a parallel context where multiple independent choices of new elements may take place simultaneously.[50] The mathematical definition will be extended to asynchronous multi-agent ASMs in Sect. 6.1. The reader is advised to skip this section and to come back to it only should the need be felt during further reading of the book.

2.4.1 Abstract States and Update Sets

The states of ASMs are *algebraic structures* as introduced in standard mathematical logic or universal algebra textbooks. Algebraic structures can be viewed as abstract memories. The arguments of the functions are the locations of the memory, whereas the values of the functions are its contents. In a similar way *relational structures* are sometimes viewed as databases. The tuples of the relations correspond to the rows of the database tables.

Definition 2.4.1 (Signature). A *signature* Σ is a finite collection of function names. Each function name f has an *arity*, a non-negative integer. Nullary function names are called *constants*. Function names can be *static* or *dynamic*. The dynamic functions are further classified according to Fig. 2.4. Every ASM signature is assumed without further mention to contain the static constants *undef*, *true*, *false*.

Signatures are also called *vocabularies*. The arity of a function name is the number of arguments that the function takes. Be aware that, as we will see below, the interpretation of dynamic nullary functions can change from one state to the next, so that they correspond to the variables of programming.

Example 2.4.1. The signature Σ_{bool} of Boolean algebras contains two constants 0 and 1, a unary function name '$-$' and two binary function names '$+$' and '$*$'.

Definition 2.4.2 (State). A *state* $\mathfrak{A}$ for the signature Σ is a non-empty set X, the *superuniverse* of $\mathfrak{A}$, together with *interpretations* of the function names of Σ. If f is an n-ary function name of Σ, then its interpretation $f^{\mathfrak{A}}$ is a function from X^n into X; if c is a constant of Σ, then its interpretation $c^{\mathfrak{A}}$ is an element of X. The superuniverse X of the state $\mathfrak{A}$ is denoted by $|\mathfrak{A}|$.

[50] For lecture slides see `AsmDefinition` (⇝ CD).

The superuniverse of a state is also called the *base set* of the state. The *elements* of a state are the elements of the superuniverse of the state. It is assumed without further mention that the interpretations of the constants *undef*, *true*, *false* are always pairwise different elements in any state. The constant *undef* represents an undetermined object, the default value of the superuniverse.

Formally, function names are interpreted in states as total functions. We view them, however, as being partial and define the *domain* of an n-ary function name f in $\mathfrak{A}$ to be the set of all n-tuples $(a_1, \ldots, a_n) \in |\mathfrak{A}|^n$ such that $f^{\mathfrak{A}}(a_1, \ldots, a_n) \neq \mathit{undef}^{\mathfrak{A}}$.

A *relation* is a function that has always the value *true*, *false* or *undef*. Think about an n-ary relation R as the set of all n-tuples $(a_1, \ldots, a_n)$ such that $R(a_1, \ldots, a_n) = \mathit{true}$. We allow relations to be partial.[51]

In applications, the superuniverse X of a state $\mathfrak{A}$ is usually divided into smaller *universes*, modeled by their characteristic functions (unary relations). The universe represented by R is the set of all elements a of $\mathfrak{A}$ for which $R(a) = \mathit{true}$. If a unary function R represents a universe, then we simply write $a \in R$ as an abbreviation for the formula $R(a) = \mathit{true}$.

Example 2.4.2. Consider the two states $\mathfrak{A}$ and $\mathfrak{B}$ for the signature Σ_{bool} of Example 2.4.1. The superuniverse of the state $\mathfrak{A}$ is the set $\{0, 1\}$. The functions are interpreted as follows, where a, b are 0 or 1:

$$\begin{array}{lcll}
0^{\mathfrak{A}} & = & 0 & \text{(zero)} \\
1^{\mathfrak{A}} & = & 1 & \text{(one)} \\
-^{\mathfrak{A}} a & = & 1 - a & \text{(logical complement)} \\
a +^{\mathfrak{A}} b & = & \max(a, b) & \text{(logical or)} \\
a *^{\mathfrak{A}} b & = & \min(a, b) & \text{(logical and)}
\end{array}$$

The superuniverse of the state $\mathfrak{B}$ is the power set of the set of non-negative integers $\mathbb{N}$. The functions are interpreted as follows, where a, b are subsets of $\mathbb{N}$:

$$\begin{array}{lcll}
0^{\mathfrak{B}} & = & \emptyset & \text{(empty set)} \\
1^{\mathfrak{B}} & = & \mathbb{N} & \text{(full set)} \\
-^{\mathfrak{B}} a & = & \mathbb{N} \setminus a & \text{(set of all } n \in \mathbb{N} \text{ such that } n \notin a) \\
a +^{\mathfrak{B}} b & = & a \cup b & \text{(set of all } n \in \mathbb{N} \text{ such that } n \in a \text{ or } n \in b) \\
a *^{\mathfrak{B}} b & = & a \cap b & \text{(set of all } n \in \mathbb{N} \text{ such that } n \in a \text{ and } n \in b)
\end{array}$$

[51] We deviate from [248] in this point. Our relations are functions that can be partial and hence take the value *undef*, whereas in [248] relations (or predicates) are total functions that can only take the values *true* or *false*. The treatment of relations as possibly partial functions simplifies, for example, the *reserve condition* in Def. 2.4.23. Unlike [248] we distinguish also between terms and formulas (see Sect. 2.4.2). The guards of **if-then-else** rules, which are boolean terms in [248], are formulas in our framework. Since the only atomic formulas are equations between terms which are either *true* or *false* by definition, formulas are always defined and we do not have to worry about undefined guards in **if-then-else** rules. Note that the main reason for restricting predicates to total boolean functions in [248] is to ensure that boolean terms are always *true* or *false*.

Both states, $\mathfrak{A}$ and $\mathfrak{B}$, are so-called Boolean algebras.

Example 2.4.3. The memory of a computer system can be viewed as an algebraic structure. Let us assume that the machine uses a 64-bit processor. Then the superuniverse of a state for the system is the set $\{0, 1\}^{64}$ consisting of all 64-bit strings. The constants $0, 1$ and the basic functions $+, *$ have their standard interpretation. In addition there is a unary dynamic function *mem*. The value $mem(i)$ is the content of cell i in the memory. The expression $mem(i + mem(j))$ denotes the content of cell $i + mem(j)$, etc.

In dynamic situations, it is convenient to view an abstract state as a kind of memory that maps locations to values.

Definition 2.4.3 (Location). A *location* of $\mathfrak{A}$ is a pair $(f, (a_1, \ldots, a_n))$, where f is an n-ary function name and $a_1, \ldots, a_n$ are elements of $|\mathfrak{A}|$. The value $f^{\mathfrak{A}}(a_1, \ldots, a_n)$ is called the *content* of the location in $\mathfrak{A}$. The *elements* of the location are the elements of the set $\{a_1, \ldots, a_n\}$.

A state $\mathfrak{A}$ can be viewed as a function that maps the locations of $\mathfrak{A}$ to its contents. We write $\mathfrak{A}(l)$ for the content of the location l in $\mathfrak{A}$.

Definition 2.4.4 (Update and update set). An *update* for $\mathfrak{A}$ is a pair (l, v), where l is a location of $\mathfrak{A}$ and v is an element of $|\mathfrak{A}|$. The update is *trivial*, if v is the content of l in $\mathfrak{A}$. An *update set* is a set of updates.

The meaning of the update is that the content of the location l in $\mathfrak{A}$ has to be changed to the value v. An update specifies how the function table of a dynamic function has to be updated at the corresponding location. Since due to the parallelism a transition rule may prescribe updating the same function at the same arguments several times, we require such updates to be consistent. Two updates *clash*, if they refer to the same location but are distinct.

Definition 2.4.5 (Consistent update set). An update set U is called *consistent*, if it has no clashing updates, i.e. if for any location l and all elements v, w, it is true that if $(l, v) \in U$ and $(l, w) \in U$, then $v = w$.

If an update set U is consistent, it can be fired in a given state. The result is a new state in which the interpretations of dynamic function names are changed according to U. The interpretations of static function names are the same as in the old state.

Definition 2.4.6 (Firing of updates). The result of firing a consistent update set U in a state $\mathfrak{A}$ is a new state $\mathfrak{A} + U$ with the same superuniverse as $\mathfrak{A}$ such that for every location l of $\mathfrak{A}$:

$$(\mathfrak{A} + U)(l) = \begin{cases} v, & \text{if } (l, v) \in U; \\ \mathfrak{A}(l), & \text{if there is no } v \text{ with } (l, v) \in U. \end{cases}$$

The state $\mathfrak{A} + U$ is called the *sequel* of $\mathfrak{A}$ with respect to U.

Since U is consistent, the state $\mathfrak{A} + U$ is determined in a unique way. Note that only those locations can have a new content in state $\mathfrak{A} + U$ with respect to state $\mathfrak{A}$ for which there is an update in U.

Given two states $\mathfrak{A}$ and $\mathfrak{B}$ with the same superuniverse and the same signature, there always exists a unique set of non-trivial updates that can be fired in state $\mathfrak{A}$ to obtain state $\mathfrak{B}$. This update set is called the difference $\mathfrak{B} - \mathfrak{A}$ of $\mathfrak{B}$ and $\mathfrak{A}$.

Definition 2.4.7 (Difference). Let $\mathfrak{A}$ and $\mathfrak{B}$ be two states with the same superuniverse. Then $\mathfrak{B} - \mathfrak{A} = \{(l, \mathfrak{B}(l)) \mid \mathfrak{B}(l) \neq \mathfrak{A}(l)\}$.

The difference $\mathfrak{B} - \mathfrak{A}$ of two states is always a consistent update set. Hence it can be fired in state $\mathfrak{A}$, and the result is the state $\mathfrak{B}$.

Lemma 2.4.1. $\mathfrak{A} + (\mathfrak{B} - \mathfrak{A}) = \mathfrak{B}$.

The basic idea of an abstract state is that the internal structure of the superuniverse is not important. The internal representation of the elements of an abstract state is hidden. What matters are the operations than can be performed on the elements. This principle of information hiding is expressed by the notion of an *isomorphism*. If the superuniverses of two abstract states can be mapped to each other one-to-one and the interpretations of the functions agree on corresponding elements, then the states are identified.

Before we introduce isomorphisms of abstract states, we define how functions that are defined on the superuniverse of an abstract state are extended to locations and update sets in the obvious way. A function α with domain $|\mathfrak{A}|$ is extended to locations of $\mathfrak{A}$ and update sets for $\mathfrak{A}$ as follows:

- If $l = (f, (a_1, \ldots, a_n))$, then $\alpha(l) = (f, (\alpha(a_1), \ldots, \alpha(a_n)))$.
- If U is an update set for $\mathfrak{A}$, then $\alpha(U) = \{(\alpha(l), \alpha(v)) \mid (l, v) \in U\}$.

A homomorphism is a function from one state into another that maps the content of a location to the content of the corresponding location in the other state.

Definition 2.4.8 (Homomorphism). Let $\mathfrak{A}$ and $\mathfrak{B}$ be two states over the same signature. A *homomorphism* from $\mathfrak{A}$ to $\mathfrak{B}$ is a function α from $|\mathfrak{A}|$ into $|\mathfrak{B}|$ such that $\alpha(\mathfrak{A}(l)) = \mathfrak{B}(\alpha(l))$ for each location l of $\mathfrak{A}$.

Example 2.4.4. Consider the two boolean algebras $\mathfrak{A}$ and $\mathfrak{B}$ from Example 2.4.2. Let $n \in \mathbb{N}$. For a subset $X \subseteq \mathbb{N}$ define

$$\alpha_n(X) = \begin{cases} 1, & \text{if } n \in X; \\ 0, & \text{otherwise.} \end{cases}$$

Then α_n is a homomorphism from $\mathfrak{B}$ to $\mathfrak{A}$.

Definition 2.4.9 (Isomorphism). An *isomorphism* from $\mathfrak{A}$ to $\mathfrak{B}$ is a homomorphism from $\mathfrak{A}$ to $\mathfrak{B}$ which is a one-to-one function from $|\mathfrak{A}|$ onto $|\mathfrak{B}|$. Two states $\mathfrak{A}$ and $\mathfrak{B}$ are called *isomorphic*, if there exists an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$.

The following lemma says that if two states are isomorphic, then their sequels with respect to a consistent update set are also isomorphic.

Lemma 2.4.2. Let α be an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$. If U is a consistent update set for $\mathfrak{A}$, then $\alpha(U)$ is a consistent update set for $\mathfrak{B}$ and α is an isomorphism from $\mathfrak{A} + U$ to $\mathfrak{B} + \alpha(U)$.

Proof. That the update set $\alpha(U)$ is consistent follows, since α is one-one.

To show that α is a homomorphism from $\mathfrak{A} + U$ to $\mathfrak{B} + \alpha(U)$, we take an arbitrary location l of $\mathfrak{A}$.

If the update (l, v) is in U, then the update $(\alpha(l), \alpha(v))$ is in $\alpha(U)$. By Def. 2.4.6, $(\mathfrak{A} + U)(l) = v$ and $(\mathfrak{B} + \alpha(U))(\alpha(l)) = \alpha(v)$.

If there is no update for l in U, then there is also no update for $\alpha(l)$ in $\alpha(U)$. Hence, by Def. 2.4.6, $(\mathfrak{A} + U)(l) = \mathfrak{A}(l)$ and $(\mathfrak{B} + \alpha(U))(\alpha(l)) = \mathfrak{B}(\alpha(l))$. Since α is a homomorphism from $\mathfrak{A}$ to $\mathfrak{B}$, we have $\alpha(\mathfrak{A}(l)) = \mathfrak{B}(\alpha(l))$. □

The composition $U \oplus V$ of two update sets U and V is the set of updates obtained from U by adding the updates of V and overwriting updates in U which are redefined in V. The composition of update sets corresponds to the sequential application of the updates.

Definition 2.4.10 (Composition of update sets).

$$U \oplus V = V \cup \{(l, v) \in U \mid \text{there is no } w \text{ with } (l, w) \in V\}$$

The composition of update sets is associative. If U and V are consistent, then $U \oplus V$ is consistent, too. Applying the update set $U \oplus V$ to state $\mathfrak{A}$ is the same as first applying U and then V.

Lemma 2.4.3. Let U, V, W be update sets.

1. $(U \oplus V) \oplus W = U \oplus (V \oplus W)$
2. If U and V are consistent, then $U \oplus V$ is consistent.
3. If U and V are consistent, then $\mathfrak{A} + (U \oplus V) = (\mathfrak{A} + U) + V$.

2.4.2 Mathematical Logic

In first-order logic (FOL), terms and formulas play a central role. Terms and formulas are syntactic objects. They are interpreted in abstract states. Terms denote elements of the states, whereas formulas denote properties of the elements. Terms are built up from variables and constants using function names. Formulas are built up from equations between terms using boolean connectives and quantifiers. Terms are denoted by r, s, t; formulas are denoted by φ, ψ. Variables are a special kind of identifier and are denoted by x, y, z.

Definition 2.4.11 (Term). Let Σ be a signature. The terms of Σ are syntactic expressions generated as follows:

1. Variables x, y, z, ... are terms.
2. Constants c of Σ are terms.
3. If f is an n-ary function name of Σ, $n > 0$, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

A term which does not contain variables is called a *ground term.* A term is called *static*, if it contains static function names only. By $t\frac{s}{x}$ we denote the result of replacing the variable x in term t everywhere by the term s (substitution of s for x in t).

Example 2.4.5. The following are terms of the signature Σ_{bool}:

$$+(x, y), \quad +(1, *(z, 0))$$

They are usually written as $x + y$ and $1 + (z * 0)$.

Since terms are syntactic objects, they do not have a meaning. A term can be evaluated in a state, if elements of the superuniverse are assigned to the variables of the term.

Definition 2.4.12 (Variable assignment). Let $\mathfrak{A}$ be a state. A *variable assignment* for $\mathfrak{A}$ is a finite function ζ which assigns elements of $|\mathfrak{A}|$ to a finite number of variables. We write $\zeta[x \mapsto a]$ for the variable assignment which coincides with ζ except that it assigns the element a to the variable x. So we have:

$$\zeta[x \mapsto a](y) = \begin{cases} a, & \text{if } y = x; \\ \zeta(y), & \text{otherwise.} \end{cases}$$

Variable assignments are also called *environments.* The range $ran(\zeta)$ of a variable assignment ζ is the set of all elements that occur in bindings of ζ.

Given a variable assignment a term can be interpreted in a state.

Definition 2.4.13 (Interpretation of terms). Let $\mathfrak{A}$ be a state of Σ, ζ be a variable assignment for $\mathfrak{A}$ and t be a term of Σ such that all variables of t are defined in ζ. By induction on the length of t, a value $[\![t]\!]^{\mathfrak{A}}_{\zeta}$ is defined as follows:

1. $[\![x]\!]^{\mathfrak{A}}_{\zeta} = \zeta(x)$
2. $[\![c]\!]^{\mathfrak{A}}_{\zeta} = c^{\mathfrak{A}}$
3. $[\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{A}}_{\zeta} = f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta})$

The interpretation of t depends on the values of ζ on the variables of t only.

Lemma 2.4.4 (Coincidence). If ζ and η are two variable assignments for t such that $\zeta(x) = \eta(x)$ for all variables x of t, then $[\![t]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\eta}$.

A homomorphism from one state into another preserves the values of terms. By $\alpha \circ \zeta$ we denote the variable assignment that binds a variable x defined in ζ to the value $\alpha(\zeta(x))$.

Lemma 2.4.5 (Homomorphism). If α is a homomorphism from $\mathfrak{A}$ to $\mathfrak{B}$, then $\alpha([\![t]\!]^{\mathfrak{A}}_{\zeta}) = [\![t]\!]^{\mathfrak{B}}_{\alpha\circ\zeta}$ for each term t.

If we substitute the term s for the variable x in t and evaluate the result $t\frac{s}{x}$ in a state $\mathfrak{A}$ under a variable assignment ζ, then we obtain the same value as when we first evaluate s in $\mathfrak{A}$ under ζ and then the term t in $\mathfrak{A}$ under the extended variable assignment, where the variable x is bound to the value of s.

Lemma 2.4.6 (Substitution). Let $a = [\![s]\!]^{\mathfrak{A}}_{\zeta}$. Then $[\![t\frac{s}{x}]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\zeta[x\mapsto a]}$.

Example 2.4.6. Consider the state $\mathfrak{A}$ for Σ_{bool} of Example 2.4.2. Let ζ be a variable assignment with $\zeta(x) = 0$, $\zeta(y) = 1$ and $\zeta(z) = 1$. Then we have:

$$[\![(x+y)*z]\!]^{\mathfrak{A}}_{\zeta} = 1.$$

The same term can be interpreted in the state $\mathfrak{B}$ of Example 2.4.2. Let $\eta(x) = \{2,3,5\}$, $\eta(y) = \{2,7\}$ and $\eta(z) = \{3,7,11\}$. Then we have:

$$[\![(x+y)*z]\!]^{\mathfrak{B}}_{\eta} = \{3,7\}.$$

In the first case, the value of the term is a non-negative integer, whereas in the second case the value of the term is a set of non-negative integers. If we take the homomorphism α_7 from Example 2.4.4, then $\zeta = \alpha_7 \circ \eta$ and $\alpha_7(\{3,7\}) = 1$.

Definition 2.4.14 (Formula). Let Σ be a signature. The formulas of Σ are generated as follows:

1. If s and t are terms of Σ, then $s = t$ is a formula.
2. If φ is a formula, then $\neg\varphi$ is a formula.
3. If φ and ψ are formulas, then $(\varphi\land\psi)$, $(\varphi\lor\psi)$ and $(\varphi\to\psi)$ are formulas.
4. If φ is a formula and x a variable, then $(\forall x\, \varphi)$ and $(\exists x\, \varphi)$ are formulas.

The logical connectives and quantifiers have the standard meaning:

symbol	name	meaning
$\neg$	negation	not
$\land$	conjunction	and
$\lor$	disjunction	or (inclusive)
$\to$	implication	if-then
$\forall$	universal quantification	for all
$\exists$	existential quantification	there is

The equivalence $\varphi \leftrightarrow \psi$ is defined by $(\varphi \to \psi) \land (\psi \to \varphi)$. A formula $s = t$ is called an *equation*. The expression $s \neq t$ is an abbreviation for the formula $\neg(s = t)$. In order to increase the readability of formulas parentheses are often omitted. For example, the following conventions are used:

Table 2.1 The semantics of formulas

$$\llbracket s = t \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket s \rrbracket_\zeta^{\mathfrak{A}} = \llbracket t \rrbracket_\zeta^{\mathfrak{A}}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \neg\varphi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{false}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \varphi \wedge \psi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true} \text{ and } \llbracket \psi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \varphi \vee \psi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true} \text{ or } \llbracket \psi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \varphi \to \psi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{false} \text{ or } \llbracket \psi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \forall x\, \varphi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket \varphi \rrbracket_{\zeta[x\mapsto a]}^{\mathfrak{A}} = \mathit{true} \text{ for every } a \in |\mathfrak{A}|; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \exists x\, \varphi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if there exists an } a \in |\mathfrak{A}| \text{ with } \llbracket \varphi \rrbracket_{\zeta[x\mapsto a]}^{\mathfrak{A}} = \mathit{true}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$\varphi \wedge \psi \wedge \chi$	stands for	$((\varphi \wedge \psi) \wedge \chi)$,
$\varphi \vee \psi \vee \chi$	stands for	$((\varphi \vee \psi) \vee \chi)$,
$\varphi \wedge \psi \to \chi$	stands for	$((\varphi \wedge \psi) \to \chi)$, etc.

The variable x is *bound* by the quantifier $\forall$ ($\exists$ resp.) in $\forall x\, \varphi$ ($\exists x\, \varphi$ resp.) and the *scope* of x is the formula φ. A variable x occurs *free* in a formula, if it is not in the scope of a quantifier $\forall x$ or $\exists x$. A formula is called a *sentence*, if it does not contain free variables. By $\varphi\frac{t}{x}$ we denote the result of replacing all free occurrences of the variable x in φ by the term t. Thereby, bound variables of φ are renamed if necessary.

Formulas can be interpreted in a state with respect to a variable assignment. Formulas are either true or false in a state and are therefore also called Boolean-valued expressions. The truth value of a formula in a state is computed recursively. The classical truth tables for the logical connectives and the classical interpretation of quantifiers are used. The equality sign is interpreted as identity.

Definition 2.4.15 (Interpretation of formulas). Let $\mathfrak{A}$ be a state of Σ, φ be a formula of Σ and ζ be a variable assignment in $\mathfrak{A}$ such that all free variables of φ are defined in ζ. By induction on the length of φ, a truth value $\llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}}$ is defined in Table 2.1.

Definition 2.4.16 (Range of a formula). The range of a formula φ with respect to the variable x in a state $\mathfrak{A}$ under ζ is the set of all elements of $\mathfrak{A}$ that make the formula true:

$$\mathit{range}(x, \varphi, \mathfrak{A}, \zeta) = \{a \in |\mathfrak{A}| : \llbracket \varphi \rrbracket_{\zeta[x\mapsto a]}^{\mathfrak{A}} = \mathit{true}\}$$

The interpretation of a formula φ depends on the values of ζ on the free variables of φ only.

Lemma 2.4.7 (Coincidence). If ζ and η are two variable assignments for φ such that $\zeta(x) = \eta(x)$ for all free variables x of φ, then $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = [\![\varphi]\!]^{\mathfrak{A}}_{\eta}$.

If we substitute a term t for a variable x in a formula φ and interpret the resulting formula $\varphi\frac{t}{x}$ in a state $\mathfrak{A}$ with respect to a variable assignment ζ, then we obtain the same truth value as when we first evaluate the term t in $\mathfrak{A}$ with respect to ζ, re-define the variable x in the environment to the value of t and interpret the formula φ in $\mathfrak{A}$ with respect to the new environment.

Lemma 2.4.8 (Substitution). Let t be a term and $a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$.
Then $[\![\varphi\frac{t}{x}]\!]^{\mathfrak{A}}_{\zeta} = [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x\mapsto a]}$.

Isomorphic structures satisfy the same formulas. An isomorphism between two states preserves the truth value of formulas.

Lemma 2.4.9 (Isomorphism). Let α be an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$. Then $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = [\![\varphi]\!]^{\mathfrak{B}}_{\alpha\circ\zeta}$.

Formulas are often used to express properties of abstract states. They are used to express properties of functions of an abstract state. They are also used to express invariants in runs of ASMs.

Definition 2.4.17 (Model). We say that a state $\mathfrak{A}$ is a *model* of φ (written $\mathfrak{A} \models \varphi$), if $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{true}$ for all variable assignments ζ for φ.

Example 2.4.7. The states $\mathfrak{A}$ and $\mathfrak{B}$ of Example 2.4.2 are models of the following equations:

$$\begin{array}{ll}
(x+y)+z = x+(y+z), & (x*y)*z = x*(y*z),\\
x+y = y+x, & x*y = y*x,\\
x+(x*y) = x, & x*(x+y) = x,\\
x+(y*z) = (x+y)*(x+z), & x*(y+z) = (x*y)+(x*z),\\
x+(-x) = 1, & x*(-x) = 0.
\end{array}$$

These formulas are called axioms of a Boolean algebra.

2.4.3 Transition Rules and Runs of ASMs

In mathematics, states like Boolean algebras are static. They do not change over time. In computer science, states are dynamic. They evolve by being updated during computations. Updating abstract states means to change the interpretation of (some of) the functions of the underlying signature. The way ASMs update states is described by transition rules of the following form which define the syntax of ASM programs. The transition rules P, Q are syntactic expressions generated as follows:

1. *Skip Rule:* **skip**
   Meaning: Do nothing.
2. *Update Rule:* $f(s_1, \dots, s_n) := t$
   Syntactic condition: f is an n-ary dynamic function name of Σ
   Meaning: Update the value of f at $(s_1, \dots, s_n)$ to t.
3. *Block Rule:* P **par** Q
   Meaning: P and Q are executed in parallel.
4. *Conditional Rule:* **if** φ **then** P **else** Q
   Meaning: If φ is true, then execute P, otherwise execute Q.
5. *Let Rule:* **let** $x = t$ **in** P
   Meaning: Assign the value of t to x and then execute P.
6. *Forall Rule:* **forall** x **with** φ **do** P
   Meaning: Execute P in parallel for each x satisfying φ.
7. *Choose Rule:* **choose** x **with** φ **do** P
   Meaning: Choose an x satisfying φ and then execute P.
8. *Sequence Rule:* P **seq** Q
   Meaning: P and Q are executed sequentially, first P and then Q.
9. *Call Rule:* $r(t_1, \dots, t_n)$
   Meaning: Call transition rule r with parameters $t_1, \dots, t_n$.

The variables x in **let**, **forall** and **choose** are logical variables (also called *read only* variables). Their values cannot be updated by a transition rule and are not stored in the state but in a finite environment. The scope of x in **let** is the rule P (but not the term t), whereas the scope of x in **forall** and **choose** is the formula φ and the transition rule P.

$$\textbf{let } x = t\ \underbrace{\textbf{in } P}_{\text{scope of } x} \qquad \textbf{forall } x\ \underbrace{\textbf{with } \varphi \textbf{ do } P}_{\text{scope of } x} \qquad \textbf{choose } x\ \underbrace{\textbf{with } \varphi \textbf{ do } P}_{\text{scope of } x}$$

An occurrence of a variable x is free in a transition rule, if it is not in the scope of a **let** x, **forall** x or **choose** x.

Definition 2.4.18 (Rule declaration). A *rule declaration* for a rule name r of arity n is an expression

$$r(x_1, \dots, x_n) = P,$$

where P is a transition rule and the free variables of P are contained in the list $x_1, \dots, x_n$.

In a rule call $r(t_1, \dots, t_n)$ the variables x_i in the body P of the rule declaration are replaced by the parameters t_i. Since here we have no concept of global variables, the formal parameters of the head are the only freely occurring variables in the body of rule declarations.

Definition 2.4.19 (ASM). An *abstract state machine* M consists of a signature Σ (including a classification of functions according to Fig. 2.4), a set of initial states for Σ, a set of rule declarations, and a distinguished rule name of arity zero called the *main rule name* of the machine.

In a given state, a transition rule of an ASM produces for each variable assignment an update set. Since the rule can contain recursive calls to other rules, it is also possible that it has no semantics at all. The semantics of a transition rule is therefore defined by a calculus in Table 2.2.

Definition 2.4.20 (Semantics of transition rules). A transition rule P yields the update set U in state $\mathfrak{A}$ under a variable assignment ζ, iff yields$(P, \mathfrak{A}, \zeta, U)$ is derivable in the calculus in Table 2.2.

The calculus in Table 2.2 depends on the rule declarations of a given ASM M. If we want to emphasize the dependence on M we say that a transition rule P yields the update set U in state $\mathfrak{A}$ under ζ *with respect to* M.

If the rule declarations of M do not contain **choose**, then for each transition rule P there is at most one update set U such that yields$(P, \mathfrak{A}, \zeta, U)$ is derivable in Table 2.2. This unique update set is sometimes denoted by $[\![P]\!]^{\mathfrak{A}}_{\zeta}$, if it exists.

Rules are called *by name*. This means that in a call $r(t_1, \ldots, t_n)$ the variables $x_1, \ldots, x_n$ are replaced in the body P of the rule by the parameters $t_1, \ldots, t_n$. The parameters are not evaluated in the state where the rule is called but only later when it is used in the body (maybe in different states due to sequential compositions). Call-by-value evaluation of rule calls can be simulated as follows:

$$r(y_1, \ldots, y_n) = (\textbf{let } x_1 = y_1, \ldots, x_n = y_n \textbf{ in } P)$$

Then upon calling $r(t_1, \ldots, t_n)$ the parameters are evaluated in the same state.

For the practice of modeling it is useful to generalize the call-by-name semantics of rule calls to rule declarations with *location variables* and *rule variables*. In this extension of ASMs each formal parameter of a rule declaration is either a logical variable as above or a location variable or a rule variable. In the body of the rule declaration, a location variable l can be used in terms and also on the left-hand side of an update as in $l := t$. A rule variable can be used at places where transition rules are expected. In a rule call, a logical variable has to be replaced by a term; a location variable has to be replaced by a location term, which is a term that starts with a dynamic function name; a rule variable has to be replaced by a transition rule. After the substitution, the body must be a correct transition rule. Location variables are used in ASM macros like the Swap macro on p. 40. Rule variables are used in *rule schemes*.

Since there are no global variables in the rule declarations of an ASM, the semantics of a transition rule depends on the values of ζ on the free variables of the rule only. Hence, if a rule is closed (does not contain free variables), its semantics depends on the state only.

Lemma 2.4.10 (Coincidence). If $\zeta(x) = \eta(x)$ for all free variables x of a transition rule P and P yields U in $\mathfrak{A}$ under ζ, then P yields U in $\mathfrak{A}$ under η.

Table 2.2 Inductive definition of the semantics of ASM rules

$\dfrac{}{\text{yields}(\textbf{skip}, \mathfrak{A}, \zeta, \emptyset)}$	
$\dfrac{}{\text{yields}(f(s_1, \dots, s_n) := t, \mathfrak{A}, \zeta, \{(l, v)\})}$	where $l = (f, (\llbracket s_1 \rrbracket^{\mathfrak{A}}_{\zeta}, \dots, \llbracket s_n \rrbracket^{\mathfrak{A}}_{\zeta}))$ and $v = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta}$
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(P \ \textbf{par} \ Q, \mathfrak{A}, \zeta, U \cup V)}$	
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta, U)}{\text{yields}(\textbf{if} \ \varphi \ \textbf{then} \ P \ \textbf{else} \ Q, \mathfrak{A}, \zeta, U)}$	if $\llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta} = \mathit{true}$
$\dfrac{\text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(\textbf{if} \ \varphi \ \textbf{then} \ P \ \textbf{else} \ Q, \mathfrak{A}, \zeta, V)}$	if $\llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta} = \mathit{false}$
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{let} \ x = t \ \textbf{in} \ P, \mathfrak{A}, \zeta, U)}$	where $a = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta}$
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U_a) \quad \text{for each } a \in I}{\text{yields}(\textbf{forall} \ x \ \textbf{with} \ \varphi \ \textbf{do} \ P, \mathfrak{A}, \zeta, \bigcup_{a \in I} U_a)}$	where $I = \mathit{range}(x, \varphi, \mathfrak{A}, \zeta)$
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{choose} \ x \ \textbf{with} \ \varphi \ \textbf{do} \ P, \mathfrak{A}, \zeta, U)}$	if $a \in \mathit{range}(x, \varphi, \mathfrak{A}, \zeta)$
$\dfrac{}{\text{yields}(\textbf{choose} \ x \ \textbf{with} \ \varphi \ \textbf{do} \ P, \mathfrak{A}, \zeta, \emptyset)}$	if $\mathit{range}(x, \varphi, \mathfrak{A}, \zeta) = \emptyset$
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A} + U, \zeta, V)}{\text{yields}(P \ \textbf{seq} \ Q, \mathfrak{A}, \zeta, U \oplus V)}$	if U is consistent
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta, U)}{\text{yields}(P \ \textbf{seq} \ Q, \mathfrak{A}, \zeta, U)}$	if U is inconsistent
$\dfrac{\text{yields}(P\frac{t_1 \cdots t_n}{x_1 \cdots x_n}, \mathfrak{A}, \zeta, U)}{\text{yields}(r(t_1, \dots, t_n), \mathfrak{A}, \zeta, U)}$	where $r(x_1, \dots, x_n) = P$ is a rule declaration of M

If we substitute a static term t for a variable x in a transition rule P and execute the rule $P\frac{t}{x}$ in a state $\mathfrak{A}$ under a variable assignment ζ, then we obtain the same update set as when we first evaluate the term t in $\mathfrak{A}$ under ζ, re-define the variable x in the environment to the value of t and execute the rule P in $\mathfrak{A}$ under the new environment.

Lemma 2.4.11 (Substitution). Let t be a static term and $a = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta}$. Then the rule $P\frac{t}{x}$ yields the update set U in state $\mathfrak{A}$ under ζ iff P yields U in $\mathfrak{A}$ under $\zeta[x \mapsto a]$.

An ASM is not allowed to depend on the internal structure of the base set of a state. Everything on which the ASM depends must be explicit in the state and the functions of the state. If two states are isomorphic, then the update sets produced by a transition rule can be mapped to each other by the same isomorphism.

Lemma 2.4.12 (Isomorphism). If α is an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$ and P yields U in $\mathfrak{A}$ under ζ, then P yields $\alpha(U)$ in $\mathfrak{B}$ under $\alpha \circ \zeta$.

A move of an ASM consists of firing the updates produced by the main rule of the machine, if they do not clash. A move is a single computation step in the run of an ASM. Since the main rule of an ASM does not have parameters and there are no free global variables in the rule declarations of an ASM, the notion of a move does not depend on a variable assignment.

Definition 2.4.21 (Move of an ASM). We say that a machine M can make a move from state $\mathfrak{A}$ to $\mathfrak{B}$ (written $\mathfrak{A} \overset{M}{\Longrightarrow} \mathfrak{B}$), if the main rule of M yields a consistent update set U in state $\mathfrak{A}$ and $\mathfrak{B} = \mathfrak{A} + U$. The updates in U are called internal updates, to be distinguished from the possible updates of monitored or shared locations; $\mathfrak{B}$ is called the next internal state.

If α is an isomorphism from state $\mathfrak{A}$ to $\mathfrak{A}'$ and M can make a move from $\mathfrak{A}$ to $\mathfrak{B}$, then by Lemma 2.4.2 and Lemma 2.4.12 the machine M can also make a move from $\mathfrak{A}'$ to $\mathfrak{B}'$, where $\mathfrak{B}'$ is the isomorphic image of $\mathfrak{B}$ under α, i.e., the following diagram commutes:

$$\begin{tikzcd}
\mathfrak{A} \arrow[r, Rightarrow, "M"] \arrow[d, "\alpha"'] & \mathfrak{B} \arrow[d, "\alpha"] \\
\mathfrak{A}' \arrow[r, Rightarrow, "M"] & \mathfrak{B}'
\end{tikzcd}$$

A run of an ASM starts in an initial state of the machine. As long as the machine can make a move, the run proceeds, requiring only that the interspersed moves of the environment, namely to update monitored or shared locations, produce a consistent state for the next machine move.[52] If in a state the machine cannot produce a consistent update set or no update set at all (due to non-termination of a recursion), then the state is considered to be the last state in the run. Because of the non-determinism of the **choose** rule and of moves of the environment, an ASM can have several different runs starting in the same initial state.

Definition 2.4.22 (Run of an ASM). Let M be an ASM with signature Σ. A *run* of M is a finite or infinite sequence $\mathfrak{A}_0, \mathfrak{A}_1, \ldots$ of states for Σ such that $\mathfrak{A}_0$ is an initial state of M and for each n, either M can make a move from $\mathfrak{A}_n$ into the next internal state $\mathfrak{A}'_n$ and the environment produces a consistent set of external or shared updates U such that $\mathfrak{A}_{n+1} = \mathfrak{A}'_n + U$, or M cannot make a move in state $\mathfrak{A}_n$ and $\mathfrak{A}_n$ is the last state in the run.

[52] Unless otherwise stated, we avoid the commitment to particular scheduling or timing schemes. We stipulate only that after every move of an ASM which produces its *next internal state*, if there are some environment moves, then they must provide a stable *next state* where the subsequent ASM move can take place. This includes the possibility of data exchange via input/output between machine and environment.

Sometimes we distinguish *internal runs*, where after the initial state the environment makes no move, from runs with interspersed environment moves which are then also called *interactive runs*. In applications, the external updates in interactive runs are often further restricted by stating *constraints* which are required to hold in *all* (but not necessarily in the internal) states of the run. The constraints express preconditions on the behavior of the environment. Often the constraints are stated as first-order formulas that have to hold in every state.

Remark 2.4.1 (Choice functions). Sometimes one can replace the **choose** operator by monitored choice functions so that the choices are made by the environment and not by the system. For example, if we know that in each possible state of an ASM the formula

$$\forall x\, (\varphi(x) \to \exists y\, \psi(x, y))$$

is true, then we can replace the transition rule

```
**forall** $x$ **with** $\varphi(x)$ **do**
  **choose** $y$ **with** $\psi(x, y)$ **do** $R(x, y)$
```

by the simpler

```
**forall** $x$ **with** $\varphi(x)$ **do** $R(x, f(x))$
```

where f is a monitored function that has to satisfy, in each state, $\psi(x, f(x))$ for every x with $\varphi(x)$. In case we do not know that for each x an appropriate y can be chosen, we have to require that the function f satisfies

$$\forall x\, (\varphi(x) \land \exists y\, \psi(x, y) \to \psi(x, f(x))$$

and to insert an additional guard as follows:

```
**forall** $x$ **with** $\varphi(x)$ **do**
  **if** $\psi(x, f(x))$ **then** $R(x, f(x))$
```

In general, each occurrence of **choose** needs its own choice function that depends on the enclosing **forall** and **choose** variables.

2.4.4 The Reserve of ASMs

Algorithms often need to increase their working space. This is reflected in ASMs by an **import** construct operating on a possibly infinite reserve set. In this section we provide three rules for this construct which guarantee that parallel imports yield fresh (pairwise different) elements and that permutations of the reserve set do not change the semantics of ASM rules, i.e. that the semantics does not depend on which reserve elements are chosen by **import**.

The ASM reserve set is part of the base set. New elements are allocated using the construct

Table 2.3 The semantics of ASMs with a reserve

$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{import } x \textbf{ do } P, \mathfrak{A}, \zeta, V)}$	if $a \in \mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ and $V = U \cup \{((\mathit{Reserve}, a), \mathit{false})\}$
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(P \textbf{ par } Q, \mathfrak{A}, \zeta, U \cup V)}$	if $\mathit{Res}(\mathfrak{A}) \cap \mathit{El}(U) \cap \mathit{El}(V) \subseteq \mathit{ran}(\zeta)$
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U_a) \quad \text{for each } a \in I}{\text{yields}(\textbf{forall } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, \bigcup_{a \in I} U_a)}$	if $I = \mathit{range}(x, \varphi, \mathfrak{A}, \zeta)$ and for $a \neq b$ $\mathit{Res}(\mathfrak{A}) \cap \mathit{El}(U_a) \cap \mathit{El}(U_b) \subseteq \mathit{ran}(\zeta)$

import x **do** P

which means "choose an element x from the reserve, delete it from the reserve and execute P". The special notation

let $x = \mathit{new}(X)$ **in** P

is then a syntactic abbreviation for importing a new element, adding it to the subuniverse X and executing P, i.e.

```
import x do
  X(x) := true
  P
```

The reserve of a state is represented using a special unary, dynamic relation *Reserve* that cannot be updated directly by an ASM but will be updated automatically upon execution of each **import**. The reserve $\mathit{Res}(\mathfrak{A})$ of a state $\mathfrak{A}$ is the set of elements a of $\mathfrak{A}$ such that $\mathit{Reserve}(a) = \mathit{true}$ in $\mathfrak{A}$. The reserve elements of a state are not allowed to be in the domain and range of any basic function of the state (except for the special relation *Reserve*). This condition is expressed by the following definition, where the elements of a location $l = (f, (a_1, \ldots, a_n))$ are the arguments $a_1, \ldots, a_n$.

Definition 2.4.23 (Reserve condition). We say that a state $\mathfrak{A}$ satisfies the *reserve condition* with respect to an environment ζ, if the following two conditions hold for each element $a \in \mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$:

R1. The element a is not the content of a location of $\mathfrak{A}$.
R2. If a is an element of a location l of $\mathfrak{A}$ which is not a location for *Reserve*, then the content of l in $\mathfrak{A}$ is *undef*.

The new rules for **import** are listed in Table 2.3. Rather than using *actions* (equivalence classes of update sets modulo permutations of the reserve, see [248]), we add additional constraints to the rules for **par** and **forall**. By $\mathit{El}(U)$ we denote the set of elements that occur in the updates of an update set U. The elements of an update (l, v) are the value v and the elements of the location l.

When a new element is imported by **import** we require that it is an element of the reserve of $\mathfrak{A}$ but does not occur in the range of the variable assignment ζ which possibly contains other new elements that are imported in the same step. The variable is bound to the new element and the body of the **import** rule is executed in the new environment. The function *Reserve* is updated at the new element to *false* (by the system and not directly by the ASM).

The constraint for P **par** Q says that the reserve elements of the update set U for P are disjoint from the reserve elements of the update set V for Q except for reserve elements in the range of ζ. We have to exclude the range of ζ, since otherwise the execution of a rule like

```
**import** $x$ **do** $(f(x) := 0$ **par** $g(x) := 0)$
```

would be impossible.

In order that the constraints in Table 2.3 work correctly, we have to require that in the scope of a bound variable the same variable is not used again as a bound variable (in a **let**, **forall**, **choose**, or **import**). Otherwise, it could happen that a reserve element that is imported and bound to a variable is hidden and then imported again as in the following example:

```
import $x$ do
  $f(x) := 0$ par
  let $x = 1$ in
    import $y$ do $f(y) := x$
```

The same reserve element could be used for x as well as for y, since the outermost x is hidden by the **let** and not visible in the environment, when y is imported.

The range of a formula has to be redefined such that reserve elements which are not in the range of the environment are excluded in the **forall** rule and cannot be chosen in the **choose** rule (cf. Def. 2.4.16).

Definition 2.4.24 (Range of a formula). The range of a formula φ with distinguished variable x in a state $\mathfrak{A}$ under ζ is the set of all elements of $\mathfrak{A}$ that are not in $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ and make the formula true:

$$\mathit{range}(x, \varphi, \mathfrak{A}, \zeta) = \{a \in |\mathfrak{A}| : a \notin \mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta),\ [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x \mapsto a]} = \mathit{true}\}$$

The elements of $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ have to be excluded, since otherwise the transition rule

```
forall $x$ with $x = x$ do $f(x) := 1$
```

would fully exhaust the reserve in one step and destroy the reserve condition.

If the constraints in Table 2.3 are enforced and a transition rule yields a consistent update set in a state that satisfies the reserve condition, then the state also satisfies the reserve condition after firing the updates. Hence, the reserve condition is preserved in runs of ASMs.

Lemma 2.4.13 (Preservation of the reserve condition). If a state $\mathfrak{A}$ satisfies the reserve condition wrt. ζ and P yields a consistent update set U in $\mathfrak{A}$ under ζ, then

1. for every element a of U which is in in the reserve of $\mathfrak{A}$ but not in the range of ζ, the update $((\mathit{Reserve}, a), \mathit{false})$ is in U,
2. the sequel $\mathfrak{A} + U$ satisfies the reserve condition wrt. ζ.

Proof. We first show that statement 1 implies statement 2. Let l be a location and v its content in $\mathfrak{A} + U$. Then either the update (l, v) is in U or v is the content of l in $\mathfrak{A}$. We have to show that the reserve conditions R1 and R2 of Def. 2.4.23 are satisfied in $\mathfrak{A} + U$. Let a be in $\mathit{Res}(\mathfrak{A} + U) \setminus \mathit{ran}(\zeta)$. Then a is also in $\mathit{Res}(\mathfrak{A})$.

Case 1. $(l, v) \in U$: Then a cannot be an element of (l, v), since otherwise, by 1, the update $((\mathit{Reserve}, a), \mathit{false})$ would be in U and a could not be in the reserve of $\mathfrak{A} + U$.

Case 2. $\mathfrak{A}(l) = v$: Since $\mathfrak{A}$ satisfies the reserve condition wrt. ζ, a is different from v and, if a is an element of l and l not a location for *Reserve*, then v is *undef*.

Now we prove statement 1 by induction on the definition of "yields". We can assume that the bound variables of P are not in the domain of the environment ζ. We consider the critical cases:

Case **let**: Assume that $a = [\![t]\!]_\zeta^\mathfrak{A}$ and

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{let } x = t \textbf{ in } P, \mathfrak{A}, \zeta, U)}$$

Since $\mathfrak{A}$ satisfies the reserve consider wrt. ζ and x is not in the domain of ζ, the state $\mathfrak{A}$ satisfies the reserve condition also with respect to the extended environment $\zeta[x \mapsto a]$. Let b be an element of U which is in $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$. Then b must be different from a, since the value of the term t in $\mathfrak{A}$ under ζ is the content of a location of $\mathfrak{A}$ or an element in the range of ζ. Hence b is in $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta[x \mapsto a])$ and, by the induction hypothesis, the update $((\mathit{Reserve}, b), \mathit{false})$ is in U.

Case **import**: Assume that $a \in \mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ and

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{import } x \textbf{ do } P, \mathfrak{A}, \zeta, U \cup \{((\mathit{Reserve}, a), \mathit{false})\})}$$

Let b be an element of U which is in $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$. If $b = a$, this case is proven, since the update $((\mathit{Reserve}, a), \mathit{false})$ is automatically added. Otherwise, b is in $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta[x \mapsto a])$ and we can apply the induction hypothesis.

Case **seq**: Assume that U is consistent and

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U) \qquad \text{yields}(Q, \mathfrak{A} + U, \zeta, V)}{\text{yields}(P \textbf{ seq } Q, \mathfrak{A}, \zeta, U \oplus V)}$$

By the induction hypothesis for P and since statement 1 implies statement 2, the state $\mathfrak{A} + U$ satisfies the reserve condition wrt. ζ. Let a be an element of $U \oplus V$ which is in $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$. If a is an element of an update of U, then by the induction hypothesis for P, the update $((\mathit{Reserve}, a), \mathit{false})$ is in U and hence also in $U \oplus V$. Otherwise, a is an element of an update of V. If a is not in the reserve von $\mathfrak{A} + U$, then a must have been deleted from the reserve of $\mathfrak{A}$ by an update $((\mathit{Reserve}, a), \mathit{false})$ of U. If a is in $\mathit{Res}(\mathfrak{A} + U)$, then by the induction hypothesis for Q, the update $((\mathit{Reserve}, a), \mathit{false})$ is in V and hence also in $U \oplus V$. □

If a state satisfies the reserve condition with respect to a variable environment, then a permutation of the elements of the reserve that are not in the range of the environment can always be extended to an automorphism of the abstract state which is the identity on the non-reserve elements and the elements in the range of the environment. Hence, the Isomorphism Lemma 2.4.12, which is also true for ASMs with a reserve, can be used to rename the reserve elements using appropriate permutations.

Lemma 2.4.14 (Permutation of the reserve). Let $\mathfrak{A}$ be a state that satisfies the reserve condition wrt. ζ. If α is a function from $|\mathfrak{A}|$ to $|\mathfrak{A}|$ that permutes the elements in $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ and is the identity on non-reserve elements of $\mathfrak{A}$ and on elements in the range of ζ, then α is an isomorphism from $\mathfrak{A}$ to $\mathfrak{A}$.

Proof. Let l be a location of $\mathfrak{A}$ with content v. If l does not contain elements of $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$, then $\alpha(l) = l$. By R1 of Def. 2.4.23, v is not in $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ and therefore $\alpha(v) = v$.

If l contains an element of $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$, then so also does $\alpha(l)$. If l is not a location for *Reserve* then, by R2 of Def. 2.4.23, we obtain that $v = \mathit{undef}$. By R1 of Def. 2.4.23, *undef* is not in the reserve of $\mathfrak{A}$. Therefore, α maps *undef* to *undef* and the content of $\alpha(l)$ is $\alpha(v)$. If l is a location for *Reserve*, then $v = \mathit{true}$ and, since α maps *true* to *true*, the content of $\alpha(l)$ is $\alpha(v)$. □

The update set computed by a transition rule in a state is unique modulo permutations of the reserve (if no **choose** is used). This is the mathematical justification for the constraints in Table 2.3. The lemma is true only for ASMs that produce finite update sets (see Exercise 2.4.15).

Lemma 2.4.15 (Independence of the choice of reserve elements). Let P be a rule of an ASM without **choose**. If $\mathfrak{A}$ satisfies the reserve condition wrt. ζ and P yields two finite update sets U and U' in $\mathfrak{A}$ under ζ, then there exists a permutation α of $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ such that $\alpha(U) = U'$.

Proof. By induction on the definition of “yields”. We assume that the bound variables of P are not in the domain of ζ. We consider the critical cases.

Case **import**: Assume that $a, a' \in \mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ and

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{import } x \textbf{ do } P, \mathfrak{A}, \zeta, U \cup \{((\mathit{Reserve}, a), \mathit{false})\})}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a'], U')}{\text{yields}(\textbf{import } x \textbf{ do } P, \mathfrak{A}, \zeta, U' \cup \{((\mathit{Reserve}, a'), \mathit{false})\})}$$

Let α be the permutation of $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ that transposes a and a'.

Since α is an automorphism of $\mathfrak{A}$ and $\alpha \circ (\zeta[x \mapsto a]) = \zeta[x \mapsto a']$, we can apply Lemma 2.4.12 and obtain yields$(P, \mathfrak{A}, \zeta[x \mapsto a'], \alpha(U))$.

Since x is not defined in ζ, we have $\mathit{ran}(\zeta[x \mapsto a']) = \mathit{ran}(\zeta) \cup \{a'\}$ and therefore $\mathfrak{A}$ satisfies the reserve condition wrt. $\zeta[x \mapsto a']$. By the induction hypothesis, there exists a permutation β of $\mathit{Res}(\mathfrak{A}) \setminus (\mathit{ran}(\zeta) \cup \{a'\})$ such that $\beta(\alpha(U)) = U'$. The composition $\beta \circ \alpha$ is a permutation of $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$. Since $\beta(\alpha(a)) = a'$, the function $\beta \circ \alpha$ maps the update $((\mathit{Reserve}, a), \mathit{false})$ to $((\mathit{Reserve}, a'), \mathit{false})$.

Case **par**: Assume that

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(P \textbf{ par } Q, \mathfrak{A}, \zeta, U \cup V)}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U') \quad \text{yields}(Q, \mathfrak{A}, \zeta, V')}{\text{yields}(P \textbf{ par } Q, \mathfrak{A}, \zeta, U' \cup V')}$$

and $\mathit{Res}(\mathfrak{A}) \cap \mathit{El}(U) \cap \mathit{El}(V) \subseteq \mathit{ran}(\zeta)$ and $\mathit{Res}(\mathfrak{A}) \cap \mathit{El}(U') \cap \mathit{El}(V') \subseteq \mathit{ran}(\zeta)$.

By the induction hypothesis there exist permutations α and β of the set $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ such that $\alpha(U) = U'$ and $\beta(V) = V'$.

Let γ be the restriction of $\alpha \cup \beta$ to $\mathit{El}(U) \cup \mathit{El}(V)$. The function γ is well-defined. Let a be an element in $\mathit{El}(U) \cap \mathit{El}(V)$. If a is not in $\mathit{Res}(\mathfrak{A})$, then $\alpha(a) = a = \beta(a)$. If a is in $\mathit{Res}(\mathfrak{A})$, then $a \in \mathit{ran}(\zeta)$ and $\alpha(a) = a = \beta(a)$.

The function γ is injective. Assume that $\alpha(a) = \beta(b)$, where $a \in \mathit{El}(U)$ and $b \in \mathit{El}(V)$. Let $c = \alpha(a)$. If c is not in $\mathit{Res}(\mathfrak{A})$, then $c = \alpha(c)$ and therefore $a = \alpha(a) = c = \beta(b) = b$. If c is in $\mathit{Res}(\mathfrak{A})$, then $c \in \mathit{Res}(\mathfrak{A}) \cap \mathit{El}(U') \cap \mathit{El}(V')$ and therefore $c \in \mathit{ran}(\zeta)$ and $c = \alpha(c)$. As in the previous case, we can conclude that $a = b$.

Since $\mathit{El}(U) \cup \mathit{El}(V)$ is finite, there exists a permutation δ of $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ such that $\delta(U) = \gamma(U) = \alpha(U) = U'$, $\delta(V) = \gamma(V) = \beta(V) = V'$ and thus $\delta(U \cup V) = U' \cup V'$.

Case **seq**: Assume that U and U' are consistent and

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A} + U, \zeta, V)}{\text{yields}(P \textbf{ seq } Q, \mathfrak{A}, \zeta, U \oplus V)}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U') \quad \text{yields}(Q, \mathfrak{A} + U', \zeta, V')}{\text{yields}(P \textbf{ seq } Q, \mathfrak{A}, \zeta, U' \oplus V')}$$

By the induction hypothesis there exists a permutation α of $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$ such that $\alpha(U) = U'$. Since α is an automorphism of $\mathfrak{A}$ (Lemma 2.4.14), it is an isomorphism of $\mathfrak{A} + U$ to $\mathfrak{A} + U'$ (Lemma 2.4.2). Since $\alpha \circ \zeta = \zeta$, by Lemma 2.4.12, it follows that yields$(Q, \mathfrak{A} + U', \zeta, \alpha(V))$.

By Lemma 2.4.13, it follows that $\mathfrak{A} + U'$ satisfies the reserve condition wrt. ζ and $\mathit{Res}(\mathfrak{A} + U') \setminus \mathit{ran}(\zeta)$ is contained in $\mathit{Res}(\mathfrak{A}) \setminus \mathit{El}(U')$.

By the induction hypothesis, there exists a permutation β of $\mathit{Res}(\mathfrak{A} + U') \setminus \mathit{ran}(\zeta)$ such that $\beta(\alpha(V)) = V'$.

Hence, $\beta \circ \alpha$ is a permutation of $\mathit{Res}(\mathfrak{A}) \setminus \mathit{ran}(\zeta)$. Since $\beta(U') = U'$, it follows that $(\beta \circ \alpha)(U \oplus V) = U' \oplus V'$. □

Remark 2.4.2 (Syntax of ASMs). Table 2.4 contains some variations of the syntax of ASMs that appear in the literature. If closing keywords are omitted, then the corresponding opening keyword extends as much as possible, indicated by indentation. In a parallel block, the keywords **do in-parallel** and **enddo** are often omitted. In a sequential block, however, the keyword **step** is often inserted before each transition rule. In writing ASMs, parallel blocks (no keywords) are treated differently from sequential blocks (additional keywords) because they are used more often. More often than the **extend** construct we use the variation **let** $x = \mathit{new}(D)$ **in** P, where the body P is executed in parallel with placing the fresh element x into the set D, as defined on p. 77.

2.4.5 Exercises

Exercise 2.4.1. (⇝ CD) Prove Lemma 2.4.1 (Difference of states).

Exercise 2.4.2. (⇝ CD) Let $\mathfrak{A}$ and $\mathfrak{B}$ be two states with the same superuniverse and the same signature. Assume that α is an isomorphism from $\mathfrak{A}$ to $\mathfrak{A}'$ and also an isomorphism from $\mathfrak{B}$ to $\mathfrak{B}'$. Show that $\alpha(\mathfrak{B} - \mathfrak{A}) = \mathfrak{B}' - \mathfrak{A}'$.

Exercise 2.4.3. (⇝ CD) Prove Lemma 2.4.3 (Properties of $\oplus$).

Exercise 2.4.4. (⇝ CD) Which of the following equations are true?

1. $U \oplus (V \cup W) = (U \oplus V) \cup (U \oplus W)$
2. $(U \cup V) \oplus W = (U \oplus W) \cup (V \oplus W)$

Prove the equation or give a counter example.

Exercise 2.4.5. (⇝ CD) Prove Lemma 2.4.5 (Terms and homomorphisms).

Exercise 2.4.6. (⇝ CD) The set of free variables of a term is defined as follows:

Table 2.4 Variations of the syntax of ASMs

if φ **then**
P
else
Q
endif
	if φ **then** P **else** Q

[**do in-parallel**] P_1 $\vdots$ P_n [**enddo**]	P_1 **par** $\ldots$ **par** P_n
$\{P_1, \ldots, P_n\}$	P_1 **par** $\ldots$ **par** P_n

do forall $x\colon \varphi$
P
enddo
	forall x **with** φ **do** P

choose $x\colon \varphi$ P **endchoose**	**choose** x **with** φ **do** P
step P **step** Q	P **seq** Q
extend D **with** x P **endextend**	**import** x **do** $D(x) := \mathit{true}$ **seq** P

1. $\mathrm{FV}(x) = \{x\}$
2. $\mathrm{FV}(c) = \emptyset$
3. $\mathrm{FV}(f(t_1, \ldots, t_n)) = \mathrm{FV}(t_1) \cup \ldots \cup \mathrm{FV}(t_n)$

The set of free variables of a formula is defined as follows:

1. $\mathrm{FV}(s = t) = \mathrm{FV}(s) \cup \mathrm{FV}(t)$
2. $\mathrm{FV}(\neg\varphi) = \mathrm{FV}(\varphi)$
3. $\mathrm{FV}(\varphi \wedge \psi) = \mathrm{FV}(\varphi \vee \psi) = \mathrm{FV}(\varphi \rightarrow \psi) = \mathrm{FV}(\varphi) \cup \mathrm{FV}(\psi)$
4. $\mathrm{FV}(\forall x\, \varphi) = \mathrm{FV}(\exists x\, \varphi) = \mathrm{FV}(\varphi) \setminus \{x\}$

The set of free variables of a transition rule is defined as follows:

1. $\mathrm{FV}(\textbf{skip}) = \emptyset$
2. $\mathrm{FV}(f(t_1, \ldots, t_n) := s) = \mathrm{FV}(t_1) \cup \ldots \cup \mathrm{FV}(t_n) \cup \mathrm{FV}(s)$
3. $\mathrm{FV}(P \textbf{ par } Q) = \mathrm{FV}(P) \cup \mathrm{FV}(Q)$
4. $\mathrm{FV}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q) = \mathrm{FV}(\varphi) \cup \mathrm{FV}(P) \cup \mathrm{FV}(Q)$
5. $\mathrm{FV}(\textbf{let } x = t \textbf{ in } P) = \mathrm{FV}(t) \cup (\mathrm{FV}(P) \setminus \{x\})$
6. $\mathrm{FV}(\textbf{forall } x \textbf{ with } \varphi \textbf{ do } P) = (\mathrm{FV}(\varphi) \cup \mathrm{FV}(P)) \setminus \{x\}$

7. $\mathrm{FV}(\textbf{choose}\ x\ \textbf{with}\ \varphi\ \textbf{do}\ P) = (\mathrm{FV}(\varphi) \cup \mathrm{FV}(P)) \setminus \{x\}$
8. $\mathrm{FV}(P\ \textbf{seq}\ Q) = \mathrm{FV}(P) \cup \mathrm{FV}(Q)$
9. $\mathrm{FV}(r(t_1, \dots, t_n)) = \mathrm{FV}(t_1) \cup \ldots \cup \mathrm{FV}(t_n)$
10. $\mathrm{FV}(\textbf{import}\ x\ \textbf{do}\ P) = \mathrm{FV}(P) \setminus \{x\}$

Let $\mathfrak{A}$ be a state. Prove the following coincidence properties:

1. If $\zeta(x) = \eta(x)$ for all $x \in \mathrm{FV}(t)$, then $[\![t]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\eta}$ (Lemma 2.4.4).
2. If $\zeta(x) = \eta(x)$ for all $x \in \mathrm{FV}(\varphi)$, then $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = [\![\varphi]\!]^{\mathfrak{A}}_{\eta}$ (Lemma 2.4.7).
3. If $\zeta(x) = \eta(x)$ for all $x \in \mathrm{FV}(P)$ and P yields U in $\mathfrak{A}$ under ζ, then P yields U in $\mathfrak{A}$ under η (Lemma 2.4.10).

Exercise 2.4.7. ($\leadsto$ CD) Prove the substitution Lemmas 2.4.6, 2.4.8, 2.4.11 for terms, formulas and transition rules.

Exercise 2.4.8. ($\leadsto$ CD) Show that the substitution Lemma 2.4.11 for transition rules is in general not true for non-static terms.

Hint: If a transition rule does contain **seq**, then different occurrences of the same term may be evaluated in different states.

Exercise 2.4.9. ($\leadsto$ CD) Prove the isomorphism Lemmas 2.4.9, 2.4.12 for formulas and transition rules.

Exercise 2.4.10. ($\leadsto$ CD) We say that two transition rules P and Q are *equivalent* (written $P \equiv Q$), if for each ASM M, for each state $\mathfrak{A}$ and each variable assignment ζ that assigns values to the free variables of P and Q the following holds: P yields U in $\mathfrak{A}$ under ζ wrt M if and only if Q yields U in $\mathfrak{A}$ under ζ wrt M. In other words, two transition rules are equivalent, if they produce the same update sets in all states.

Show that the following transition rules are equivalent:

1. $(P\ \textbf{par}\ \textbf{skip}) \equiv P$
2. $(P\ \textbf{par}\ Q) \equiv (Q\ \textbf{par}\ P)$
3. $((P\ \textbf{par}\ Q)\ \textbf{par}\ R) \equiv (P\ \textbf{par}\ (Q\ \textbf{par}\ R))$
4. $(P\ \textbf{par}\ P) \equiv P$ [if P is deterministic (without **choose**)]
5. $(\textbf{if}\ \varphi\ \textbf{then}\ P\ \textbf{else}\ Q)\ \textbf{par}\ R \equiv \textbf{if}\ \varphi\ \textbf{then}\ (P\ \textbf{par}\ R)\ \textbf{else}\ (Q\ \textbf{par}\ R)$
6. $(P\ \textbf{seq}\ \textbf{skip}) \equiv P$
7. $(\textbf{skip}\ \textbf{seq}\ P) \equiv P$
8. $((P\ \textbf{seq}\ Q)\ \textbf{seq}\ R) \equiv (P\ \textbf{seq}\ (Q\ \textbf{seq}\ R))$
9. $(\textbf{if}\ \varphi\ \textbf{then}\ P\ \textbf{else}\ Q)\ \textbf{seq}\ R \equiv \textbf{if}\ \varphi\ \textbf{then}\ (P\ \textbf{seq}\ R)\ \textbf{else}\ (Q\ \textbf{seq}\ R)$

Show that the following transition rules are in general not equivalent:

1. $((P\ \textbf{par}\ Q)\ \textbf{seq}\ R) \not\equiv ((P\ \textbf{seq}\ R)\ \textbf{par}\ (Q\ \textbf{seq}\ R))$
2. $(P\ \textbf{seq}\ (Q\ \textbf{par}\ R)) \not\equiv ((P\ \textbf{seq}\ Q)\ \textbf{par}\ (P\ \textbf{seq}\ R))$
3. $(\textbf{let}\ x = t\ \textbf{in}\ P) \not\equiv P\frac{t}{x}$

Exercise 2.4.11 (Substitutivity of equivalent rules). ($\rightsquigarrow$ CD) Let $R[P]$ be a transition rule with occurrences of P. By $R[Q]$ we denote the result of replacing the occurrences of P by Q. Prove the following substitutivity property: If $P \equiv Q$, then $R[P] \equiv R[Q]$.

Exercise 2.4.12. ($\rightsquigarrow$ CD) We say that two transition rules P and Q are *extensionally equal*, if for each ASM M, for each state $\mathfrak{A}$ and each variable assignment ζ that assigns values to the free variables of P and Q the following holds: if P yields a consistent update set U in $\mathfrak{A}$ under ζ wrt. M, then there exists a consistent update set V such that Q yields V in $\mathfrak{A}$ under ζ wrt. M and $\mathfrak{A} + U = \mathfrak{A} + V$ and *vice versa*. In other words, two transition rules are extensionally equal, if they have the same moves.

1. Give an example of two transition rules P and Q which are extensionally equal but not equivalent.
2. Show that Exercise 2.4.11 is not true for the extensional equality.

Hence, the equivalence notion between transition rules of Exercise 2.4.10 has better properties than the extensional equality.

Exercise 2.4.13. ($\rightsquigarrow$ CD) Prove the isomorphism Lemma 2.4.12 for ASMs with a reserve (Table 2.3).

Exercise 2.4.14. ($\rightsquigarrow$ CD) Add the remaining cases to the proof of Lemma 2.4.15 about the independence of the choice of reserve elements.

Exercise 2.4.15. ($\rightsquigarrow$ CD) Show that Lemma 2.4.15 about the independence of the choice of reserve elements is in general not true for ASMs that produce infinite update sets.

2.5 Notational Conventions

Throughout the book we stick to standard mathematical and programming terminology. For a quick reference we nevertheless list here some frequently used notation.

We freely mix mathematical and programming notations. For example we write both $f(x)$ and $x.f$ for the value of f at argument x. We also write both $f(x)(y)$ and $f(x, y)$ to denote the value of f for argument (x, y). The composition of two functions f and g is written as $f \circ g$ and is defined by $(f \circ g)(x) = f(g(x))$.

We also follow a common practice and often distinguish two functions with the same name by their signature (the types of their domains or ranges). Frequently we identify sets and predicates with their characteristic functions. For example we write $P(s) := \mathit{true}$ to express that s is added to the set $\{x \mid P(x)\}$. We often abbreviate $P(s) = \mathit{true}$ by $P(s)$ and $P(s) = \mathit{false}$ by $\neg P(s)$ or **not** $P(s)$.

X^* denotes the set of all sequences of elements of X. We use list and sequence as synonyms. $[a_1, \ldots, a_n]$ is the list containing the elements $a_1, \ldots, a_n$; $[\,]$ denotes the empty list; $\mathit{length}(l)$ returns the number of elements in list l.

We use Hilbert's choice operator ϵ and description operator ι, where $\epsilon x(P(x))$ denotes an object satisfying $P(x)$ and $\iota x(P(x))$ denotes the unique object satisfying $P(x)$. Both operators yield an undefined value if there is no such object respectively if the uniqueness is violated. We write $\mathit{Powerset}(a)$ for the set of all subsets of a.

For ASM-related concepts we write $U \setminus \mathit{Updates}(F)$ for the set of updates in U minus every update whose function name belongs to a function in F. We write also $\mathit{Updates}(f_1, \ldots, f_n)$ for $\mathit{Updates}(\{f_1, \ldots, f_n\})$. We write $U \restriction \mathit{Loc}$ for the restriction of U to locations in Loc. For the set of locations appearing in an update set we write $\mathit{Loc}(U)$:

$$U \restriction \mathit{Loc} = \{(l, v) \in U \mid l \in \mathit{Loc}\}, \quad \mathit{Loc}(U) = \{l \mid (l, v) \in U \text{ for some } v\}$$

The set of locations determined by a set T of ground terms in a state is denoted by $\mathit{Loc}(T)^{\mathfrak{A}}$:

$$\mathit{Loc}(T)^{\mathfrak{A}} = \{(f, ([\![t_1]\!]^{\mathfrak{A}}, \ldots, [\![t_n]\!]^{\mathfrak{A}})) \mid f(t_1, \ldots, t_n) \in T\}$$

We write $\mathit{Loc}(t)^{\mathfrak{A}}$ for $\mathit{Loc}(\{t\})^{\mathfrak{A}}$.

Sources and Historical Remarks

The *definition of (basic) ASMs* appeared in [245, 248]. The *ground model* and *refinement* methods were introduced into ASMs in [71, 72, 76]. The use of ASM models to develop practical, high-level design based *testing* methods, suggested in [86, p. 36],[87, p. 6] has been started in [121, 28, 208, 237]. Generalizing FSMs to *control state ASMs* appeared in [86], the extension of basic ASMs by sequencing and submachine operations in [134]. The *function classification* in [245] was extended in [80, 86] from where part of Sect. 2.1 is taken. The treatment of the *Reserve* set and the justification of the **import** construct are taken from [203] and simplify the definitions in [248]. See Chap. 9 for historical details.

3 Basic ASMs

In this chapter[1] we illustrate the ASM *ground model method* for reliable requirements capture, and the *refinement method* for crossing levels of abstraction to link ASM models through well-documented incremental development steps. Due to their introductory character, the examples in Sects. 3.1, 3.2 can be used as finger exercises. The reader who looks for more challenging refinements supporting design-driven verifications may prefer to switch directly to the database recovery and shortest path algorithms in Sect. 3.2 and to Sect. 3.3 where we use ASMs to stepwise refine and verify-on-the-fly a pipelined microprocessor out of its serial register transfer level ground model. Chapters 4, 5, 6 contain further case studies for the ground model and the refinement method. For more complex applications which do not fit the dimensions of a text book we refer the reader to the ASM models in the Java/JVM book [406] and to the literature which is surveyed in Chap. 9.

3.1 Requirements Capture by Ground Models

In this section we formulate six fundamental categories of questions to be used as guidelines for capturing requirements. We apply them to build ground model control state ASMs for some simple devices (ATM, Password Change, Telephone Exchange). We then illustrate by a command-line debugger control model the use of ground models in a reverse engineering context, where the ground model is the concrete one from which more abstract models are derived by *abstraction*, the reverse of refinement. Finally we exemplify the use of ground model ASMs for the development of accurate conceptual frameworks by defining some basic component model notions.

Real-life ground model ASMs are obviously larger than what can be presented here. Representative examples are the ground model ASMs for language standards (e.g. the ISO standard for PROLOG [101, 291, 131], the IEEE standard for VHDL'93 [111, 112], the ITU standard for SDL'2000 [292, 194]), for the semantics of major programming and design languages (in-

[1] Lecture slides can be found in `GroundModelsSimpleExls` (⇝ CD), `RefinemtMeth` (⇝ CD), `DbRecovery` (⇝ CD), `ShortestPath` (⇝ CD), `PipeliningRISC` (⇝ CD), `Debugger` (⇝ CD), `Backtracking` (⇝ CD), `ComponentModel` (⇝ CD).

cluding C [285], C++ [420], Java [406], Occam with its characteristic non-determinism and parallelism [105, 104],[2] Oberon [310], UML [98, 99, 153], some industrial domain-specific languages [385, 312]), for industry standards [224] and for industrial-size control systems (e.g.[121]), etc., as surveyed in Chap. 9. However, the problems encountered when building such models have been solved by applying the techniques and ideas explained through the examples in this book.

Problem 3 (Framework for architecture description languages). Define a unifying ASM model for architecture description languages, such that different instantiations of parameters and of macros reflect the differences between specific languages. Use the model to compare different concrete architecture description languages.

Problem 4 (Framework for security models). Define practical security models and relate them to the security models of current systems, e.g. as used for smart cards. As an example to start with, restrict the Java Virtual Machine model JVM in [406] to an appropriate submachine JVMC which executes Java Card instructions [155] and then put the execution of single instructions under an additional guard *FirewallCheck* which formalizes when a security exception has to be raised, to be dealt with by new rules for firewall-related methods in the Java Card API.

3.1.1 Fundamental Questions to be Asked

In this section we formulate six categories of questions we found useful as a practical guide for the formalization task leading from loosely formulated requirements to accurate, application-domain-oriented models. We motivate the questions through an analysis of a short list of requirements for invoicing orders, which served for a comparative investigation of specification methods to find out "What questions are prompted by one's particular method of specification?"[12, p. XIII].[3] Due to the vagueness and incompleteness of these requirements, which are characteristic of a certain class of informal requirement descriptions, their analysis brings out very clearly the need to ask and answer each of the six groups of questions. Although the questions are not specific to any system-description approach, building ground model ASMs naturally leads us to ask and answer all of them.

Here is the list of fundamental requirements capture questions.

1. Who are the system **agents** and what are their relations? In particular, what is the relation between the *system* and its *environment*?

[2] For Occam programs which present non-determinism and parallelism we define an interpreter in Sect. 6.5.1, adopting multi-agent async ASMs tailored to support the use of graphical UML diagrams.

[3] An Invoice machine is treated in more detail in `GroundModelsSimpleExls` (⇝ CD), data-refining ground model operations, see Exercise 3.1.8.

2. What are the system **states**?
   - What are the domains of objects and what are the functions, predicates and relations defined on them? This question is stressed by the object-oriented approach to system design.
   - What are the *static* and the *dynamic* parts (including input/output) of states?
3. How and by which **transitions** (actions) do system states evolve?
   - Under which conditions (*guards*) do the state transitions (actions) of single agents happen and what is their effect on the state?
   - What is supposed to happen if those conditions are not satisfied? Which forms of *erroneous use* are to be foreseen and which *exception handling* mechanisms should be installed to catch them? What are the desired *robustness* features?
   - How are the transitions of different agents related? How are the "internal" actions of agents related to "external" actions (of the environment)?
4. What is the **initialization** of the system and who provides it? Are there **termination** conditions and, if yes, how are they determined? What is the relation between initialization/termination and input/output?
5. Is the system description **complete and consistent**?
6. What are the system **assumptions** and what are the desired system **properties**?

We quote below from [12] (slightly rephrased) the problem description for invoicing orders ,which we will analyze guided by the above questions.

R0.1 The subject is to invoice orders.
R0.2 To invoice is to change the state of an order (to change it from the state *pending* to *invoiced*).
R0.3 On an order, we have one and only one reference to an ordered product of a certain quantity. The quantity can be different from other orders.
R0.4 The same reference can be ordered on several different orders.
R0.5 The state of the order will be changed to invoiced if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product.
Case 1:
R1.1 All the ordered references are in stock.
R1.2 The stock or the set of the orders may vary due to the entry of new orders or cancelled orders, or due to having a new entry of quantities of products in stock at the warehouse. But we do not have to take these entries into account.
R1.3 This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state.

Case 2:

> You do have to take into account the entry of new orders, cancellation of orders, and entries of quantities in the stock.

What do the requirements say about the state of the system? By R0.1 there is a set *ORDER* which is static in case 1 (R1.3) and dynamic in Case 2; no initialization and no bounds are specified. By R0.2 there is a dynamic function $state\colon ORDER \rightarrow \{pending, invoiced\}$. R0.1/2 seem to imply that initially $state(o) = pending$ for all orders o. By R0.3/5 there are two functions, both static in Case 1 (R1.3) and maybe dynamic in Case 2; their initialization and dynamics are unspecified. The function $product\colon ORDER \rightarrow PRODUCT$ represents the (or an?) ordered product in stock order. The function $orderQuantity\colon ORDER \rightarrow QUANTITY$ by R0.3/4 is not injective, not constant. By R0.5 there is a function $stockQuantity\colon PRODUCT \rightarrow QUANTITY$ which apparently is thought to be dynamic – a static interpretation is not reasonable – but nothing is specified when and by whom it should be updated.

What do the requirements say about the agents of the system and how the system evolves? By R0.1/2 there is only one transition. R0.5 does not mention the update of *stockQuantity*. It leaves open whether the invoicing is done simultaneously for all orders, or only for a subset of orders (with a synchronization for concurrent access of the same product by different orders?). In case the update is meant to be made for one order at a time it remains unspecified in which succession and with what successful termination or abruption mechanism this should be realized. R0.5 leaves open the time model (duration of invoicing) as well as questions concerning error, exceptions and robustness conditions.

Additionally for Case 2, the informal description does not specify the agents for dynamic manipulation of orders and stock products, how they interact for shared data (namely the elements of *ORDER* and the function $stockQuantity(Product)$), whether they act independently or following a schedule (interleaving?), whether the arrival and cancellation sequence of orders is to be reflected by invoicing, and whether also *Product* is dynamic, etc.

Modulo all those missing pieces of information, one can nevertheless reason upon possible rules for invoicing orders. In the static case, a single-order rule can be formalized as follows. Per step at most one order is invoiced, with an unspecified schedule (thus also not taking into account any arrival time of orders) and with an abstract deletion function (whose update in an asynchronous multiple-order environment may be defined to have cumulative effect, see the definition of *Total* in the rule AllOrNone below).

```
SingleOrder =
  choose $Order \in ORDER$ with $state(Order) = pending$ and
    $orderQuantity(Order) \leq stockQuantity(product(Order))$
  do
    $state(Order) := invoiced$
    $\textsc{DeleteStock}(orderQuantity(Order), product(Order))$
```

An "all-or-none" strategy, where simultaneously all orders for one product are invoiced (or none if the stock cannot satisfy the request), is expressed by the following rule. For variations see Exercise 3.1.1.

```
AllOrNone = choose $Product \in PRODUCT$
  let $Pending = \{o \mid state(o) = pending,\ product(o) = Product\}$
    $Total = \sum_{Order \in Pending} orderQuantity(Order)$
    if $Total \leq stockQuantity(Product)$ then
      forall $Order \in Pending$
        $state(Order) := invoiced$
      $\textsc{DeleteStock}(Total, Product)$
    else report "stock cannot satisfy all orders of chosen $Product$"
```

For the dynamic case the following rule formalizes invoicing en bloc all entering orders, using a (user-determined?) monitored function *in*. Similarly one can proceed for cancellation of orders and for entering new items into the stock (see Exercise 3.1.2).

```
IncomingOrders = if $in = (Prod_1\, Qty_1 \ldots Prod_n\, Qty_n)$ then
  forall $1 \leq i \leq n$ let $o = new(ORDER)$ in
    $orderQuantity(o) := Qty_i$
    $product(o) := Prod_i$
    $state(o) := pending$
```

The preceding discussion of even so small a set of so elementary requirements confirms the basic features that a satisfactory method of requirements capture should possess. Among them are a most general (application-oriented) notion of state, of simultaneous independent local transformations with global effect (as provided by the ASM construct **forall**), and of loose scheduling schemes (as provided by the ASM construct **choose**) – the ingredients of basic ASMs. A pragmatic naming principle, concerning high-level abstraction and suggested by the example, is to *use meaning-conveying names*, possibly chosen from the application domain, so that the comprehensibility of the system by the user is enhanced and that too early a consideration of problem-irrelevant representation issues is avoided. (Certainly one has to be aware of the danger that anthropomorphic terms may bring with them tacit assumptions one better avoids; but the concern about the danger of using anthropomorphic terms has been vastly exaggerated in the literature on formal methods.) Another principle is to avoid details which are irrelevant for the problem domain, such as detailed type or procedure declarations, or structuring of classes or modules, which belong to further design steps or to the implementation, not to the problem. Methodologically, ground model ASMs as a result of the elicitation of requirements are closer to the application domain than the nowadays fashionable UML requirements models, which from the very beginning are concerned about the class structure underlying the system to be developed – a structure one can certainly impose on a ground model ASM as additional refinement, as was hinted at above for the Lift example in Sect. 2.3. But following a clear separation of different concerns this should be done only when the analysis of an appropriate software architecture is started.

3.1.2 Illustration by Small Use Case Models

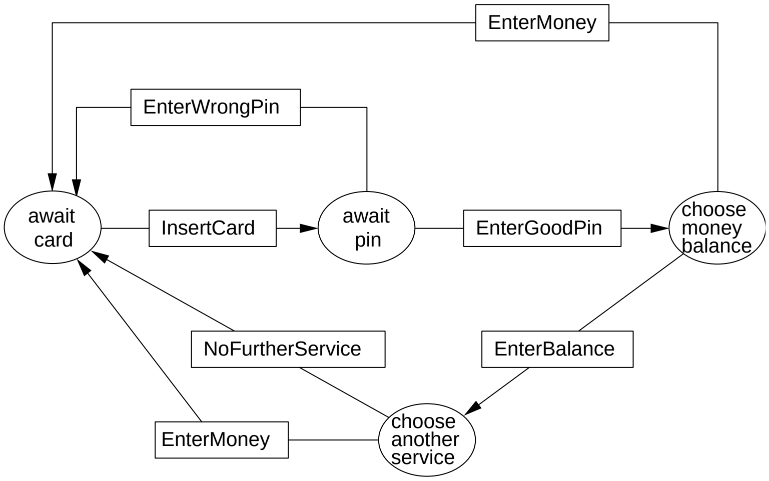
In this section we illustrate through simple examples some typical issues of requirements capture. We start with ground model ASMs for two elementary state control devices (ATM, Password Change) to explain the concept of *use case descriptions* of a system and their refinements. Use cases, often also called scenarios, can be seen as segments of abstract computations defining intended interactions between users and the system proper, and thus provide a way to piecemeal determine the system requirements. This includes the special case where the system is entirely defined by an interface offering a set of single one-step operations, see the Invoice Machine in Exercise 3.1.8. A telephone exchange example serves to illustrate the use of control state ASMs for specifications resembling message sequence charts, namely by a use case description of interactions of members of teams of agents. By a debugger control model we show how scenarios can also be exploited as *high-level system test cases*. The last example illustrates using a ground model ASM to precisely define basic concepts for a *conceptual framework* for language and platform independent component-based programming.

The refinement steps in this section lead from use case descriptions with symbolic ("stateless") operations to use case models with state-based operations, which are then extended incrementally (by adding new features) or by data refinements (increasing the degree of detailing for operations). For the ATM we exhibit a data refinement and for the Password Change a $(1, n)$-refinement.

ATM (Cash machine). Here is a typical software engineering textbook description of requirements for an ATM.

> Design the control for an ATM, and show it to be well functioning, where via a GUI the customer can perform the following operations:
>
> Op1. *Enter* the ID. Only one attempt is allowed per session; upon failure the card is withdrawn.
>
> Op2. Ask for the *balance* of the account. This operation is allowed only once and only before attempting to withdraw money.
>
> Op3. Withdraw *money* from the account. Only one attempt is allowed per session. A warning is issued if the amount required exceeds the balance of the account.

Fig. 3.1 UseCaseAtm model



Acc. The central system, which is supposed to be designed separately, receives the information about every withdrawal and updates the account balance correspondingly. The ATM becomes *inaccessible* for the customer for any other transaction until this update has become effective.

Ref. Extend the ATM to go out-of-service when not enough money is left.

Clearly this description views an ATM as offering customers specific sequences of operations, e.g. the sequence Op1, Op2, Op3, but not the sequence Op1, Op3, Op2. Some operation is missing in the list: apparently entering the ID requires having inserted a card. As a first modeling step we extract from the description the complete intended sequencing of user operations. The operations themselves remain symbolic, determined only by their name without describing the effect of their execution upon the underlying state. This leads to the control state ASM defined in Fig. 3.1, representing a compact use case description.

The next requirements elicitation step provides a meaning to the symbolic operations of the use case model, determining the effect of their execution upon the underlying state.[4] A monitored function *CurrIn* represents customer input with values belonging to abstract sets *ID*, *MONEY* etc. Another pair of monitored functions *CurrCard*, *inserted* serves to indicate the currently inserted card. To let the ATM communicate with the customer we use for simplicity an output function *Out* whose values abstractly represent

[4] It represents a data refinement step as defined in Sect. 3.2.

the appropriate messages (to be displayed on the screen; see below), the returned card and possibly the granted money. According to requirement Op1, the insertion of a card preceding entering an ID can then be formalized as follows, where *inserted* is viewed as a flag:

```
$\textsc{InsertCard} = \textbf{if } \mathit{inserted} \textbf{ then } \mathit{Out} := \mathit{EnterPinMsg}$
```

To model requirement Op1 one has to clarify the meaning of "failure". Presumably the ID is a pin number, required to be the pin number of the inserted card and known to the system by some static function *pin*. By the accessibility requirement Acc, access should be granted only if *account*(*CurrCard*) is *accessible*, where the shared function *accessible* indicates whether a previous customer ATM operation is still pending in the central system;[5] *account* is a static function. This interpretation of requirement Op1 is summarized by the following definitions (see the footnote to the proof of Lemma 3.1.1 concerning the withdrawal of the card in the case also of an inaccessible account):

```
$\textsc{EnterGoodPin} = \textbf{if } \mathit{CurrIn} = \mathit{pin}(\mathit{CurrCard}) \textbf{ and}$
  $\mathit{accessible}(\mathit{account}(\mathit{CurrCard}))$
$\textbf{then } \mathit{Out} := \mathit{ChooseServiceMsg}$

$\textsc{EnterWrongPin} = \textbf{if } \mathit{CurrIn} \in \mathit{ID} \textbf{ then}$
  $\textbf{if } \mathit{CurrIn} \neq \mathit{pin}(\mathit{CurrCard})$
    $\textbf{then } \mathit{Out} := \mathit{WrongPinMsg}$
  $\textbf{elseif } \neg \mathit{accessible}(\mathit{account}(\mathit{CurrCard}))$
    $\textbf{then } \mathit{Out} := \mathit{AccountNotAccessible}$
```

To model requirement Op3 an additional static function is needed to check that the required amount of money is *allowed* for the account of *CurrCard*. By requirement Acc one also has to reflect that the granted amount of money is deleted from the account by the central system. We formalize this by a deletion macro which is not specified further here, for which we assume the property invoked in the correctness proof below, namely that the execution of the macro has the effect of calling the appropriate deletion operation in the central system. We make sure that further access to the ATM is denied for the customer until the deletion has become effective, making the account again *accessible*. This results in the following rule. We leave the definition of similar rules EnterBalance and NoFurtherService as Exercise 3.1.3.

```
$\textsc{EnterMoney} = \textbf{if } \mathit{CurrIn} \in \mathit{MONEY} \textbf{ then}$
  $\textsc{ProcessMoneyRequest}(\mathit{CurrIn})$
```

[5] The requirements do not determine whether *accessible* is shared between the central system and a particular ATM or for a class of ATMs, but since this feature does influence the behavior of ATMs, it is part of the requirements elicitation to ask such a question and to document the answer for the contract.

```
ProcessMoneyRequest($In$) =
  if $allowed(In, CurrCard)$ then GrantMoney($In$)
  else $Out := \{NotAllowedMsg, CurrCard\}$

GrantMoney($In$) =
  $Out := \{ExitMsg, money(In), CurrCard\}$
  SubtractFrom($account(CurrCard), In$)
  $accessible(account(CurrCard)) := false$
```

For the ASM resulting from the above refinement of the use case ASM in Fig. 3.1 one can prove that it functions correctly with respect to the given requirements (Ref is discussed below). The operational character of the intuitive understanding of correct functioning is reflected by the notion of ASM run which underlies the following lemma.

Lemma 3.1.1 (ATM correctness). The use case ASM in Fig. 3.1 with operations labeling the arcs defined as above, satisfies the following properties requested by the requirements:

1. Per session only one attempt is possible to enter a correct pin number; upon failure the card is withdrawn.
2. Per session only one attempt is possible to withdraw money; a warning is issued if the amount required exceeds the balance.
3. Asking for the balance of the account is possible only once and only before attempting to withdraw money.
4. The central system receives the information about every withdrawal and updates the account balance correspondingly. Any further transaction is disallowed until the account has been updated.

Proof. One can proceed by an analysis of ASM runs. Property 1 follows from the definition of rule EnterWrongPin whereby the ATM is brought back to the *AwaitCard* control state, informing the customer and withholding the card.[6] Property 2 follows from the definition of the rule EnterMoney, which brings the ATM back to the *AwaitCard* state, giving back the card. Property 3 follows from the sequencing of operations in the control state diagram. Property 4 is guaranteed by the assumption for the deletion operation in rule EnterMoney and by setting there $accessible(account(CurrCard))$ to false so that the rule guards for entering a pin number prevent further transactions until the central system changes the accessibility predicate back to true. □

To satisfy the requirement Ref we now extend the ATM ASM by an out-of-service feature. What is meant by this requirement? Probably a new rule GoOutOfService should be added to bring the current control into

[6] Certainly the card withdrawal in the case of an inaccessible account specifies a rather severe security policy. Other decisions can be described by a more realistic WrongPinOrInaccessibleAccount refining EnterWrongPin.

an *OutOfService* state when there is not enough money left. This is easily accommodated by adding a new rule for a spontaneous transition of the ATM (say in its idle control state *AwaitCard*) when the amount of money left has reached a minimum, which can be predefined statically or updatable via service and maintenance:

```
GoOutOfService =
  if $MoneyLeft < min$ then $ctl\_state := OutOfService$
```

For correctness reasons, which the requirements engineer should point out to the customer, one should also prevent the machine from being executed when MoneyLeftBelowRequest(*CurrIn*), e.g. when $MoneyLeft - CurrIn < 0$.[7] This can be obtained by guarding the processing of money requests, using the conservative extension scheme already explained for adding exception handling to the LIFT ASM in Fig. 2.15. In this case we restrict the ASM for the "normal" ATM behavior by the negation of the new guard MoneyLeftBelowRequest(*CurrIn*) and add a new rule to inform the customer (and to call a bank-note supply service) in the other case. One then has to refine also the successful execution of money request processing, namely to update the *MoneyLeft* after a money request has been granted. This illustrates a frequent type of special refinement step consisting in replacing one operation by another one, which represents a $(1, 1)$-refinement step in Fig. 2.1. Using ASMs one can deal with "operations" which are defined by a step of an entire machine; in this case the operation to be refined is the macro GrantMoney(*In*), which is extended by the additional update $MoneyLeft := MoneyLeft - In$. This leads to the following refinement for entering money requests, where their processing is triggered.

```
EnterMoney = if $CurrIn \in MONEY$ then
  if not MoneyLeftBelowRequest($CurrIn$) then
    ProcessMoneyRequest($CurrIn$)
  else $Out := \{NotEnoughMoneyLeftMsg, CurrCard\}$
```

[7] As B. Thalheim commented in an e-mail of December 11 (2002) to E. Börger, this prevents an unsatisfactory behavior found in the ATM software of some German banks where a money withdrawal transaction is programmed as a sequence of three sub-transactions: customer validation (PIN control etc.), posting (account update) and money dispensing. As a result it can happen that the account update transaction is executed, debiting the customer's account for the *allowed* sum, even if the dispensing sub-transaction does not complete normally, for instance when the amount cannot be paid because the money is not physically available in the ATM. Apparently in such cases, the banks undo the debiting of the account when the customer complains (sic!). In the ground model the faulty situation springs up through inspection, by elementary application-driven reasoning, prior to any software testing. Furthermore a simple ground model analysis (see below) reveals one among numerous ways of how the program can handle the problem correctly – whereas apparently the banks judge it difficult, or doable only at prohibitive cost, to fix the incriminated code.

```
ProcessMoneyRequest($In$) =
  if $allowed(In, CurrCard)$ then
    GrantMoney($In$)
    $MoneyLeft := MoneyLeft - CurrIn$
  else $Out := \{NotAllowedMsg, CurrCard\}$
```

The atomic interpretation we have used up to now for the meaning of "money withdrawal" is subject to discussion. Namely one may think that entering the request for money withdrawal implies a further choice between standard amounts (say from a set *Std*) and a user-defined amount. We show how to accommodate such a further detailing of requirements by a $(1, n)$-refinement with $n > 1$, where one step of the abstract machine is replaced by a sequence of steps of the refined machine.

Determining the operation sequences which constitute the refined interpretation of EnterMoney leads to the use case description in Fig. 3.2 with the following interpretation of the rules labeling the arcs. We leave it to the reader as Exercise 3.1.4 to convince himself that the ATM Correctness Lemma 3.1.1 remains true for the refined ASM.

```
ChooseWithdrawal = if ($CurrIn = Withdrawal$) then $skip$
EnterStandardAmount = if $CurrIn \in Std$ then
  ProcessMoneyRequest($CurrIn$)
ChooseOtherAmount = if ($CurrIn = OtherAmount$) then $skip$
EnterAmount = if $CurrIn \in MONEY$ then
  ProcessMoneyRequest($CurrIn$)
```

The machine in Fig. 3.2 satisfies all the initial requirements – except for the request of GUI support for the customer operations. To determine the place of this non-functional requirement in the model a *data refinement* suffices, a step-by-step or $(1, 1)$-refinement[8] which for our example is *conceptually* pretty simple but offers a precise interface to determine in terms of appropriate quality criteria the properties expected from a professional screen layout implementation. In fact one can implement the current control state *ctl_state* as the current screen *CurrScreen* and define for each control state c the desired *screen* through which the customer inputs values for *CurrIn* and receives messages from the ATM.[9] Formally we define for every ATM control state c:

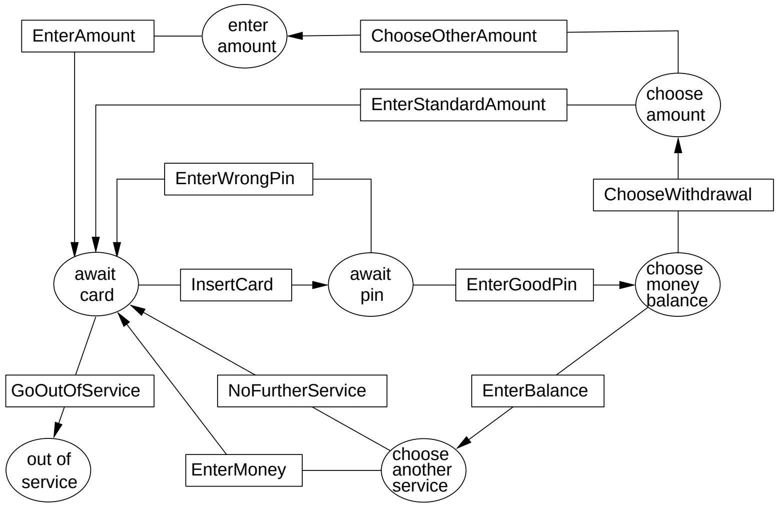
$ctl\_state = c$ is refined as $CurrScreen = screen(c)$

We leave it as exercise to show that the GUI refinement preserves the ATM Correctness Lemma 3.1.1 (Exercise 3.1.5), also under additional robustness constraints (Exercise 3.1.6).

[8] See the definition in Sect. 3.2.

[9] At the level of detail reached by this refinement step, the assumption about input to be consumed by firing rules is easily satisfied. This illustrates the advantage one can take in the high-level model to simply assume properties which one knows to be easily satisfiable at a more detailed level.

Fig. 3.2 RefinedAtm use case model



Problem 5 (Modeling business rules). Select an interesting set of business rules, complex enough to be worth building and analyzing a high level model.

Password change. This example serves to illustrate that for control state ASMs, $(1, n)$-refinements consisting of the replacement of a machine step by a sequence of n steps of another machine, where n may be fixed or variable, are conceptually supported by replacements of nodes by graphs in the underlying control state diagrams. We start with the problem formulation.

> Design a program, and show it to be well functioning, which allows a user to change his password using the following sequence of operations:
>
> Op1. Enter the user ID and the (current) password. Access should be refused if the ID or the password is incorrect.
>
> Op2. Enter twice the new password. The new password request should be rejected if the new password is syntactically incorrect or not correctly repeated. Otherwise the new password should be stored as the password valid from now on.
>
> R1. Refine the machine by a line editor for inputting passwords.
>
> R2. Refine the machine by restricting the number of attempts to define the new password.

The requirements suggest a clear sequencing of user operations, which is formalized by the control state ASM in Fig. 3.3.

In this case also the definition of the abstract operations is clear, formulated using a static function *IsKnownID* and a dynamic function *pwd*, for

Fig. 3.3 PasswordChange use case model

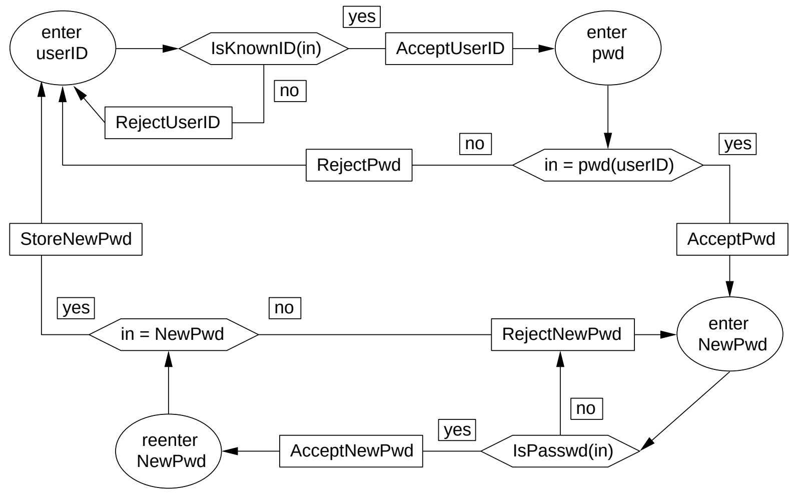
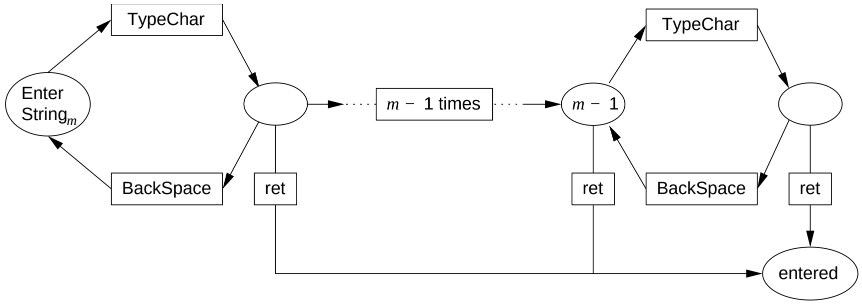


Fig. 3.4 Character inputting machine

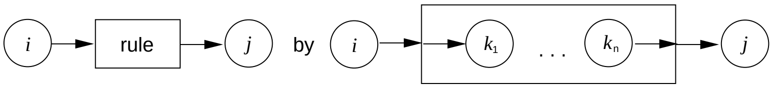


```
TypeChar =
  if $in \in CHAR$ then
    $str := str * in$
    $display := display * in$
```

```
BackSpace =
  if $in = backspace$ then
    $str := tail(str)$
    $display := tail(display)$
```

example AcceptUserID = $UserID := in$, AcceptNewPwd = $NewPwd := in$, etc. In every control state of the machine, the user has to provide a value for the input function in before it can be checked to represent a correct $UserID$ or password. If one wants to reflect the structure of these inputting operations, character typing submachines like EnterString(n, m) for entering a string of length $n \leq l \leq m$ (e.g. see Fig. 3.4 for EnterString$(1, m)$) can be placed in Fig. 3.3 between each control state and the following test rhomb.

Fig. 3.5 $(1, n)$-refinement of control state ASMs



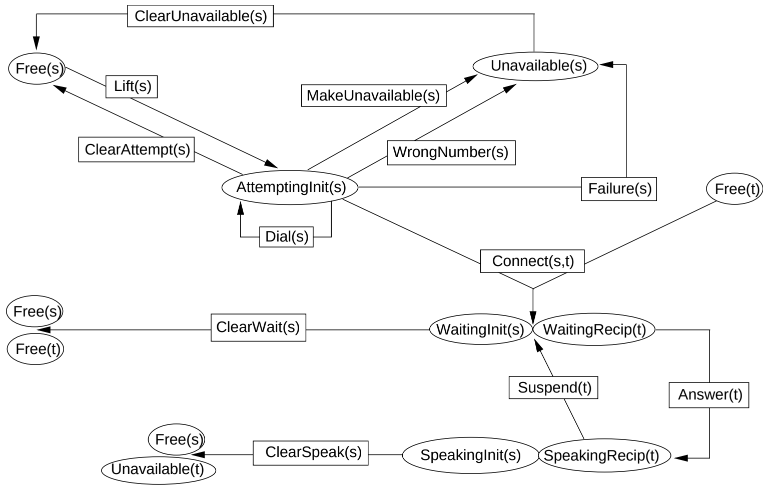
The form of these submachines can vary; the simplest type corresponds to a $(1, n)$-refinement diagram with fixed n, as depicted in Fig. 3.5. The number n is dynamically determined when the subprogram contains a loop, like in Fig. 3.4. What these submachines have in common is a unique start and exit state which makes it possible to "hang" them into a control state of another machine.

We leave it as Exercise 3.1.10 to refine the Password Change machine to satisfy R2 and to justify its correctness.

Telephone exchange. The following example shows how the successive interactions of members of teams of largely independent agents can be controlled by a basic ASM with parameterized rules, to be instantiated by appropriate agents (single agents or agents which stand for a team, i.e. a possibly ordered set of agents). In this way Message Sequence Chart specifications turn out to correspond to a special class of control state ASMs (see also the Session Initiation Protocol slide in `GroundModelsSimpleExls` (⇝ CD)). We define an ASM to control a network through which pairs of subscribers establish and clear phone conversations, according to the problem description (paraphrased) from [430].

> Subscribers at each moment are in one status out of the following:
> *Free*: neither engaged in, nor attempting, any phone conversation;
> *Unavailable*: due to a timeout or an unsuccessful attempt to call;
> *AttemptingInit*: attempting as initiator of a phone conversation to call somebody by dialing his number;
> *WaitingInit*: waiting as initiator for somebody to answer, after having been connected;
> *SpeakingInit*: speaking as initiator to the called subscriber;
> *WaitingRecip*: status of a recipient when his phone is ringing or when he has suspended an established conversation;
> *SpeakingRecip*: status of a recipient when he is speaking to the initiator of the phone conversation.
>
> The system and the subscribers can execute the following operations:
>
> – Free Subscribers can LIFT their handset, thus becoming attempting initiators.
> – In every initiator status, initiators (and only them) can CLEAR their call, thus becoming again free.
> – Initiators in attempting status can DIAL, trying to complete a subscriber's number.

Fig. 3.6 TelephoneExchange use case ASM



- The system can MakeUnavailable an initiator, due to a timeout, or due to an unsuccessful attempt to call because of Failure (when the called subscriber is not free) or WrongNumber.
- The system can Connect an initiator and a free recipient, making them both waiting (for the recipient to answer).
- A recipient can Answer and Suspend a phone call.

The description lends itself to a formalization by a control state ASM where the status of a subscriber is reflected by a control state, but it remains to beclarified how different subscribers and teams (pairs of connected subscribers) can be handled by one basic ASM. The fact that the requirements are described in an operation-oriented way provides a clue to a solution. We parameterize the rules of the ASM by single subscribers or pairs of subscribers so that in reality at each moment there can be many active instances of the machine. Thus the above *status requirements* are modeled by the sequencing of subscriber and system operations defined via the use case ASM in Fig. 3.6. Each subscriber s has its own control state $\mathit{ctl\_state}(s)$ and its own input $\mathit{in}(s)$. The predicate $\mathit{Free}(s)$, for example, is an abbreviation for $\mathit{ctl\_state}(s) = \mathit{Free}$. Control states of pairs of connected subscribers are depicted as pairs of their single control states.

To formalize the above *requirements for system and subscriber operations* we refine the symbolic operations of Fig. 3.6, using again a pure data refinement. Due to the control nature of the telephone exchange problem, most of the operations consist merely in the control state transition of the use case ASM, so that their defining rules have the form **if** *Cond* **then** *skip* and only the guard remains to be defined. This is the case for the three internal (input-free) transitions {MakeUnavailabe, WrongNumber, Failure}, but also for the following operations where the guard consists only in the subscriber's inputting the call of that operation:

```
if $in(s) \in \{ClearAttempt, ClearUnavailable, Answer, Suspend\}$
then $skip$
```

The guard for MakeUnavailable(s) is

$$Currtime - LastOpenTime(s) > MaxPause$$

with a dynamic function $LastOpenTime(s)$ which has to be initialized by Lift(s) and to be refreshed by each operation Dial(s), recording the value of the monitored increasing function $Currtime$ when s last called an operation. The guard for WrongNumber(s) is $IncorrectNr(DialedSofar(s))$ with a static predicate $IncorrectNr$ and a dynamic function $DialedSofar(s)$ which also has to be initialized at Lift(s) and to be refreshed at each operation Dial(s). The guard for Failure(s) can be formulated as the conjunction of $CorrectNr(DialedSofar(s))$ and $ctl\_state(subscriber(DialedSofar(s))) \neq Free$ with abstract static function $subscriber$.

The guard for Lift(s) is simply that this operation is called by the subscriber, and its updates initialize the two dynamic functions $DialedSofar$ and $LastOpenTime$. Analogously, Dial(s) also updates these two functions, but it is guarded by the condition that the number dialed so far can be completed. Certainly we assume that the domains where the predicates $CompletableNr$, $IncorrectNr$, $CorrectNr$ yield value $true$ are pairwise disjoint.

```
Lift($s$) = if $in(s) = Lift$ then
  $DialedSofar(s) := []$
  $LastOpenTime(s) := Currtime$
Dial($s$) = if $CompletableNr(DialedSofar(s))$ then
  $DialedSofar(s) := Append(in(s), DialedSofar(s))$
  $LastOpenTime(s) := Currtime$
```

The team-building operation Connect(s, t) establishes the initiator s of the phone conversation as the $caller$ of the subscriber t who has the dialed number.

```
Connect($s, t$) = if $t = subscriber(DialedSofar(s))$ then
  if $CorrectNr(DialedSofar(s))$ and $Free(t)$ then
    $caller(t) := s$
```

Correspondingly the ClearSpeak(s), ClearWait(s) operations, whose guard is simply that the operation is called by the first team member s, reset the caller of t to uncouple the team formed by Connect(s, t), e.g. by

```
let $t = (\iota r \; caller(r) = s)$ in $caller(t) := undef$.
```

Problem 6 (Internet telephony protocols). Model a real-life protocol for Internet telephony, e.g. the Session Initiation Protocol (SIP) which has the potential of becoming fundamental for an integrated use of XML-based web services through the Internet, see `http://www.cs.columbia.ed/~hgs/sip/` or `http://ietf.org/rfc/rfc3261.txt`. The model will consist in an asynchronous ASM (See Chap. 6) whose components are control state ASMs.

Command-line debugger. The debugger control state model serves to illustrate how scenarios can be formulated in terms of ground models and be exploited as *high-level system test cases*. The model is extracted from an industrial re-engineering case study reported in [26].[10]

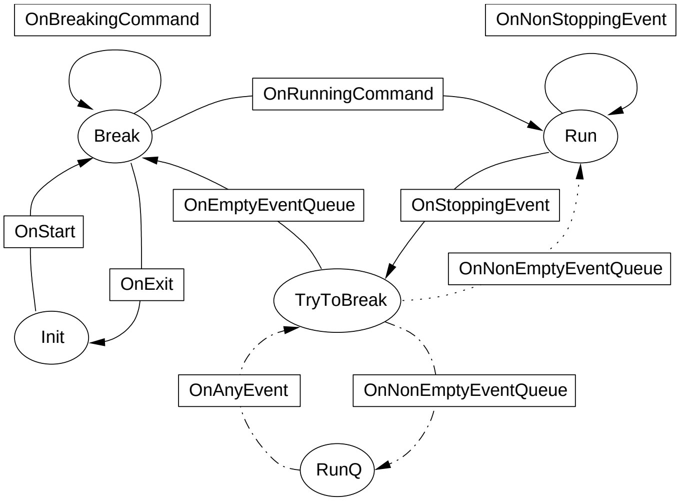
The goal of the case study was to reverse engineer a command line debugger (of 30 K lines of C++ code) which works in a stack-based COM execution environment. Three debugger models have been abstracted, forming a refinement hierarchy[11] which specifies the Application Programming Interfaces by which the debugger components interact with each other. The control model we explain here is a control state ASM which has been used for the simulation of various scenarios.

The role of the control state ASM is to model how the debugger control *dbgMode* moves between the user, the debugger and the run-time system. $dbgMode = Init$ means that the user is in control to start the debugger. $dbgMode = Break$ means that the user can emit commands to the debugger for exiting it, or for controlling the execution (e.g. to run a program, to step through managed code, setting breakpoints or stopping events, etc.), or for inspecting the program state (e.g. to view threads, stack frames, variables, events, etc.). When $dbgMode = Run$, the run-time system has the control and the debugger waits for a callback; eventually the run-time system will switch *dbgMode* to *TryToBreak* passing the control to the debugger, namely upon a callback event. Since the focus in this model is on sequencing the interactions between the three agents involved, most of the relevant state information is represented by functions which are monitored for some of the parties involved and implicitly set by others. For example the user clearly sets *command* and *dbgEvents* (indicating whether stopping upon certain events is turned *On* or *Off*, e.g., $dbgEvents(classLoadEvent) = Off$). The run time system sets *callback*, indicating for example that the executed process was exited, the required step was completed, the module to be observed was (un)loaded, a thread was created, a breakpoint was hit, etc. The debugger watches *callback* to know when to proceed, with the assumption that the run-time issues at most one event at a time, and only in mode *Run*, and then waits for an acknowledgment from the debugger. The run-time system also

[10] Case studies like [26, 121, 102, 97] represent examples where the ground model is the most concrete model, extracted for re-engineering purposes from existing code. For the sake of analysis such a detailed ground model typically is further abstracted into higher-level models before being re-implemented.

[11] Characteristic examples of these refinement steps are explained in Sect. 3.2.

Fig. 3.7 DEBUGGER control state ASM



sets the boolean function *eventQueueFlag* indicating whether there are events still to be handled before the control goes back to the user (by giving him the prompt). The result of this analysis is the control state ASM in Fig. 3.7 (disregarding the transitions to and from *RunQ* explained below).

Test experiments have been made with user scenarios for this debugger control model, one of them reported in an animation on slide number 13 in `Debugger` (⇝ CD) and reproducible also as Gedankenexperiment.[12] This exhibited an undesired behavior of the debugger and indicated a natural way to correct it in the control state machine, namely by leading transition ONNONEMPTYEVENTQUEUE not any more back to control state *Run*, but to a new control state *RunQ* from where a new transition leads back to *TryToBreak* so that *Break* mode remains reachable. Later this bug was found to have been fixed also in the C++ code, independently and at the same time, by the development team. An analogous experience is reported also in another industrial project [121] where ASMs were used for re-engineering C++ code.

Problem 7 (ASM synthesis from use cases). Develop a tool to transform use cases into executable ASMs whose runs realize the given use cases. See Harel's play-in/play-out approach in [270] which is based upon a temporal-logic extension [165] of message sequence charts and UML sequence diagrams for specifying system scenarios.

[12] Imagine upon hitting a breakpoint in the run-time system followed by switching the control to *TryToBreak*, there is an event in the queue so that the machine goes back to *Run*. Imagine that then a non-stopping class loading event happens. Then the control cannot go back to *Break* – although it should because a breakpoint was hit – since there is no direct transition from *Run* to *Break*.

Component model. We show here how to use ground model ASMs to define an abstract framework for the analysis of component composition issues. The material is extracted from [410]. The goal is to provide concepts for components which

- export and import parameterized services under usage constraints,
- come with interface specifications depending on views,
- can be composed and refined by connectors,

based upon abstract notions of component, connector, service, specification, view, and constraint (related to the underlying logic). We formulate signature and constraints for these notions and use them in a scheme for a dynamic consistency check of component structures. We illustrate connectors by *communicators* appearing in the *UPnP* networking architecture model with an ASM for routing messages [223]. The purpose of this section is to motivate some reader to work on the following problem. For a reverse engineering technique to build a real-life processor out of its formally specified basic architectural components see [102].

Problem 8 (Modeling middleware techniques). Develop a comprehensive ASM model for real-life multi-platform middleware techniques for component composition (e.g. Corba) or for the component concept of a real-life component based programming system (e.g. JavaBeans).

The structure of the component model provides for each of the above mentioned notions an abstract set equipped with related functions satisfying certain axioms. Functions *Exports*, *Imports*, *Constraint*, *Name* : yield for each component $c \in COMPONENT$ the sets of exported and imported services, the constraints which express allowed usages (e.g. an application ordering) of the services exported by a component, and its name.

$$\begin{aligned}
&Exports\colon COMPONENT \rightarrow Powerset(EXP\_SERVICE)\\
&Imports\colon COMPONENT \rightarrow Powerset(IMP\_SERVICE)\\
&Constraint\colon COMPONENT \rightarrow CONSTRAINT\\
&Name\colon COMPONENT \rightarrow STRING
\end{aligned}$$

Services can be exported or imported.

$$SERVICE = EXP\_SERVICE \cup IMP\_SERVICE$$

Each service is parameterized by a list of pairs of types and modes, has a result type, a unique component to which it belongs, and a name.

$$\begin{aligned}
&ServiceParam\colon SERVICE \rightarrow (TYPE \times MODE)^*\\
&MODE = \{in, out, inout\}\\
&ServiceResType\colon SERVICE \rightarrow TYPE\\
&Component\colon SERVICE \rightarrow COMPONENT\\
&Name\colon SERVICE \rightarrow STRING
\end{aligned}$$

Each exported service has an associated *use structure* determining its allowed usages in relation to other services which are imported in the body of the exporting component and may be used to execute the exported service. The set of these imported services in a given use structure is provided by a function *ContainedServices*. The use structure has to meet the constraints of the components from where those imported services are imported, a property which can be checked by a function.

$$\begin{aligned}&\mathit{ImportStructure}\colon \mathit{EXP\_SERVICE} \rightarrow \mathit{USE\_STRUCTURE}\\ &\mathit{ContainedServices}\colon \mathit{USE\_STRUCTURE} \rightarrow \mathit{Powerset}(\mathit{IMP\_SERVICE})\\ &\mathit{MeetsConstraint}\colon \mathit{USE\_STRUCTURE} \times \mathit{CONSTRAINT} \rightarrow \mathit{BOOL}\end{aligned}$$

The class *SPEC* of interface descriptions for services under given views is deliberately kept abstract, together with functions determining for each exported service the specification provided for it and for each imported service the specification required for it.[13] Therefore also the satisfaction relation is kept abstract which determines whether a given (expservice) specification satisfies a given (impservice) specification so that the service to be exported can safely be plugged into the component where to import it. Its instantiation depends also on the logic adopted to describe the intended semantical relations between specifications.

$$\begin{aligned}&\mathit{ProvidedSpec}\colon \mathit{VIEW} \times \mathit{EXP\_SERVICE} \rightarrow \mathit{SPEC}\\ &\mathit{RequiredSpec}\colon \mathit{VIEW} \times \mathit{IMP\_SERVICE} \rightarrow \mathit{SPEC}\\ &\mathit{SatisfiesSpec}\colon \mathit{VIEW} \times \mathit{SPEC} \times \mathit{SPEC} \rightarrow \mathit{BOOL}\end{aligned}$$

Similarly the domain of connectors, whose goal is to support the interaction of components, comes with a *Connector* function together with its two projection functions, connecting the service required (imported) by a component to an exported service of another component. Two axioms are imposed on connectors.

$$\begin{aligned}&\mathit{Connector} : \mathit{IMP\_SERVICE} \times \mathit{EXP\_SERVICE} \rightarrow \mathit{CONNECTOR}\\ &\mathit{ImportService} : \mathit{CONNECTOR} \rightarrow \mathit{IMP\_SERVICE}\\ &\mathit{ExportService} : \mathit{CONNECTOR} \rightarrow \mathit{EXP\_SERVICE}\end{aligned}$$

Connector Axiom 1. Every import service is connected to at most one exported service by either a connector or an abstract connector.

Connector Axiom 2. No export service uses in its body an import service to which it is linked by a connector-chain (non-cyclic connectors).

An export service s_{exp} uses in its body an import service s_{imp}, if

$$s_{imp} \in \mathit{ContainedServices}(\mathit{ImportStructure}(s_{exp})).$$

[13] See [31] for an interesting proposal to implement behavioral interface specifications, including component interaction, via AsmL on the .NET platform; ASMs are used as vehicle to convey to the client an understanding of component behavior without relying on implementation details.

Component systems are defined as sets S of component structures as described above. In such a system the sets and functions of each of its component structures S are parameterized by S, although notationally we suppress the parameter when it is clear from the context. The following well-formedness condition is imposed on component systems.

Component System Axiom. Every component and every connector in a component system belongs to exactly one component structure of the system.

Using the above defined signature of component structures and component systems, we now define a dynamic consistency check for component structures, based upon a *consistency notion* for component structures which captures the correctness of the connections and the satisfaction of the constraints. The *consistency check machine* below is defined to be implementable as stand-alone process for each single component structure; this requirement is satisfied using the **forall** construct to govern the component wise sequential check of the connectors and the constraints.

```
$\textsc{ComponentConsistencyCheck}(S) = \textbf{forall}\ c \in \mathit{COMPONENT}(S)$
  $\textsc{Fsm}(\mathit{checkSpec}, \textsc{CheckConnectSpec}(\textsc{c}), \mathit{checkConstr})$
  $\textsc{Fsm}(\mathit{checkConstr}, \textsc{CheckConstraints}(\textsc{c}), \mathit{checked})$
```

$\textsc{CheckConnectSpec}(c)$ outputs an error message if some import service is without connector or if some connector connects an import service to an export service without matching signature or with a specification which for some view is not satisfied. The use of the ι-operator to describe for an import service *the* export service to which it is connected is justified by the first connector axiom. The relation $\mathit{equivSignatures}(s, s')$ expresses that the elements of the parameter lists $\mathit{ServiceParam}(s)$ and $\mathit{ServiceParam}(s')$ as well as $\mathit{ServiceResType}(s)$ and $\mathit{ServiceResType}(s')$ are pairwise equivalent.

```
$\textsc{CheckConnectSpec}(c) = \textbf{forall}\ s_{imp} \in \mathit{Imports}(c)$
  $\textbf{if}\ \forall s\ \mathit{Connector}(s_{imp}, s) = \mathit{undef}\ \textbf{then}$
    $\textsc{ImpServiceNotConnected}(s_{imp})$
  $\textbf{else let}\ s_{exp} = \iota s(\mathit{Connector}(s_{imp}, s) \neq \mathit{undef})$
    $\textbf{if not}\ \mathit{equivSignatures}(s_{imp}, s_{exp})\ \textbf{then}$
      output $\mathit{errorInConnector}(s_{imp}, s_{exp}, \mathit{NoSignatureMatch})$
    $\textbf{forall}\ v \in \mathit{VIEW}\ \ \textbf{if not}$
      $\mathit{SatisfiesSpec}(v, \mathit{ProvidedSpec}(v, s_{exp}), \mathit{RequiredSpec}(v, s_{imp}))$
    $\textbf{then}$ output $\mathit{errorInConnector}(s_{imp}, s_{exp}, \mathit{NoSpecMatch})$
```

The submachine $\textsc{CheckConstraints}(c)$ checks for each exported service the constraints of all used components (those from where services are imported to execute a service exported by the component) to meet the related use structure.

```
CheckConstraints($c$) =
  forall $s_{exp} \in Exports(c)$  forall $c' \in UsedComponents(s_{exp})$
    if not $MeetsConstraint(ImportStructure(s_{exp}), Constraint(c'))$
    then output $error(s_{exp}, c', ComponentConstraintViolated)$
  where $UsedComponents(s) = \{c \in COMPONENT \mid$
    $\exists s_{imp} \in ContainedServices(ImportStructure(s))$
      $\exists s_{exp} \in Exports(c)\ (Connector(s_{imp}, s_{exp}) \neq undef)\}$
```

In a similar way one can define a notion of system refinement with a checking machine. In [410] such a refinement is defined independently from each other for types, views and components, so that the check can be defined by independent machines

```
CheckTypeRefinement($R, S$)
CheckViewRefinement($R, S$)
CheckComponentRefinement($R, S$)
```

which can be executed in parallel.

To conclude we illustrate connectors by their instance in the networking architecture *UPnP* where they have the role of *communicators*, in particular to route messages between agents. To perform this task each communicator is equipped with a *mailbox* of *MESSAGE*s which may or may not be *ReadyToDeliver*, as expressed by a monitored predicate. Communicators *ResolveMessage*s selected among those *ReadyToDeliver* in their *mailbox*. More precisely they *Transform* an inbound message into a set of outbound messages addressed to all recipients at the message *destination*. The information on these recipients is supposed to be retrievable from an *addressTable*. The chosen messages are discarded from the *mailbox* and are forwarded to every recipient, which can be reached using the *routingTable* for the message *destination* address, which is an external function that represents the global network topology by mapping addresses to neighboring agents. This is formalized by the following ASM taken from[223] (where the auxiliary functions are refined to AsmL executable value returning procedures; see Sect. 4.1.2):

```
Communicator =
  choose $SelMsg \subseteq \{m \in mailbox \mid ReadyToDeliver(m)\}$
    forall $msg \in SelMsg$
      Delete($msg, mailbox$)
      forall $m \in ResolveMessage(msg)$
        if $Recipient(m) \neq undef$ then
          InsertMsg($m, Recipient(m)$)
  where $ResolveMessage(m) =$
    $\{Transform(m, a) \mid a \in addressTable(destination(m))\}$
  $Recipient(m) = routingTable(destination(m))$
```

3.1.3 Exercises

Exercise 3.1.1. ($\rightsquigarrow$ CD) Formulate a variant of rule AllOrNone which chooses a maximal satisfiable subset of simultaneously invoiced pending orders for the same product. Formulate a maximal-sale rule where the strategy is to obtain a maximal quantity of sold items.

Exercise 3.1.2. Formulate rules similar to IncomingOrders for the cancellation of orders and for entering new items into the stock.

Exercise 3.1.3. Define refined rules for the transitions EnterBalance and NoFurtherService of machine UseCaseAtm.

Exercise 3.1.4. Explain why the ATM Correctness Lemma 3.1.1 remains true for the ASM with refined submachine EnterMoney.

Exercise 3.1.5. Explain why the ATM Correctness Lemma 3.1.1 is preserved by the GUI data refinement step.

Exercise 3.1.6. ($\rightsquigarrow$ CD) Formulate desirable robustness properties for the ATM and refine the ASM to incorporate them.

Exercise 3.1.7. Refine the ATM ASM to an executable program in your favorite programming language or ASM execution engine. In case you use Java, can you justify the correctness of your implementation on the basis of the ASM model for Java in [406]?

Exercise 3.1.8 (Data refinement for an invoice machine). ($\rightsquigarrow$ CD)

1. Specify a one-user system to handle invoices for orders from clients and to accordingly maintain the record of products (prices and availability in stock), showing also how to incorporate subsystems for error detection and statistics. The system should provide a ("for change") extendable set of operations, including, for example, the following ones: creating/modifying products, clients, invoices; reporting errors in handling products, invoices, clients; retrieving information on clients, products, invoices and their past. For customers' inspection, use abstract and application-domain-oriented operations and then data refine them by more detailed equivalent operations.
2. Refine the specification by specifying also an appropriate GUI.
3. Illustrate the restrictions of the "one-user one-operation per time system" with respect to a distributed multiple-user version.

Exercise 3.1.9 (Data refinement of line-editor operations).

1. Define a buffer with the operations InsertChar, DeleteChar and ForwardCursor, BackwardCursor from the following given operations:

$insert(String, Character, Position)$
$delete(String, Position)$

2. Add display-oriented features, distinguishing between printable and unprintable characters, using operation $prefix(String, l)$ providing the prefix of length l of *String*, operation $blanks(Length)$ yielding a string of blanks of given *Length*, and operation $fill(String, l)$ filling *String* with blanks to reach at least length l.

Exercise 3.1.10. (⇝ CD) Refine the Password Change machine to satisfy R2 and justify its correctness for the requirements.

Exercise 3.1.11 (AlarmClock**).** (⇝ CD) Design an alarm clock, which automatically every second updates and displays *currtime*, and allows setting of *currtime* or alarm time (hour, minute) by the user. *currtime* is displayed at each update. To initiate (re)setting *currtime*/alarm time, the user pushes a time/alarm button. The clock, upon reaching the alarm time, rings until either the user cancels the alarm by pressing the "alarm stop" button or 30 seconds have elapsed. The execution of "ring" may be made dependable upon the position of an alarm-on/off-button, to be set by the user. Refine the alarm to provide for three consecutive ringings with an interval of five minutes between them.

Exercise 3.1.12. Refine the AlarmClock of the previous exercise (a) by adding more time zones, e.g. allowing the display of the local time and the time of another time zone, (b) by adding two independent stopwatch timers (which can be set, started, stopped and read off as usual), (c) by adding the possibility of adjusting the time to the exact time of the atomic clock in Braunschweig.

Exercise 3.1.13. Refine the TelephoneExchange ASM to handle also phone conferences with up to n partners.

Exercise 3.1.14 (Refining forall machines). (⇝ CD) Refine the 1-step machine ComponentConsistencyCheck(S) by an iterated control state ASM working in each iteration on one component only, but with an abstract scheduling for the iteration order. Prove the equivalence.

3.2 Incremental Design by Refinements

In this section we define the notion of ASM refinement, illustrate the frequent special patterns of conservative extension, of procedural (submachine) refinement and of pure data refinement, and explain the general scheme for proving the correctness of ASM refinements. As the first examples for proving some system properties by building a sequence of proven-to-be-correct stepwise refined ASMs we consider in Sect. 3.2.2 two well-known algorithms for database recovery and for the shortest path problem. For other examples see Sect. 3.3 and Chapters 5, 6. In Sect. 3.2.3 we explain Schellhorn's scheme for modularizing ASM refinement correctness proofs.

3.2.1 Refinement Scheme and its Specializations

We formulate here the meaning underlying the ASM refinement step scheme in Fig. 2.1 by defining in what sense every (infinite) refined run correctly simulates an (infinite) abstract run with equivalent corresponding states.

Definition 3.2.1 (Correct refinement). Fix any notions $\equiv$ of equivalence of states[14] and of initial and final states. An ASM M^* is a correct refinement of an ASM M if and only if for each M^*-run $S_0^*, S_1^*, \ldots$ there is an M-run $S_0, S_1, \ldots$ and sequences $i_0 < i_1 < \ldots, j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S_{j_k}^*$ for each k and either

- both runs terminate and their final states are the last pair of equivalent states, or
- both runs and both sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ are infinite.

The states $S_{i_k}, S_{j_k}^*$ are the corresponding states of interest. They represent the end points of the corresponding computation segments (those of interest) in Fig. 2.1, for which the equivalence is defined in terms of a relation between their corresponding locations (those of interest).[15] We leave it as Exercise 3.2.1 to show that in Def. 3.2.1, the sequences of corresponding states can be chosen to be minimal in the sense that between two sequence elements there are no other equivalent states. We refer to Fig. 2.1 when using the term (m, n)-refinement (steps), as defined in Sect. 2.1.2.

When every (infinite) abstract run correctly simulates an (infinite) refined run with equivalent corresponding states, the refinement is called complete. Correct refinements which are not complete frequently occur in practice.

Definition 3.2.2 (Complete refinement). M^* is a complete refinement of M if and only if M is a correct refinement of M^*.

Corollary 3.2.1. For deterministic ASMs (without **choose**), refinement correctness and completeness imply for terminating runs the equivalence of the input/output behavior of the abstract and the refined machine.

[14] As explained in Sect. 2.1.2 such a notion of equivalence typically is an equivalence between corresponding states of interest which is based upon the equivalence of the data in the locations of interest in these states.

[15] Sometimes it is convenient to assume that terminating runs are extended to infinite sequences which become constant in the final state.

Purely incremental refinement, also known as *conservative extension*, has been described for the lift control in Sect. 2.3 and illustrated by the extension of the ATM machine by GoOutOfService in Sect. 3.1.2. Another example was given in Fig. 2.11 where a bytecode verifier is added to the trustful Java interpreter component of the Java Virtual Machine.

Procedural refinement, also called submachine refinement, consists in replacing one machine by another (usually more complex) machine. A characteristic example is the refinement of a Prolog machine which uses an abstract function *unify* to a machine which calls a submachine implementing a unification procedure [132] – example of a $(1, n)$-refinement where n can be determined only dynamically since it depends on the size of the to be unified terms. $(1, n)$-refinements with $n > 1$ have their typical use in compiler verification when replacing a source code instruction by a chunk of target code; for numerous examples see [104, 102, 439, 273, 231].

It is important for the practicability of ASM refinements that the size of m and n in (m, n)-refinements is allowed to depend dynamically on the state. Characteristic examples appear in [132]. Lemma 6.4.2 provides a case where n is fixed, but grows with the number of protocol members, or where $n = *$ is finite but without a priori bound, depending on the execution time of the participating processes. The correctness proof of a Java-to-JVM compiler in [406, Sect. 14.2] uses $(1, n)$-refinements with $0 \leq n \leq 3$ depending on the length of the computation which leads the JVM machine from one to its next state of interest (i.e. one corresponding to a state of the Java machine). In [141] the correctness proof for exception handling in Java/JVM uses (m, n)-refinements where m is determined by the number of Java statements jumped over during the search for the exception handler. Although by a theorem of Schellhorn [387, Theorem 12] every (m, n)-refinement with $n > 1$ can be reduced to $(m, 1)$-refinements, this is typically at the price of having more involved equivalence notions which may complicate the proofs. Practical experience shows that (m, n)-refinements with $n > 1$ and including $(m, 0), (0, n)$-steps support the feasibility of decomposing complex (global) actions into simpler (locally describable) ones whose behavior can be verified in practice.

For control state ASMs the graph structure provides a frequent special case of procedural refinement, namely replacing a control state transition – a machine *rule* at a node with well-defined entries i and exits j – by a new submachine M with the same number of entries and exits. This is illustrated in Fig. 3.5 for the replacement of Fsm$(i, rule, j)$ by Fsm(i, M, j) (tacitly assuming the renaming of the entry/exit nodes of M to the given ones i, j, which is incorporated into the diagram notation).

As an example we refine the operation *OnStart* which in the command-line debugger in Fig. 3.7 leads from control state *Init* to *Break*. The Object Model, intermediate between the control state and the ground model, reflects the static (compile time) program components of the run-time system (such as module, class, function) and offers a partial view of the dynamic model (e.g. processes, threads). It thus refines the related user commands, e.g. setting breakpoints, stepping commands, requesting to control the execution by inspection of the run-time stack, of frames. In the debugger object model, *OnStart* is refined to first *initializeCOM*, then to *createNewShell* (initializing its environment, process, thread, frame, breakpoints, etc.) with a pointer to the services, followed by *setDbgCallback* to provide the services access to the client's callback methods. The resulting refined machine is illustrated on slide 19 of `RefinemtMeth` (⇝ CD).

Due to the synchronous parallelism of ASMs, in a $(1,1)$-refinement an action – a part of a parallel step, not limited to a single "operation" – can be replaced by multiple parallel actions which are viewed as part of a new parallel step. Formally speaking, in an ASM a rule can be refined by finitely many other rules which are executed in parallel. An interesting example has been put on slide 20 of `RefinemtMeth` (⇝ CD), where the callback for loading modules in the control state debugger is refined in the object model to first bind in parallel each of the shell's breakpoints to the module in question, and only then to call the debugee to continue. Analyzing this rule and the symmetric rule refinement for unloading of modules in the debugger object model, a mismatch was detected between the way loading and unloading of module callbacks was implemented; see [26, Sect. 4.3]. `Debugger` (⇝ CD) contains further examples of such procedural refinements of control state ASMs.

A pure *data refinement* is given by $(1,1)$-refinements where abstract states and ASM rules are mapped to concrete ones in such a way that the effect of each concrete operation on concrete data types is the same as the effect of the corresponding abstract operation on abstract data types.

A frequent type of an ASM data refinement, which exploits the generalization of "operation" to "ASM rule", is the transition from a use case model with abstract (symbolic) rules to a model which assigns a state transformation meaning to the rule names; see the use case model refinements for the lift ASM in Sect. 2.3 and for the ATM machine in Sect. 3.1.2. As another example we show how the notion of backtracking can be captured by an ASM in such a way that applying to it appropriate data refinements yields well-known logic and functional programming patterns and generative grammars (context free and attribute grammars).

The Backtrack machine dynamically constructs a tree of alternatives and controls its traversal. When its *mode* is *ramify*, it creates as many new children nodes to be computation *candidates* for its *currnode* as there are computation *alternatives*, provides them with the necessary *env*ironment and switches to *selec*tion mode. In *mode* = *select*, if at *currnode* there are no more candidates the machine Backtracks, otherwise it lets the control move to TryNextCandidate to get *execute*d. The external function *alternatives* determines the solution space, depending upon its parameters and possibly the current state. The dynamic function *env* records the information every new node needs to carry out the computation determined by the alternative it is associated with. The macro BACK moves *currnode* one step up in the tree, to *parent*(*currnode*), until the *root* is reached where the computation stops. TRYNEXTCANDIDATE moves *currnode* one step down in the tree to the *next* candidate, where *next* is a possibly dynamic choice function which determines the order for trying out the alternatives. Typically, the underlying execution machine will update *mode* from *execute* to *ramify*, in the case of a successful execution, or to *select* if the execution fails. This model is summarized by the following definition:

```
    BACKTRACK = {RAMIFY, SELECT} where
    RAMIFY =
      if $mode = ramify$ then
        let $k = |alternatives(Params)|$
        let $o_1, \ldots, o_k = new(NODE)$
          $candidates(currnode) := \{o_1, \ldots, o_k\}$
          forall $1 \leq i \leq k$
            $parent(o_i) := currnode$
            $env(o_i) := i\text{-th}(alternatives(Params))$
          $mode := select$
    SELECT =
      if $mode = select$ then
        if $candidates(currnode) = \emptyset$ then BACK
        else
          TRYNEXTCANDIDATE
          $mode := execute$
  BACK =
    if $currnode = root$
      then $mode := Stop$
      else $currnode := parent(currnode)$
  TRYNEXTCANDIDATE =
    $currnode := next(candidates(currnode))$
    DELETE$(next(candidates(currnode)), candidates(currnode))$
```

We show now that by data refinements BACKTRACK can be turned into the backtracking engine for the core of ISO Prolog [131], of IBM's constraint logic programming language CLP(R) [133], of the functional programming language Babel [118], of context free and of attribute grammars [297].

To obtain the backtracking engine for Prolog, we instantiate *alternatives* to the function *procdef*(*stm*, *pgm*), yielding a sequence of clauses in *pgm*, which have to be tried out in this order to execute the current goal *stm*, together with the needed state information from *currnode*. We determine *next* as *head* function on sequences, reflecting the depth-first left-to-right tree traversal strategy of ISO Prolog. It remains to add the execution engine for Prolog specified as an ASM in [131], which switches *mode* to *ramify* if the current resolution step succeeds and otherwise switches *mode* to *select*.

The backtracking engine for CLP(R) is the same. One only has to extend *procdef* by an additional parameter for the current set of *constraints* for the indexing mechanism and to add the CLP(R) engine specified as ASM in [133].

The functional language Babel uses the same function *next*, whereas *alternatives* is instantiated to $fundef(currexp, pgm)$, yielding the list of defining rules provided in *pgm* for the outer function of *currexp*. The Babel execution engine specified as an ASM in [118] applies the defining rules in the given order to reduce *currexp* to normal form (using narrowing, a combination of unification and reduction).

To instantiate Backtrack for context free grammars G generating leftmost derivations we define $alternatives(currnode, G)$ to yield the sequence of symbols $Y_1, \ldots, Y_k$ of the conclusion of a G-rule whose premise X labels *currnode*, so that *env* records the label of a node, either a variable X or terminal letter a. The definition of *alternatives* includes a choice between different rules $X \rightarrow w$ in G. For leftmost derivations *next* is defined as for Prolog. As machine in $mode = execute$ one can add the following rule. For nodes labeled by a variable it triggers further tree expansion, and for terminal nodes it extracts the yield (concatenating the terminal letter to the word generated so far) and moves the control to the parent node to continue the derivation in $mode = select$.

```
Execute(G) = if $mode = execute$ then
  if $env(currnode) \in VAR$ then $mode := ramify$ else
    $output := output * env(currnode)$
    $currnode := parent(currnode)$
    $mode := select$
```

For attribute grammars it suffices to extend the instantiation for context free grammars as follows. For the synthesis of the attribute $X.a$ of a node X from its children's attributes we add to the else-clause of the Back macro the corresponding update, e.g. $X.a := f(Y_1.a_1, \ldots, Y_k.a_k)$ where $X = env(parent(currnode))$ and $Y_i = env(o_i)$ for children nodes o_i. Inheriting an attribute from the parent and siblings can be included in the update of *env* (e.g. upon node creation), extending it to update also node attributes. The attribute conditions for grammar rules are included in Execute(G) as an additional guard to yielding output, of the form $Cond(currnode.a, parent(currnode).b, siblings(currnode).c)$. We leave it as Exercise 3.2.3 to formulate an ASM for tree adjoining grammars, generalizing Parikh's analysis of context free languages.

We conclude this section with a particular case of (n, m)-refinement where $n < m$, which is known in the literature under the name of *premature choice*. It deals with implementations of an early choice in an abstract model by a later choice in the refined model. For the traditional trace-based refinement approach which works with abstractions of type $(1, 1)$ this case presents a technical problem of proof-theoretic nature, usually solved by introducing so-called *prophecy variables*. For a clear and succinct exposition see [316, Sect. 8]. The ASM refinement notion covers this case, as we are going to illustrate here by the reliable channel example taken from [316, p. 8.10]: "A reliable channel accepts messages and delivers them in FIFO order, except that if there is a crash, it may lose some messages." The following basic ASM describes the crucial three operations of putting and getting *Message*s, where the *queue* operations to *insert*, *delete*, *return* a message are kept abstract, and of crashes which may result in the loss of some or all messages in the *queue*:

```
$\textsc{EarlyChoice} = \textsc{Put}$ or $\textsc{Get}$ or $\textsc{Crash}$
where
  $\textsc{Put}(m) = insert(m, queue)$
  $\textsc{Get} = \{delete(head(queue), queue), return(head(queue))\}$
  $\textsc{Crash} =$ choose $q \in \{s \mid s$ subsequence of $queue\}$
    forall $m \in q$ $delete(m, queue)$
```

A typical implementation, like the Internet's TCP protocol, ensures FIFO delivery and gets rid of retransmitted duplicates by consecutive message numbering, discarding any message whose number is smaller than the last assigned number. Since the underlying Internet message transport is not FIFO, if after a crash a later outstanding undelivered message overtakes an earlier one, the latter will be lost – but this will be known only after the overtaking has taken place, which may be long after the crash and the subsequent recovery. One can model such an implementation by "marking" the messages which are queued at the time of a crash and letting the get-operation choose whether to drop or to keep a marked message. This yields a refined basic ASM LateChoice where the set *Message* is refined to pairs $(x.mssg, x.mark)$ with attributes *mssg* and a Boolean-valued *mark* (with default value "false" to be assigned when *insert*ing a new message into the *queue*). LateChoice has the same rule as EarlyChoice but with a redefined crash operation, which marks all messages in the *queue*; it has a redefined return macro, which has the choice whether to skip or to in fact return a marked message.

```
$\textsc{Crash}_{ref} =$ forall $x \in queue$ $x.mark := true$
$return_{ref} =$ (if $head(queue).mark$ then $skip$) or $return(head(queue))$
```

We leave it as Exercise 3.2.6 to formulate conditions under which the machine EarlyChoice is correctly refined by LateChoice.

Problem 9 (Framework for communication models). Formulate a uniform model of communication from which practical communication mechanisms and communication models proposed in the literature can be obtained by instantiating the abstractions. Use the framework for a comparison of different communication models.

3.2.2 Two Refinement Verification Case Studies

In this section we show how to use ASM refinements for proving system properties. We use two well-known algorithms for the shortest path problem and for database recovery, specified following [409, 260] by a ground model ASM which is detailed incrementally by proven to be correct refinement steps.

Shortest path (graph traversal) problem. The goal is to exhibit an efficient program which computes the reachability set of a given graph and can be proved to do so. The task is performed by defining the following hierarchy of ASMs leading from a ground model to Dijkstra's algorithm [182] and to an executable program, where each refinement step implements a design decision which is documented by the correctness proof:[16]

- a ground model computing graph reachability sets: $\textsc{ShortestPath}_0$,
- wave propagation of frontier: $\textsc{ShortestPath}_1$,
- neighborhoodwise frontier propagation: $\textsc{ShortestPath}_2$,
- edgewise frontier extension per neighborhood: $\textsc{ShortestPath}_3$,
- queue/stack implementation of frontier/neighborhoods: $\textsc{ShortestPath}_4$,
- introducing weights for measuring paths and computing shortest paths: $\textsc{ShortestPath}_5$ (Dijkstra's algorithm,)
- instantiating data structures for measures and weights: a C++ program.

The ground model serves to specify an algorithm which, given a directed graph $(\mathit{NODE}, E, \mathit{source})$ with a distinguished *source* node, labels every node which is reachable from *source* via edges in E and terminates for finite graphs. The idea is to start at *source*, move along edges to neighbor nodes and *label* every reached node as visited. In the initial state only *source* is labeled as visited. Termination is achieved by pushing in each step the set of already visited nodes one edge further without revisiting nodes which have already been *labeled as visited*. This algorithmic idea is formalized by the following ASM for which the correctness and termination property can be proved easily.

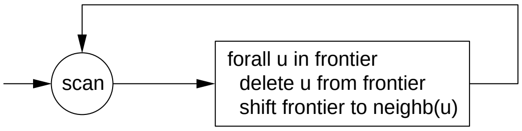
```
$\textsc{ShortestPath}_0 =$
  $\textbf{forall } (u, v) \in E \textbf{ with } u \in \mathit{visited} \textbf{ and } v \notin \mathit{visited}$
    $\mathit{visited}(v) := \mathit{true}$
```

Lemma 3.2.1 (Correctness). Each node which is reachable from *source* is exactly once labeled as visited.

Proof. The existence claim follows by an induction on the length of the paths from *source*. The initialization assumption guarantees the basis of the induction. The uniqueness property follows from the rule guard which ensures that only nodes which are not yet labeled as visited are considered for being labeled as visited. □

[16] An asynchronous shortest path ASM, $\textsc{MinPathToLeader}$, which works in arbitrary connected networks, is defined in Sect. 6.1.5.

Fig. 3.8 ShortestPath$_1$



Lemma 3.2.2 (Termination). The machine ShortestPath$_0$ terminates for finite graphs, in the sense that it reaches a state in which there is no longer any edge $(u, v) \in E$ whose tail u is labeled as visited but whose head v is not.

Proof. Follows from the fact that by each rule application, the (finite) set of nodes which are not labeled as visited decreases. □

The goal of the first refinement step is to identify the *frontier* of the wave propagation, namely as the dynamic set of nodes which have been labeled as visited in the last step. The assumption that initially only *source* is labeled as visited is turned into the equation $frontier = \{source\}$ which holds in the initial state. This leads to the following ASM where the frontier is moved simultaneously for each node in *frontier* to all its neighbors (restricted to those which have not yet been labeled as visited). Nodes are labeled as visited when they become members of *frontier*, hence *frontier* is a subset of *visited*. See Fig. 3.8.

```
ShortestPath$_1$ =
  forall $u \in frontier$
    $frontier(u) := false$
    ShiftFrontierToNeighb($u$)

ShiftFrontierToNeighb($u$) =
  forall $v \in neighb(u)$ do ShiftFrontierTo($v$)

ShiftFrontierTo($v$) = if $v \notin visited$ then
  $visited(v)  := true$
  $frontier(v) := true$
```

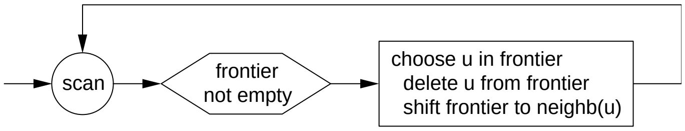
Neighbors of a node u are those connected to u by an edge:

$$neighb(u) = \{v \mid (u, v) \in E\}.$$

It is easy to show that all nodes which are reachable from *source* by a path of length $\leq t$ are labeled as visited by ShortestPath$_1$ in $\leq t$ steps (Exercise 3.2.7).

Lemma 3.2.3. The steps of ShortestPath$_i$ for $i = 0, 1$ are in 1-1 correspondence and perform the same labelings.

Fig. 3.9 SHORTESTPATH$_2$



Proof. Ths follows by an induction on the runs, proving that in each state the set of nodes u such that u is visited and there is an edge $(u, v) \in E$ such that v is not visited is contained in *frontier* (so that SHIFTFRONTIERTONEIGHB(u) is applied). □

Corollary 3.2.2. SHORTESTPATH$_1$ terminates and satisfies the correctness property of Lemma 3.2.1.

The goal of the second refinement step is to start reducing the parallelism by shifting the frontier in each step to the neighborhood of only *one* node. To leave the design space open, we keep the scheduling of the node for the next frontier propagation step abstract and determine those nodes by a choice function *select*. When it comes to prove certain properties, this function will be constrained by appropriate conditions (e.g. fairness to obtain completeness of node visits). This leads to the following ASM which has the same macros as SHORTESTPATH$_1$, not repeated here. See the equivalent description in Fig. 3.9 which realizes the idea that each run of SHORTESTPATH$_1$ can be simulated by a breadth-first run of SHORTESTPATH$_2$ producing the same labelings of nodes as visited (Exercise 3.2.8).

```
SHORTESTPATH$_2$ = **choose** $u \in frontier$
    $frontier(u) := false$
    SHIFTFRONTIERTONEIGHB$(u)$
```

Lemma 3.2.4 (Slow down by nodewise frontier propagation).

1. For each step t and each $u \in frontier_t$(SHORTESTPATH$_2$) there exists a $t' \leq t$ such that $u \in frontier_{t'}$(SHORTESTPATH$_1$).
2. If SHORTESTPATH$_2$ in step t labels a node as visited, then SHORTESTPATH$_1$ does the same in some step $t' \leq t$.

Proof. Both statements follow by an induction on t. □

Corollary 3.2.3. SHORTESTPATH$_2$ terminates for finite graphs and satisfies the correctness property of Lemma 3.2.1.

The goal of the second refinement step is to continue reducing the parallelism by edgewise frontier extension per neighborhood. This comes up to refine the SHORTESTPATH$_2$-rule SHIFTFRONTIERTONEIGHB(u) to an iterating

Fig. 3.10 ShiftFrontierToNeighb(u)

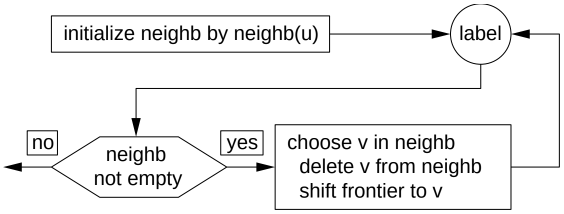
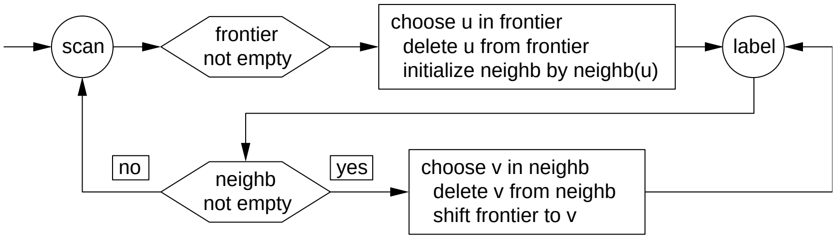


Fig. 3.11 ShortestPath$_3$



submachine which – after appropriate initialization for *neighb* = *neighb*(u) – selects one by one every node v of *neighb* to edgewise ShiftFrontierTo(v), so that the same labeling of nodes as visited is obtained. Replacing the machine ShiftFrontierToNeighb(u) in ShortestPath$_2$ by the iterating machine defined in Fig. 3.10 yields the machine ShortestPath$_3$ in Fig. 3.11.

Corollary 3.2.4. The correctness and termination of ShortestPath$_2$ are preserved by the refinement to ShortestPath$_3$.

The next step is a data refinement which implements *frontier* by a queue and *neighb* by a stack. Therefore, *choose* of *frontier* becomes the function which selects the first element at the left end, and *insert* becomes the function which appends its argument at the right end, together with the standard queue functions like *delete*, meaning *frontier* := *rest*(*frontier*). We have to maintain the property that no node occurs more than once in *frontier*. For the stack we have the obvious functions *choose* = *top*, *delete* = *pop* and assume for the initialization that *neighb*(u) is given as stack. Let ShortestPath$_4$ be the resulting ASM which is easily shown to preserve correctness and termination of ShortestPath$_3$ (Exercise 3.2.9).

To refine ShortestPath$_4$ to a machine which computes a *shortest* path for each reachable node, the *weight* of paths from *source* has to be defined and computed. The idea is to measure paths by the accumulated weight of their edges. The weight of edges is determined by an abstract function *weight*: $E \to \mathbb{R}^+$ that assigns a non-negative real number to each edge of the graph. The path $\mathit{weight}\colon \mathit{PATH} \to \mathbb{R}$ is defined in the usual inductive manner from edge *weight* (from which it is distinguished from by its type); ϵ stands for the empty path:

$$\mathit{weight}(\epsilon) = 0, \mathit{weight}(pe) = \mathit{weight}(p) + \mathit{weight}(e).$$

One can then define $\mathit{minWeight}\colon \mathit{NODE} \to \mathbb{R}$ by

$$\mathit{minWeight}(u) = \mathit{inf}\{\mathit{weight}(p) \mid p \text{ is a path from } \mathit{source} \text{ to } u\}.$$

The algorithmic idea is to compute *minWeight* by successive approximations of an upper bound $\mathit{upbd}\colon \mathit{NODE} \to \mathbb{R}$ for each node encountered on a path from *source*. Approximations come into the picture because nodes may be connected to *source* by more than one path, whose weights have to be compared to determine a minimal one. The assumption for the initial state is refined to $\mathit{upbd}(u) = \infty$ for every node u except $\mathit{upbd}(\mathit{source}) = 0$.

Consider now any frontier shift from a node u to one of its neighbors v, i.e. a step where a path from *source* to u is extended by the edge $e = (u, v)$ to reach v. If the current upper bound $\mathit{upbd}(v)$ as candidate for $\mathit{minWeight}(v)$ can be improved by the weight of the newly considered path from *source* to u followed by e, then $\mathit{upbd}(v)$ is *lowered via* u, namely to $\mathit{upbd}(u) + \mathit{weight}(e)$.

This idea is realized by refining the operation ShiftFrontierTo(v) in ShortestPath$_4$ to the following LowerUpbd(v, u):

```
LowerUpbd($v, u$) = let $e = (u, v)$ in
  if $upbd(u) + weight(e) < upbd(v)$ then
    $upbd(v) := upbd(u) + weight(e)$
    if $v \notin visited$ then
      $frontier(v) := true$
      $visited(v) := true$
```

Let ShortestPath$_5$ be the result of replacing in ShortestPath$_4$ the machine ShiftFrontierTo(v) by LowerUpbd(v, u) and initializing *neighb* by $(u, \mathit{neighb}(u))$ (since the parameter u is needed for lowering $\mathit{upbd}(v)$). Moreover, *frontier* is implemented as a *priority queue* in which the node which is selected is always the one with the least *upbd* value:

$$u = \mathit{choose}(\mathit{frontier}) \iff \forall v \in \mathit{frontier}(\mathit{upbd}(u) \le \mathit{upbd}(v))$$

This machine is known as Dijkstra's algorithm [182].

Lemma 3.2.5. Dijkstra's algorithm terminates for finite graphs.

Proof. Consider any run of ShortestPath$_5$. Each node of the graph is added at most once to the priority queue *frontier*. Hence the control state *scan* is reached at most $|\mathit{NODE}|$ times. □

Theorem 3.2.1 (Correctness). $\mathit{minWeight}(u) = \mathit{upbd}(u)$ holds for every node u when Dijkstra's algorithm terminates.

Proof. We call a node u *complete*, if its upper bound is already the weight of the shortest path from *source* to u, i.e., if $\mathit{upbd}(u) = \mathit{minWeight}(u)$.

The following two invariants hold whenever ShortestPath$_5$ is in the control state *scan*:

1. If u is *visited* but not in *frontier*, then u is complete.
2. If u is complete, u is not in *frontier* and e is an edge from u to v, then $\mathit{upbd}(v) \leq \mathit{minWeight}(u) + \mathit{weight}(e)$.

The two invariants are certainly satisfied in the initial state, where all nodes different from *source* are not complete and $\mathit{visited} = \mathit{frontier} = \{\mathit{source}\}$.

Assume now that the machine is control state *scan* and that u is the least element of the priority queue *frontier*. Since u is removed from *frontier*, we have to show that u is complete and that the invariant 2 for u is still true in the next state, where the machine is in control state *scan* (after all neighbors of u have been lowered).

Consider a shortest path from *source* to u. (Since u is in frontier, it is reachable from *source*.) Starting at *source* we proceed on the path until we reach the first node which is in *frontier*. We call this node v and claim that v is complete. Why? First observe that each initial segment of a shortest path from *source* to a node is also a shortest path. Hence, if $e = (a, b)$ is an edge on a shortest path, then $\mathit{minWeight}(b) = \mathit{minWeight}(a) + \mathit{weight}(e)$. Hence we can start at *source* and repeatedly use invariant 2 and see that all nodes on the path up to and including v are complete.

Since u is the least element of *frontier* and v is complete, it follows that $\mathit{upbd}(u) \leq \mathit{minWeight}(v)$. Since v is on a shortest path from *source* to u, we also have $\mathit{minWeight}(v) \leq \mathit{minWeight}(u)$. Hence, it follows that u is complete.

All neighbors of u are lowered in the following *label* phase and therefore invariant 2 is satisfied for u in the next *scan* state. □

A similar refinement step of ShortestPath$_4$ with an abstract measure leads to Moore's algorithm [336] and a solution of the constrained shortest path problem. For more information on this refinement step and its implementation in C++ see [409].

As one among numerous other possible refinements of ShortestPath$_2$ we mention the algorithm defined in [237] to compute from an ASM an FSM to be used for testing the ASM. ShiftFrontierToNeighb(u) is refined as including into *frontier*, which initially contains an initial state of the ASM, every "relevant" *nextState* v which results from applying to u any machine action a from a set *action*. This parameter typically contains the rules of the original ASM which are to–be–tested. The algorithm keeps track of the used *transition*s (triples (u, a, v)); for relevant next states it also keeps track of the corresponding so-called *hyper*states. These hyperstates are equivalence classes of states of the original ASM. They form the states of the to–be–generated FSM and are computed from the ASM states by a function *hyperstate*. Thus the algorithm is parametric in the underlying notions of *action*, *nextState*, *relevant*, *hyperstate* and initial states which can be adapted to the particular testing goals.[17]

```
AsmGenFsm = choose $u \in frontier$
  $frontier(u) := false$
  forall $a \in action$ forall $v$ with $nextState(u, a, v)$
    $transition(u, a, v) := true$
    if $relevant(u, a, v)$ then
      $frontier(v) := true$
      $hyper(hyperstate(v)) := true$
```

Database recovery. The goal of this example is to design and verify by stepwise refinement an ASM model for a recovery algorithm for a multiple-user, concurrently accessed database, which guarantees despite system failures the *atomicity* of the effect of durative database transactions (either committing all the transaction updates or aborting all of them) and the *durability* of the effect of committed transactions. We construct a database ground model for which it is easy to prove the atomicity and durability of transactions under appropriate run constraints. The model uses three kinds of storage – stable, volatile, committed – with abstract operations Read, Write, Commit, Abort, Recover, Fail, Flush and some auxiliary functions. We then refine volatile and committed storage with related operations by stable and cache memory with log management and prove it to be correct. In exercises we formulate three more provably correct refinements, namely by run-time computation of auxiliary functions, by the sequentialization of parallelism and by computing run constraints.

Ground model DbRecovery**.** The database is accessed by elements of an external set *TRANSACTION* of sequences of user *OPERATION*s. In each step the database executes one operation, the one currently issued as the value of a monitored function *currOp*. The atomicity property requires that, independently of whether the operations issued by any transaction t are interleaved with operations issued by other transactions, transaction t completes in one of the following two ways:

normally with the *commit* operation. By the durability requirement committing means that the values of all *write* operations of t have to remain in the database until they are overwritten by a subsequent transaction.

[17] In [237] *frontier* is also refined to a sequence of reachable ASM states to–be–traversed, with *head* as the selection function. This realizes an implementation step of the sort considered above in passing from ShortestPath_3 to ShortestPath_4.

abnormally with the *abort* operation. This requires the values of all writes of t to be replaced with the previous values, which remain until the next overwrite. Every system failure occurring when t is active aborts t so that the database values after a failure reflect only updates which have been made by transactions committed before the failure.

Each operation is of one of the four types *read, write, commit, abort*, and each has its issuing user and possibly a read/write location and a write value. This is formalized by the following four external functions:

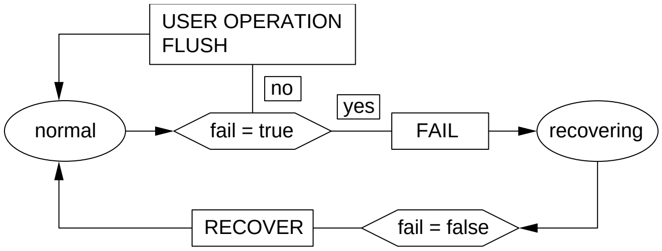
$$\begin{aligned}
&\mathit{issuer}\colon \mathit{OPERATION} \to \mathit{TRANSACTION}\\
&\mathit{type}\colon \mathit{OPERATION} \to \{\mathit{read}, \mathit{write}, \mathit{commit}, \mathit{abort}\}\\
&\mathit{loc}\colon \mathit{OPERATION} \to \mathit{LOC}\\
&\mathit{val}\colon \mathit{OPERATION} \to \mathit{VALUE}
\end{aligned}$$

The goal of the recovery algorithm we are going to define is to install the committed database – *commDb* consisting of the last committed values – as current database *currDb*, consisting of the most recent values over volatile or stable storage. At the time of recovery, the recovery information is required to be in the stable database *stableDb* (as specified in the ground model below by the rule Fail, which transfers *stableDb* to *currDb*, possibly overwritten in the sequel by Recover to *commDb*. The first refinement step provides a more detailed version of this requirement.). For each transaction an auxiliary controlled function *writeSet* records the set of locations for which the given transaction has issued a write operation. A monitored function *fail* indicates the presence or absence of a system failure, treated as an event which is consumed by firing the corresponding Fail rule below. Not to commit to any particular flushing policy, we treat also *cacheFlush* as a monitored function indicating when the cache value in the given location has to be copied to stable storage. In the ground model flushing appears only for specification purposes; its detailed meaning is realized through the refinements. Initially, *stableDb*, *currDb* and *commDb* are supposed to be everywhere undefined and *writeSet* to yield everywhere $\emptyset$.

$$\begin{aligned}
&\mathit{stableDb}, \mathit{currDb}, \mathit{commDb}\colon \mathit{LOC} \to \mathit{VALUE}\\
&\mathit{writeSet}\colon \mathit{TRANSACTION} \to \mathit{PowerSet}(\mathit{LOC})\\
&\mathit{fail}\colon \mathit{BOOL}\\
&\mathit{cacheFlush}\colon \mathit{LOC} \to \mathit{BOOL}
\end{aligned}$$

The database recovery ASM is a machine which – when operating in its *normal* mode – either executes the issued user operation and flushes some values or reacts to a failure. In mode *recovering* it tries to recover from a failure together with the corresponding flush operation. This yields the use case definition of the control state ASM in Fig. 3.12.[18] Note, that UserOperation and

Fig. 3.12 DbRecovery ASM



Flush, which appear in the same box in Fig. 3.12, are executed in parallel according to the standard ASM semantics.

We now define the ground model meaning for the symbolic operations occurring in Fig. 3.12. Of the four user operations, at this level of abstraction reading has no recovery effect (Read = **skip**). Write means to write the given value into the given location in volatile memory (*currDb*) and to record that location as a location being written and therefore subject to later commit or abort. Commit and Abort are inverse to each other and mean to transfer all written locations from volatile storage *currDb* to *commDb* and vice versa. Flush transfers the *toBeFlushed* values from *currDb* to the stable storage *stableDb*, and Failing inversely restores all stable locations in *currDb*. Hence, the effect of Fail is that all values that are in *currDb* but not yet in *stableDb* get lost (the cache memory is cleared). To Recover means to restore all committed locations in *currDb*. This is summarized by the following definition of the above-mentioned symbolic operations, in terms of macros which will be further refined below.

```
UserOperation = case type(currOp) of
  read  → Read          commit → Commit
  write → Write         abort  → Abort
Read = skip
Write = {WriteIntoVolatileLoc, RecordLoc}
Commit = Commit(issuer(currOp))
Abort = Abort(issuer(currOp))
Flush = forall l with toBeFlushed(l) do Flush(l)
Fail = forall l ∈ LOC do RestoreStableValue(l)
Recover = forall l ∈ LOC do RestoreCommValue(l)

WriteIntoVolatileLoc = currDb(loc(currOp)) := val(currOp)
```

[18] The guard *fail* = *false* in the rule leading from *recovering* back to *normal* becomes necessary only when the above-formulated assumption that the event *fail* is consumed by firing Fail is no longer satisfied and that during recovery no new failure can occur.

$\textsc{RecordLoc} = \textbf{let } t = \mathit{issuer}(\mathit{currOp}) \textbf{ in}$
$\quad \mathit{writeSet}(t) := \mathit{writeSet}(t) \cup \{\mathit{loc}(\mathit{currOp})\}$
$\textsc{Commit}(t) = \textbf{forall } l \in \mathit{writeSet}(t) \textbf{ do } \mathit{commDb}(l) := \mathit{currDb}(l)$
$\textsc{Abort}(t) = \textbf{forall } l \in \mathit{writeSet}(t) \textbf{ do } \mathit{currDb}(l) := \mathit{commDb}(l)$
$\mathit{toBeFlushed}(l) \iff (\mathit{cacheFlush}(l) = \mathit{true})$
$\textsc{Flush}(l) = \mathit{stableDb}(l) := \mathit{currDb}(l)$
$\textsc{RestoreStableValue}(l) = \mathit{currDb}(l) := \mathit{stableDb}(l)$
$\textsc{RestoreCommValue}(l) = \mathit{currDb}(l) := \mathit{commDb}(l)$

A transaction is called *active* in state S, if since its first read/write, in some preceding state $T \leq S$, neither does it commit nor abort nor does the system recover in the open interval (T, S). We use here the following expressions:

- t does operation o iff $\mathit{type}(\mathit{currOp}) = o$ and $\mathit{issuer}(\mathit{currOp}) = t$ and $\mathit{ctl\_state} = \mathit{normal}$ and $\mathit{fail} = \mathit{false}$,
- t commits/aborts a write to l iff t commits/aborts and $l \in \mathit{writeSet}(t)$,
- t encounters a failure in S iff t is active and $\mathit{fail} = \mathit{true}$ in S,
- t terminates in S iff t commits or aborts or encounters a failure in S.

Runs of the recovery algorithm are restricted to satisfy the constraint of *strictness* of the external operation schedule. In strict runs, each transaction is terminated by a *commit*, *abort* or a system failure. Moreover, transactions are prevented from writing to locations which contain uncommitted values, so that any transaction that writes to a location l terminates before the next read/write to l.

Definition 3.2.3. A run of the database recovery algorithm is *strict*, if

- operations are issued only by *active* transactions,
- if transaction t in state S writes to a location l and transaction t' writes to or reads from l in a later state $T > S$, then t is not active any more in state T.

It should be noted that in a strict run a transaction may not read from or write to a data location once it has written to it.

Lemma 3.2.6 (Strictness). If a transaction t is active in state S of a strict run and the location l is in $\mathit{writeSet}(t)$, then there exists a state $T < S$ where t writes to l. Hence, if two different transactions t and t' are active in state S, then $\mathit{writeSet}(t) \cap \mathit{writeSet}(t') = \emptyset$.

The first property to show is the durability for *commDb*-values in runs of $\textsc{DbRecovery}$. In the proofs below we say that in a state S the machine is running when $\mathit{fail} = \mathit{false}$ holds in S. Similarly we say that S is "normal" or "recovering" if $\mathit{fail} = \mathit{false}$ holds in S and the control state is *normal* or *recovering*, respectively.

Proposition 3.2.1 (Durability). In strict runs of DbRecovery, when a *write* of a value v to a location l in state R is committed in a state $S > R$, then $commDb(l) = v$ holds from the next state $S+1$ until some transaction commits a new *write* to l.

Proof. A *write* by transaction t of v to l in state R yields $currDb(l) = v$ and $l \in writeSet(t)$ in state $R+1$. A *commit* of t in state $S > R$ yields $commDb(l) = currDb(l)$ in state $S+1$. This implies $commDb(l)_{S+1} = v$ for the following reason: by strictness, from state R (excluded) until state S (included) no transaction writes to l or aborts a write to l and there is no system failure followed by an application of Recover. Therefore, $currDb(l)$ is not updated in $(R, S]$, so that $commDb(l)_{S+1} = v$. If for some T no transaction commits a write to l in $(S, T]$, $commDb(l) = v$ continues to hold in $(S, T]$. □

The second property to show is the atomicity for values of *currDb* in runs of DbRecovery.

Proposition 3.2.2 (Atomicity). In runs of DbRecovery, if transaction t in state R writes value v to location l and thereafter terminates in state S, in every following normal state $T > S$ until which no other Write to l is issued the value $currDb(l)_T$ is

- the value v, if t terminates by a *commit*,
- the old value $currDb(l)_R$ before the last writing, if t terminates by an *abort* or a system failure.

Proof. After t issued a Write in state R, resulting in $currDb(l) = v$ and $l \in writeSet(t)$ in state $R+1$, there are two cases upon termination of t in a state $S \in (R, T)$: either t commits in S (Case 1) or t aborts or encounters a failure in S (Case 2).

Case 1: t commits in S. By strictness this implies that (1) $currDb(l)$ is not updated in $(R, S]$ – since from state R (excluded) until state S (included) no transaction writes to l or aborts a write to l and there is no system failure followed by an application of Recover. Applying Commit in S yields (2) $currDb(l)_{S+1} = currDb(l)_S = commDb(l)_{S+1}$. This implies $currDb(l)_T = v$ by Lemma 3.2.7 below and (1).

Case 2: t aborts or encounters a failure in S. By strictness, (1) $commDb(l)$ is not updated in $[R, T]$, since no transaction commits a write to l in $[R, T]$. By Lemma 3.2.8 below, (2) $currDb(l)_R = commDb(l)_R$. We now distinguish whether t aborts or encounters a failure in S.

Case 2.1: t aborts in $S \in (R, T)$. Then

$$\begin{array}{rcll} currDb(l)_T & = & currDb(l)_{S+1} & \text{by Lemma 3.2.7} \\ & = & commDb(l)_S & \text{since } t \text{ aborts in } S \\ & = & commDb(l)_R & \text{by (1)} \\ & = & currDb(l)_R & \text{by (2)} \end{array}$$

Case 2.2: t encounters a failure in S. Let S' be the first non-failure state in (S, T) (which does exist since a normal state T is reached). This state is the result of applying Recover so that the following holds:

$$\begin{array}{rcll} \mathit{currDb}(l)_{S'+1} & = & \mathit{commDb}(l)_{S'} & \text{by rule Recover} \\ & = & \mathit{commDb}(l)_R & \text{by (1)} \\ & = & \mathit{currDb}(l)_R & \text{by (2)} \\ & = & \mathit{currDb}(l)_T & \text{by Lemma 3.2.7} \end{array}$$

□

Lemma 3.2.7 (Preservation of committed values). Committed values in $\mathit{currDb}(l)$ are preserved between normal states without writes to l, i.e. if $\mathit{currDb}(l)_S = \mathit{commDb}(l)_S$ in a normal state S and if T is a later normal state until which no transaction writes to l, then $\mathit{currDb}(l)_T = \mathit{currDb}(l)_S$.

Proof. The following property can be proved by induction on $T - S$:

> If $S \leq T$, S is normal, $\mathit{currDb}(l)_S = \mathit{commDb}(l)_S$, and no *write* is issued to l in $[S, T)$, then
>
> – if T is normal, then $\mathit{currDb}(l)_T = \mathit{commDb}(l)_T = \mathit{commDb}(l)_S$,
>
> – if T is recovering, then $\mathit{commDb}(l)_T = \mathit{commDb}(l)_S$.

If $S = T$, nothing has to be shown. Otherwise, if $S < T$, the induction hypothesis can be applied to $T - 1$ and depending on whether $T - 1$ is *normal* or *recovering*, the property is carried over from $T - 1$ to T. □

Lemma 3.2.8. In a state where a *write* to l is issued, the equation $\mathit{commDb}(l) = \mathit{currDb}(l)$ holds.

Proof. In a strict run the following invariant holds for each state S:

> If S is normal, then $\mathit{commDb}(l)_S = \mathit{currDb}(l)_S$ or there exists an active transaction t in S that has issued a *write* to l in some state $T < S$.

Hence, if a *write* to l is issued in S, then by strictness the second possibility is excluded. □

Refinement of DbRecovery. The refinement consists in distributing the current database *currDb* into *stableDb* and a *cache* memory. The committed database *commDb* becomes a derived function that can be computed from a *log* of all writes using the set of *committed* transactions. This implies on the one hand that *currDb* is computed as follows from *stableDb* and the volatile $\mathit{cache}\colon \mathit{LOC} \to \mathit{VALUE}$:

$$\mathit{currDb}(l) = \begin{cases} \mathit{stableDb}(l), & \text{if } \mathit{cache}(l) = \mathit{undef}; \\ \mathit{cache}(l), & \text{otherwise.} \end{cases}$$

On the other hand it implies that the dynamic recording of *commDb* is computed in stable storage.

Since the cache flush policy is independent of the recovery mechanism, the problem is that, upon system failure, writes in the cache by uncommitted transactions (with values possibly flushed to stable storage) must be undone, and writes in the cache of committed transactions (whose values might reside only in the cache) must be redone (reinstalled). The idea consists in keeping track of writes/commits via log records of all writes, to be stored in stable or volatile memory, and a dynamic list of committed transactions residing in stable storage.

We therefore introduce a dynamic subset $\mathit{log} \subseteq \mathit{LOG}$ of an ordered set of records of all current writes, equipped with standard order-related functions $<$, *next*, *max* and functions to retrieve the transaction which issued a write, the location where it has written and the new value written there:

$$\begin{aligned}&\mathit{issuer}\colon \mathit{LOG} \rightarrow \mathit{TRANSACTION}\\ &\mathit{loc}\colon \mathit{LOG} \rightarrow \mathit{LOC}\\ &\mathit{afterImage}\colon \mathit{LOG} \rightarrow \mathit{VAL}\end{aligned}$$

The *commDb* is now computed by the equation

$$\mathit{commDb}(l) = \mathit{afterImage}(\mathit{lastRcd}(l, \mathit{commRcds}))$$

using the following auxiliary derived functions to determine the last record $\mathit{lastRcd}\colon \mathit{LOC} \times 2^{\mathit{LOG}} \rightarrow \mathit{LOG}$ for a location in a set of log entries, the records of committed transactions (*commRcds*), the set of last committed records (*lastCommRcds*) and the dynamic function

$$\mathit{committed}\colon \mathit{TRANSACTIONS} \rightarrow \mathit{BOOL}$$

representing the set of committed transactions.

$$\begin{aligned}&\mathit{lastRcd}(l, L) = \mathit{max}\{r \in L \mid \mathit{loc}(r) = l\}\\ &\mathit{commRcds} = \{r \in \mathit{log} \mid \mathit{committed}(\mathit{issuer}(r))\}\\ &\mathit{lastCommRcds} = \{r \in \mathit{commRcds} \mid r = \mathit{lastRcd}(\mathit{loc}(r), \mathit{commRcds})\}\end{aligned}$$

When the database has to UNDO a recorded write, the following derived function $\mathit{undoRcd}(r)$ is used to determine the last committed record r' of a write to the location in question, with the understanding that $\mathit{undoRcd}(r) = \mathit{undef}$ if (as happens initially) no appropriate record r' exists:

$$\mathit{undoRcd}(r) = \mathit{max}\{r' \in \mathit{commRcds} \mid r' < r,\ \mathit{loc}(r') = \mathit{loc}(r)\}$$

The *log* is stored in stable or volatile memory and has a monitored subset $\mathit{stableLog} \subseteq \mathit{log}$ satisfying the following two run constraints:

Run Constraint RCS1. Upon system failure, records of last committed values ("last committed records") must be in stable storage: $\mathit{lastCommRcds} \subseteq \mathit{stableLog} \subseteq \mathit{log}$.

Run Constraint RCS2. If a location has no record in the stable *log* portion, then its value in *stableDb* must be *undef*:
if $\neg\exists r \in \mathit{stableLog}$ with $\mathit{loc}(r) = l$, then $\mathit{stableDb}(l) = \mathit{undef}$.

The Run Constraint RCS1 ensures that, in the case of a system failure when the records in $\mathit{log} \setminus \mathit{stableLog}$ are lost, the record $\mathit{lastRcd}(l, \mathit{commRcds})$ does not change for any location l. The Run Constraint RCS2 ensures that if a value of an uncommitted *write* to location l has already been flushed to $\mathit{stableDb}(l)$ and therefore $\mathit{stableDb}(l) \neq \mathit{undef}$, then there is also a record of the *write* to l in *stableLog*.

The decision for which locations the cache values should be removed upon flushing is reflected by a monitored function $\mathit{cacheRemove}\colon \mathit{LOC} \to \mathit{BOOL}$ subject to the following constraint:

Run Constraint RCS3. $\mathit{cacheRemove}(l) = \mathit{false}$ upon reading/writing to l and upon undoing/redoing an l-record.

Without this constraint inconsistent update sets could be created, for example, when in the same computation step a value v is written to a location l of the *cache* with the update $((\mathit{cache}, l), v)$ and the previous value of l is flushed and removed from the *cache* with the update $((\mathit{cache}, l,), \mathit{undef})$.

The rules of the refined ASM DbRecovery′ are obtained from those of DbRecovery by extending the two rules Read and Fail and by detailing the macros as follows. In Write, to WriteIntoVolatileLoc to *currDb* is replaced by *cache*, to RecordLoc the fields of the next free *LOG* entry are written with the write information by Writelog. Commit(t) becomes inserting t into the list of *committed* transactions. Abort(t) means to Undo all *log* entries issued by t. The predicate *toBeFlushed* is strengthened to flush only values from locations which have a defined *cache* value, in Flush *currDb* is replaced by *cache* and a cache removal is added where requested by the monitored function *cacheRemove*. In Fail, the update of *log* to its part in stable log memory is added and to RestoreStableValue(l) becomes $\mathit{cache}(l) := \mathit{undef}$. To RestoreCommValue$(l)$ becomes to Redo the *cache* value for l from its last record in *log* in case (the transaction of) this record is committed, and otherwise to Undo it from the record provided by *undoRcd*.

```
$\textsc{WriteIntoVolatileLoc} = \mathit{cache}(\mathit{loc}(\mathit{currOp})) := \mathit{val}(\mathit{currOp})$
$\textsc{RecordLoc} = \textsc{Writelog}(\mathit{next}(\mathit{max}(\mathit{log})))$
$\textsc{Commit}(t) = \mathit{committed}(t) := \mathit{true}$
$\textsc{Abort}(t) = \mathbf{forall}\ r \in \mathit{log}\ \mathbf{with}\ \mathit{issuer}(r) = t\ \mathbf{do}\ \textsc{Undo}(r)$
$\mathit{toBeFlushed}(l) \iff (\mathit{cacheFlush}(l) = \mathit{true}\ \mathbf{and}\ \mathit{cache}(l) \neq \mathit{undef})$
$\textsc{Flush}(l) =$
  $\mathit{stableDb}(l) := \mathit{cache}(l)$
  $\mathbf{if}\ \mathit{cacheRemove}(l)\ \mathbf{then}\ \mathit{cache}(l) := \mathit{undef}$
$\textsc{RestoreStableValue}(l) = \mathit{cache}(l) := \mathit{undef}$
$\textsc{RestoreCommValue}(l) = \mathbf{let}\ r = \mathit{lastRcd}(l, \mathit{log})\ \mathbf{in}$
```

```
    if $r \neq undef$ then
      if $committed(issuer(r))$ then Redo$(r)$ else Undo$(r)$
  Read = let $l = loc(currOp)$ in
    if $cache(l) = undef$ then $cache(l) := stableDb(l)$
  Fail = forall $l \in LOC$ do RestoreStableValue$(l)$
    $log := stableLog$
  Writelog$(r)$ =
    $issuer(r) := issuer(currOp)$
    $loc(r) := loc(currOp)$
    $afterImage(r) := val(currOp)$
    $log := log \cup \{r\}$
  Redo$(r)$ = $cache(loc(r)) := afterImage(r)$
  Undo$(r)$ = $cache(loc(r)) := afterImage(undoRcd(r))$
```

For the definition of equivalence between the refined ASM DbRecovery′ and DbRecovery there is a one-to-one correspondence between homonymous rules. Furthermore, we stipulate for the equivalence notion that homonymous locations are identical via the following definition of *currDb*, *commDb*, *writeSet* in DbRecovery′:

- *currDb* as defined on p. 128,
- *commDb* as defined on p. 129,
- If t is active, then

$$writeSet(t) = \{l \in LOC \mid \exists r \in log \mid issuer(r) = t \wedge loc(r) = l\}.$$

Theorem 3.2.2 (Recovery equivalence theorem). In runs of the machines DbRecovery and DbRecovery′ started in equivalent initial states, in every pair of corresponding states corresponding locations are equivalent.

Proof. It follows by a run induction from the following equivalence of the effect of every refined DbRecovery′-operation macro to the effect of the homonymous DbRecovery-macro in every state S DbRecovery′ reaches:

Flush: if $cacheFlush(l)$ holds in S, then $stableDb(l)_{S+1} = currDb(l)_S$,

Write: if t writes v to l in S, then $currDb(l)_{S+1} = v$ and $l \in writeSet(t)_{S+1}$,

Commit: if t commits in S and $l \in writeSet(t)_S$, then $commDb(l)_{S+1} = currDb(l)_S$,

Abort: if t aborts in S and $l \in writeSet(t)_S$, then $currDb(l)_{S+1} = commDb(l)_S$,

Fail: if $fail = true$ in S, then $currDb(l)_{S+1} = stableDb(l)_S$ for every l,

Recover: if the machine recovers in S, then $currDb(l)_{S+1} = commDb(l)_S$ for every l.

It therefore remains to prove those equations in DbRecovery′ together with the equivalence of corresponding locations.

ad **FLUSH**. By the definition of $currDb(l)$ on p. 128 there are two cases to distinguish.

Case 1. $cache(l)_S = undef$. Then by definition $currDb(l)_S = stableDb(l)_S$. Since by applying Flush $stableDb(l)$ is not changed, $stableDb(l)_{S+1} = stableDb(l)_S = currDb(l)_S$.

Case 2. Otherwise. Then by definition $currDb(l)_S = cache(l)_S$ and by firing Flush $stableDb(l)_{S+1} = cache(l)_S = currDb(l)_S$.

ad **WRITE**. Write of v to l in S yields $cache(l)_{S+1} = v$, so that by definition $currDb(l)_{S+1} = v$. log has a new l-record with issuer t, so that $l \in writeSet(t)_{S+1}$.

ad **FAIL**. Fail in S yields $cache(l)_{S+1} = undef$ for every l, so that by definition $currDb(l)_{S+1} = stableDb(l)_S$.

ad **RECOVER**. We distinguish two cases depending on whether the function $lastRcd(l, log)$ is defined.

Case 1: in S there is no l-record in log. Then there has been no committed write to l, so that $commDb(l)_S = undef$. As a consequence of the Fail in state $S - 1$ we have $cache(l)_S = undef$. The Recover in state S does not update $cache(l)$. Hence, $cache(l)_{S+1} = undef$. By the Run Constraint RCS2, we have $stableDb(l)_S = undef = stableDb(l)_{S+1}$. Hence we obtain, $currDb(l)_{S+1} = stableDb(l)_{S+1} = undef = commDb(l)_S$.

Case 2: $r = lastRcd(l, log)$ is defined. Let $t = issuer(r)$.

Case 2.1. $committed(t)_S = true$. Then $cache(l)_{S+1} = afterImage(r)_S = commDb(l)_S$ by Redo and definition.

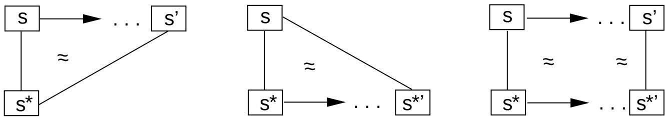
Case 2.2. Otherwise, $currDb(l)_{S+1} = cache(l)_{S+1} = afterImage(r')_S$ for the last committed l-record $r' < r \in log_S$ (by Undo). Since r is the last l-record in log_S though not committed, r' is the last committed l-record in log_S. Thus $afterImage(r')_S = commDb(l)_S$.

ad **COMMIT**. By $l \in writeSet(t)_S$ there is a record in log with issuer t and location l, so that t writes a value v to l in some state $R < S$. By the strictness constraint on runs, t is active in S. By Lemma 3.2.9 this implies for the last l-record $r \in log_S$ that $currDb(l)_S = v = afterImage(r)_S$. Applying Commit in S yields $committed(t)_{S+1} = true$. Therefore, $currDb(l)_S = afterImage(r)_S = commDb(l)_{S+1}$ by the definition of $commDb$.

ad **ABORT**. By $l \in writeSet(t)_S$ there is an l-record $r \in log_S$ with issuer t. By strictness, t is active in S. By Lemma 3.2.9 r is the last l-record $r \in log_S$. Undo in S yields $cache(l)_{S+1} = afterImage(r')_S$ for the last committed l-record $r' < r$ in log_S. Since r is the last l-record in log_S, r' is the last committed l-record in log_S. Thus $afterImage(r')_S = commDb(l)_S$ by the definition of $commDb$. Therefore $currDb(l)_{S+1} = cache(l)_{S+1} = commDb(l)_S$.

□

Fig. 3.13 Components of ASM refinement diagrams



Lemma 3.2.9 (Persistence of write effects of active transactions). If t writes v to l in state R and is still active in state $S > R$, then the last l-record in log_S is the record r written at R and $currDb(l)_S = v$.

Proof. By induction on the number of states $S > R$. See [260, Lemma 5]. □

3.2.3 Decomposing Refinement Verifications

We formulate here Schellhorn's [387] scheme for establishing invariants to prove the correctness of an ASM refinement. The idea consists in decomposing the commuting diagram in Fig. 2.1 into more basic diagrams with end points s, s^* which satisfy an invariant $\approx$ implying the to be established equivalence $\equiv$. The method is to follow the two runs, for each pair of corresponding states – not both final – satisfying $\approx$, looking for a successor pair $s', s^{*\prime}$ (of corresponding states satisfying $\approx$). Three cases are possible for such run extensions: only one of the two runs can be extended or both are extendable. These cases give rise to three types of basic diagram, as shown in Fig. 3.13:

- $(m, 0)$-triangles representing a computation segment which leads in $m > 0$ steps to an $s' \approx s^*$,
- $(0, n)$-triangles representing a computation segment which leads in $n > 0$ steps to an $s^{*\prime} \approx s$,
- (m, n)-trapezoids representing computation segments which lead in $m > 0$ steps to an s' and in $n > 0$ steps to an $s^{*\prime}$ such that $s' \approx s^{*\prime}$.

Definition 3.2.4 (Forward simulation condition). *FSC* is defined as the following run condition: for every pair (s, s^*) of states, if $s \approx s^*$ and not both are final states, then

- either the abstract run can be extended by an $(m, 0)$-triangle leading in $m > 0$ steps to an $s' \approx s^*$ satisfying $(s', s^*) <_{m0} (s, s^*)$ for a well-founded relation $<_{m0}$ limiting successive applications of $(m, 0)$-triangles,
- or the refined run can be extended by a $(0, n)$-triangle leading in $n > 0$ steps to an $s^{*\prime} \approx s$ satisfying the condition $(s, s^{*\prime}) <_{0n} (s, s^*)$ for a well-founded relation $<_{0n}$ limiting successive applications of $(0, n)$-triangles,[19]
- or both runs can be extended by an (m, n)-trapezoid leading in $m > 0$ abstract steps to an s' and in $n > 0$ refined steps to an $s^{*\prime}$ such that $s' \approx s^{*\prime}$. Any of the three possible subcases $m < n$, $m > n$ (typical for optimizations) or $m = n$ is allowed here.

In [387] Schellhorn proves the following theorem, the basis for the machine verification in KIV [388, 389] of the proven to be correct hierarchy of ASMs relating Prolog to its compilation to WAM code [132].

Theorem 3.2.3. (Decomposition of ASM refinement diagrams). M^* is a correct refinement of M with respect to an equivalence notion $\equiv$ and a notion of initial/final states if there is a relation $\approx$ (a coupling invariant) such that

1. the coupling invariant implies the equivalence,
2. each refined initial state s^* is coupled by the invariant to an abstract initial state $s \approx s^*$,
3. the forward simulation condition FSC holds.

Problem 10 (ASM refinement theory). Develop the refinement theory of ASMs further, providing practical refinement schemes which reflect frequently used patterns. Implement these schemes in a theorem-proving system. Compare them to specific refinement schemes in the literature (see [24, 334, 337, 25, 167, 176]).

3.2.4 Exercises

Exercise 3.2.1. Show that in the refinement Def. 3.2.1, the sequences of corresponding states can be chosen to be minimal in the sense that between two sequence elements there are no other equivalent states, i.e. there are no $i_k < i < i_{k+1}, j_k < j < j_{k+1}$ such that $S_i \equiv S^*_j$.

Exercise 3.2.2. Define a data refinement of sets to lists. Show why the abstraction function is not necessarily total nor injective.

Exercise 3.2.3. (⇝ CD) Refine BACKTRACKto an ASM for tree adjoining grammars, generalizing Parikh's analysis of context free languages by "pumping" of context free trees from *basis trees* (with terminal yield) and *recursion trees* (with terminal yield except for the root variable).

Exercise 3.2.4 (Communication via offer and request matching). (⇝ CD) The communication in the programming language Occam is realized via channels, each of which for a communication to take place requires exactly one reader x and one writer y [20]: the reader is *pos*itioned to execute an *instr*uction $c?v$ to read into $val(v, env(x))$ the value coming through a channel $bind(c, env(x))$, and the writer is *pos*itioned to execute an *instr*uction $d!t$ to write $val(t, env(y))$ into that channel $bind(d, env(y))$. In the instantaneous communication rule the channel synchronizes the reader's request

[19] This well-founded order condition is guaranteed in refinements of event-based B systems by the VARIANT clause, containing an expression for a natural number which has to be shown to decrease for each rule application [11].

[20] Their uniqueness is guaranteed by constraints on the channels and shared variables in Occam programs; see [105, Sect. 2].

and the writer's offer without being updated itself, serving only as a medium which realizes an agreement between the two processes (matching request and offer).

```
OccamCommunication$(x, v, c, y, t, d) =$
  if $mode(x) = running$ and $instr(pos(x)) = c?v$ and
    $mode(y) = running$ and $instr(pos(y)) = d!t$ and
    $bind(c, env(x)) = bind(d, env(y))$
  then $\{val(v, env(x)) := val(t, env(y)),$ proceed $x,$ proceed $y\}$
  where proceed $z = (pos(z) := next(pos(z)))$
```

The rule can be refined by introducing a channel agent which establishes the communication once a *reader* and a *writer* have arrived independently, recording as the channel attributes their identity, the variable and the message value. Prove the machine OccamChannel consisting of the following three rules to be a correct refinement of OccamCommunication; determine the corresponding states, the locations of interest, their equivalence, the refinement type.

```
In$(x, c, v) =$ if $mode(x) = running$ and $instr(pos(x)) = c?v$ then
  put $x$ asleep at $next(pos(x))$
  $\{reader(bind(c, env(x))) := x, var(bind(c, env(x))) := v\}$
Out$(x, c, t) =$ if $mode(x) = running$ and $instr(pos(x)) = c!t$ then
  put $x$ asleep at $next(pos(x))$
  $\{writer(bind(c, env(x))) := x, msg(bind(c, env(x))) := val(t, env(x))\}$
Chan$(c) =$ if $reader(c), writer(c) \neq nil$ then
  $val(var(c), env(reader(c))) := msg(c)$
  {wake up $reader(c),$ wake up $writer(c),$ clear $c$}
where
  put $z$ asleep at $p = \{mode(z) := sleeping, pos(z) := p\}$
  wake up $z = (mode(z) := running)$
  clear $c = \{reader(c) := nil, writer(c) := nil\}$
```

Exercise 3.2.5 (Communication via handshaking). In process algebra systems like LOTOS [68], process communication is specified via so-called *gates* where the participating agents have to "agree on offered values". This can be viewed as a special case of shared memory communication, namely via gate locations g shared for reading and/or writing. The general scheme for two processes P, Q is as follows, where we use predicates α, β to determine the choice the processes may have for agreeing upon a consistent update of gate g with a value determined by terms s, t:

```
Handshaking$(P, \alpha, s, Q, \beta, t) =$
  choose $x$ with $\alpha(x)$ in $\{g := s(x), P(x)\}$
  choose $y$ with $\beta(y)$ in $\{g := t(y), Q(y)\}$
```

Use Handshaking to express the parallel execution of $\{g := 17, P\}$ and **choose** $y \in \mathbb{N}$ **in** $\{g := y, Q(y)\}$. Simulate OccamCommunication by a Handshaking ASM for appropriately defined $P, \alpha, s, Q, \beta, t$ (assuming that at each moment for each channel c at most one process is allowed to read and at most one to write to c). See also the description by AspHandshaking on p. 185.

Exercise 3.2.6. ($\leadsto$ CD) Formulate appropriate conditions under which EarlyChoice is correctly refined by LateChoice and prove the correctness of the refinement.

Exercise 3.2.7. Prove that all nodes which are reachable from *source* by a path of length $\leq t$ are labeled as visited by ShortestPath$_1$ in $\leq t$ steps.

Exercise 3.2.8. ($\leadsto$ CD) Show that each run of ShortestPath$_1$ can be simulated by a breadth-first run of ShortestPath$_2$ producing the same labelings of nodes as visited.

Exercise 3.2.9. Prove that the refinement to ShortestPath$_4$ preserves the correctness and termination of ShortestPath$_3$.

Exercise 3.2.10. Refine ShortestPath$_4$ to efficient code.

Exercise 3.2.11. ($\leadsto$ CD) Formulate a stack ASM with operations *Add(e)* and *Remove* using a static concatenation function. Data refine this machine by dynamic functions *head, next*. Prove that the refinement is correct in the sense that corresponding runs have equivalent stacks and that executions of *Add(e), Remove* operations correspond to each other.

Exercise 3.2.12. ($\leadsto$ CD) Refine DbRecovery$'$ to M^* by *computing derived functions*, e.g. $afterImage(undoRcd(r))$. Formulate the correctness of this refinement and prove it.

Exercise 3.2.13. ($\leadsto$ CD) Refine M^* of the preceding exercise to $M^\sharp$ by sequentializing **forall** as iterated log scanning in Fail, Abort, Recover. Formulate and prove the correctness of this refinement.

Exercise 3.2.14. ($\leadsto$ CD) Refine $M^\sharp$ of the preceding exercise to $M^\flat$ by *implementing run conditions*, e.g. the following two constraints on cache policy for log.

RCS1. The records of all last committed writes must be in stable storage (for reinstallment during recovery), formally: $lastCommRcds \subseteq stableLog \subseteq log$. Hint: Refine Commit by flushing *log* to *stableLog*.

RCS2. If there is no *stableLog* record of a write to l, then $stableDb(l) = undef$. Hint: maintain for each l the last log record with l (refining Write); constrain flushing from *cache* to *stableDb* by a check of the index of the last record for the location of the value to be flushed against the index of the last record in stable storage.

Formulate and prove the correctness of this refinement.

3.3 Microprocessor Design Case Study

In this section we illustrate the ASM method for ground model construction and stepwise refinement by a real-life architecture design case study, namely the provably correct optimization of a microprocessor, starting from its serial ground model at the register transfer level and leading to its pipelined parallel version. The example also illustrates how one can use the ASM method to deal with hardware/software co-design problems in an accurate but nevertheless transparent way. The reader who is not interested in architecture problems may skip this section, which is a re-elaboration of [119]. For applications of the method to commercial processors and for its extension to derive from the ASM models simulators and debuggers see [411] and the references at the end of the chapter.

Building a ground model in this context means modeling a microprocessor in architectural terms, avoiding any overhead which does not belong to the application domain but only to the needs of the formalization (e.g. by FSMs tailored for model checking or by logical theories which support mechanical verification in PVS or HOL). Refinement means to optimize the processor model by standard techniques, verifying on-the-fly the correctness of the optimization. As an example we choose pipelining, a key implementation technique to make fast CPUs. In fact the guideline for the refinement steps we introduce below is to mimic as closely as possible state-of-the-art incremental hardware design techniques and forms of reasoning the designers use to justify their design, which typically are expressed in terms of the local state to describe an overall "global-state" transformation. As a result the approach introduced in this section can be used (a) to teach the basics of RISC processors and more generally of architectures and their implementation, (b) to refine the ASM models for architectures further to support their mechanical verification (see the analysis of the models below in KIV and PVS [217, 407]), their validation (through experimentation with a machine-executable version,[21] and their proven-to-be-correct synthesis by correctness-preserving transformations (see [279]).

We concentrate our attention on control, where notoriously most errors are found during the design of a processor. As a typical microprocessor we consider the processor DLX developed by Hennessy and Patterson [278] exhibiting the core features of RISC processors with a standard five-stage instruction pipeline. Pipelining provides a simultaneous execution of multiple instructions exploiting independences between segments into which instruction execution can be decomposed, as a result of which the overall execution speed for programs is improved despite a possible slow-down for the latency of single instructions. Since pipelining is not visible to the programmer, it is all the more crucial to ensure that the semantics of instructions is preserved

[21] An architecture similar to the ASM model DLX^{pipe} developed below for the pipelined version of DLX has been implemented in [168].

by the concurrency inherent in this technique. We prove the correctness of Hennessy and Patterson's pipelined processor with respect to its serial model which comes with a one-instruction-at-a-time view of the processor. The task therefore consists in (a) defining a ground model DLX^{seq} for the datapath and the serial control of DLX, (b) refining this model to the pipelined version DLX^{pipe} of DLX (in fact in three steps, gradually exposing the complications of pipelining), (c) defining in which sense (and under which conditions on the underlying hardware and compiler) the machines are equivalent and (d) justifying the correctness of the refinements. In other words we are going to make the following statement precise enough to make it subject to a mathematical proof.

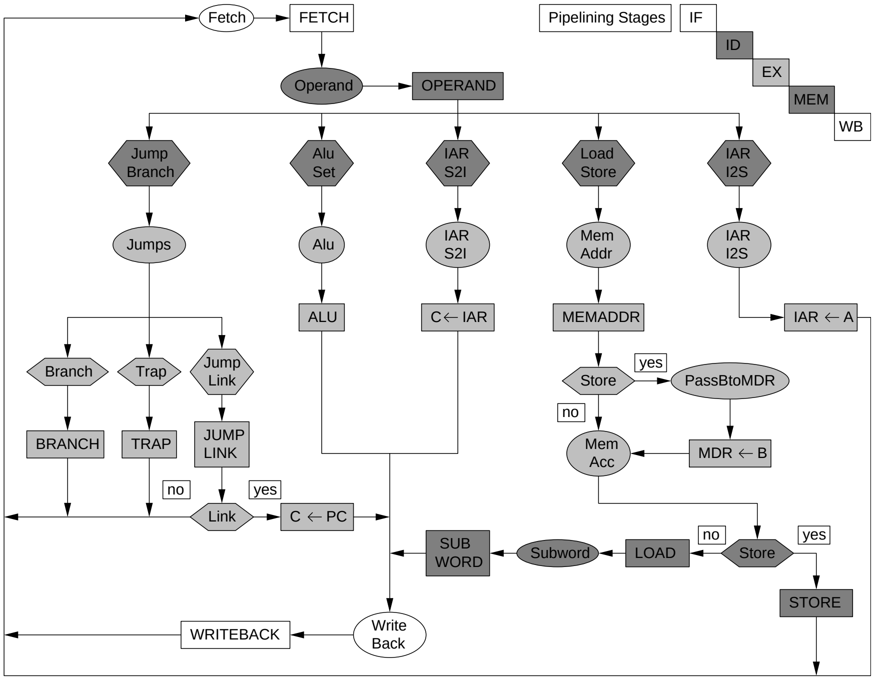
Theorem 3.3.1 (Correctness of DLX^{pipe}**).** For each DLX program, the result of its serial (one-instruction-at-a-time) execution in the ground model DLX^{seq} is the same as the result of its pipelined (up to five instructions at–a–time) execution in the pipelined model DLX^{pipe}.

DLX incorporates pipelining techniques which resolve structural, data and control hazards: instruction scheduling, forwarding, new hardware links with additional control logic, and stalling. This suggests three application-domain-driven refinement steps of the ground model DLX^{seq}: a parallel version DLX^{par} which resolves structural hazards, a refinement to DLX^{data} which resolves data hazards, and a refinement to DLX^{pipe} which resolves control hazards. Each refined model is proven to be correct with respect to the preceding model, lifting piecemeal the compiler assumptions which in the abstract models guarantee hazard-freeness for the conflict types that the model does not resolve. The proof principles we use are inductions and case distinctions which correspond to the pipelining conflict types, that is to say to standard methods to solve the conflicts and to justify the solution.

3.3.1 Ground Model DLX^{seq}

DLX comprises instructions for arithmetical and set operations, absolute and conditional jumps (branches), interrupts, and memory access which we classify in corresponding sets *Alu, Set, Jump, Branch, Interrupt*, $Mem = Load \cup Store$. We abstract from particularities of the instruction format by working with abstract functions which provide what is encoded in an instruction, namely its operation *opcode*, its first and second operand *fstop*, *sndop* (names of registers in an abstract register file whose contents are given by a function *reg*),[22] the destination register *dest* for the result computed by an instruction, and a flag *iop* indicating whether the instruction operation is an immediate one or not, i.e. whether it works with an immediate argument value *ival* which is encoded in the instruction itself and not taken from the

Fig. 3.14 The serial DLX model DLX^{seq}



register file. As for the size of the register file, we also make no assumption about the bandwidth of the memory which is represented by a function *mem*.

In the *one-instruction-at-a-time-view* DLX executes at each clock cycle exactly one instruction and does this in up to five successive steps. During *I*nstruction *F*etch an instruction is fetched from *mem*ory to a datapath register *IR*, followed by *I*nstruction *D*ecoding which includes extracting the operands from the register file registers *fstop*, *sndop* – we suppress the standard argument *IR*, writing *nthop* instead of *nthop*(*IR*) – and assigning them to register file exit ports A, B which are connected to the ALU input. During the *EX*ecution proper, in the case of an *Alu* or *Set* instruction the ALU is used to compute the value of the operation *opcode* for the input values in A, B and to assign it to the register file entry port C. In the case of a

[22] The practice to notationally suppress the standard interpretation of registers by the register content function *reg* and to write $R \leftarrow S$ for $reg(R) := reg(S)$ comes close to ASM terms, where a register can be interpreted as a 0-ary function whose value represents the register content, so that one could also write directly $R := S$. The difference between the ASM notation and the notation which is common in architecture texts becomes visible in examples like $dest(IR) \leftarrow C$, which stands for $reg(dest(reg(IR))) := reg(C)$ and not for $dest(IR) := C$, since the range of *dest* is the register file and not the register contents.

Jump, *Branch* or *Mem*ory instruction the needed address is computed from $A, PC, ival$ and stored in PC (involving also an interrupt address register IAR in the case of an interrupt instruction) or in a memory address register MAR. In the MEMory stage the memory data register MDR is used for loading from or storing to *mem* at MAR, followed in case of a *Load* instruction by transferring the relevant portion of the MDR-value to C. Finally, in the Write$B$ack stage the result value is written back from C to the *dest*ination register in the register file. The stages of this standard execution of a single instruction are reflected by the ground model control states in Fig. 3.14. The macros for DLX^{seq} are defined in Table 3.1 (together with their refinement for DLX^{par}, to be explained below).

For the sake of completeness we explain some more details which appear in Fig. 3.14. The rhombs denote tests of whether the fetched instruction in IR is of the indicated type, formally $JumpBranch = opcode(IR) \in Jump \cup Branch$, etc. The jump instructions of DLX comprise a system jump called TRAP – which saves the current value of PC in the interrupt address register IAR and updates PC to $ival(IR)$ – as well as jumps which establish also a so-called “link”, meaning that they save the current value of PC in C from where it will be written back to the destination register in the register file. We write $JumpLink = Jump \setminus \{\mathtt{TRAP}\} \supseteq Link$. DLX has besides TRAP two interrupt instructions $Interrupt = \{\mathtt{MOVS2I}, \mathtt{MOVI2S}\}$ to move addresses between the register file and the special interrupt address register IAR. We use *next* to denote the next instruction address. When no confusion is to be feared, we omit the superscripts *seq* or *par*.

The idea of pipelining is to fire simultaneously the rules of all stages, one per instruction in its current stage, exploiting their independence where possible. In doing this one has to resolve conflicts which may arise due to a simultaneous access to a hardware resource – the so-called structural hazards – or because an instruction execution may need data which have to be computed by a preceding instruction whose pipelined execution is not yet terminated – the so-called data hazards – or because the instruction fetched after a jump or branch instruction may not be the one to jump to – the so-called control hazards. We split this task into three subtasks by stepwise refining DLX^{seq} to first DLX^{par}, then DLX^{data} and then DLX^{pipe}, verifying on-the-fly at each step the corresponding conflict resolution.

3.3.2 Parallel Model DLX^{par} Resolving Structural Hazards

We refine here DLX^{seq} to DLX^{par}, obtained by replacing the global control states by local rule guards, which are applied to instructions in their pipelining stages, and by detailing the rule macros to avoid structural conflicts. In addition we illustrate what is called “speeding up pipe stages”, namely by incorporating the SubWord-step into the DLX^{par}-rule WB. The motivation for such speed ups derives from the fact that when all pipe stages proceed simultaneously, the time needed for moving an instruction one step

Table 3.1 The macros for DLX^{seq} and DLX^{par}

	DLX^{seq}	DLX^{par}
Fetch	$IR \leftarrow mem(PC)$ $PC \leftarrow next(PC)$	$IR \leftarrow mem_{instr}(PC)$ **if not** $jumps$ **then** $PC \leftarrow next(PC)$
Operand	$A \leftarrow fstop$ $B \leftarrow sndop$	$A \leftarrow fstop$ $B \leftarrow sndop$

Alu	**if** $iop(opcode)$ **then**
	$C \leftarrow opcode(A, ival)$
	else
	$C \leftarrow opcode(A, B)$
		if $opcode(IR1) \in Alu \cup Set$ **then**
		if $iop(opcode(IR1))$ **then**
		$C \leftarrow opcode(IR1)(A, ival(IR1))$
		else
		$C \leftarrow opcode(IR1)(A, B)$

Memaddr	$MAR \leftarrow A + ival$	**if** $opcode(IR1) \in Load \cup Store$ **then** $MAR \leftarrow A + ival(IR1)$
PassBtoMDR	$MDR \leftarrow B$	**if** $opcode(IR1) \in Store$ **then** $SMDR \leftarrow B$
Movs2i	$C \leftarrow IAR$	**if** $opcode(IR1) = \texttt{MOVS2I}$ **then** $C \leftarrow IAR$
Movi2s	$IAR \leftarrow A$	**if** $opcode(IR1) = \texttt{MOVI2S}$ **then** $IAR \leftarrow A$

Branch	**if** $reg(A) = 0$ **then**
	$PC \leftarrow PC + ival$
		if $opcode(IR1) \in Branch$ **then**
		if $reg(A) = 0$ **then**
		$PC \leftarrow PC1 + ival(IR1)$

Trap	$IAR \leftarrow PC$ $PC \leftarrow ival$	**if** $opcode(IR1) = \texttt{TRAP}$ **then** $IAR \leftarrow PC1$ $PC \leftarrow ival(IR1)$

Jumplink	**if** $iop(opcode)$ **then**
	$PC \leftarrow PC + ival$
	else $PC \leftarrow A$
		if $opcode(IR1) \in Jump \cup Link$ **then**
		if $iop(opcode(IR1))$ **then**
		$PC \leftarrow PC1 + ival(IR1)$
		else $PC \leftarrow A$

Link	$C \leftarrow PC$	**if** $opcode(IR1) \in Link$ **then** $C \leftarrow PC1$
Load	$MDR \leftarrow mem(MAR)$	**if** $opcode(IR2) \in Load$ **then** $LMDR \leftarrow mem(MAR)$
Store	$mem(MAR) \leftarrow MDR$	**if** $opcode(IR2) \in Store$ **then** $mem(MAR) \leftarrow SMDR$
Subword	$C \leftarrow opcode(MDR)$	
Writeback	$dest \leftarrow C$	**if** $opcode(IR3) \in$ $Alu \cup Set \cup Link \cup \{\texttt{MOVS2I}\}$ **then** $dest(IR3) \leftarrow C1$ **if** $opcode(IR3) \in Load$ **then** $dest(IR3) \leftarrow opcode(IR3)(LMDR)$

down the pipeline is a machine cycle, so that the length of the latter is determined by the time required for the slowest pipe stage. The parallelization of the rules of all five instruction stages leads to the following definition. We use IF, ID, EX, MEM, WB to denote a pipe stage as well as the submachine corresponding to a stage. When we want to denote a pipe stage PS of an *instr*uction we also write $PS(instr)$; we are confident that given the context any confusion with the application of machine PS to $instr$ is avoided.

$$\begin{aligned}
&\mathrm{DLX}^{par} = \{\mathrm{IF}, \mathrm{ID}, \mathrm{EX}, \mathrm{MEM}, \mathrm{WB}\}\ \textbf{where}\\
&\mathrm{IF} \quad\ \ = \textsc{Fetch}\\
&\mathrm{ID} \quad\ \ = \{\textsc{Operand}, \textsc{Preserve}\mathrm{ID}\}\\
&\mathrm{EX} \quad\ \ = \{\textsc{Alu}, \textsc{Memaddr}, \textsc{PassBto}\mathrm{MDR}, \textsc{Movi2s}, \textsc{Movs2i}\} \cup\\
&\qquad\qquad \{\textsc{Branch}, \textsc{Trap}, \textsc{Jumplink}, \textsc{Link}, \textsc{Preserve}\mathrm{EX}\}\\
&\mathrm{MEM} = \{\textsc{Load}, \textsc{Store}, \textsc{Preserve}\mathrm{MEM}\}\\
&\mathrm{WB} \quad = \textsc{Writeback}
\end{aligned}$$

We explain now through commenting upon the refined definition of the macros in Table 3.1[23] how DLX^{par} resolves the structural conflicts. Four groups of resources have to be doubled so that any combination of operations can occur in pipe stages which are executed simultaneously in one clock cycle, namely the memory access (to fetch instructions independently from other load or store operations), an addition mechanism (to increment the program counter PC independently from other ALU operations), the memory data register (for overlapping load and store instructions), and latches for the instruction register IR, for PC and for the register file entry C (to hold values which are needed later in a pipeline cycle of an instruction).

A memory access conflict between instruction fetching and load/store instructions is avoided by increasing the memory bandwidth, introducing an additional memory access function mem_{instr}, used only for fetching instructions and assumed to be a subfunction of mem. This models the Harvard-architecture principle and abstracts from any particular implementation feature related to using separate instruction and data caches.[24] To avoid using the ALU for incrementing PC a separate PC-incrementer is provided, another abstract function $next$. The guard defined below for the $\textsc{Fetch}$ macro prevents PC from being updated by the IF–rule when a jump or branch rule has to update it in its execution phase.

$$\begin{aligned}
&jumps = opcode(IR1) \in Jump\ \textbf{or}\\
&\quad (opcode(IR1) \in Branch\ \textbf{and}\ opcode(IR1)(A) = true)
\end{aligned}$$

[23] When we want to differentiate between the different definitions for a macro we add as a superscript the corresponding refinement.

[24] The proof below uses the fact that DLX programs are not self-modifying, i.e. that mem_{instr} is static (defined by the initialization) whereas mem is dynamic. For a formalization of this property in the context of the KIV verification of Theorem 3.3.2 see [217].

Some of the values which appear during the execution of an instruction at a certain pipe stage are needed at later pipe stages and have to be copied in order not to get overwritten by a subsequent instruction occurring in the pipeline. This is the case for segments of IR, for PC and C. Since we abstract from the instruction format and decoding details we use additional registers $IR1$, $IR2$, $IR3$, $PC1$, $C1$ to keep copies through the pipe stages EX, MEM, WB by the following preservation rules (see below for the instruction scheduling reasons which motivate the update of $PC1$).

$$\begin{aligned}&\textsc{Preserve}\mathrm{ID} = \{IR1 \leftarrow IR, PC1 \leftarrow next(next(PC))\}\\&\textsc{Preserve}\mathrm{EX} = IR2 \leftarrow IR1\\&\textsc{Preserve}\mathrm{MEM} = \{IR3 \leftarrow IR2, C1 \leftarrow C\}\end{aligned}$$

In DLX^{seq} the memory data register MDR is the only interface between the register-file and the memory and serves for both loading and storing. In DLX^{par} a load instruction I which in the pipeline immediately precedes a store instruction I' would compete with I' for writing into MDR in its pipe stage MEM (when I' in its pipe stage EX wants to write B into MDR). This resource conflict is resolved by doubling MDR into two registers $LMDR$ and $SMDR$ and by correspondingly refining the Mem-related rules. The new rule PassBtoMDR requires a new direct hardware link from the exit of B to the entry of $SMDR$ in order to avoid the use of the ALU for this data transfer. Similarly the price for speeding up the pipe stages by integrating Subword into WB is linking the exit of $LMDR$ directly (without passing through $C1$) to the entry of the register-file and adding to the latter a selector for choosing among $C1$ and (the required portion of) $LMDR$.

3.3.3 Verifying Resolution of Structural Hazards (DLX^{par})

As a first proof step to establish Theorem 3.3.1 we now provide a meaning and a proof for the correctness of the parallelization of DLX^{seq} stated in Theorem 3.3.2 below. The separation of the resolution of structural hazards, which results from the parallelization, from the resolution of data and control hazards is reflected by assuming here that the instruction scheduling by the underlying compiler prevents data or control hazards from occurring. In the following refinement steps this assumption will be lifted first for data and then for control hazards.

Theorem 3.3.2 (Correctness of DLX^{par}). For an arbitrary DLX program P, let C be the DLX^{seq}-computation started with P and let C^{par} be the corresponding DLX^{par}-computation started with the corresponding program P^{par}. Then C and C^{par} compute the same result if C^{par} is data-hazard-free.

Table 3.2 The result locations for DLX instructions

Result location of *instr*	Updated by *instr* of type	To be collected after the end of pipe state
$\langle reg, dest(instr)\rangle$	$Alu \cup Set \cup Load \cup Link \cup \{\texttt{MOVS2I}\}$	$WB(instr)$
$\langle reg, IAR\rangle$	$\{\texttt{TRAP}, \texttt{MOVI2S}\}$	$EX(instr)$
$\langle reg, PC\rangle$	$Jump \cup Branch$	$EX(instr)$
	$\notin Jump \cup Branch$	$IF(instr)$
$\langle mem, arg\rangle$	$Store$	$MEM(instr)$

where arg = value of $reg(fstop(instr)) + ival(instr)$ when fetching $instr$

Proof. The task to accomplish consists in first defining (a) the correspondence and equivalence of computations in terms of their result, (b) the notion of data-hazard and data-hazard-freeness together with the instruction scheduling for programs P by transformed programs P^{par} which prevents control hazards from occurring, and then proving the statement. We start by defining the notions of result location, of used location and of relevant location, which allows us to recover DLX^{seq}–states from successive DLX^{par}–states by local projections.

We say that two computations *correspond* to each other if their initializations coincide on the common signature except where explicitly stated otherwise. For DLX^{seq}–initializations we assume $reg(IR) = undef$ and $ctl\_state = Fetch$, and for DLX^{par}–initializations we assume $reg(PC1) = reg(C1) = reg(IRi) = undef$ for $i = 1, 2, 3$. We say that a computation is initialized or starts with an instruction $instr$ if $mem(PC) = mem_{instr}(PC) = instr$.

We decompose computations as follows into segments which are relevant for a given instruction. An *instruction cycle* for an occurrence of *instr* is defined for DLX^{seq} as any subcomputation which starts with $ctl\_state = Fetch$ and $mem(PC) = instr$ and leads to the next state with $ctl\_state = Fetch$. In DLX^{par}–computations it is defined as any subcomputation which starts with fetching *instr* and ends with the first following pipe stage of *instr* at the end of which the values of all the *result locations* of *instr*, as defined below, are computed. We call this pipe stage the end stage of (that occurrence of) *instr*; whether it is $EX(instr)$, $MEM(instr)$ or $WB(instr)$ depends on *instr* and is defined in Table 3.2. The result of an occurrence of *instr* or of an *instr* cycle is defined by the values $f(a)$ which are assigned to the result locations $\langle f, a\rangle$ of *instr* as part of its execution. The result of a computation is defined as the sequence of the results of its instruction cycles.

The notion of instruction cycles allows us to provide the proof instructionwise by showing that in every pair $(C,\ C^{par})$ of corresponding DLX^{seq}, DLX^{par}–computations, any *corresponding instruction cycles* compute the same result.

Table 3.3 The critical stages for usage of locations in DLX

Location	Updated by *instr* of type	Critical use in stage
$\langle reg, nthop\rangle$	$Alu \cup Set \cup \{\texttt{MOVI2S}\} \cup Branch \cup Jump \setminus \{\texttt{TRAP}\}$	$EX(instr)$
	$Load \cup Store$	$MEM(instr)$
$\langle reg, IAR\rangle$	$\{\texttt{MOVS2I}\}$	$EX(instr)$
$\langle reg, PC\rangle$	$Jump \cup Branch$	$EX(instr)$
	$\notin Jump \cup Branch$	$IF(instr)$
$\langle mem, arg\rangle$	$Load$	$MEM(instr)$

where arg = value of $reg(fstop(instr)) + ival(instr)$ when fetching *instr*

The correspondence between instruction cycles in C and in C^{par} is defined by the order in which they occur: if $I_1, I_2, \ldots$ and $I'_1, I'_2, \ldots$ are the instruction cycles of C and C^{par}, respectively, in the order in which they appear there, then I_i and I'_i correspond to each other. By I_0, I'_0 we indicate the initial state. We say that I_0, I'_0 formalize the "result" of "no computation step".

The result of an instruction *instr* execution depends on the data used by the instruction, namely besides the static information encoded in *instr* the value of the PC, of the operands in the register file or of *mem* locations and possibly of the interrupt address register. In Table 3.3 we list the locations and instruction uses, together with the critical pipelining stage at which the correct value is needed. Conflicts can arise in two ways, namely (a) if I' uses, as one of its operands, the content of the destination register of a preceding instruction I in the pipe, and (b) if I' enters in the pipe shortly after a jump or branch instruction. To separate the analysis of data from that of control hazards we distinguish whether or not the data dependence concerns a jump or branch instruction.

Definition 3.3.1. Denote by $I \overset{1,2,3}{<} I'$ that an instruction cycle for I' is starting 1,2 or 3 steps after the one for I. We say that I' is *data-dependent* on I iff $I \overset{1,2,3}{<} I'$ and one of the following two conditions holds:

- $dest(I) \in \{fstop(I'), sndop(I')\}$ and $I' \notin Jump \cup Branch$
- $dest(I) = fstop(I')$ and $I' \in Jump \cup Branch$

A DLX^{par}-computation is *data hazard free* if it contains no occurrence of an instruction which is in the pipe together with an occurrence of an instruction on which it is data-dependent.

When a jump or branch instruction I is fetched, the two instruction cycles starting 1 and 2 steps later generate results which could spoil the continuation of the computation once the jump has been executed (after the stage $EX(I)$ which updates PC to its correct value). Therefore we assume in this section that P is transformed to P^{par} by placing two empty instructions after each jump or branch instruction. We stipulate that these empty instructions do not start an instruction cycle[25] and that they are put into new locations which are linked by the extended *next* function to preserve the given connections. This explains the update of $PC1$ in OPERAND^{par}.[26]

The decomposition of computations into instruction cycles allows us to prove the theorem instructionwise, using an inductive argument. For the induction step we need a stronger inductive hypothesis than that stated in the theorem. For its formulation we introduce a notion of *relevant locations* which allows us to locally relate the global serial states of DLX^{seq} and their pipelined virtual counterparts.[27] For the proof of the following lemma, which implies Theorem 3.3.2, the notion will be refined by admitting as additional irrelevant locations all those where in a hazardous situation – which may occur because it is not prevented any more by the instruction scheduler or compiler – the refined architecture takes care of providing the right values for the locations when needed. In this way we make it explicit where and how the compiler assumptions can be weakened if the hardware is strengthened to solve a given type of conflicts. This illustrates the potential of ASM modeling to describe hardware/software co-design problems.

Lemma 3.3.1 (DLX^{par} **Lemma).** For $n \geq 0$ let IC_n, IC_n^{par} be the nth instruction cycle in computations C and C^{par}, respectively.

Completeness: If C^{par} is data hazard free, then IC_n and IC_n^{par} are instruction cycles for the same occurrence of a DLX–instruction *instr* and start with the same values for the relevant locations used by *instr*.

Correctness: If IC and IC^{par} are instruction cycles for any *instr* in C and C^{par}, respectively, which start with the same values for the relevant locations used by *instr* and if *instr* is not data dependent on any instruction in the pipe, then IC and IC^{par} compute the same result.

A location l used by *instr* is called *relevant* except in the following two cases:

[25] This is a typical example of a clear though loosely expressed hypothesis, made for the sake of a proof, which for a mechanical proof verification has to be detailed further. For example see the KIV verification of Theorem 3.3.2 in [217].

[26] When a jump or branch instruction I is fetched at address $l = reg(PC)$, PC is updated to $l' = next(l)$, which in P^{par} is the address of *undef*. But in the EX(I)–stage the new value of PC must be computed on the basis of the value of $next(next(l'))$, i.e. the value of $next(l)$ for P. Therefore, $PC1$ has to store this value when PC – in case a jump or branch instruction has been fetched – contains the address of the empty instruction.

[27] This replaces the use of the flushing technique which is common in the pipelining verification literature.

Irrelev 1. $l = \langle reg, IAR\rangle$ and $instr =$ `MOVS2I` enters the pipe 1, 2 or 3 stages after an occurrence of `MOVI2S` or of $TRAP$.[28]

Irrelev 2. $l = \langle mem, arg\rangle$ and $instr \in Load$ enters the pipe 1, 2, or 3 stages after an occurrence of a *Store* instruction for the same value arg.[29]

We leave it as Exercise 3.3.1 to prove that this lemma implies Theorem 3.3.2. It therefore remains to prove the DLX^{par} Lemma. We proceed by induction on n. The claim for $n = 0$ holds by the assumption that C and C^{par} correspond to each other and therefore are initialized with the same static functions and with the same dynamic functions *reg*, *mem*. In the induction step, by inductive hypothesis, for each $i \leq n$, the ith instruction cycle IC^p_i in C^{par} starts with the same values for the relevant locations used by $instr_i$ as does the ith instruction cycle IC_i in C and they both compute the same result. Therefore IC_{n+1} and IC^p_{n+1} are instruction cycles for the same instruction *instr* and start with the same values for the relevant locations used by that instruction. To prove the correctness claim we first observe that due to the absence of stalls, the $(n+1)$th instruction cycle in C^{par} starts after the first step of IC^p_n in case the instruction $instr_n$ is neither a branch instruction with true branching condition nor a jump; otherwise the $(n+1)$th instruction cycle in C^{par} starts after the third step of IC^p_n due to the following *Jump Lemma* whose proof we leave as Exercise 3.3.2.

Lemma 3.3.2 (Jump Lemma). If a jump or branch instruction I is fetched in a DLX^{par}-computation, then the following two fetched instructions are empty and at stage $ID(I)$ the register $PC1$ is updated by the correct value to be used for the computation of the possible new PC-value in stage $EX(I)$.

Therefore, for the location PC the two machines produce the same result. Since the other result locations depend on the instruction type, the claim of the DLX^{par} lemma can be established by a case distinction. For each case one shows that through corresponding updates in IC and IC^{par}, by the

[28] No conflict can arise from using $\langle reg, IAR\rangle$ because `MOVS2I`, the only instruction which uses IAR, can never be in conflict with any preceding instruction. If I writes into IAR, then $I \in \{TRAP, \texttt{MOVI2S}\}$ and I writes into IAR in its third pipe stage; therefore if $I \overset{1,2,3}{<} I'$, then I has already written into IAR when I' uses it.

[29] No conflict can arise from using a memory location because load instructions – the only ones which use memory locations – can never be in conflict with preceding store instructions – the only ones which write into memory locations. Indeed if $I \in Store$ and $I' \in Load$, then I updates its result location $\langle mem, reg(fstop(I)) + ival(I)\rangle$ in its fourth pipe stage and I' reads the value of the location $\langle mem, reg(fstop(I)) + ival(I)\rangle$ in its fourth pipe stage too. Therefore if $I \overset{1,2,3}{<} I'$ and I' loads the value of the result location of I as updated by I, then I has already updated this result location when I' loads from there.

definition of the rule macros in Table 3.1 the same value is computed for the result location in question.[30] The proof exploits the fact that in ASM computations every controlled location keeps its value unchanged as long as it is not updated by a rule execution. □

3.3.4 Resolving Data Hazards (Refinement DLX^{data})

We now enrich the architecture DLX^{par} so that it can handle data hazards for linear code, i.e. conflicts for non-jump/branch instructions I', freeing the compiler from the work of avoiding those conflicts. This means to weaken the data-hazard-freeness assumption, allowing data hazards between I' and I to occur but guaranteeing their resolution by the architecture. The three standard methods to do this, namely the forwarding technique, new hardware links coming with appropriate additional control logic (multiplexers), and stalling, appear in the form of corresponding rule refinements. Thus the refined model is defined as follows, with locally defined refinements at each stage.

$$\mathrm{DLX}^{data} = \{\mathrm{IF}^{data}, \mathrm{ID}^{data}, \mathrm{EX}^{data}, \mathrm{MEM}^{data}, \mathrm{WB}^{data}\}$$

Therefore, in this section let I' be a non-jump/branch instruction which is *data-dependent* on I, i.e. satisfies the first case of Def. 3.3.1. We will specify the rule refinements piecemeal, following the case distinctions between whether the data hazard to be handled involves a memory access or not and whether the distance between the data dependent instructions in the pipe is 1, 2 or 3. In each case more hazardous locations are identified to be handled by a new rule branch and as a consequence to become “irrelevant”[31] for providing the correct argument values which are needed by I' in stage EX or MEM. In the case of no data dependence in the pipe, the rule branch of DLX^{par} applies which makes the refinement into a conservative one. In the case analysis we justify the correctness of the refined architecture DLX^{data} and therefore establish the following theorem.

Theorem 3.3.3 (Correctness of DLX^{data}). For an arbitrary DLX program P, let C be the DLX^{seq}-computation started with P and let C^{data} be the corresponding DLX^{data}-computation started with P^{par}. Assume that in

[30] In [217, 407] this case distinction is detailed for a mechanical verification using KIV.

[31] The notion of “irrelevant” locations realizes an abstract form of the standard implementation technique by a scoreboard to keep track of data dependences. See Exercise 3.3.7. Not surprisingly, the formalization of an equivalent of this notion for the mechanical verification of Theorem 3.3.2 in KIV and PVS has led to a rather complex predicate; see [217, 407]. Part of the complication stems from the need to formalize the basic semantical ASM property that in any computation a location remains unchanged unless it is updated (by a rule execution for controlled locations and by the environment for monitored locations).

C^{data} no occurrence of a jump or branch instruction is in the pipe together with an occurrence of an instruction on which it is data dependent. Then C and C^{data} compute the same result.

Proof. Localizing the hazardous locations and turning them into irrelevant ones allows us again to establish the claim instructionwise, proving the following analogon of Lemma 3.3.1. We leave it as Exercise 3.3.4 to show that this lemma implies Theorem 3.3.3.

Lemma 3.3.3 (DLX^{data} **Lemma).** For $n \geq 0$ let IC_n and IC_n^{data} be the nth instruction cycle in computations C and C^{data}, respectively.

Completeness: If C^{data} is free of hazards for jump or branch instructions, then IC_n, IC_n^{data} are instruction cycles for the same occurrence of a DLX–instruction I' and start with the same values for the relevant locations used by I'.

Correctness: Let IC, IC^{par} and IC^{data} be instruction cycles for any I' in C, C^{par} and C^{data}, respectively, which start with the same values for the relevant locations used by I'. The following holds:

Conservativity: If I' is not data dependent on any instruction in the pipe, then IC^{data}, IC^{par} compute the same result.

Refinement: If $I' \notin \mathit{Jump} \cup \mathit{Branch}$ is data dependent on some $I \overset{1,2,3}{<} I'$, then IC^{data}, IC compute the same result.

The proof for the completeness part of the DLX^{data} lemma follows the lines of the proof of Lemma 3.3.1, including the proof of the Jump Lemma 3.3.2. For the conservativity claim one can easily see that if I' is data independent of any I which precedes it in the pipe, then for each DLX^{data}-rule applied in IC^{data} for the execution of (this occurrence of) I', in any of its five pipe stages, the branch is taken which constitutes the corresponding DLX^{par}-rule. Since by assumption IC^{par} and IC^{data} start with the same values for the relevant locations used by I', the effect of these rule applications to I' in IC^{data} is the same as in IC^{par} and in particular the values of the result locations of I' computed in IC^{par} and IC^{data} coincide. From Lemma 3.3.1 it follows that IC and IC^{data} also compute the same result.

It remains to define the refinement of the rule macros and to prove the refinement claim for them. The assumption on jump/branch instructions implies a) $I \in \mathit{Alu} \cup \mathit{Set} \cup \mathit{Load} \cup \mathit{Link} \cup \{\texttt{MOVS2I}\}$ and b) $I' \neq \texttt{MOVS2I}$, i.e. $I' \in \mathit{Alu} \cup \mathit{Set} \cup \mathit{Load} \cup \mathit{Store} \cup \{\texttt{MOVI2S}\}$. The reason is that only in these cases are *dest(I), fstop(I'), sndop(I')* defined (see Table 3.4 and the assumption $\mathit{dest}(I) = R31$ for $I \in \mathit{Link}$). Therefore, we distinguish three cases, depending on whether the data hazard involves a memory access or not, together with two subcases, depending on the distance between data dependent-instructions in the pipe. For each case we show that the values of the result

Table 3.4 Domain of definition of DLX instruction parameters

function	defined for *instr*
$dest(instr)$	$\in Alu \cup Set \cup Load \cup Link \cup \{\texttt{MOVS2I}\}$
$fstop(instr)$	$\in Alu \cup Set \cup Mem \cup JumpLink \cup Branch \cup \{\texttt{MOVI2S}\}$
$sndop(instr)$	$\in Alu \cup Set \cup Store$

locations of I' in IC are the same as the ones produced by executing I' through the refined rules in IC^{data}.

Case $I \notin Mem$**.** In this case $I \in Alu \cup Set \cup Link \cup \{\texttt{MOVS2I}\}$. *dest(I)* receives the value needed by I' through its update to the value of *C1* in stage *WB(I)*; this value has been copied in stage *MEM(I)* from *C* where it appeared in stage *EX(I)*, as the result of an $Alu \cup Set$–operation or as content of *PC1* or of *IAR*. Therefore in case I' enters the pipe 3 or 2 steps after I, the *ID(I')*-stage overlaps with stage *WB(I)* or *MEM(I)* so that the correct updates of the arguments of I' can be made by directly copying $C' \in \{C1, C\}$ to $nthReg \in \{A, B\}$, instead of waiting for $nthop \in \{fstop, sndop\}$ to receive the correct value. In case I' enters the pipe one step after I, the expected operand value val_{nth} is computed during the stage *ID(I')* and is available in stage *EX(I')* but not before. As a consequence the data hazard can be resolved for the first subcase by refining the *ID*-rule Operand and for the second subcase by refining the *EX*-rules concerned in this subcase, namely Alu, Movi2s, Memaddr, PassBtoMDR.

Subcase $I \overset{\mathbf{2,3}}{<} I'$**.** To forward values directly from the register file entries $C' \in \{C, C1\}$ to the register file exits $nthReg \in \{A, B\}$, direct hardware links between them are needed to support the following refinement of Operand. For brevity we write $\text{Operand}^{par}_{nth}$ for $nthReg \leftarrow nthop$ and express by a derived function C', to be refined by a third case below, the necessary distinction if the code requires two successive updates of $dest(I)$. In that case, as defined by the serial semantics of DLX^{seq} the last update counts.

$$
\begin{aligned}
&Operand^{data} = \{Operand^{data}_{fst}, Operand^{data}_{snd}\} \textbf{ where } Operand^{data}_{nth} = \\
&\quad \textbf{if } nthop \in \{dest(IR3), dest(IR2)\} \textbf{ then } nthReg \leftarrow C' \\
&\quad \textbf{else } \text{Operand}^{par}_{nth}
\end{aligned}
$$

$$
C' = \begin{cases} C1, & \text{if } nthop(IR) = dest(IR3) \neq dest(IR2); \\ C, & \text{if } nthop(IR) = dest(IR2). \end{cases}
$$

Since the rule refinement strengthens the architecture to resolve a possible data conflict, in this subcase by loading anyway the correct arguments for the *EX*-or *MEM*-stage rules of I' into *A, B*, we can weaken the assumptions in Lemma 3.3.3 by enlarging the set of non-relevant locations used by I':

Irrelev 3. $< reg, nthop(I') >$ such that $I' \notin Jump \cup Branch$ and for some $I \overset{3,2}{<} I'$ with $I \notin Mem$ holds $nthop(I') = dest(I)$.

Subcase $I \overset{1}{<} I'$. In this case I' immediately follows I in the pipe, so that the result to be computed by I comes out of the ALU and goes into the register file entry C at the end of stage *EX(I)* and can be forwarded directly, without passing through the register file exits *A, B*, as next ALU-input to compute the *EX* rules for I' with the correct arguments. This is at the expense of introducing a direct link between C and both ALU ports (for $I' \in Alu \cup Set$), *IAR* (for $I' =$ `MOVI2S`), *MAR* and *SMDR* (for $I' \in Mem$) together with some control logic (multiplexers) for selecting the forwarded value as the ALU input rather than the value from the register file. Since these rules are refined furthermore for other conflict cases below, we show here only in the example of the Memaddr macro how the conflict is resolved in the present subcase. For brevity of exposition we use a derived function val_nth, to be refined by a third case below, to express values taken either regularly (from the *nth* register file exit: $fstReg = A$ or $sndReg = B$) or forwarded from C.

$$val_{nth} = \begin{cases} C, & \text{if } nthop(IR1) = dest(IR2); \\ nthReg, & \text{otherwise.} \end{cases}$$

```
if $opcode(IR1) \in Load \cup Store$ then
  if $fstop(IR1) = dest(IR2)$ then $MAR \leftarrow val_{fst} + ival(IR1)$
  else $\text{Memaddr}^{par}$
```

Irrelev 4. $< reg, nthop(I') >$ such that for some $I \overset{1}{<} I'$ with $I \notin Mem$ one of the following holds:

- $opcode(I') \in Alu \cup Set, iop(opcode(I')), dest(I) = fstop(I'), nth = fst;$
- $opcode(I') \in Alu \cup Set, iop(opcode(I')) = false, dest(I) = nthop(I');$
- $opcode(I') \in Mem \cup \{\texttt{MOVI2S}\}, dest(I) = fstop(I'), nth = fst;$
- $opcode(I') \in Store, dest(I) = sndop(I'), nth = snd.$

Case $I \in Mem, I' \notin Mem$. In this case $I \in Load$ and $I' \in Alu \cup Set \cup \{\texttt{MOVI2S}\}$. The value *val* loaded by I is available only at the end of stage $MEM(I)$, namely in *LMDR* or (due to a possible stall; see below) in a new latch *LMDR*1. Therefore, instructions $I' \notin Mem$ which enter the pipe 3 or 2 steps later than I can grep such an operand *val*ue in their stage $ID(I')$ or $EX(I')$, so that it suffices to furthermore refine Operand and the relevant *EX*-stage rules Alu, `MOVI2S`. If, however, I' enters the pipe immediately after I, then the pipeline has to be stopped for one stage, starting at the latest just before stage $EX(I')$, in such a way that after the pipeline takes off again, I' can grep from $LMDR$ the value I has in the meantime loaded there. Since the MEM-stage rule applied to I may overwrite the value $LMDR$ of an immediately preceding load instruction,[32] we add a preservation rule to let immediately after stage WB a copy of $LMDR$ be available also outside the register file for possible use in $\textsc{Alu}^{data}$ and $\texttt{MOVI2S}^{data}$.

$$
\begin{aligned}
&\mathrm{WB}^{data} = \{\textsc{Writeback}^{par}, \textsc{Preserve}\mathrm{WB}\}\ \textbf{where}\\
&\quad \textsc{Preserve}\mathrm{WB} = \{IR4 \leftarrow IR3, LMDR1 \leftarrow LMDR\}
\end{aligned}
$$

Subcase $I \overset{\mathbf{2,3}}{<} I'$. We refine $\textsc{Operand}^{data}$ by refining the derived function C' as follows:

$$
C' = \begin{cases}
C1, & \text{if } nthop(IR) = dest(IR3) \neq dest(IR2)\\
 & \text{and } opcode(IR3) \notin Load;\\
LMDR, & \text{if } nthop(IR) = dest(IR3) \neq dest(IR2)\\
 & \text{and } opcode(IR3) \in Load;\\
C, & \text{if } nthop(IR) = dest(IR2).
\end{cases}
$$

Similarly we refine val_{nth} together with $\textsc{Alu}^{par}$ and $\textsc{Movi2s}^{par}$ as follows:

$$
val_{nth} = \begin{cases}
C, & \text{if } nthop(IR1) = dest(IR2);\\
LMDR, & \text{if } nthop(IR1) = dest(IR3) \neq dest(IR2)\\
 & \text{and } opcode(IR3) \in Load;\\
LMDR1, & \text{if } nthop(IR1) = dest(IR4) \neq dest(IR2), dest(IR3)\\
 & \text{and } opcode(IR3), opcode(IR4) \in Load;\\
nthReg, & \text{otherwise.}
\end{cases}
$$

```
$\textsc{Alu}^{data}$ = if $opcode(IR1) \in Alu \cup Set$ then
  if not ($dataDep1$ or $dataDep2$) then $\textsc{Alu}^{par}$ else
    if $iop(opcode(IR1))$ and $dataDep1$ then
      $C \leftarrow opcode(IR1)(val_{fst}, ival(IR1))$
    if not $iop(opcode(IR1))$ and $dataDep2$ then
      $C \leftarrow opcode(IR1)(val_{fst}, val_{snd})$
$\textsc{Movi2s}^{data}$ = if ($opcode(IR1) = \texttt{MOVI2S}$) then
  if $dataDep1$ then $IAR \leftarrow val_{fst}$ else $\textsc{Movi2s}^{par}$
where
  $dataDep1 = (fstop(IR1) = dest(IR2))$ or  for some $n \in \{3, 4\}$
    $fstop(IR1) = dest(IRn)$ and $opcode(IRn) \in Load$
```

[32] This case has been forgotten in [119], as observed by Holger Hinrichsen (e-mail to E. Börger of February 11, 1998) pointing to the case of an ADD instruction whose two operand registers are loaded by two immediately preceding load instructions. See also Exercise 3.3.6.

$$\begin{aligned}
&\mathit{dataDep}2 = (\mathit{dest}(\mathit{IR}2) \in \{\mathit{fstop}(\mathit{IR}1), \mathit{sndop}(\mathit{IR}1)\})\ \textbf{or}\\
&\quad \text{for some } n \in \{3,4\}\\
&\quad (\mathit{dest}(\mathit{IRn}) \in \{\mathit{fstop}(\mathit{IR}1), \mathit{sndop}(\mathit{IR}1)\}\ \textbf{and}\ \mathit{opcode}(\mathit{IRn}) \in \mathit{Load})
\end{aligned}$$

Since in this way the correct arguments for the *EX*, *MEM*-stage rules applied to I' are fed into A, B or directly forwarded to the operation in question, the corresponding argument locations become irrelevant:

Irrelev 5. $< \mathit{reg}, \mathit{nthop}(I') >$ such that $I' \notin \mathit{Mem} \cup \mathit{Jump} \cup \mathit{Branch}$ and for some $I \in \mathit{Load}$ with $\mathit{dest}(I) = \mathit{nthop}(I')$ the following holds: $I \overset{3}{<} I'$ or $(I \overset{2}{<} I', \mathit{iop}(\mathit{opcode}(I')), \mathit{nth} = \mathit{fst})$ or $(I \overset{2}{<} I', \mathit{otiop}(\mathit{opcode}(I')))$.

Subcase $I \overset{1}{<} I'$. In this case the pipelined execution of I' (and therefore also of later instructions) has to be stopped at the latest just before stage $\mathit{EX}(I')$, until the value to be loaded by I becomes available in *LMDR*. We reflect the common practice of adding a *pipeline interlock*, which detects this situation and stops the pipelining until the conflict has been resolved, by introducing the following additional rule guard (the second clause is explained below):

$$\begin{aligned}
&\mathit{loadRisk} = \mathit{opcode}(\mathit{IR}2) \in \mathit{Load}\ \textbf{and}\\
&\quad (\mathit{IR}1 \notin \mathit{Mem} \cup \mathit{Jump} \cup \mathit{Branch}\ \textbf{and}\\
&\qquad \mathit{dest}(\mathit{IR}2) \in \{\mathit{fstop}(\mathit{IR}1), \mathit{sndop}(\mathit{IR}1)\})\\
&\quad \textbf{or}\ (\mathit{IR}1 \in \mathit{Mem}\ \textbf{and}\ \mathit{dest}(\mathit{IR}2) = \mathit{fstop}(\mathit{IR}1))
\end{aligned}$$

By putting the rules of stages *EX, ID, IF* under this additional guard we achieve that in the case of a load risk they are not executed whereas the rules of *MEM*, *WB* continue to be executed. This leads to the following refinement of the rules for these three stages. Since in this model we abstract from control hazards, the *EX*-rules which update *PC* do not change from DLX^{par} to DLX^{data}. The final refined DLX^{data}-macros are explained below.

$$\begin{aligned}
&\mathrm{ID}^{data} = \textbf{if not}\ \mathit{loadRisk}\ \textbf{then}\ \{\textsc{Operand}^{data}, \textsc{PreserveID}\}\\
&\mathrm{EX}^{data} = \textbf{if not}\ \mathit{loadRisk}\ \textbf{then}\\
&\quad \{\textsc{Alu}^{data}, \textsc{MemAddr}^{data}, \textsc{PassBtoMDR}^{data}, \textsc{Movi2s}^{data}\}\\
&\quad \cup\{\textsc{Movs2i}^{par}, \textsc{Branch}^{par}, \textsc{Trap}^{par}, \textsc{Jumplink}^{par}, \textsc{Link}^{par}\}\\
&\quad \cup\{\textsc{PreserveEX}^{par}\}
\end{aligned}$$

To reset *loadRisk* to false once it has become true, in that case we fetch the default element $\mathit{undef} = \mathit{opcode}(\mathit{undef}) \notin \mathit{Load}$ to update $\mathit{IR}2$.

$$\mathrm{IF}^{data} = \textbf{if}\ (\textbf{not}\ \mathit{loadRisk})\ \textbf{then}\ \textsc{Fetch}^{par}\ \textbf{else}\ \mathit{IR}2 \leftarrow \mathit{undef}$$

In this way we obtain that after the execution of this new rule, the full pipelined execution will be resumed. At this point $I' = reg(IR1)$ still holds but I has been copied from $Ir2$ to $IR3$. Now consider the two cases arising from whether I' also depends on the instruction fetched right before I or not. If not, the data dependence considered here can be proved with the arguments used for the previous subcase to be resolved by the refined *EX*-rules. In the other case we need to distinguish whether that instruction is a load instruction or not. In the first subcase the data conflict is resolved by the *LMDR*1-value taken as the argument by the $EX(I')$-rule. In the second subcase the data conflict is resolved by the $ID(I')$-rule. This analysis shows that we can add the following locations to the irrelevant ones.

Irrelev 6. $< reg, nthop(I') >$ such that $I' \notin Mem \cup Jump \cup Branch$ and for some $I \in Load$ with $dest(I) = nthop(I')$ holds $I \overset{1}{<} I'$.

Case $I, I' \in Mem$. The two subcases complete the definition of EX^{data} and MEM^{data}.

Subcase $I \overset{2,3}{<} I'$. If $I \overset{3}{<} I'$, the refined rule Operand^{data} provides the correct value loaded by I as operand for I'. If $I \overset{2}{<} I'$, this value can be forwarded to I' in its stage EX, namely through refining the rules Memaddr, PassBtoMDR as defined below, at the hardware price of new direct links between *LMDR* and *MAR*, *SMDR*.

```
$\text{Memaddr}^{data} =$ if $opcode(IR1) \in Load \cup Store$ then
  if not $dataDep1$ then $\text{Memaddr}^{par}$ else
    $MAR \leftarrow (val_{fst}, ival(IR1))$
$\text{PassBtoMDR}^{data} =$ if $opcode(IR1) \in Store$ then
  if not $dataDep1$ then $\text{PassBtoMDR}^{par}$ else
    $SMDR \leftarrow val_{snd}$
where $dataDep1 = (fstop(IR1) = dest(IR2))$ or for some $n \in \{3,4\}$
  $fstop(IR1) = dest(IRn)$ and $opcode(IRn) \in Load$
```

The locations resulting as irrelevant via this refinement are the following:

Irrelev 7.

- $< reg, nthop(I') >$ for $I' \in Mem$ and some $I \in Load$ satisfying $I \overset{3}{<} I'$ and $nthop(I') = dest(I)$;
- $< reg, fstop(I') >$ for $I' \in Mem$ and some $I \in Load$ satisfying $I \overset{2}{<} I'$ and $fstop(I') = dest(I)$;
- $< reg, sndop(I') >$ for $I' \in Store$ and some $I \in Load$ satisfying $I \overset{2}{<} I'$ and $sndop(I') = dest(I)$.

Subcase $I \overset{1}{<} I'$. Since the *Mem*-instruction I' can use the value loaded by the preceding instruction I in two ways, as datum to be stored or as address for the load or store operation, we distinguish these two cases.

Case $\mathit{dest}(I) = \mathit{sndop}(I')$. Then $I' \in \mathit{Store}$ and the value loaded by I is needed by I' in its *MEM*-stage, during which it is available in *LMDR*. Therefore, this case can be handled again by forwarding, refining Store, at the expense of a direct link between *LMDR* and the memory input port and of adding the following non-relevant locations:

Irrelev 8. $< \mathit{reg}, \mathit{sndop}(I') >$ for $I' \in \mathit{Store}$ and some $I \in \mathit{Load}$ satisfying $I \overset{1}{<} I'$ and $\mathit{sndop}(I') = \mathit{dest}(I)$.

```
$\mathrm{MEM}^{data} = \{\textsc{Store}^{data}, \textsc{Load}^{par}, \textsc{PreserveMEM}\}$ where
$\textsc{Store}^{data} =$ if $\mathit{opcode}(\mathit{IR}2) \in \mathit{Store}$ then
  if $\mathit{opcode}(\mathit{IR}3) \in \mathit{Load}$ and $\mathit{dest}(\mathit{IR}3) = \mathit{sndop}(\mathit{IR}2)$
  then $\mathit{mem}(\mathit{MAR}) \leftarrow \mathit{LMDR}$
  else $\textsc{Store}^{par}$
```

Case $\mathit{dest}(I) = \mathit{fstop}(I')$. In this case I' needs its first operand during its *EX*-stage when the memory address is computed. But $\mathit{dest}(I)$ is loaded into *LMDR* only during stage $\mathit{MEM}(I)$ so that the pipeline must be interrupted again for one clock cycle; namely we have to uphold the execution of the rules for stage $\mathit{EX}(I')$ and therefore also for the two preceding stages ID, IF. This explains the second clause in the definition of *loadRisk* above. Thereby the modified rules resolve the data conflict in this case, establishing the claim of the lemma with the following additional non-relevant locations:

Irrelev 9. $< \mathit{reg}, \mathit{fstop}(I') >$ for $I' \in \mathit{Mem}$ and some $I \in \mathit{Load}$ satisfying $I \overset{1}{<} I'$ and $\mathit{fstop}(I') = \mathit{dest}(I)$. □

Problem 11 (Cost evaluation of hardware links). Devise a notation for ASM architecture rules which exhibits together with the updates also the hardware links (i.e. the "connection" between terms and their value providing subterms), to serve as a measure for the hardware cost of a proposed refinement.

Problem 12 (Abstract analysis of out-of-order pipelining). Enhance the method shown above to verify common out-of-order-completion techniques, appropriate for pipelines with long-running operations, where an instruction fetched early may complete after an instruction fetched later (see for example the architecture in [102]).

Problem 13 (Abstract analysis of superscalar pipelining). Enhance the method shown above to verify common superscalar architecture techniques where multiple instructions can be fetched simultaneously. See [147] for a superscalar DLX version.

3.3.5 Exercises

Exercise 3.3.1. Prove that Lemma 3.3.1 implies Theorem 3.3.2.

Exercise 3.3.2. (⇝ CD) Prove the Jump Lemma 3.3.2.

Exercise 3.3.3. (⇝ CD) Go through the details of the case distinction for the proof of Lemma 3.3.1, checking the correctness of the refined macros.

Exercise 3.3.4. Prove that Lemma 3.3.3 implies Theorem 3.3.3.

Exercise 3.3.5. (⇝ CD) Refine DLX^{data} to a machine DLX^{pipe} which is fully pipelined, i.e. resolves also control hazards coming with nonlinear code; prove its correctness.

Exercise 3.3.6. (⇝ CD) Formulate a machine DLX^{data} where the data conflict between an instruction $I' \notin Mem \cup Jump \cup Branch$ and two preceding load instructions is resolved without using a latch $LMDR1$.

Exercise 3.3.7. Refine DLX^{data} to a machine where the data conflict recognition and stalling are implemented using a scoreboard which keeps track at run-time for each register whether the pipe contains an instruction modifying that register. The scoreboard technique supports pipelining of architectures where, in contrast to DLX, instructions may differ significantly with respect to their execution time, e.g. when involving besides an integer-arithmetic ALU also a floating point unit or a graphical unit.

Sources and Historical Remarks

The ground model and refinement methods were introduced into ASMs in [71, 72, 76], further developed in [132, 114, 42, 41, 104, 119, 120, 138, 406] and adopted in numerous ASM projects, see Chap. 9 for details. The commutative diagram of Fig. 2.1 underlying Def. 3.2.1, 3.2.2 was introduced in [129] as scheme for the refinements used to prove the correctness of the Prolog-to-Wam compilation. An investigation of mechanizable proof support for the ASM refinement scheme, together with a detailed comparison of various specializations of the scheme to refinement notions in the literature, appeared in [387]. Pure data refinements are the basis for numerous algebraic and set-theoretic refinement notions [167, 176], including those used in VDM [199] and Z [431].

The use of the stepwise ASM-refinement method for design-driven architecture verification was initiated by the Transputer case study in [104] and was extended to pipelining DLX in [119] (which contains also a detailed comparison with other methods to verify pipelined RISC machines, in particular model checking and the use of theorem provers like PVS or HOL). In [217, 407] KIV and PVS have been used to machine-verify the parallelization of the serial ground model ASM (Theorem 3.3.2). The use of ASMs for modeling and verifying pipelining methods grew out from the reverse engineering project of a special-purpose parallel architecture in [102] where pipelining comes together with VLIW parallelism and where the modularity of the hardware description technique is exploited to structure a real-life processor into simple and rigorously defined basic components. The method has been adopted in [286] for the verified layered specification of the early commercial RISC microprocessor ARM2 with a simpler three-stage pipeline; that model in turn is used in [412] to illustrate how to automatically transform register transfer descriptions of microprocessors into executable ASMs. In [411] the method is enhanced to using ASMs for behavioral and structural descriptions of application-specific instruction set processors, from which bit-true and cycle-accurate simulators and debuggers are derived. See Sect. 9.3 for details.

We were led by [12, 48] to formulate the guideline questions for requirements capture. The Telephone Exchange ASM in Sect. 3.1.2 has been inspired by the B machine in [5, Sect. 8.2]. The backtracking ASM is extracted from the two core Prolog rules in [131].

4 Structured ASMs (Composition Techniques)

The characteristics of basic ASMs – simultaneous execution of multiple atomic actions in a global state – come at a price, namely the lack of direct support for practical composition and structuring principles. To make such features available as standard refinements for high-level system design and abstract programming in the large, we define in this chapter[1] two classes of ASMs which offer as building blocks sequential composition, iteration, and parameterized (possibly recursive) submachines extending the macro-notation used with basic ASMs. The chapter can be read independently of Chap. 3 and most of Chap. 2; it suffices to know the definition of basic ASMs.

Turbo ASMs as defined in Sect. 4.1 capture the mentioned submachine notions in a black-box view which matches the *synchrony hypothesis* of synchronous programming languages [266], as in Esterel where the program reaction is considered as instantaneous although it is made up of a sequence of elementary actions ("micro-steps") which are performed in a fixed order. Turbo ASMs hide the internals of subcomputations by compressing them into one step (hence the name) and thus fit the synchronous parallelism of basic ASMs (as well as the parallelism of asynchronous ASMs defined in Chap. 6).*Abstract State Processes* as defined in Sect. 4.2 realize them in a white-box view, where interleaving permits one within a context of parallel execution to also follow the single steps of a component computation.

Turbo ASMs permit us to integrate into their semantical composition principles the common syntactical forms of encapsulation and state hiding, such as the notion of a *local state* and a mechanism for *returning values* and *error handling*. We illustrate the use of these composition techniques by succinct turbo ASMs for standard programming constructs, including the concepts underlying the celebrated Structured Programming Theorem 4.1.1 and some widely used forms of recursion (Sect. 4.1.2). In fact the scheme we provide for computing recursively defined functions by turbo ASMs naturally *integrates functional description and programming techniques* into the "high-level programming" by ASMs. In Sect. 4.1.3 we identify the standard tree structure of turbo ASM subcomputations. A logic for turbo ASMs is investigated in Sect. 8.1.1.

[1] Lecture slides can be found in `TurboASM` (⇝ CD), `RecursionAsm` (⇝ CD), `ASP` (⇝ CD).

4.1 Turbo ASMs (seq, iterate, submachines, recursion)

We extend the basic ASMs in Sect. 4.1.1 by operators for *sequential composition* and for *iteration* of ASMs, and in Sect. 4.1.2 by *parameterized submachines* which may recursively call themselves and thus genuinely enrich the notational macro-shorthand. The definitions realize a black-box view of the compound machine, defined to hide the details of its internal subcomputation and to yield its global effect (if any) in *one* step, executable in parallel with the other rules.[2] We call *turbo ASM* every ASM which can be obtained from basic ASMs by applying finitely often and in any order the operators of sequential composition, iteration and submachine call. Since a turbo ASM subcomputation may execute an a priori unlimited number n of basic machine steps, which could go to infinity, this naturally leads to the possibility of non-terminating subcomputations. In such a case the overall computation step into which the subcomputation is inserted as a turbo step is undefined.

For logical reasons explained in Sect. 8.1.4, for turbo ASMs where sequential composition is combined with recursion one better restricts nondeterminism to external functions, avoiding the use of the **choose**-construct. The resulting ASM runs are deterministic modulo the external functions since choices if any are made outside the machine. This is not to forbid the **choose**-construct, but to make us aware of the proof-theoretic problems related to its indiscriminate use.

Another proviso dictated by reasons of practicability concerns changes of monitored and external updates of shared functions for turbo ASMs. To make the analysis of the global effect of hidden subcomputations feasible it is assumed for turbo ASMs that "during" a black-box step the values of monitored and shared functions are not updated externally, i.e. their values are fixed for every turbo computation participating in the considered single global machine step. Otherwise the behavior of turbo ASMs easily becomes hard to follow.

4.1.1 Seq and Iterate (Structured Programming)

In this section we define the turbo ASM **seq**-construct, which combines simultaneous atomic updates of basic ASMs in a global state with *sequential execution*. It naturally extends to the generic turbo ASM **iterate**-construct from which the classical *iteration*-operators are derived.

To get a clue for the intended smooth integration of ("the effect of") subcomputations into atomic state changes, consider the way ASMs avoid the frame problem: the global one-step effect of an ASM – the transformation of

[2] The definition of turbo ASMs avoids the complicated details of the standard pseudo-code solution of the problem where a non-atomic command is broken into atomic subcommands which are connected by a program counter, as in SPEC [316, Sects. 3, 17].

the given state into the *next* state – is determined in a fixed manner by the function *next* defined in Lemma 4.1.2 as the overall result of a set of "local" updates (if this set is consistent). Therefore, submitting to *next* in a given state an update set which may result from any finite number of elementary computation steps suffices to encapsulate those steps and turn their effect into an "atomic" step.[3] It is a natural extension of the parallel synchronism of basic ASMs to collect into one set not only updates which have been made in one step, but to allow two or more "basic" steps – taking obviously into account that sequential execution should allow later overwriting of earlier updates. In this way the apparent dichotomy between "atomic" and "compound" actions turns into a question of machine *view*, whether one wants to analyze it at the high level of abstraction of a turbo-ASM (black-box view) or at the more detailed level of a basic ASM (white-box view). What is "basic" and what is "turbo" is relative to the level of abstraction chosen by the designer for machine executions, in analogy to the different roles that controlled and external functions play for the structure of the underlying state.

To separate the computational issues of composition from concerns related to state sharing we assume in the following that the submachines we are considering all have the same signature.

We denote the sequential composition of two ASM rules P, Q by P **seq** Q and define its semantics as the effect of first executing P in the given state $\mathfrak{A}$ and then Q in the resulting state $\mathfrak{A} + U$ (if it is defined), where U is the set $[\![P]\!]^{\mathfrak{A}}$ of updates produced by P in $\mathfrak{A}$. To reflect the fact that Q may overwrite a location which has been updated by P we use the notation $U \oplus V$ for the merging of two update sets, which is defined as follows. We merge only if U is consistent, otherwise we stick to U, because then we want both $\mathfrak{A} + U$ and $\mathfrak{A} + (U \oplus V)$ to be undefined.

Definition 4.1.1. Let P and Q be ASM rules.

$$[\![P \textbf{ seq } Q]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}} \oplus [\![Q]\!]^{\mathfrak{A}+[\![P]\!]^{\mathfrak{A}}}$$

$$U \oplus V = \begin{cases} \{(loc, val) \in U | loc \notin Locs(V)\} \cup V, & \text{if consistent(U);} \\ U, & \text{otherwise.} \end{cases}$$

The definition implies that a sequential computation gets stuck once an inconsistency is encountered in its first part.

Lemma 4.1.1 (Persistence of inconsistency). If $[\![P]\!]^{\mathfrak{A}}$ is not consistent, then $[\![P \textbf{ seq } Q]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}}$.

[3] In this way the definition of turbo ASMs avoids the fixpoint problem synchronous programming languages have to cope with due to their synchrony hypothesis. For example in Esterel [266] the current event is a fixpoint of a function which may not be monoton and therefore may have more than one fixpoint; as a consequence the language associates a meaning only to those programs where the fixpoint is unique. This property is not necessarily transparent to the user but is checked by the compiler.

The next lemma expresses that the definition of the ASM **seq** constructor captures the classical meaning of the sequential composition of machines.[4] We leave the proof as Exercise 4.1.2.

Lemma 4.1.2 (Compositionality and semi-ring properties of seq). Denote by *next* the function defined by $\mathit{next}(R)(\mathfrak{A}, \zeta) = \mathfrak{A} + [\![P]\!]^{\mathfrak{A}}_{\zeta}$.

$$\mathit{next}(P \textbf{ seq } Q) = \mathit{next}(Q) \circ \mathit{next}(P)$$

$$[\![\mathit{skip} \textbf{ seq } P]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}} = [\![P \textbf{ seq } \mathit{skip}]\!]^{\mathfrak{A}}$$

$$[\![P \textbf{ seq } (Q \textbf{ seq } R)]\!]^{\mathfrak{A}} = [\![(P \textbf{ seq } Q) \textbf{ seq } R]\!]^{\mathfrak{A}}$$

Iterating **seq** encapsulates computations with a finite number of iterated steps into one step, namely defined by $R^0 = \mathit{skip}$ and $R^{n+1} = R^n \textbf{ seq } R$. Denote by $\mathfrak{A}_n$ the state (if defined) which is obtained by firing the update set produced by R^n in state $\mathfrak{A}$.

There are two natural stop situations for iterated rule applications without a priori fixed bounds, namely when the update set becomes empty (the case of *successful termination*) and when it becomes inconsistent (the case of *failure*, given the persistence of inconsistency as formulated in Lemma 4.1.1).[5] Both cases provide a fixpoint $\lim_{n\to\infty}[\![R^n]\!]^{\mathfrak{A}} = [\![R^{n-1}]\!]^{\mathfrak{A}}$ for the first n where the update set produced by R in the state obtained by firing R^{n-1} in $\mathfrak{A}$ is empty or inconsistent. This motivates the following definition and explains the corollary.

Definition 4.1.2. $[\![\textbf{iterate } R]\!]^{\mathfrak{A}} = \lim_{n\to\infty}[\![R^n]\!]^{\mathfrak{A}}$, if for some $n \geq 0$ it holds that $[\![R]\!]^{\mathfrak{A}_n} = \emptyset$ or $\mathit{inconsistent}([\![R]\!]^{\mathfrak{A}_n})$.

Corollary 4.1.1 (Well-definedness of iterate R). If $[\![R]\!]^{\mathfrak{A}_{n-1}}$ is inconsistent or empty, then $[\![R^n]\!]^{\mathfrak{A}} = [\![R^m]\!]^{\mathfrak{A}}$ for all $m \geq n > 0$.

The sequence $([\![R^n]\!]^{\mathfrak{A}})_{n>0}$ eventually becomes constant only upon termination or failure. Otherwise, the computation diverges and the update set for the iteration is undefined. A famous example for a turbo ASM with diverging (though, if viewed differently, useful) computation is **iterate** $a := a + 1$.

We now illustrate the sequential iteration of turbo ASMs by two examples and by deriving some standard constructs of structured programming.

Example 4.1.1 (Turbo ASM starting the Java class initialization). In Java each class is automatically initialized upon its first use. The order of class initialization is required to respect the class hierarchy, i.e. the superclass of a class c has to be initialized before c. Therefore, when upon the first use

[4] We assume strictness for every f, meaning that $f(x)$ is undefined if x is undefined.

[5] We do not include here the case of an update set whose firing does not change the given state, although including this case would provide an alternative stop criterion which is also viable for implementations of ASMs.

of a class its initialization is triggered, this trigger must be passed along the class hierarchy until an initialized class c' is encountered (i.e. satisfying *initialized*(c'), as eventually will happen towards the top of the class hierarchy). To abstract from the standard sequential implementation (where obviously the class initialization is started in a number of steps depending on how many not yet initialized classes there are above the given class) the **iterate**-construct turns out to be handy, offering an atomic operation to push all initialization methods in the right order onto the frame stack. This is expressed by the following turbo ASM which uses a macro *createInitFrame* of simple frame updates (for details see [406], from where the machine is taken).

```
Initialize($class$) $\equiv c := class$ seq
  iterate
    if $\neg initialized(c)$ then $createInitFrame(c)$
    if $\neg initialized(superClass(c))$ then $c := superClass(c)$)
```

The finiteness of the acyclic class hierarchy in Java guarantees that this machine yields a well-defined update set.

Example 4.1.2 (Turbo ASM for iterative ASM ***While****).* The following iterative ASM *while* repeats the execution of its body rule as long as it produces a non-empty update set and the *cond*ition holds:[6]

while $(cond)$ R = **iterate** (**if** $cond$ **then** R).

This *while* loop, if started in state $\mathfrak{A}$, terminates if eventually $[\![R]\!]^{\mathfrak{A}_n}$ becomes empty or the condition *cond* becomes *false* in $\mathfrak{A}_n$ (with consistent and non-empty previous update sets $[\![R]\!]^{\mathfrak{A}_i}$ and previous states $\mathfrak{A}_i$ satisfying *cond*). If the iteration of R reaches an inconsistent update set (failure) or yields an infinite sequence of consistent non-empty update sets, then the state resulting from executing the while loop starting in $\mathfrak{A}$ is not defined (divergence of the while loop). In these two cases the function *next*(**while** $(cond)$ R) is undefined on $\mathfrak{A}$.

A *while* loop may satisfy more than one of the above conditions, like **while** (*false*) *skip*. There are four typical cases:

success: **while** $(cond)$ *skip* or **while** $(false)$ R
failure: **while** $(true)$ $a := 1, a := 2$
divergence: **while** $(true)$ $a := a$

Structured programming constructs. Turbo ASMs provide the conceptual ingredients of structured programming. We illustrate this here in a highly abstract manner with sequentially iterated turbo ASMs, namely by providing a surprisingly elementary proof for a general form of the celebrated Structured Programming Theorem of Böhm and Jacopini [66]. In doing this we construct by sequential iteration simple turbo ASMs to compute arbitrary computable

[6] See Example 4.1.4 for a slightly different recursively defined **While**.

functions, in a way which combines the advantages of Gödel–Herbrand-style functional and of Turing-style imperative programming.

The structured programming example shows more than what it states, and it would be mistaken to take it as of only theoretical interest. The atomicity of the turbo ASM sequentialization and iteration is the key for a rigorous definition of the semantics of some fundamental UML notions, e.g. the event-triggered exiting from compound actions of UML activity and state machine diagrams, where the intended instantaneous effect of exiting has to be combined with the request to exit nested diagrams sequentially following the subdiagram order; see [98, 99].

Problem 14 (Analysis of turbo control state ASM networks). Generalize the work done for Mealy automata in [145], investigating networks of turbo control state ASMs built up using **seq**, **iterate**, and the synchronous parallelism of basic ASMs to provide sequencing, parallel composition and feedback operators.

Definition 4.1.3. We call *Böhm–Jacopini-ASM* any turbo ASM M which can be defined, using only **seq, while**, from basic ASMs whose non-controlled functions are restricted to one (a 0-ary) input function (whose value is fixed by the initial state), one (a 0-ary) output function, and the initial functions of recursion theory (see below) as static functions. The purpose of the 0-ary input function, which we write in_M, is to contain the number sequence which is given as the input for the computation of the machine. Similarly out_M is used to receive the output of M. The *initial* functions of recursion theory are the following functions from Cartesian products of natural numbers into the set of natural numbers: $+1$, all the projection functions U_i^n, all the constant functions C_i^n and the characteristic function of the predicate $\neq 0$.

Following the standard definition we call a number theoretic function $f\colon \mathbb{N}^n \to \mathbb{N}$ *computable by an ASM M* if for every n-tuple $x \in \mathbb{N}^n$ of arguments on which f is defined, the machine *started with input x terminates with output $f(x)$*. By "M started with input x" we mean that M is started in the state where all the dynamic functions different from in_M are completely undefined and where $in_M = x$. Assuming the external function in_M not to change its value during an M-(turbo)computation, it is natural to say that M "terminates in a state with output" y, if in this state out_M gets updated for the first time, namely to y. In all the machines constructed below this will always be the state in which the intended turbo-computation reached its final goal.

Theorem 4.1.1 (Structured programming theorem).
Every computable function can be computed by a Böhm–Jacopini ASM.

Proof. We define by induction for each partial recursive definition of a computable function f a machine F computing it. Each initial function f is computed by the following machine F consisting of only one function update which reflects the defining equation of f.

$$F \equiv out_F := f(in_F)$$

In the inductive step we construct, for every partial recursive definition of a function f from its constituent functions f_i, a machine F which mimics the standard evaluation procedure underlying that definition. We use the following macros which describe (a) inputting from some external input source *in* to a machine F before it gets started, and (b) extracting the machine output upon termination of F to some external target location *out*. These macros reflect the mechanism for providing arguments and yielding values which is implicit in the standard use of functional equation systems to determine the value of a function for a given argument.

$$F(in) \equiv in_F := in \textbf{ seq } F$$
$$out := F(in) \equiv in_F := in \textbf{ seq } F \textbf{ seq } out := out_F$$

We start with the case of function composition. If functions $g, h_1, \ldots, h_m$ are computed by Böhm–Jacopini-ASMs $G, H_1, \ldots, H_m$, then their composition f defined by $f(x) = g(h_1(x), \ldots, h_m(x))$ is computed by the following machine $F = \text{FctCompo}$:[7]

$$\text{FctCompo}(G, H_1, \ldots, H_m) =$$
$$\{H_1(in_F), \ldots, H_m(in_F)\} \textbf{ seq } out_F := G(out_{H_1}, \ldots, out_{H_m})$$

Unfolding this structured program reflects the order one *has* to follow for evaluating the subterms in the defining equation for f, an order which is implicitly assumed in the equational (functional) definition. First, the input is passed to the constituent functions h_i to compute their values, whereby the input functions of H_i become controlled functions of F. The parallel composition of the submachines $H_i(in_F)$ reflects that their computations are completely independent from each other;[8] what counts and is expressed is that all of them have to terminate before the next "functional" step is taken. That next step consists in passing the sequence of out_{H_i} as input to the constituent function g. Finally the value of g on this input is computed and assigned as output to out_F.

Similarly, let a function f be defined from g, h by primitive recursion:

$$f(x, 0) = g(x), \quad f(x, y+1) = h(x, y, f(x, y))$$

and let Böhm–Jacopini-ASMs G, H be given which compute g, h. Then the following machine $F = \text{PrimitiveRecursion}$ computes f, composed as a sequence of three submachines. The start submachine evaluates the first defining equation for f by initializing the recursor *rec* to 0 and the intermediate

[7] For reasons of simplicity but without loss of generality we assume that the submachines have pairwise disjoint signatures. Remember that we use sets to denote the rules of an ASM which are to be executed in parallel.

[8] As a consequence they can be done in any order, with or without interleaving of their substeps. See the discussion of ASMs for forms of recursion in Sect. 4.1.2.

value *ival* to $g(x)$. The *while* submachine evaluates the second defining equation for f for increased values of the recursor as long as the input value y has not been reached. The output submachine provides the final value of *ival* as output. As in the case of simultaneous substitution, the sequentialization and iteration described here make explicit the bare minimum on ordering computational substeps, which is assumed and in fact needed in the standard functional use of the defining equations for f.[9]

```
$\textsc{PrimitiveRecursion}(G, H) = \mathbf{let}\ (x, y) = in_F\ \mathbf{in}$
  $\{ival := G(x),\ rec := 0\}\ \mathbf{seq}$
  $(\mathbf{while}\ (rec < y)\ \{ival := H(x, rec, ival),\ rec := rec + 1\})\ \mathbf{seq}$
  $out_F := ival$
```

If f is defined from g by the μ-operator, i.e. $f(x) = \mu y(g(x, y) = 0)$, and if a Böhm–Jacopini-ASM G computing g is given, then the following machine $F = \textsc{MuOperator}$ computes f. The start submachine computes $g(x, rec)$ for the initial recursor value 0, and the iterating machine computes $g(x, rec)$ for increased values of the recursor until 0 shows up as computed value of g, in which case the reached recursor value is set as output.

```
$\textsc{MuOperator}(G) = \{G(in_F, 0),\ rec := 0\}\ \mathbf{seq}$
$(\mathbf{while}\ (out_G \neq 0)\ \{G(in_F, rec + 1),\ rec := rec + 1\})\ \mathbf{seq}$
$out_F := rec$
```

□

Remark 4.1.1 (Functional versus imperative programming). The construction of Böhm–Jacopini-ASMs illustrates, through the idealized example of computing recursive functions, how ASMs allow us to pragmatically reconcile the often discussed conceptual dichotomy between functional and imperative programming. In this context functional programs are characterized as different from imperative ones because "rather than telling the computer what to do, they *define* what it is that the computer is to provide" (quoted from [166]). The equations which appear in the Gödel–Herbrand-type definition of partial recursive functions "define what it is that the computer is to provide" only on the basis of the implicit assumptions made for the procedure to be followed for the manipulation of arguments and values during the evaluation of terms. The corresponding Böhm–Jacopini-ASMs constructed above make this machinery explicit, exhibiting how to evaluate the subterms when using the equations, as much as is needed to make the functional shorthand

[9] Of course different uses of such equations can be imagined, but that would mean that the equations come with other underlying evaluation mechanisms which are taken for granted and could be made explicit as an abstract machine.

work correctly in the way it was hardwired in our brains through training at school.[10]

4.1.2 Submachines and Recursion (Encapsulation and Hiding)

In this section the purely notational macro technique is extended by a constructor to build large turbo ASMs from parameterized submachines which justifies also recursive submachine calls. The resulting atomic submachine view is illustrated by a hierarchical decomposition of the Java Virtual Machine. We use turbo submachines to define encapsulation and state hiding mechanisms for ASMs with *local state*, *return values* and *error handling*. Value-returning turbo ASMs exactly reflect the abstraction made in functional programming from everything except the input–output (argument–value) relation. Since the turbo ASM submachine concept expresses in abstract form the usual imperative calling mechanism, one can use it in particular to make the common intuitive understanding of recursion precise in terms of single-agent ASMs. We illustrate this by turbo ASMs for Mergesort and Quicksort.

Turbo ASM submachines. The notion of basic ASM avoids domain theoretic complications – arising when explaining by denotational methods what it means to iterate the execution of a machine "until …" – by defining only the one-step computation relation and by relegating fixpoint (termination) concerns to the metatheory. In the same spirit we define the semantics of submachine calls only for the case where the possible chain of nested calls of that machine is finite, leaving it to the programmer to guarantee that a potentially infinite chain of recursive procedure calls is indeed well founded with respect to some order. The resulting definition directly supports the practitioners' algorithmic understanding and use of submachine calls.

The definition for rules is therefore extended to allow also named parameterized rule (submachine) calls $R(a_1, \ldots, a_n)$ with actual parameters $a_1, \ldots, a_n$, coming with a rule definition of the following form:

$$R(x_1, \ldots, x_n) = \mathit{body}$$

where *body* is a rule. R is called the rule *name*, and $x_1, \ldots, x_n$ are the formal parameters of the rule definition, comprising all the free occurrences of variables in *body* and binding them.

The basic intuition the practice of computing provides for the interpretation of a named rule is to define its semantics as the interpretation of the rule body with the formal parameters replaced by the actual arguments. In other words, nested calls of a recursive rule R are unfolded into a sequence

[10] See Sect. 4.1.2, where we provide abstract machines for forms of recursion, illustrating further how ASMs allow one to capture the encapsulation techniques which come with the functional programming paradigm.

$R_1, R_2, \dots$ of rule incarnations where each R_i may trigger one more execution of the rule body, relegating the interpretation of possibly yet another call of R to the next incarnation R_{i+1}. This may produce an infinite sequence, namely if there is no ordering of the procedure calls with respect to which the sequence will decrease and reach a basis for the recursion. In this case the semantics of the call of R is undefined. If, however, a basis for the recursion does exist, say R_n, it yields a well-defined value for the semantics of R through the chain of successive calls of R_i; namely for each $0 \leq i < n$ with $R = R_0$, R_i inherits its semantics from R_{i+1}. This is formalized as follows where without loss of generality we assume that rules are called by name, i.e. the formal parameters are substituted in the rule body by the actual parameters so that these are evaluated only when used (this is not necessarily in the state in which the rule is called, due to the presence of **seq**). Call by value can be obtained by $R(t) \equiv$ (**let** $x = t$ **in** *body*).

Definition 4.1.4 (Turbo submachine). Let R be a named rule declared by $R(x_1, \dots, x_n) = \mathit{body}$ and let $\mathfrak{A}$ be a state. If $[\![\mathit{body}[a_1/x_1, \dots, a_n/x_n]]\!]^{\mathfrak{A}}$ is defined, then

$$[\![R(a_1, \dots, a_n)]\!]^{\mathfrak{A}} = [\![\mathit{body}[a_1/x_1, \dots, a_n/x_n]]\!]^{\mathfrak{A}}$$

As expected, a rule definition $R(x) = R(x)$ yields no value for any $[\![R(a)]\!]^{\mathfrak{A}}$.

Example 4.1.3. Passing of parameters in recursive calls circumvents the need we had in Example 4.1.1 to initially assign the parameter value to a local variable, yielding the following recursive turbo ASM to start the Java class initialization. As in Example 4.1.1, the termination depends on the finiteness of the class hierarchy and the termination of *createInitFrame*(c).

```
InitializeRec(c) =
  if initialized(superClass(c)) then createInitFrame(c)
  else createInitFrame(c) seq InitializeRec(superClass(c))
```

Example 4.1.4 (Turbo ASM for recursive ***While****).* The following recursive *whileRec* behaves differently from the iterative ASM-*while* in Example 4.1.2. It leads to termination only if the condition *cond* eventually becomes false, and not in the case where eventually the update set of the body rule becomes empty. For example *whileRec*(*true*, *skip*) does not terminate.

```
whileRec(cond, R) =if cond then R seq whileRec(cond, R)
```

Atomic view of turbo ASM submachines. This view is illustrated by the JVM submachine *verifyVM* in Fig. 2.11, viewed as a component added to the trustful interpreter and called from there as a turbo ASM when a new class is loaded. Independently from its use as a turbo ASM it can be investigated also in a white-box manner, as a stand-alone component, in support of a separation of different verification concerns concerning the different levels of the hierarchical decomposition of the Java Virtual Machine into loading, verifying and executing machines for the five principal language layers (imperative core, static classes, object-oriented features, exception handling and concurrency). For example *verifyVM* is shown in [406, Chap. 17] to always terminate and to be sound and complete with respect to correctly typeable compiled Java programs; in terms of its interface it is also shown (in [406, Chap. 18]) to correctly interact with the trustful JVM interpreter and the class loader.

The atomicity of turbo ASM submachines served in [99] to model the event-triggered run-to-completion scheme of UML state machines (see Sect. 6.5.1). Using turbo ASMs we are going to introduce now encapsulation and state-hiding mechanisms which are common for submachine concepts, such as *local state*, *return values* and *error handling*.

Turbo ASMs with local state. In traditional ASMs all the dynamic functions are global. However, the use of only locally visible parts of the state can naturally be incorporated into turbo ASMs. It suffices to extend the definition of named rules by allowing some dynamic functions to be declared as local. It means that each call of the rule works with its own incarnation of local dynamic functions f, which are initialized upon rule invocation by an initialization rule $\mathit{Init}(f)$ preceding the execution of the rule body.

We therefore allow definitions of named rules to contain also (horizontally or vertically displayed) declarations of dynamic functions as local, together with an optional initialization rule, e.g. in the following form:

```
$\mathit{name}(x_1, \ldots, x_n) =$
  $\textbf{local}\ f_1[\mathit{Init}_1] \ldots \textbf{local}\ f_k[\mathit{Init}_k]$
  $\mathit{body}$
```

where *body* and Init_i are rules. The formal parameters $x_1, \ldots, x_n$ bind their free occurrences in *body* and Init_i. The functions $f_1, \ldots, f_k$ are treated as local functions whose scope is the rule where they are introduced. They are not considered to be part of the signature of the main ASM. Init_i is a rule used for the initialization of f_i. We write **local** $f := t$ for **local** $f[f := t]$.

For the semantic interpretation of a call of a rule with local dynamic functions, the updates to the local functions are collected together with all other function updates made through executing the body. This includes the updates required by the initialization rules. The restriction of the scope of the local functions to the rule definition is obtained by then removing from the update set U, which is available after the execution of the body of the call, the set $\mathit{Updates}(f_1, \ldots, f_k)$ of updates concerning the local functions $f_1, \ldots, f_k$. This is summarized by the following definition.

Definition 4.1.5 (Turbo ASMs with local functions). Let R be a rule declaration with local functions as given above. The two terms in the following equation are either both undefined or both defined and equal:

$$[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![(\{Init_1, \ldots, Init_k\} \textbf{ seq } body)[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}} \\ \setminus Updates(f_1, \ldots, f_k)$$

We assume that there are no name clashes for local functions between different incarnations of the same rule (i.e. each rule incarnation has its own set of local dynamic functions).

Error handling. Modern programming languages support exception-handling techniques which separate error handling from normal execution of code. Producing an inconsistent update set is an abstract form of throwing an exception. This allows us to introduce for turbo ASMs an abstract method for catching an inconsistent update set and of executing error handling rules.

Let T be a set of terms. The semantics of **try** P **catch** T Q is the update set of P, if this update set is consistent on the locations determined by elements of T. Otherwise it is the update set of Q.[11]

Definition 4.1.6. Let P and Q be turbo ASMs and T a set of terms. We define

$$[\![\textbf{try } P \textbf{ catch } T\ Q]\!]^{\mathfrak{A}} = \begin{cases} [\![P]\!]^{\mathfrak{A}}, & \text{if consistent } [\![P]\!]^{\mathfrak{A}} \upharpoonright Loc(T)^{\mathfrak{A}}; \\ [\![Q]\!]^{\mathfrak{A}}, & \text{otherwise.} \end{cases}$$

Turbo ASMs with return values. A frequent special use of turbo ASMs is to compute functions of the input. Storing an output value in a global dynamic function *out*, as we did for the proof of Theorem 4.1.1, violates good information-hiding principles. A mechanism is needed which allows one to retrieve the intended return value of a named rule R from a location determined by the rule caller, independently of R. It suffices to provide a notation for locations from where to extract such result values from the final state of the turbo submachine computation. We use for named rules with n parameters the common notation $l \leftarrow R(a_1, \ldots, a_n)$ to denote the overall effect (update set) of executing the rule body, where the 0-ary dynamic function l has been substituted for a reserved 0-ary function **result**. **result** and therefore its replacements by programmer-defined locations l play the role of placeholders in which to store the intended return value. They represent an abstract interface offered for communicating results from a rule execution to the caller, typically implemented as the top of a stack from where the caller gets the computed value when the control comes back from the callee.

Definition 4.1.7. Let $R(x_1, \ldots, x_n) = body$ be the declaration for R.

$$[\![l \leftarrow R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![body[l/\textbf{result}, a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$$

[11] Since for turbo ASMs the rule enclosed by the **try** block is executed either completely or not at all, there is no need for any **finally** clause to remove trash.

Two rules $l \leftarrow R(a_1, \ldots, a_n)$ and $l' \leftarrow R(a'_1, \ldots, a'_n)$ with different locations l, l' can be applied simultaneously with different return values for l and l'. When a term l of the form $f(t_1, \ldots, t_n)$ is used in $l \leftarrow R(a_1, \ldots, a_n)$, a good encapsulation discipline will take care that R does not modify the values of t_i, because they contribute to determining the location where the caller expects to find the return value.

We use this notation now to model frequent forms of recursion, illustrating thereby how to naturally reflect functional programming features by value-returning turbo ASMs.

Modeling recursion by turbo ASMs. There are many ways to explain the meaning of various forms of recursion. Turbo ASM submachines abstractly model the standard imperative calling mechanism, which provides the key for expressing the common intuitive understanding of recursion in terms of single-agent ASM computations. By the atomicity of their black-box computations, turbo ASMs allow us to reflect exactly the machinery which underlies the common mathematical use of *functional* equations to evaluate function values. We illustrate this here for the classical recursive definitions of the Quicksort and Mergesort algorithms.[12] Our intention is not to suggest replacing succinct and perfectly clear recursive definitions by ASM specifications, but to show that WITHIN the ASM framework one is justified in freely using recursion in the way we have learnt to.

The update set produced by executing a turbo ASM call represents the total effect of executing the submachine in the call state (atomicity of the turbo ASM computation). Using the machine to return a value adds a form of *functional abstraction* from everything in that computation except the resulting input–output (argument–value) relation.[13] Technically we combine the turbo ASM notation for value returning machines with the **let**-construct to pass the result of machines R_i by value to other machines S, mimicking the use of activation records to store parameters as local variables. Since for each submachine call a dedicated placeholder is needed to record the result of the subcomputation, in the following definition we apply an external function *new* to the dynamic set FUN_0 of 0-ary dynamic functions ("write variables"). As stated in Sect. 2.2.3, *new* is supposed to provide each time a completely fresh

[12] This answers the question raised in [340]: "If algorithms are machines, then which machine is the Mergesort?" The answer is different from the one in [261, 60], where it is argued that "recursive computations are to be viewed as a special case of distributed computation": each recursive call is executed by a newly created callee agent expected to return his result to the caller. The way we define the result of the turbo ASM call implies that from the caller's view it is returned immediately, directly reflecting the functional view which only uses the result in the given evaluation process and abstracts from how and by whom the result has been obtained. As a consequence our explanation of recursion does not need to invoke multiple agents, but it is compatible with using them for a distributed implementation.

[13] The pure functional effect of course is achieved only if the submachine computation on the caller's side affects only the result location.

location, i.e. a location which has never been used before and is also not used for any other simultaneous call of *new*. This suffices to make the assumptions underlying the functional handling of intermediate values explicit.[14]

Definition 4.1.8 (Using return values in turbo ASMs). Let R_i, S be turbo ASMs with formal parameter sequences x_i of R_i and parameters y_i of S. We define:

```
let $\{y_1 = R_1(a_1), \ldots, y_n = R_n(a_n)\}$ in $S$ =
  let $l_1, \ldots, l_n = new(FUN_0)$ in
    forall $1 \leq i \leq n$ do $l_i \leftarrow R_i(a_i)$ seq
      let $y_1 = l_1, \ldots, y_n = l_n$ in $S$
```

Note that **let** $\{x_1 = R_1, x_2 = R_2\}$ **in** S can be different from **let** $x_1 = R_1$ **in** (**let** $x_2 = R_2$ **in** S) (Exercise 4.1.4).

The preceding definition allows one to explicitly capture the abstract machine which underlies the common mathematical evaluation procedure for functional expressions, including those defined by forms of recursion. We illustrate this by the following turbo ASM definitions of Quicksort and of Mergesort which exactly mimic the usual recursive definition of the algorithms to provide as *result* a sorted version of any given list. The computation suggested by the well-known recursive equations to quicksort L proceeds as follows: FIRST partition the *tail* of the list into the two sublists $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ of elements $< head(L)$ respectively $\geq head(L)$ and quicksort these two sublists separately (independently of each other), THEN *concatenate* the results taking $head(L)$ between them. The fact that this description uses various auxiliary list and comparison operations is reflected by the appearance of corresponding auxiliary functions in the following turbo ASM.

```
QUICKSORT($L$) = if$| L | \leq 1$ then result := $L$ else
  let
    $x$ = QUICKSORT($tail(L)_{<head(L)}$)
    $y$ = QUICKSORT($tail(L)_{\geq head(L)}$)
  in result := $concatenate(x, head(L), y)$
```

[14] Strictly speaking this definition introduces a new rule into Table 2.2, given that the signature becomes dynamic: new 0-ary functions (parameterless locations) are introduced which can be updated, namely to keep incarnations of variables for the entire subcomputation. One can however easily avoid this and separate the "logical" (read-only) variables from the variables for locations (write-variables) by using a monadic function *result* which takes locations as arguments; to guarantee that the results of invocations with different arguments are stored in different locations it suffices to pass to *result* different arguments l, l'. Formally in Def. 4.1.7 instead of l use $result(l)$, and in Def. 4.1.8 write $y_i = result(l_i)$. Then **let** $\{y_1 = R_1(a_1), \ldots, y_n = R_n(a_n)\}$ **in** S can be eliminated via Def. 4.1.8 and be reduced to the rules in Table 2.2.

In Exercise 4.1.5 we formulate a procedural refinement of list partitioning in Quicksort.

The computation suggested by the usual recursive equations to mergesort a given list L consists in FIRST splitting it into a *LeftHalf*(L) and a *RightHalf*(L) (if there is something to split) and mergesort these two sublists separately (independently of each other), THEN to *Merge* the two results by an auxiliary elementwise *Merge* operation. This is expressed by the following turbo ASM which besides two auxiliary functions *LeftHalf*, *RightHalf* comes with an external function *Merge* defined below as a submachine.

```
$\textsc{Mergesort}(L) =$ if $|L| \leq 1$ then result $:= L$ else
  let
    $x = \textsc{Mergesort}(\mathit{LeftHalf}(L))$
    $y = \textsc{Mergesort}(\mathit{RightHalf}(L))$
  in result $\leftarrow \textsc{Merge}(x, y)$
```

Usually MERGE is defined by a recursion, suggesting the following computation scheme which is formalized by the turbo ASM below. If both lists are non-trivial, by a case distinction the smaller one of the two list heads is determined and placed as the first element of the *result* list, concatenating it with the result of a separate and independent MERGE operation for the two lists remaining after having removed the chosen smaller head element.

```
$\textsc{Merge}(L, L') =$
  if $L = \emptyset$ or $L' = \emptyset$ then result $:= \iota l(l \in \{L, L'\}$ and $l \neq \emptyset)$
  elseif $\mathit{head}(L) \leq \mathit{head}(L')$ then
    let $x = \textsc{Merge}(\mathit{tail}(L), L')$ in result $:= \mathit{concatenate}(\mathit{head}(L), x)$
  else
    let $x = \textsc{Merge}(L, \mathit{tail}(L'))$ in result $:= \mathit{concatenate}(\mathit{head}(L'), x)$
```

For a data refinement of MERGESORT and MERGE see Exercise 4.1.6.

Other ways of adding recursion to ASMs. The inductive semantics for turbo ASMs in Table 2.2 describes the semantical basis for Schmid's AsmGofer system [390]. A slightly different approach to recursion is taken in XASM [17]. In this section, we point out the essential differences. We focus just on recursion and omit other features of XASM like the evaluation of terms with side effects (update sets) or the so-called external functions of XASM.

For the definition of the semantics of recursive calls in XASM we need in addition to the predicate yields$(P, \mathfrak{A}, \zeta, U)$ two new predicates run$(P, \mathfrak{A}, \zeta, \mathfrak{B})$ and final$(P, \mathfrak{A}, \zeta)$. The predicate run$(P, \mathfrak{A}, \zeta, \mathfrak{B})$ means that there exists a finite run of P from $\mathfrak{A}$ into state $\mathfrak{B}$ where the free variables of P are defined in the environment ζ. The predicate final$(P, \mathfrak{A}, \zeta)$ means that $\mathfrak{A}$ is a final state for P under ζ.

The predicates yields$(P, \mathfrak{A}, \zeta, U)$, run$(P, \mathfrak{A}, \zeta, \mathfrak{B})$ and final$(P, \mathfrak{A}, \zeta)$ are defined by *simultaneous* induction. The clauses for "yields" are the same as in Table 2.2 except for rule calls. The new clauses are listed in Table 4.1.

Table 4.1 Inductive definition of the semantics of XASM rule calls

$\dfrac{}{\mathrm{run}(P, \mathfrak{A}, \zeta, \mathfrak{A})}$	
$\dfrac{\mathrm{run}(P, \mathfrak{A}, \zeta, \mathfrak{B}) \quad \mathrm{yields}(P, \mathfrak{B}, \zeta, U)}{\mathrm{run}(P, \mathfrak{A}, \zeta, \mathfrak{B} + U)}$	if U is consistent
$\dfrac{\mathrm{yields}(P, \mathfrak{A}, \zeta, \emptyset)}{\mathrm{final}(P, \mathfrak{A}, \zeta)}$	
$\dfrac{\mathrm{run}(P, \mathfrak{A}, \eta, \mathfrak{B}) \quad \mathrm{final}(P, \mathfrak{B}, \eta)}{\mathrm{yields}(r(t), \mathfrak{A}, \zeta, \mathfrak{B} - \mathfrak{A})}$	where $r(x) = P$ is a rule declaration, $a = [\![t]\!]_{\zeta}^{\mathfrak{A}}$ and $\eta = \zeta[x \mapsto a]$

The predicates “yields” and “run” mutually depend on each other. In the definition of a finite “run” the predicate “yields” is used. Conversely, in the definition of “yields” for rule calls, the predicate “run” is used.

How does a (possibly recursive) rule call $r(t)$ in XASM work? Assume that $r(x) = P$ is a rule declaration. First the argument t is evaluated in the current state $\mathfrak{A}$ and the current environment ζ. The value of t is assigned to the variable x in a new environment η. Then a new run is started for the body P in state $\mathfrak{A}$ with the environment η. If after finitely many steps, the run terminates in a final state $\mathfrak{B}$ (where P yields an empty update set), then the rule call $r(t)$ yields the difference between the final state $\mathfrak{B}$ and the initial state $\mathfrak{A}$.

Hence the essential difference between a rule call in XASM and a rule call of a turbo ASM is that in XASM the rule call means the *repeated* execution of the body, whereas for turbo ASMs it is just the one-time execution. In the presence of sequential composition (**seq**) the two approaches have the same expressive power, since the iterated execution of an ASM can be obtained as one step of an enclosing turbo ASM. From the logical point of view, the turbo ASM calls are simpler, since they just mean that the call has to be replaced by its body (see Sect. 8.1.1).

4.1.3 Analysis of Turbo ASM Steps

In this section we analyze the runs which are mapped to single steps of turbo ASMs, elucidating the microsteps which are hidden in a turbo ASM step.

To focus on the crucial aspects and without loss of generality we disregard in this section the **choose** construct, given that its effect can be replaced by the use of external selection functions. The inductive definition of the big-step semantics of turbo ASMs in Fig. 2.2 shows that the microsteps of a turbo ASM step are either basic ASM steps – yielding a set of updates – or the steps encapsulated in the execution of sequential machines P **seq** Q or of submachine calls $r(t)$. Purely sequential machines, i.e. defined without **par** or **forall**, represent classical imperative programs with recursive procedure calls.

Table 4.2 Partial evaluation of turbo ASM rules

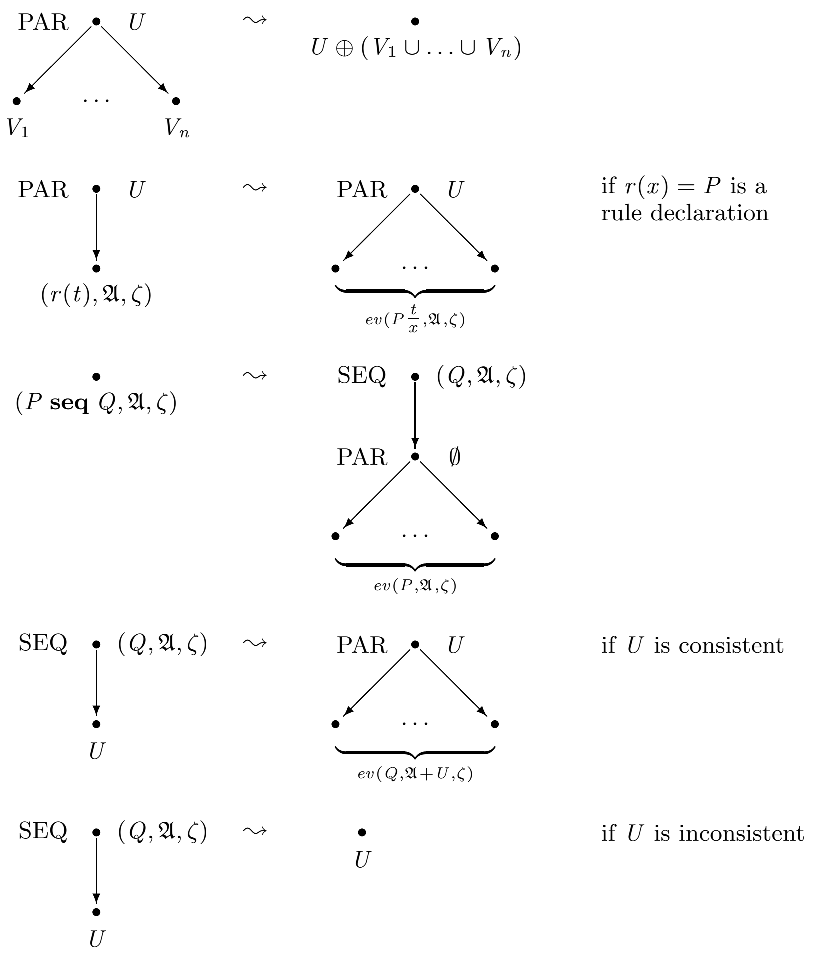
$$
\begin{array}{l}
ev(\mathbf{skip}, \mathfrak{A}, \zeta) = \emptyset \\
ev(f(s) := t, \mathfrak{A}, \zeta) = \{((f, a), b)\} \quad \text{where } a = \llbracket s \rrbracket^{\mathfrak{A}}_{\zeta} \text{ and } b = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta} \\
ev(P \ \mathbf{par}\ Q, \mathfrak{A}, \zeta) = ev(P, \mathfrak{A}, \zeta) \cup ev(Q, \mathfrak{A}, \zeta) \\
ev(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, \zeta) = \begin{cases} ev(P, \mathfrak{A}, \zeta), & \text{if } \llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta} = true; \\ ev(Q, \mathfrak{A}, \zeta), & \text{otherwise.} \end{cases} \\
ev(\mathbf{let}\ x = t\ \mathbf{in}\ P, \mathfrak{A}, \zeta) = ev(P, \mathfrak{A}, \zeta[x \mapsto a]) \quad \text{where } a = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta} \\
ev(\mathbf{forall}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P, \mathfrak{A}, \zeta) = \bigcup_{a \in range(x, \varphi, \mathfrak{A}, \zeta)} ev(P, \mathfrak{A}, \zeta[x \mapsto a]) \\
ev(P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta) = \{(P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta)\} \\
ev(r(t), \mathfrak{A}, \zeta) = \{(r(t), \mathfrak{A}, \zeta)\}
\end{array}
$$

In computing a (macro)step of a purely sequential turbo ASM P in state $\mathfrak{A}$, the intermediate states can be analyzed as pairs (U, S) of the update set U accumulated so far and a sequence (stack) S of frames $(P, \mathfrak{B}, \zeta)$ still to be executed, each consisting of the residual program P, the state $\mathfrak{B}$ in which it has to be executed and the environment (interpretation of free variables) ζ for the execution. Starting with $(\emptyset, (P, \mathfrak{A}, \zeta))$, the intermediate micro-computation is finished when the stack has become empty, and the accumulated update set is the yield of the computed turbo macrostep. In computing macrosteps of purely parallel machines, i.e. defined without **seq**, one encounters successively at each level of submachine call unfolding a computed update set (the yield of a basic ASM step, all of which have to be collected at the end to form the turbo yield) and a set of call frames $(r(t), \mathfrak{A}, \zeta)$ of the rule calls still to be unfolded. These intermediate states are naturally represented in a tree with one level for each unfolding.

Since in general several sequential machines can run in parallel, to label the nodes in the trees which represent intermediate states during the computation of a turbo macrostep we need besides update sets (to which we add the keyword PAR to remind us that this update set has to be collected as part of the final yield) and *call frames* $(r(t), \mathfrak{A}, \zeta)$ also *sequential frames* $(P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta)$. To reflect the moments where a new turbo subcomputation step appears we define in Table 4.2 a partial *ev*aluation function for turbo ASMs, which differs from the big-step semantics of turbo ASMs defined in Fig. 2.2 by leaving rule calls and sequential rules unevaluated.

Figure 4.3 illustrates the five types of microsteps whose iteration describes a macrostep of a turbo program. The basic ASM TurboMicroStep, not recursive and defined below without using **seq** or submachine calls, computes the sequence of microsteps of any macrostep of a turbo ASM which yields an update set U. Its initial state is defined for any turbo program P applied in state $\mathfrak{A}$ with environment ζ as the following tree *StartTree*$(P, \mathfrak{A}, \zeta)$: the root is labeled with $(\mathrm{PAR}, \emptyset)$ and has a subtree whose nodes are labeled with

Table 4.3 Operations on PAR/SEQ trees for hidden turbo ASM steps


PAR
U
V_1
V_n
$U \oplus (V_1 \cup \ldots \cup V_n)$
PAR
U
$(r(t), \mathfrak{A}, \zeta)$
PAR
U
$ev(P\frac{t}{x}, \mathfrak{A}, \zeta)$
if $r(x) = P$ is a rule declaration
$(P$ **seq** $Q, \mathfrak{A}, \zeta)$
SEQ
$(Q, \mathfrak{A}, \zeta)$
PAR
$\emptyset$
$ev(P, \mathfrak{A}, \zeta)$
SEQ
$(Q, \mathfrak{A}, \zeta)$
U
PAR
U
$ev(Q, \mathfrak{A}+U, \zeta)$
if U is consistent
SEQ
$(Q, \mathfrak{A}, \zeta)$
U
U
if U is inconsistent

the elements of $\overline{ev}(P, \mathfrak{A}, \zeta)$, denoting the list of elements of $ev(P, \mathfrak{A}, \zeta)$ with updates combined into a single set:

$$\overline{ev}(P, \mathfrak{A}, \zeta) = [U, F_1, \ldots, F_k] \text{ for } ev(P, \mathfrak{A}, \zeta) = U \cup \{F_1, \ldots, F_k\}$$

where U is an update set and F_i are frames.

At each frame TurboMicroStep performs a further microstep. Rule SpawnCall unfolds one submachine call. Rule SpawnSeqBegin starts the computation of the first submachine P of a sequential machine P **seq** Q. It transforms the node labeled with the sequential frame $(P$ **seq** $Q, \mathfrak{A}, \zeta)$ into a SEQ subtree whose root records the "residual" frame $(Q, \mathfrak{A}, \zeta)$ and has as subtree $StartTree(P, \mathfrak{A}, \zeta)$. Once the first submachine has successfully computed a consistent update set U, the second submachine Q is going to be executed by rule SpawnSeqCont, which transforms the SEQ tree into the continuation tree $ContTree(U, (Q, \mathfrak{A} + U, \zeta))$, which is defined like $StartTree(Q, \mathfrak{A} + U, \zeta)$ except for recording in its root the intermediate update set U instead of the $\emptyset$. In case of an inconsistent yield U, this failure set is collected by rule CollectSeqFail to be collected by the parent node via rule Collect.

```
TurboMicroStep = if $\exists n \in NODE\ (label(n) \notin UpdateSet)$ then
  forall $n \in NODE$ do
    SpawnCall($n$)
    SpawnSeqBegin($n$)
    SpawnSeqCont($n$)
    CollectSeqFail($n$)
    Collect($n$)
```

Since the three spawn rules are variations of the OccamParSpawn machine on p. 43, with "activation" of nodes meaning here to *label* them by the appropriate frame, we use without further explanation tree extension macros like "create children of n for F" or "insert m as child of n". As usual we write $children(n)$ for the set of all nodes m such that $parent(m) = n$. If the set contains exactly one element, we denote it by $child(n)$. The following submachines of TurboMicroStep formalize the tree transformation steps illustrated in Table 4.3.

```
SpawnCall($n$) = if $label(n) = (\mathrm{PAR}, U)$ then
  forall $m \in children(n)$ with $label(m)$ is a call frame do
    let $(r(t), \mathfrak{A}, \zeta) = label(m)$, $V \cup Frames = ev(body(r)\frac{t}{x}, \mathfrak{A}, \zeta)$ in
      $label(m) := V$
      create children of $n$ for $Frames$

SpawnSeqBegin($n$) = if $label(n) = (P$ seq $Q, \mathfrak{A}, \zeta)$ then
  $label(n) := (\mathrm{SEQ}, (Q, \mathfrak{A}, \zeta))$
  let $m = new(NODE)$ in
    insert $m$ as child of $n$
    $label(m) := (\mathrm{PAR}, \emptyset)$
    create children of $m$ for $\overline{ev}(P, \mathfrak{A}, \zeta)$

SpawnSeqCont($n$) =
  if $label(n) = (\mathrm{SEQ}, (Q, \mathfrak{A}, \zeta))$ and $label(child(n)) \in UpdateSet$
    then let $U = label(child(n))$ in
      if $consistent(U)$ then
        $label(n) := (\mathrm{PAR}, U)$
        let $V \cup Frames = ev(Q, \mathfrak{A} + U, \zeta)$ in
          $label(child(n)) := V$
```

```
              create children of $n$ for $\mathit{Frames}$
    $\textsc{CollectSeqFail}(n) =$
      if $\mathit{label}(n) = (\mathrm{SEQ}, (Q, \mathfrak{A}, \zeta))$ and $\mathit{label}(\mathit{child}(n)) \notin \mathit{UpdateSet}$
        then $\mathit{label}(n) := \mathit{label}(\mathit{child}(n))$

    $\textsc{Collect}(n) =$
      if $\mathit{label}(n) = (\mathrm{PAR}, U)$ and $\forall m \in \mathit{children}(n)\, (\mathit{label}(m) \in \mathit{UpdateSet})$
      then $\mathit{label}(n) := U \oplus \bigcup_{m \in \mathit{children}(n)} \mathit{label}(m)$
```

Proposition 4.1.1 (Turbo step analysis). For every turbo ASM P, state $\mathfrak{A}$, environment ζ and update set U it holds that P in $\mathfrak{A}$ with environment ζ yields in one step U if and only if TurboMicroStep started in $\mathit{StartTree}(P, \mathfrak{A}, \zeta)$ terminates with the tree consisting of its root labeled by U.

Proof. The claim from left to right follows by an induction on the big-step semantics of turbo ASMs defined in Table 2.2. The other direction follows by an induction on the TurboMicroStep computation. For the inductive steps in the case of SEQ nodes or nodes labeled with a call frame or a sequential frame use the following lemma.

Lemma 4.1.3. If $ev(P, \mathfrak{A}, \zeta) = U \cup \{(P_i, \mathfrak{A}, \eta_i) \mid 1 \leq i \leq k\}$ and $\mathit{yields}(P_i, \mathfrak{A}, \eta_i, V_i)$, then $\mathit{yields}(P, \mathfrak{A}, \zeta, U \cup \bigcup_{1 \leq i \leq k} V_i)$.

The lemma follows from the definition of ev in Table 4.2 by an induction on the size of P. For details (though formulated in slightly different terms) see [203, Sect. 4]. □

Problem 15 (Definition of the notion of transactions). Use turbo ASMs for an abstract definition of the notion of database transaction, exploiting the black-box view to hide events within transactions. For a detailed analysis of the problem and an async ASM model for transactions see [368].

4.1.4 Exercises

Exercise 4.1.1 (Problems with ASM programming solutions for P seq Q). Given two basic ASMs P, Q, define a basic control state ASM which simulates P **seq** Q. Consider the problem of initialization when your machine is to be called successively by different machines. Consider the problems when one tries to incorporate, say, a $(1,2)$-refinement P' of P and a $(1,3)$-refinement Q' of Q into (a refinement $(P \textbf{ seq } Q)'$ of) P **seq** Q.

Exercise 4.1.2. Prove Lemma 4.1.2.

Exercise 4.1.3. Rephrase the proof for Theorem 4.1.1 replacing the function *out* by the value returning mechanism $l \leftarrow R(a_1, \ldots, a_n)$.

Exercise 4.1.4. Construct an example where **let** $\{x_1 = R_1, x_2 = R_2\}$ **in** S is different from **let** $x_1 = R_1$ **in** (**let** $x_2 = R_2$ **in** S).

Exercise 4.1.5. (Procedural refinement of list partitioning in Quicksort). Refine QUICKSORT to a control state turbo ASM where the partitioning of L into $L_{<head(L)}$ and $L_{\geq head(L)}$ is computed using the following basic ASM $Partition(l, h, p)$, working on the representation of lists as functions $L\colon [r, s] \rightarrow VAL$ from intervals of natural numbers to a set of values. When $r < s$, $Partition$ is started with the search boundaries $l = r, h = s$ and the list head $pivot = L(r)$. It terminates when reaching $l = h$ with $L(l) = pivot$, all L-elements smaller than the pivot to the left of l, and all the others at l or to the right of l. Until reaching $l = h$, the partitioning procedure alternates between searching from above for list elements $L(h) \leq pivot$ and searching from below for list elements $L(l) \geq pivot$. When such an element is encountered and it is $\neq$ the element at the other current search boundary – one of them is the pivot – then the boundary elements $L(l), L(h)$ are swapped (using the SWAP-machine from p. 40) and the search switches to the other boundary. When $L(h) \leq pivot \leq L(l) \leq L(h)$ before $l = h$ is encountered (as happens if pivot has multiple occurrences in the list), h can be decreased by one. Show the refinement to be correct.

```
PARTITION($l, h, pivot$) =
  if $L(h) > pivot$ then $h := h - 1$
  elseif $L(l) < pivot$ then $l := l + 1$
  elseif $L(l) > L(h)$ then SWAP($L(l), L(h)$)
  elseif $l < h$ then $h := h - 1$
```

Exercise 4.1.6. (⇝ CD) (Data refinement of MERGESORT and MERGE). Prove that the following definition of the equivalence of corresponding locations provides a $(1, 1)$-refinement to a model where lists are represented as functions $L\colon [l, h] \rightarrow VAL$ from intervals of natural numbers to a set of values.

- $L = \emptyset \equiv l > h$, $\mid L \mid \leq 1 \equiv l \geq h$
- $head(L) = L(l)$, $tail(L) = L \setminus \{(l, L(l))\}$
- $LeftHalf(L) = [l, half(l + h)]$, $RightHalf(L) = [half(l + h) + 1, h]$ where $half(2x) = x$, $half(2x + 1) = x + 1$ (for example)
- $concatenate(v, L) = \{(l, v)\} \cup RightShift(L, 1)$ where $RightShift(L, n) = \lambda x.L(x - n)$

Exercise 4.1.7. Refine the turbo ASM of Exercise 4.1.6 to an executable program in the programming language of your choice.

Exercise 4.1.8. Define TURBOMICROSTEP as composition of a submachine for purely sequential turbo ASMs and a machine for purely parallel turbo ASMs.

4.2 Abstract State Processes (Interleaving)

By interleaving evolving programs, in this section we integrate into the ASM model of synchronous parallel rule execution a *white-box view of subcomputations* which allows one to control at the top level the single steps of a structured component program. This naturally leads us to provide a name for the program constructor which describes the non-deterministic rule scheduling known as *interleaving*, viewed as a mechanism to let multiple processes proceed independently from each other, one at a time (mono-agent view of "parallel" processes).[15] This entails another constructor, namely for *selective synchronization* whereby interleaved processes can be constrained to act in parallel only if they all contribute to a selected update (read: "share a selected event"). This leads us to extend basic ASMs to *Abstract State Processes* (ASPs) by adding three process-algebraic operators **then**, **intlea** and **sync** (t).

To describe the program evolution of an ASP implied by the white-box view of sequential execution steps we adopt the structured programming approach. Therefore, to define the semantics of an ASP construct P we have to indicate which pair (P', U) of the so-called *residual* program P' and of the update set U it yields in an arbitrary state $\mathfrak{A}$. We concentrate our attention here upon the definition of the semantics of ASPs, given that the extension of the syntax of ASMs to that of ASPs is a routine matter. For the sake of clarity we split the definition into two tables. Table 4.4, which we shall explain below, extends Table 2.2 to derive statements of the form yields$(P, \mathfrak{A}, P', U)$ for standard ASPs, defined by the constructs defining turbo ASMs where we distinguish white-box sequencing from its turbo black-box version by writing **then** instead of **seq**.

The execution of **skip** and assignment processes yields the empty residual process **nil**. A conditional process as defined in Table 4.4 blocks other processes with which it may be synchronized if, in the case where the condition is true, its subprocess P, or otherwise Q, cannot proceed. Therefore, one has to distinguish two interpretations of **if** *Cond* **then** R. The *persistent* one is defined as **if** *Cond* **then** R **else nil**; it has a blocking effect in case *Cond* is false, since no inference rule is provided to execute the empty program **nil**. The *transient* version is defined as **if** *Cond* **then** R **else skip**, which in case *Cond* is false uses the inference rule for **skip**. We use the persistent version as the default. One may compare this with the blocking evaluation of guards, e.g. in the high-level design language COLD [197], whereas in ASMs the rule of a "blocked" process does not prevent other rules from being executed in parallel.

For the formulation of processes which involve the manipulation of logical variables it is notationally convenient to view states $\mathfrak{A}$ as sets of pairs (l, v)

[15] The same integration of interleaving into the synchronous parallelism of basic ASMs appears in [379].

Table 4.4 Inductive definition of the semantics of standard ASP rules

$\dfrac{}{\text{yields}(\mathbf{skip}, \mathfrak{A}, \mathbf{nil}, \emptyset)}$	
$\dfrac{}{\text{yields}(f(s_1, \ldots, s_n) := t, \mathfrak{A}, \mathbf{nil}, \{((f, (\llbracket s_1 \rrbracket^{\mathfrak{A}}, \ldots, \llbracket s_n \rrbracket^{\mathfrak{A}})), \llbracket t \rrbracket^{\mathfrak{A}})\})}$	
$\dfrac{\text{yields}(P, \mathfrak{A}, P', U)}{\text{yields}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, P', U)}$	$\llbracket \varphi \rrbracket^{\mathfrak{A}} = true$
$\dfrac{\text{yields}(Q, \mathfrak{A}, Q', V)}{\text{yields}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, Q', V)}$	$\llbracket \varphi \rrbracket^{\mathfrak{A}} = false$
$\dfrac{\text{yields}(P\tfrac{x'}{x}, \mathfrak{A}[x' \mapsto a], P', U)}{\text{yields}(\mathbf{let}\ x = t\ \mathbf{in}\ P, \mathfrak{A}, P', U \cup \{(x', a)\})}$	$a = \llbracket t \rrbracket^{\mathfrak{A}}$ x' fresh
$\dfrac{\text{yields}(P\tfrac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \text{for each } a \in I = range(x, \varphi, \mathfrak{A})}{\text{yields}(\mathbf{par}\{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{par}\{P_a \mid a \in I\}, \bigcup_{a \in I} U_a \cup \{(y_a, a)\})}$	y_a fresh
$\dfrac{\text{yields}(P\tfrac{x'}{x}, \mathfrak{A}[x' \mapsto a], P', U) \quad \text{for some } a \in I = range(x, \varphi, \mathfrak{A})}{\text{yields}(\mathbf{choose}\{P(x) \mid \varphi(x)\}, \mathfrak{A}, P', U \cup \{(x', a)\})}$	x' fresh
$\dfrac{\text{yields}(P, \mathfrak{A}, P', U)}{\text{yields}(P\ \mathbf{then}\ Q, \mathfrak{A}, P'\ \mathbf{then}\ Q, U)}$	$P' \neq \mathbf{nil}$
$\dfrac{\text{yields}(P, \mathfrak{A}, \mathbf{nil}, U)}{\text{yields}(P\ \mathbf{then}\ Q, \mathfrak{A}, Q, U)}$	
$\dfrac{\text{yields}(P\tfrac{t_1 \cdots t_n}{x_1 \cdots x_n}, \mathfrak{A}, P', U)}{\text{yields}(r(t_1, \ldots, t_n), \mathfrak{A}, P', U)}$	$r(x_1, \ldots, x_n)$ $= P$

of locations l and their values v and to identify free variables with 0-ary function symbols so that their interpretation is incorporated into $\mathfrak{A}$, instead of being kept in a separate environment function ζ. Thus we write $\mathfrak{A}[x' \mapsto a]$ for the extension of $\mathfrak{A}$ by the new location x' with value a.

In the **let**-construct local variables appear to which precomputed values are assigned. In basic or turbo ASMs this preliminary computation step remains implicit as part of the unique computation step associated with the machine **let** $x = t$ **in** P, whose execution implies a form of sequentialization which is typical for the call-by-value discipline, namely to first compute t in the given state and then to execute P with the computed value recorded in the local variable x. These variables are called logical because their binding to the current value of t holds only for the atomic execution of P. Since in ASPs such a program P is not executed atomically, but may lead after one step to a residual program P' which involves further steps, the incarnation x' of the variable x with its interpretation by the value t has to survive until the residual process has become empty (reduced to **nil**). This holds analogously also for the other ASP constructors.

We therefore use for each execution of an ASP constructor $c(x)$ P in a given state a fresh instance of x, say x', standing for a new 0-ary function which records for the entire execution of the constructor body P the value assigned in the given state to the parameter x. Formally this makes the signature of ASPs dynamic, though the dynamics is restricted to creating new incarnations of local variables, whereas in basic or turbo ASMs the signature is static. By imposing the condition of x' being fresh (meaning by this that it is sufficiently new not to be mixed up with any other variable,[16] namely that it is not used before and not simultaneously anywhere else) we guarantee that the currently determined value for x will not collide with any other value determined in a different process or in a different step for the same parameter x ("same" in the syntactic sense). For brevity we write x, although we allow it to denote a tuple of parameters.

For notational uniformity we write in this section **par** $\{P(x) \mid \varphi(x)\}$ instead of **forall** x **with** φ **do** P. In case φ evaluates to finitely many elements, **par** $\{P_1, \ldots, P_n\}$ stands for **par** $\{P(x) \mid \varphi(x)\}$, where $\{P(x) \mid \varphi(x)\} = \{P_1, \ldots, P_n\}$. This avoids having to fuss with fixed parameter instances. We do so similarly for **choose** $\{P(x) \mid \varphi(x)\}$ and for the synchronization and interleaving operators **sync** (t) and **intlea** defined below. See Table 4.6 for a summary of some alternative notations borrowed from the process-algebraic literature. When applying ASP operators to sets of processes, it is notationally convenient to assume the following implicit transformation, which will not be mentioned furthermore: whenever the application of an inference rule leads to a residual program *oper*$\{P\}$, where *oper* is any of the operators and $\{P\}$ is a singleton set of processes, the residual program is considered as automatically rewritten into P. Similarly, *oper*$\{\}$ is rewritten into **nil**. By $\varphi\frac{t}{x}$ we denote the result of replacing all free occurrences of the variable x in φ by the term t.

choose-processes as defined in Table 4.4 are blocking in case there is nothing to choose from (empty choice set), since for this case we provide no inference rule. This is in contrast to the basic or turbo ASM **choose**-construct, which in the case of an empty choice set is treated as equivalent to **skip**.

The submachine call defined in Table 4.4 is by reference. The call-by-value version can be defined using the **let**-construct.

In Table 4.5 we add to standard ASPs the interleaving operator **intlea** and the related selective synchronization operator **sync** (t).

Interleaving is a choice among processes to perform the next step where, however, the programs of the not-chosen processes remain in force for subsequent choices. To exhibit the analogy of interleaving to synchronous par-

[16] See Sect. 2.4.4 for the characterization of this use of the **import**-construct to obtain fresh local variables x'. One can use a similar expedient to the one explained in the footnote to Def. 4.1.8 to separate the "logical" (read-only) variables from the variables for locations (write-variables). It suffices to write *var*(x') instead of x' with a monadic function *var* which takes variables as arguments; to guarantee that different incarnations of x are stored in different locations it suffices to pass to *var* different arguments x', x''.

Table 4.5 Semantics of interleaving and selective synchronization

$$\frac{\mathrm{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \text{for some } a \in I = \mathit{range}(x, \varphi, \mathfrak{A})}{\mathrm{yields}(\mathbf{intlea}\ \{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{intlea}\ \{P_c \mid c \in I\}, U_a \cup \bigcup_{c \in I}\{(y_c, c)\})}$$

where y_c fresh for all $c \in I$, $P_b = P\frac{y_b}{x}$ for $b \in I \setminus \{a\}$

$$\frac{\mathrm{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \mathit{Loc}(t)^{\mathfrak{A}} \in \mathit{Loc}(U_a) \text{ for each } a \in I}{\mathrm{yields}(\mathbf{sync}\ (t)\{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{sync}\ (t)\{P_a \mid a \in I\}, \bigcup_{a \in I} U_a \cup \{(y_a, a)\})}$$

where I=$\mathit{range}(x, \varphi, \mathfrak{A})$, y_a fresh for all $a \in I$

$$\frac{\mathrm{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \mathit{Loc}(t)^{\mathfrak{A}} \notin \mathit{Loc}(U_a) \text{ for some } a \in I}{\mathrm{yields}(\mathbf{sync}\ (t)\{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{sync}\ (t)\{P_c \mid c \in I\}, U_a \cup \bigcup_{c \in I}\{(y_c, c)\})}$$

where $I = \mathit{range}(x, \varphi, \mathfrak{A})$, y_c fresh for all $c \in I$, $P_b = P\frac{y_b}{x}$ for $b \in I \setminus \{a\}$

allelism, which involves a form of universal quantification, we formulate the **intlea**-rule for an arbitrary set of processes determined by a property $\varphi(x)$. As a consequence, to determine the set of the processes P_b which have not been chosen but are put into interleaving with the residual process P_a of the one process chosen for execution, one needs to keep track of the instantiations of $P(x)$ by the elements b which satisfy the condition φ. Formally, this requires us to bind pairwise different new incarnations y_b of the parameter x to b. In the case of an explicitly given set $P_1, \ldots, P_n$ of processes where the parameter instances are fixed, our notational convention eliminates the need to reinstantiate the fixed parameter indices $1 \leq i \leq n$.

The synchronization operator **sync** (t) allows one to prevent the occurrence of actions which do not involve an update of (the location determined by) t by all the synchronized processes.

Whereas every basic ASM program yields in every state an update set (though it may be inconsistent, in which case the computation is abrupted), when executing an ASP it may happen that the current program in the current state has no yield because no axiom or inference rule can be applied. This produces a form of ASP termination which does not exist for basic ASMs, that of a (static) deadlock. It can arise in the following cases for the residual program:

– a conditional expression with blocked argument process,
– a sequential expression with blocked first argument,
– a choice expression where all alternatives are blocked,
– a parallel expression with at least one blocked process,
– an interleaving expression where all component processes are blocked.

Local sequential scheduling of subprocesses. For an illustration of the combination of interleaving with white-box sequential execution imagine a set $\mathfrak{R}$ of processes R of form

Table 4.6 Syntactic variations of some ASP constructs

choose x **with** φ **do** P $\mid \{P(x) \mid \varphi(x)\}$	**choose** $\{P(x) \mid \varphi(x)\}$
forall x **with** φ **do** P $\parallel \{P(x) \mid \varphi(x)\}$	**par** $\{P(x) \mid \varphi(x)\}$
$\mid t \mid \{P(x) \mid \varphi(x)\}$	**sync** $(t)\{P(x) \mid \varphi(x)\}$
$\mid\mid\mid \{P(x) \mid \varphi(x)\}$	**intlea** $\{P(x) \mid \varphi(x)\}$
operator$\{P_1, \ldots, P_n\}$ (**operator** $=\mid, \parallel, \mid t \mid, \mid\mid\mid$)	**operator**$\{P(x) \mid \varphi(x)\}$ **where** $\{P(x) \mid \varphi(x)\} = \{P_1, \ldots, P_n\}$

$$\mathrm{R} = \mathit{First}_R \ \textbf{then} \ \mathit{Second}_R$$

which are constrained to be interleaved such that the order of execution of the subprocesses Second_R is determined by the order of execution of the subprocesses First_R. For a local realization of such a scheduling we equip each process **self** with its instance **self**.*ticket* of a location *ticket* which keeps track of the order in which the interleaving operator chooses the subprocesses First_R for execution. This allows one to (a) locally record "when" **self** has been called to start the execution of its first subprocess, and (b) to locally advance *ticket* to the next free position in the dynamic ordering of calls of subprocesses First_R, namely by putting two updates to get and advance the current ticket in parallel with $\mathit{First}_{\textbf{self}}$: **self**.$\mathit{ticket} := \mathit{ticket}$ and $\mathit{ticket} := \mathit{ticket} + 1$. Then $\mathit{Second}_{\textbf{self}}$ can be scheduled when its ticket is, say, "displayed", i.e. when the guard **self**.$\mathit{ticket} = \mathit{display}$ is true, adding to $\mathit{Second}_{\textbf{self}}$ an update to advance the *display*. This yields the following transformation of **intlea** $(\mathfrak{R})$ into **intlea** (LocalSeqSched$(\mathfrak{R})$).

```
LocalSeqSchedule($R$) = $\mathit{First}'_R$ then $\mathit{Second}'_R$ where
  $\mathit{First}'_R$ = par $\{\mathit{First}_R$, GetAndAdvanceOrderPos$(R)\}$
  GetAndAdvanceOrderPos$(R)$ =
    par $\{R.\mathit{ticket} := \mathit{ticket}, \mathit{ticket} := \mathit{ticket} + 1\}$
  $\mathit{Second}'_R$ = if $\mathit{displayed}(R.\mathit{ticket})$ then
    par $\{\mathit{Second}_R$, AdvanceDisplay$(R)\}$
  $\mathit{displayed}(\mathit{Ticket}) = (\mathit{Ticket} = \mathit{display})$
  AdvanceDisplay$(R) = (\mathit{display} := \mathit{display} + 1)$
```

Handshaking. For an illustration of the effect of selective synchronization in the ASP context consider again Exercise 3.2.5 on process communication via rendez-vous. If the ASM defined there is interpreted as an ASP by replacing the outer **par** by **sync** (g), the two subprocesses are allowed to proceed independently of each other, in an interleaved manner, as long as their updates do not affect the synchronization gate g, whereas the Handshaking ASM forces both processes to always execute simultaneously and with a consistent gate update.

```
AspHandshaking$(P, \varphi, s, Q, \psi, t)$ = sync $(g)\{R, S\}$ where
  $R$ = choose $x$ with $\varphi(x)$ in $(g := s(x)$ then $P(x))$
  $S$ = choose $y$ with $\psi(y)$ in $(g := t(y)$ then $Q(y))$
```

This ASP formulation of handshaking realizes the "agreement on values offered at a gate" by the consistency condition for the gate updates. If one wants to faithfully reflect also that gates, in the process-algebraic view, are only virtually updated, serving only as communication medium, one can declare the updates of g to be transient, i.e. not relevant for the resulting state transformation. This can be done by simply not considering these transient updates in the definition of the next-state function, which associates new states to given states and update sets, in analogy to the restriction of update sets for the treatment of local functions and for defining the error handling of turbo ASMs in Sect. 4.1.2.

Problem 16 (Analysis of ASP runs). Characterize the possible runs of ASPs, similarly to the characterization of turbo ASM runs in Sect. 4.1.3.

Problem 17 (ASP refinement techniques). Adapt the structured and proof-oriented process-algebra refinement techniques defined in [176] to ASPs.

Problem 18 (ASP verification techniques). Investigate which process-algebraic proof rules can be generalized to support deductive and possibly mechanically verified reasoning about ASP runs.

Problem 19 (ASP implementation). Extend an implementation of basic or turbo ASMs to an implementation of ASPs to pave the way for an implementation of (an approximation of) asynchronous ASMs defined in Chap. 6.

Sources and Historical Remarks

The definition of turbo ASMs, though not the name, appeared in [134] where also related notions in the ASM literature are discussed. The section on modeling recursion by turbo ASMs is extracted from [95]. The turbo ASM composition concepts are realized in the ASM engine AsmGofer [390] and apparently also in AsmL [201]. The analysis of turbo ASMs is based upon [203]. A logic for turbo ASMs appeared in [405] and is investigated in Sect. 8.1.1. ASPs appeared for the first time in [67] where references can be found to other approaches to combine state-based methods with behavioral process-algebraic concepts.

5 Synchronous Multi-Agent ASMs

In this chapter[1] the single-agent ASMs of Chaps. 3, 4 are extended to multi-agent synchronous ASMs (*sync* ASM) which support modularity for the design of large systems. We illustrate this by sync ASMs for two popular benchmark case studies for the verified design of reactive control systems: a controller for the Production Cell [323] (Sect. 5.1), solving a typical industrial plant control problem, and a real-time gate controller for the Generalized Railroad Crossing [277] (Sect. 5.2), both controllers coming with to-be-verified safety, liveness and performance requirements. Although the case studies are really small (leading to roughly 1 K lines of controller code), they allow us to explain how to apply practically useful software architecture principles to modularize systems, starting from ground model ASMs and leading to verified code. The chapter can be read independently of the preceding Chaps. 4 and 3 and most of Chap. 2; it suffices to know the definition of basic ASMs and of ASM refinements.

A multi-agent synchronous ASM is defined as a set of agents which execute their own basic or turbo ASMs in parallel, synchronized using an implicit global system clock. Semantically a sync ASM is equivalent to the set of all its constituent single-agent ASMs, operating in the global states over the union of the signatures of each component. Examples *par excellence* are offered by programs in synchronous programming languages [266] where runs are totally ordered sets of "logical instants" at which "events" occur (read: sets of simultaneous occurrences of possibly value-carrying signals through which the programs communicate among themselves and with the environment) to which all subprocesses of the executed program react instantaneously. The sequence of events determining a run is the sequence of states forming the run of the underlying multi-agent synchronous ASM, where the global clock tick (a built-in signal which is supposed to be present in every event) plays the role of a step counter.

The practical usefulness of sync ASMs derives from the possibility of equipping each agent with its own set of states and rules and of defining and analyzing the interaction between components using precise interfaces over common locations. To denote the instance of a function f for an agent a we write $a.f$ and often omit mentioning a when it is clear from the context.

[1] Lecture slides can be found in `ProdCell` (⇝ CD), `GateController` (⇝ CD).

5.1 Robot Controller Case Study

This section reviews the ASM solution [120, 332] for the Production Cell control problem [323], which together with the ASM solution [43] for the frequently used Steam Boiler case study [9] constituted the first explicit test of the integratability of the ASM method into the various phases of an industrial software development cycle (see Sect. 9.4.1 for historical details). The declared goal was to cover the following major development steps (see the V-scheme levels in Fig. 2.3):

1. elicit the given requirements by capturing them into a ground model ASM, inspectable by an application-domain expert,
2. stepwise refine this abstract model to executable (in this case C++) code whose module structure reflects the application-domain-driven component architecture of the ground model,
3. mathematically verify the required safety, performance and liveness properties (which were model checked and PVS-verified in [424, 362, 207]),
4. validate the code by extensive experimentation with the production cell simulator built at the FZI in Karlsruhe,
5. provide for maintenance purposes a transparent and complete documentation of the design (which in fact was submitted as an inspection case study to a Dagstuhl seminar on “Practical Methods for Code Documentation and Inspection” [117]).

Since in this section a production cell controller is developed to illustrate how sync ASMs enhance the modularity of specifications and their implementations, we concentrate here upon the design – ground model construction and its refinement[2] – and refer the reader to the above-mentioned publications concerning the verification, validation and documentation aspects for which this book presents other more challenging examples. One example is in Sect. 5.2.2, which explains the verification of a real-time controller (there the design task is pretty obvious).

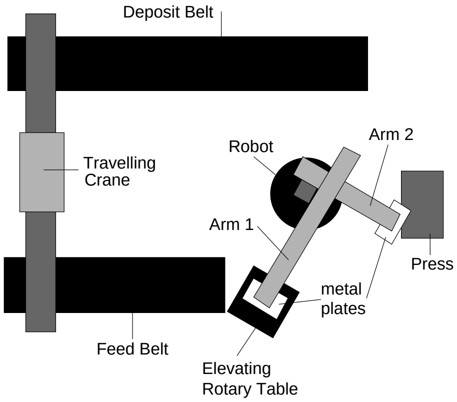
5.1.1 Production Cell Ground Model

We start by extracting from the task description [323] what are the agents and the basic objects, operations and interactions of the system.

> . . . the production cell is composed of two conveyor belts, a positioning table, a two-armed robot, a press, and a traveling crane. Metal plates inserted in the cell via the feed belt are moved to the press. There, they are forged and then brought out of the cell via the other belt and the crane.

[2] For a further analysis of the compositional aspects of the production cell ASM in terms of its submachines see [353].

Fig. 5.1 Production cell plant



We reflect the structure of the plant (see Fig. 5.1) by mapping each system device to an agent executing a basic ASM, to be refined to a C++-module of the final controller.[3] The reader may wonder whether it is appropriate to synchronize the production cell agents. Their runs could certainly be considered as those of an asynchronous ASM as defined in Chap. 6, but at the price of proof complications which are unnecessary semantically (from the system behavior viewpoint), given that the dependences among the production cell components are sequential – each device interacts in order with exactly one predecessor and one successor – and determined by the order in which the metal blanks pass through the system, as will become clear from the analysis of the device task descriptions below.

Transport belts. Although in [323] the two transport belts come with different descriptions, for reuse purposes we define an abstract TransportBelt and instantiate it to a FeedBelt and a DepositBelt.

> The task of the feed belt consists in transporting metal blanks to the elevating rotary table. The belt is powered by an electric motor, which can be started up or stopped by the control program. A photoelectric cell is installed at the end of the belt; it indicates whether a blank has entered or left the final part of the belt. . . . the photoelectric cells switch on when a plate intercepts the light ray. Just after the plate has completely passed through it, the light barrier switches off. At this precise moment, the plate . . . has just left the belt to land on the elevating rotary table – provided of course that the latter machine is correctly positioned . . . the feed belt may only

[3] We skip here the rather unnatural traveling crane specification which has been added to the case study only to make the system closed for the simulator.

Fig. 5.2 TransportBelt ground model

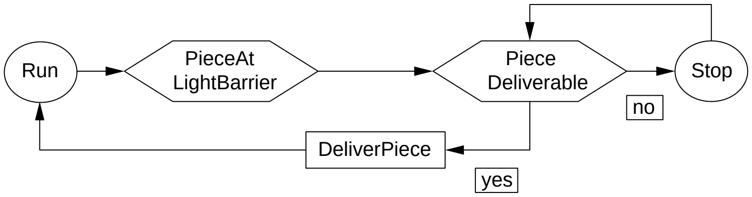
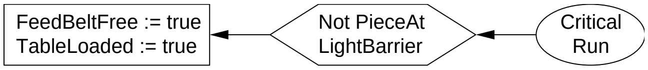


Fig. 5.3 Durative version of DeliverPiece



> convey a blank through its light barrier, if the table is in loading position ... do not put blanks on the table, if it is already loaded ...

Abstracting from the motors one arrives at the control state ASM in Fig. 5.2 with a monitored function *PieceAtLightBarrier* representing the sensor values and delivery macros *PieceDeliverable*, *DeliverPiece*.

The formalization of the above feed belt task description can be completed by instantiating the delivery macros as follows. We define *PieceDeliverable* as a derived predicate *TableReadyForLoading* which interfaces the elevating rotary table (ERT).

$$\begin{array}{l} \mathit{TableReadyForLoading} \iff \\ \quad \mathit{ERT.ctl\_state} = \mathit{StoppedInLoadPos}\ \textbf{and not}\ \mathit{TableLoaded} \end{array}$$

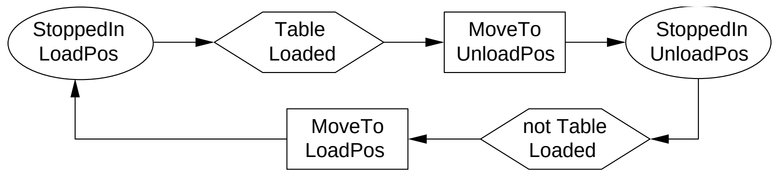
With the definition of the DeliverPiece macro in Fig. 5.3 we illustrate an abstract durative delivery action. The location *FeedBeltFree* which is monitored for the insertion of new blanks to the feed belt realizes a 0-1-counter.[4]

$$\begin{array}{l} \text{FeedBelt} = \text{TransportBelt}\ \textbf{where} \\ \quad \mathit{PieceAtLightBarrier} = (\mathit{PhotoelectricCell} = \mathit{on}) \\ \quad \mathit{PieceDeliverable} = \mathit{TableReadyForLoading} \end{array}$$

We leave it as an exercise to formulate a turbo ASM for the DeliverPiece macro in Fig. 5.3 (Exercise 5.1.1) and to instantiate TransportBelt to a DepositBelt (Exercise 5.1.2).

[4] For reasons of uniformity of exposition the feed belt ASM here slightly differs from the one in [120], which reflects that by the original requirements in [323] the feed belt is allowed to carry two pieces, one of which at most is at the light barrier. What is needed to modify the definition of TransportBelt correspondingly?

Fig. 5.4 ElevRotTable ground model



Elevating rotary table. For the ground model we again abstract from the peculiar way in which the ERT appears in the task description to rotate and lift pieces, driven by motors. This leads to capture the table task description below by the control state ASM ElevRotTable defined in Fig. 5.4, with monitored function *TableLoaded* (which is shared by the feed belt and the robot) and abstract actions *MoveToUnloadPos*, *MoveToLoadPos*. We leave it as Exercise 5.1.3 to refine the rule MoveToUnloadPos to a durative abstract action.

> The task of the elevating rotary table is to rotate the blanks by about 45 degrees and to lift them to a level where they can be picked up by the first robot arm. The vertical movement is necessary because the robot arm is located at a different level than the feed belt and because it cannot perform vertical translations. The rotation of the table is also required, because the arm's gripper is not rotary and is therefore unable to place the metal plates into the press in a straight position by itself.

Robot. The robot is the central component of the production cell.

> The robot comprises two orthogonal arms. For technical reasons, the arms are set at two different levels. Each arm can retract or extend horizontally. Both arms rotate jointly. Mobility on the horizontal plane is necessary, since elevating rotary table, press, and deposit belt are all placed at different distances from the robot's turning center. The end of each robot arm is fitted with an electromagnet that allows the arm to pick up metal plates. The robot's task consists in: taking metal blanks from the elevating rotary table to the press; transporting forged plates from the press to the deposit belt.

It is advisable to capture these robot requirements at the ground model level by a simple control state ASM Robot as defined in Fig. 5.5, where abstracting from the details of the movements of the robot or its arms, the sequence of the load or unload actions is controlled by derived or monitored functions which indicate when to start or stop a device action or moving to the position of the next device action. *Yield(DevAction)* and *Next(DevAction)*

Fig. 5.5 Robot ground model

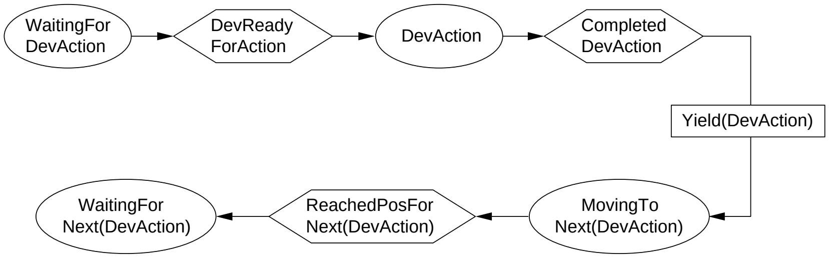


Table 5.1 Robot macros

DevAction	*Next(DevAction)*	*Yield(DevAction)*
TableUnload	*PressUnload*	*TableLoaded* := *false*
PressUnload	*DepBeltLoad*	*PressLoaded* := *false*
DepBeltLoad	*PressLoad*	*DepBeltLoadable* := *false*
PressLoad	*TableUnload*	*PressLoaded* := *true*

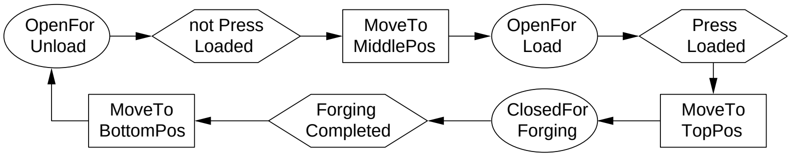
are defined by Table 5.1, formalizing the indications given in the task description for the order of the robot actions. The derived action guards represent interfaces which are defined in the ground model as follows:

```
RobotActionGuards =
  $DevReadyForUnload = DevInUnloadPos$ and $DevLoaded$ where
    $Dev \in \{Table, Press\}$
    $TableInUnloadPos = (ERT.ctl\_state = StoppedInUnloadPos)$
  $PressReadyForLoad = PressInLoadPos$ and not $PressLoaded$ where
    $PressInActionPos = (Press.ctl\_state = OpenForAction)$
    $Action \in \{Load, Unload\}$
  $DepBeltReadyForLoad = (DepBeltLoadable = true)$
```

Press. The press goes through the cycle of loading, forging and unloading under the control of the robot which loads it with blanks and retrieves forged pieces. Abstracting from the motors the press task description below can be captured by the simple control state ASM in Fig. 5.6, using the monitored function *PressLoaded* (which is controlled by the robot). Another monitored function signals when the forging has been completed.

> The task for the press is to forge metal blanks. The press consists of two horizontal plates, with the lower plate being movable along a vertical axis. The press operates by pressing the lower plate against

Fig. 5.6 PRESS ground model



> the upper plate. Because the robot arms are placed on different horizontal planes, the press has three positions. In the lower position, the press is unloaded by arm 2, while in the middle position it is loaded by arm 1. The operation of the press is coordinated with the robot arms as follows: 1. Open the press in its lower position and wait until arm 2 has retrieved the metal plate and left the press, 2. Move the lower plate to the middle position and wait until arm 1 has loaded and left the press, 3. Close the press, i.e. forge the metal plate. This processing sequence is carried out cyclically.

At the level of abstraction of the Production Cell ground model some of the safety properties required in [323] can be proved only from appropriate assumptions on the abstract device actions. We leave it as Exercise 5.1.5 to formulate such assumptions.

5.1.2 Refinement of the Production Cell Component ASMs

The refinements in this section are procedural or data refinements and are easily seen to be correct, once it is established that they concern the way the functionality of the single devices is achieved internally, maintaining the interfaces through which the ground model ASM components interact.[5] A first refinement step for movements – making them durative – has already been indicated in Exercises 5.1.1 and 5.1.3. The refinements that we introduce now instantiate the movements as driven by the actuators (electric motors and electromagnets) and as monitored via the additional sensors (switches and potentiometers) mentioned in the above task descriptions. The level of abstraction achieved by this refinement step turned out to be already appropriate for the last refinement step, a direct and structure preserving translation to C++ modules, for which we refer to [332].[6]

[5] As observed in [120] the "ground model as composition of independent submachines which interact through rigorously defined interfaces yields the possibility to combine components even if they are chosen from different levels of abstraction", a feature which has been used also for the Java/JVM component machines in [406] and is systematically exploited in applications of AsmL to "substitute low-level implementations by high-level specifications" [29] for testing and checking purposes at run-time.

[6] A main requirement for the ASM compiler developed in [391] was that the translation scheme preserves the specification structure without generating inefficient

Refining belts, ERT, press. It is easy to data-refine the FEEDBELT movement as motor-driven, namely by characterizing the control states as reflecting the motor states *FeedBeltMot* $\in \{on, off\}$, combined with a flag *Delivering* to distinguish between *Run* and *CriticalRun*.

$ctl\_state = Run \leadsto FeedBeltMot = on$ **and not** $Delivering$
$ctl\_state = CriticalRun \leadsto FeedBeltMot = on$ **and** $Delivering$
$ctl\_state = Stop \leadsto FeedBeltMot = off$

Correspondingly the refined (non-optimized) feed belt control state update $ctl\_state := Run$ becomes $FeedBeltMot := on$ **and** $Delivering := false$, etc. Refining in the same way the elevating rotary table, following the detailed indications of the above task description, leads to a (1,2)-refinement because of the independence of the motors

$TableElevationMot \in \{Idle, Up, Down\}$
$TableRotationMot \in \{Idle, clockwise, counterClockwise\}$

which drive the vertical and the rotary movement (see the disjunction defining *MovingToActionPos*). The movements come with monitored boundary values *BottomPosition, TopPosition, MinRotation, MaxRotation.*

$ctl\_state = StoppedInLoadPos \leadsto BottomPosition$ **and** $MinRotation$
  and $TableElevationMot = TableRotationMot = Idle$
$ctl\_state = StoppedInUnloadPos \leadsto TopPosition$ **and** $MaxRotation$
  and $TableElevationMot = TableRotationMot = Idle$
$ctl\_state = MovingToLoadPos \leadsto$
  $TableElevationMot = Up$ **or** $TableRotationMot = clockwise$
$ctl\_state = MovingToUnloadPos \leadsto TableElevationMot = Down$ **or**
  $TableRotationMot = counterClockwise$

We leave it as Exercise 5.1.6 to similarly refine the PRESS ground model by detailing the control states *Open/ClosedForAction* and *MovingToAnyPos* and to prove the refinements to be correct (Exercise 5.1.7).

Refining the robot. We now refine the actions of the ROBOT ground model. A *RobotRotationMot* drives the rotation, which is measured by a monitored function *Angle* with values in the intervals shown in Table 5.2 with boundary values at which the rotation motor has to stop. Motors *Arm1Mot, Arm2Mot* with values *idle, extend, retract* drive the extension or retraction of the two arms to reach either the table and the press or the press and the deposit belt. The monitored functions *ArmiExt* $(i = 1, 2)$ indicate the current arm extensions and have the significant boundary values *Arm1AtTable, ArmiAtPress, Arm2AtDepBelt*, defining the position at the device where an arm has to pick up or drop a piece. Power switches for magnets *ArmiMagnet* $\in \{on, off\}$ at

code. As a consequence the code generated by this compiler (e.g. for the Production Cell ASM) is easy to inspect, an important feature for supporting compiler inspection and verification.

the end of the arms ($i = 1, 2$) serve to pick up or release the metal pieces. The refinement in these terms of $ctl\_state = WaitingForDevAction$ as well as of $ctl\_state = MovingToDevAction$ is defined in Table 5.2. It yields a (1,1)-refinement for each robot wait and move rule (for each relevant *DevAction*), and for each device action rule it yields a (1,3)-refinement describing the succession of arm extension, action on a metal piece and arm retraction.[7] To avoid repetitions we use abbreviations which convey their meaning, like *ArmToDev* for one of *Arm1ToTable, Arm2ToDepBelt* or *ArmiToPress* ($i = 1, 2$), and similarly *ArmAtDev* for one of *Arm1AtTable, Arm2AtDepBelt* or *ArmiAtPress*. We denote by $Act(ArmAtDev)$ the (Un)Load action the robot is supposed to perform when the arm is positioned at the indicated device. *ArmsRetracted* stands for $Arm1Ext = Arm2Ext = retracted$, *RobotIdle* for $RobotRotationMot = Arm1Mot = Arm2Mot = idle$. Summarizing, RobotRefined consists of the following five groups of rules:

> RobotWaiting, RobotActionExtension, RobotActionProper, RobotActionRetraction, RobotMoving,

one for each instance of parameters.

```
RobotWaiting($DevAction$) =
  if $WaitingForDevAction$ and $DevReadyForAction$ then
    $Arm(DevAction)Mot := extend$

RobotActionExtension($ArmToDev, ArmAtDev$) =
  if $extending\ ArmToDev$ and $ArmExt = ArmAtDev$ then
    $ArmMot := idle$
    where let $mov \in \{extend, retract\}$ in $moving\ ArmToDev =$
      $(Angle = ArmToDev$ and $ArmMot = mov)$
RobotActionProper($ArmToDev, ArmAtDev$) =
  if $extended\ ArmAtDev$ then
    $ArmMagnet := on/off(ArmAtDev)$
    $ArmMot := retract$
    $Yield\ Act(ArmAtDev)$
  where
    $on/off(Arm1OverTable) = on/off(Arm2IntoPress) = on$
    $on/off(Arm2OverDepBelt) = on/off(Arm1IntoPress) = off$
    $extended\ ArmAtDev = (Angle = ArmToDev$ and
      $ArmExt = ArmAtDev$ and $ArmMot = idle)$
RobotActionRetraction($ArmToDev$) =
  if $retracting\ ArmToDev$ and $ArmExt = retracted$ then
    $RobotRotationMot := rot(Arm, Dev)$
```

[7] The placement of the *Yield* of an action as part of the proper action rule corrects an obvious oversight which entered [120] by a misleading symmetry argument for the two transport belts and was discovered in [362] when model checking the refined Production Cell ASM.

Table 5.2 Refining robot waiting/moving

WaitingFor	*Angle*	*Arm*1	*Arm*2
DevAction		*Magnet*	*Magnet*
TableUnload	*Arm*1*ToTable*	*off*	*off*
PressUnload	*Arm*2*ToPress*	*on*	*off*
DepBeltLoad	*Arm*2*ToDepBelt*	*on*	*on*
PressLoad	*Arm*1*ToPress*	*on*	*off*

and *ArmsRetracted* and *RobotIdle*

MovingTo	*RobotRotation*	*Arm*1	*Arm*2	*Angle*
DevAction	*Mot*	*Magn.*	*Magn.*	
PressUnload	*counterClock*	*on*	*off*	$[\mathit{Arm1ToTable}, \mathit{Arm2ToPress}]$
DepBeltLoad	*counterClock*	*on*	*on*	$[\mathit{Arm2ToPress}, \mathit{Arm2ToDepBelt}]$
PressLoad	*counterClock*	*on*	*off*	$[\mathit{Arm2ToDepBelt}, \mathit{Arm1ToPress}]$
TableUnload	*clockwise*	*off*	*off*	$[\mathit{Arm1ToPress}, \mathit{Arm1ToTable}]$

and *ArmsRetracted* and $\mathit{Arm1Mot} = \mathit{Arm2Mot} = \mathit{Idle}$

ReachedPos	*Press*	*DepBelt*	*Press*	*Table*
ForDevAction	*Unload*	*Load*	*Load*	*Unload*
Angle	*Arm*2*ToPress*	*Arm*2*ToDepBelt*	*Arm*1*ToPress*	*Arm*1*ToTable*

```
    $ArmMot := idle$
  where $rot(Arm, Dev) =$
    if $Arm = Arm1$ and $Dev = Press$
      then $clockwise$
      else $counterclockwise$

RobotMoving($DevAction$) =
  if $MovingToDevAction$ and $ReachedPosForDevAction$ then
    $RobotRotationMot := idle$
```

We leave it as Exercise 5.1.8 to similarly define the refined notion of (*Un*)*LoadingDevice*. To establish the required safety and liveness properties for the refined Production Cell ASM, it suffices to show that it satisfies the assumptions under which these properties have been proved for the ground model (Exercise 5.1.9).

5.1.3 Exercises

Exercise 5.1.1. Formulate a turbo ASM for the DeliverPiece macro in Fig. 5.3.

Exercise 5.1.2. (⇝ CD) Instantiate TransportBelt to a DepositBelt which reflects the following task description in [323]: "The task of the deposit belt is to transport the work pieces unloaded by the second robot arm to the traveling crane. A photoelectric cell is installed at the end of the belt; it reports when a work piece reaches the end section of the belt. The control program then has to stop the belt. The belt can restart as soon as the traveling crane has picked up the work piece. ... photoelectric cells switch on when a plate intercepts the light ray. Just after the plate has completely passed through it, the light barrier switches off. At this precise moment, the plate is in the correct position to be picked up by the traveling crane."

Exercise 5.1.3. (⇝ CD) Refine the rule MoveToUnloadPos in rule ElevRotTable to a durative abstract action.

Exercise 5.1.4. (⇝ CD) The order of robot actions suggested in [323] implies the Last Piece Problem: if in a run a last piece is loaded to the press, the robot cannot unload it. Solve the problem.

Exercise 5.1.5. (⇝ CD) Formulate assumptions for the Production Cell ground model ASM and prove from them the following safety properties for appropriately initialized runs.

The feed belt never puts pieces on the table when the table is loaded or not stopped in its loading position. The robot never rotates over its bounds, never puts a piece into the press when the press is loaded and never drops pieces outside safe areas. The loaded first robot arm is never moved above the loaded table if the table is in unloading position; the loaded second robot arm is never moved over the deposit belt unless the deposit belt is loadable. The press never closes when a robot arm is positioned in it; it is never moved downwards/upwards from its bottom/top position.

Prove in the same way the following liveness property: every piece which enters the feed belt eventually is forged and leaves the deposit belt.

Exercise 5.1.6. (⇝ CD) Refine the Press ground model by detailing the control states *Open*/*ClosedForAction* and *MovingToAnyPos* in terms of the functions

$PressMot \in \{Idle, Up, Down\}$,
$ActionPos \in \{BottomPos, MiddlePos, TopPos\}$,
$PressLoaded$.

Exercise 5.1.7. Prove the refinements of the Belts, ERT and the Press in the Production Cell ASM to be correct.

Exercise 5.1.8. (⇝ CD) Define the refined notion of (*Un*)*LoadingDevice* and its being completed.

Exercise 5.1.9. (⇝ CD) Prove that the refined Production Cell ASM satisfies the assumptions which were used in Exercise 5.1.5 to prove the safety and liveness properties required by the task description.

5.2 Real-Time Controller (Railroad Crossing Case Study)

In this section we define *real-time controller ASMs* which mediate between continuous processes and discrete computations controlling them. Real-time controller ASMs reflect Parnas' four-variable model for process control systems. We illustrate the concept by a real-time controller ASM which we verify to correctly control the gate of the real-time Railroad Crossing case study proposed in [277]. For a non-standard approach to real-time ASMs based upon the use of infinitesimals see [379].

5.2.1 Real-Time Process Control Systems

A fundamental problem one has to solve for software controllers of real-time processes consists in mapping continuous objects to approximations in discrete states. This comprises a sampling and a synchronization problem. The *sampling problem* consists in finding appropriate methods to approximate the continuous flow of physical process data by a finite number of discrete samples, taken at real-time moments which form an increasing discrete sequence $(t_n)_{n \in \mathbb{N}}$. The *synchronization problem* consists in synchronizing the control program with input/output devices to guarantee for the software system – which needs stable states for reading its input, for internal controller steps, and for sending output – a correct reactive behavior with respect to the physical environment. In Parnas' *Four-Variable Model* [360] the requirements for the physical process with its MONitored and CONtrolled physical variables are mapped to requirements for the software program with its discrete software input and output via INput and OUTput devices (Fig. 5.7). The input device transforms continuous physical quantities, e.g. those measured by sensors, into discrete software input values, whereas the output device transforms discrete software output into values for actuators that manipulate the continuous physical process.

To guarantee stable states for reading input, internal controller steps, and sending output, many process control systems are based upon the *Environment-Controller Separation Principle* : at each moment, the software controller either takes input from the environment through its input device, without involving any internal computation step, or it makes an internal computation step (including the preparation of the next output to be sent to the output device), without involving any further input reading. Its appropriateness for the control of the physical system depends on the correctness of the sampling mechanism and on sufficient hardware speed to guarantee the timely execution of the internal controller steps with respect to the frequency of sampling. The correctness of synchronous (e.g. Esterel) programs for reactive systems largely depends on strict time requirements on input rate and system response time (read: on simultaneity and precedence conditions for

Fig. 5.7 Parnas' four-variable model

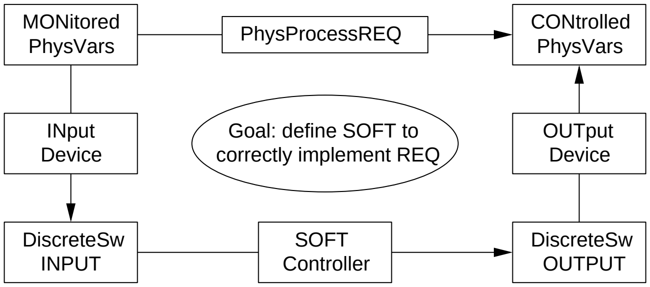
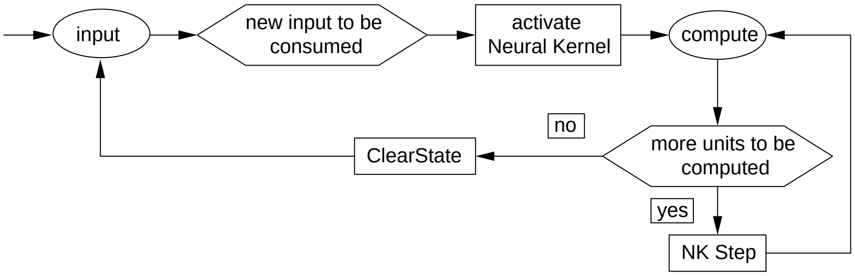


Fig. 5.8 Neural abstract machine model



the event ordering). The separation principle underlies the black-box view of neural nets modeled by the turbo ASM NeuralAM in Fig. 5.8 (taken from [142]). In its input phase the neural kernel is activated by the arrival of new input from the environment, to perform on that input an internal computation which ends with emitting an output to the environment and going to the input state. The internal computation consists of a finite sequence of atomic actions which are performed by the basic computing units (nodes of a directed data-flow graph); e.g. in the so-called forward propagation mode, the network input is transmitted by the input units to the internal units, which propagate their results through the graph until the output units are reached and the machine signals to the environment its readiness to take new input.

An analogous environment–controller separation is used to synchronize circuits by a clock which alternates between low and high pulse, like FlipFlop on p. 47. The circuit is viewed as a black box with input lines (representing the environment) and output lines. When the clock pulse is low, the box and the output are idle and the input may change; when the clock pulse is high, the box and the output may change and the input is idle.

The definition of ASM runs implicitly incorporates the environment–controller separation principle by splitting a step into internal and external updates. We now make this distinction explicit for ASMs which work in a real-time context. To ease the exposition we continue to stick to the idealization that actions are atomic,[8] meaning that the controller executes instantaneously and that the environment changes take place instantaneously.[9] To be sure that at each real-time moment there is a well-defined state, the runs are supposed to satisfy the environment–controller separation principle. This leads to the following definition (which we formulate for ASMs *tout court*, since M may be basic, turbo, sync or async).

Definition 5.2.1 (Real-time controller ASM). A real-time controller ASM is an ASM M whose runs – called real-time controller runs – satisfy the following properties:

- There is a discrete sequence $0 = t_0 < t_1 < \dots$ of real numbers, called the computationally significant real-time moments of M, with associated states $S(t_n)$ in which $\mathit{currtime} = t_n$ holds for the monitored real-valued system time *currtime*. Set $\sigma(t_n) = S(t_n) - \{\mathit{currtime}\}$.
- Every state $\sigma(t_{n+1})$ is obtained from the preceding computationally significant state $\sigma(t_n)$ either by an internal machine step or by an external environment step, i.e.:

  internal step: either by a move of M[10] to its next internal state, and between t_n and t_{n+1} no update occurred for any external location or any shared location which was not updated in the internal move of M,

  external step: or by environmental updates of external or shared locations, and between t_n and t_{n+1} no internal update of M occurred.

The sequences $(t_n)_{n \in \mathbb{N}}$ of real-time controller runs are usually restricted to discrete ones where each real number t has only finitely many preceding computationally significant run moments $t_n < t$. This excludes so-called Zeno-runs.

Definition 5.2.2 (Extension of real-time controller runs). A run $(t_n, S(t_n))_{n \in \mathbb{N}}$ of a real-time controller ASM M is canonically extended by states $S(t)$ for arbitrary real-time moments $t_n < t < t_{n+1}$ as follows:

- $\mathit{currtime} = t$ holds in $S(t)$,
- in the case of an internal step: $\sigma(t) = \sigma(t_{n+1})$, the state resulting from the M-step at t_n,

[8] In Sect. 6.4 it is shown how a natural refinement of the notion of an ASM run gently leads from atomic to durative actions.

[9] It is typical for the so-called synchronization hypothesis, which underlies synchronous languages, that events are assumed not to consume time so that time "passes" only between two events. See also the proposal in [14] to view real-time systems as resulting from discrete systems by the inclusion of clock variables.

[10] In the case of an async ASM M read: of some agent of M.

– in the case of an external step: $\sigma(t) = \sigma(t_n)$, the unchanged M-state (since between t_n and t_{n+1} no internal M-step takes place and the instantaneous change of external locations by definition becomes effective only at t_{n+1}).

5.2.2 Railroad Crossing Case Study

In this section we analyze the ground model ASM in [253] for the real-time controller of the Railroad Crossing problem [277] to illustrate how the ASM method allows one to couple high-level design with an on-the-fly verification of the design correctness. We concentrate here on analyzing and proving safety and liveness and formulate other provable system properties as exercises. To simplify the formulation we treat the machine as a sync ASM of a track and a gate controller, without restricting the generality of the solution.

Ground model construction. The system is required to operate a gate at a railroad crossing. A set of trains travel on multiple tracks in both directions. Each track has four sensors, detecting when a train enters or exits the crossing: L_1, L_2 for trains coming from the left, R_1, R_2 for trains coming from the right. Based on these sensor signals, a controller should signal the gate to open/close. It is part of the contract to verify for the system that when a train is in the crossing, the gate is closed (Safety) and that the gate is open when no train is in the crossing (Liveness).

We start with unfolding the assumptions for the motion of trains and for the time relation between trains and the gate. The motion of trains can be characterized for every track by a train motion sequence $t_0 < t_1 < \ldots$ of real numbers with the following three properties:

Initial State: at the initial moment t_0, the observed track segment $[L_1, R_1]$ is empty.

Train Pattern: If t_{3i+1} appears in the sequence, then t_{3i+2} and t_{3i+3} appear in the sequence,

– at t_{3i+1}, an incoming train is detected at either L_1 or R_1,

– at t_{3i+2} this train reaches the crossing,

– at t_{3i+3} this train is detected to have left the crossing at L_2 or R_2.

Completeness: The sequence $t_0 < t_1 < \ldots$ covers exactly the trains that appear on this track.

The train approaching time is determined by the minimum and maximum time $d_{min} \leq d_{max}$ for a train to reach the crossing after having been detected at L_1 or R_1, formally:

$$\forall i \text{ in a train motion sequence holds } t_{3i+2} - t_{3i+1} \in [d_{min}, d_{max}]$$

The gate closure and opening time d_{close} and d_{open} are characterized by the assumption that during no interval $[t, t + d_{close}]$ or $[t, t + d_{open}]$ is a signal to close or open in force without at some moment in this interval the gate being closed or opened.[11] For the gate versus the train time one obviously has to assume that the gate closes in time before the fastest train reaches the crossing: $d_{close} < d_{min}$.

In the signature we find, besides the monitored system time *currtime*, a monitored function *TrackStatus*: $TRACK \rightarrow \{empty, coming, inCrossing\}$ and a controlled function *Deadline*: $TRACK \rightarrow REAL \cup \{\infty\}$ to measure the allowable $WaitTime = d_{min} - d_{close}$ between the appearance of a train and the latest possible moment to start the gate closing (for the gate to be closed in time). A function $Dir \in \{open, close\}$ controlled by the track control signals when to open or close the gate. The actual gate status *opened* or *closed* is the value of the gate control state, which therefore is called *GateStatus*.

The sync RailCrossCtl ASM consists of the basic ASMs TrackCtl and GateCtl controlling the tracks and the gate in the presence of the environment which sets the monitored function *TrackStatus*. For each track the deadline is set upon arrival of a train, the signal to close is sent to the gate control upon deadline expiration, and the deadline is cleared when the track becomes empty. The signal to open is sent to the gate only when it is safe to do so. GateCtl is an instance of the FlipFlop on p. 47. We refer to its two control state transitions as *OpenGate* and *CloseGate*.

```
TrackCtl =
  forall $x \in TRACK$
    $SetDeadline(x)$
    $SignalClose(x)$
    $ClearDeadline(x)$
  $SignalOpen$
where
  $SetDeadline(x)$ = if $TrackStatus(x) = coming$ and
    $Deadline(x) = \infty$ then $Deadline(x) := currtime + WaitTime$
  $SignalClose(x)$ = if $currtime = Deadline(x)$ then $Dir := close$
  $ClearDeadline(x)$ = if $TrackStatus(x) = empty$ and
    $Deadline(x) < \infty$ then $Deadline(x) := \infty$
  $SignalOpen$ = if $Dir = close$ and $SafeToOpen$ then $Dir := open$
  $SafeToOpen = \forall x \in TRACK$
    $TrackStatus(x) = empty$ or $currtime + d_{open} < Deadline(x)$

GateCtl = Switch($(Dir = open, opened), (Dir = close, closed)$)[12]
```

[11] These stipulations foresee that the gate control may react to closure/opening commands from the controller with some (bounded) delay. Since we assume the controller to react immediately, i.e. at the instant when it is enabled, and since we want to consider the machine consisting of controller and gate control as a sync ASM, we assume that should a possible gate reaction delay be desired, then it is taken into account by the system clock. For an async ASM such an assumption is not necessary.

[12] Obviously one can sharpen the guards in GateCtl and *SignalClose* so that each step changes the state. See Exercise 5.2.1.

Runs of RailCrossCtl are defined as real-time controller ASM runs which satisfy the train motion condition via the following constraint on the moments of change of *TrackStatus*: among the computationally significant moments of the run, every track x has a subsequence $0 = t_0 < t_1 < \ldots$ of significant moments of x such that the value of $TrackStatus(x)$ is

- *empty* over every interval $[t_{3i}, t_{3i+1})$,
- *coming* over every interval $[t_{3i+1}, t_{3i+2})$, and $d_{min} \leq (t_{3i+2} - t_{3i+1}) \leq d_{max}$ holds,
- *inCrossing* over every interval $[t_{3i+2}, t_{3i+3})$,
- *empty* over $[t_k, \infty)$ if t_k is the final significant moment in the sequence, in which case k is a multiple of 3.

In those runs, the TrackCtl is assumed to react immediately, the reaction time of GateCtl is assumed as bounded in the sense of the above gate closure/opening time property.

Verifying safety and liveness. Supported by the above defined precise forms of intuitive railcrossing related concepts, the design of RailCrossCtl can be certified to be correct by proving that under the indicated assumptions the runs of the machine satisfy the required safety and liveness properties (Theorem 5.2.1). In a similar way one can also certify product-quality features, in this case that e.g. the closing of the gate and – under appropriate assumptions – its opening are never interrupted and that the liveness fails if d_{open} or D_{close} are replaced with smaller values. For proofs of this and of other run-time system properties see the exercises below and [253]. The proof for Theorem 5.2.1 below is intended to be read by a design or certification expert and therefore is formulated in intuitive terms made mathematically precise, following a successful and longstanding tradition of applied mathematics. One should be aware that a higher level of certification where all proof details are filled in and checked by machines comes at the price of a considerably higher cost, due to the fact that it increases tremendously the labor intensive and error prone formalization effort.[13]

Theorem 5.2.1. In every run of RailCrossCtl the following holds:

Safety: Whenever a train is in the crossing, the gate is closed. Formally: if $GateStatus(x) = inCrossing$ over a real-time interval $[t_{3i+2}, t_{3i+3})$ for some track x, then $GateStatus = closed$ over $[t_{3i+1} + d_{min}, t_{3i+3}] \supseteq [t_{3i+2}, t_{3i+3}]$.

Liveness: Whenever the crossing is empty (i.e. $TrackStatus(x) \neq inCrossing$ for every x) in an open real-time interval (α, β) with $\alpha + d_{open} < \beta - D_{close}$, the gate is open in the closed interval $[\alpha + d_{open}, \beta - D_{close}]$ with $D_{close} = d_{max} - WaitTime = d_{close} + (d_{max} - d_{min})$.

[13] In [35, 34] a timed logic is developed for the verification of a class of ASMs with explicit continuous time and is applied to a formal verification of RailCrossCtl.

Proof. **Safety**. If $\mathit{GateStatus}(x) = \mathit{inCrossing}$ over $[t_{3i+2}, t_{3i+3})$, the properties of the train motion sequence for x and the immediate reaction of TrackCtl imply that $\mathit{SetDeadline}(x)$ fires at t_{3i+1}, setting $\mathit{Deadline}(x)$ to $\alpha = t_{3i+1} + \mathit{WaitTime}$, so that $\mathit{TrackStatus}(x) \neq \mathit{empty}$ over $I = (\alpha, t_{3i+3})$. Hence over I, x makes *SafeToOpen* false and thereby disables *SignalOpen*. This implies that *OpenGate* is disabled over I since $\mathit{SignalClose}(x)$ fires at α so that, immediately after α, $\mathit{Dir} = \mathit{close}$ holds and remains unchanged over I. But $\mathit{Dir} = \mathit{close}$ immediately after α implies that $\mathit{GateStatus} = \mathit{closed}$ holds for some $\alpha < t < \alpha + d_{close} = t_{3i+1} + d_{min}$ and that it remains so as long as *SignalOpen* is disabled, i.e. over $I = (\alpha, t_{3i+3})$. But then it continues to hold over $[t_{3i+1} + d_{min}, t_{3i+3}]$ since only *OpenGate* – which is fireable only after *SignalOpen* – can change it to *opened*.

Liveness. The empty crossing premise for (α, β) implies by the Deadline Lemma 5.2.1 that for every track x the following inequalities hold:

$$\mathit{Deadline}(x) \geq \beta - D_{close} > \alpha + d_{open} \text{ over } (\alpha, \beta).$$

Therefore, *SafeToOpen* holds when $\mathit{currtime} = \alpha$ and thus $\mathit{Dir} = \mathit{open}$ holds immediately after α (possibly through firing of *SignalOpen*). *Dir* remains *open* over $(\alpha, \beta - D_{close})$, since over this interval $\mathit{Deadline}(x) \geq \beta - D_{close} > \mathit{currtime}$ holds for every x and thus disables $\mathit{SignalClose}(x)$. As a consequence also *CloseGate* is disabled over $(\alpha, \beta - D_{close})$.

$\mathit{Dir} = \mathit{open}$ over $(\alpha, \beta - D_{close})$ together with the assumption on the gate opening time d_{open} imply that $\mathit{GateStatus} = \mathit{opened}$ for some $\alpha < t < \alpha + d_{open}$; this remains so as long as *CloseGate* is disabled, namely over $(\alpha, \beta - D_{close})$. Therefore, $\mathit{GateStatus} = \mathit{opened}$ over $[\alpha + d_{open}, \beta - D_{close})]$, since only *CloseGate* – which is fireable only after some $\mathit{SignalClose}(x)$ has been executed – can change it to *closed*. □

Lemma 5.2.1 (Deadline Lemma). Let x be a track with significant moments $0 = t_0 < t_1 < \ldots$. Then the following holds:

1. $\mathit{Deadline}(x) = \infty$ over $(t_{3i}, t_{3i+1}]$ and $\mathit{Deadline}(x) = t_{3i+1} + \mathit{WaitTime}$ over $(t_{3i+1}, t_{3i+3}]$.
2. If $\mathit{TrackStatus}(x) \neq \mathit{inCrossing}$ over an interval (α, β), then over this interval $\mathit{Deadline}(x) \geq \beta - D_{close}$ with $D_{close} = d_{close} + (d_{max} - d_{min}) = d_{max} - \mathit{WaitTime}$.

Proof. Property 1 follows by induction on i. Property 2 is proved indirectly. Assume $\mathit{Deadline}(x) < \beta + \mathit{WaitTime} - d_{max}$ at some $t \in (\alpha, \beta)$. Property 1 implies that $\mathit{Deadline}(x) = t_{3i+1} + \mathit{WaitTime}$ holds at t for some $t \in (t_{3i+1}, t_{3i+3}]$. Then $t_{3i+1} < t < \beta \leq t_{3i+2}$ since, by hypothesis, (α, β) and the *inCrossing* interval $[t_{3i+2}, t_{3i+3})$ are disjoint. Therefore, $\beta - t_{3i+1} \leq t_{3i+2} - t_{3i+1} \leq d_{max}$, and thus $\beta - d_{max} \leq t_{3i+1}$. This implies $\beta - D_{close} = \beta - d_{max} + \mathit{WaitTime} \leq t_{3i+1} + \mathit{WaitTime} = \mathit{Deadline}(x)(\text{at } t) < \beta - D_{close}(\text{by assumption})$, which is a contradiction. □

5.2.3 Exercises

Exercise 5.2.1. Sharpen the guards in *SignalClose* and GATECTL to obtain a machine where each step changes the state.

Exercise 5.2.2. ($\leadsto$ CD) Internalize the timing conditions of train motion sequences to represent the environment by agents, one for each track, executing a TIMEDAUTOMATON (p. 288).

Exercise 5.2.3. ($\leadsto$ CD) Refine RAILCROSSCTL by introducing gate positions between $Closed = 0^o$ and $Opened = 90^o$. Clarify the real-world timing assumptions which allow that $Dir = close/open$ only when $GateStatus = opened/closed$.

Exercise 5.2.4 (Verifying RAILCROSSCTL run-time properties[253]). ($\leadsto$ CD)
Prove for significant moments $0 = t_0 < t_1 < \ldots$ of track x:

1. $SetDeadline(x)$ fires exactly at every t_{3i+1} (when $TrackStatus(x)$ has become $coming$).
2. $SignalClose(x)$ fires exactly at every $t_{3i+1} + WaitTime$.
3. $ClearDeadline(x)$ fires exactly at every t_{3i} for $i > 0$ (when $TrackStatus(x)$ has become empty).
4. Define $s(x)$ as the local $SafeToOpen(x)$ condition, namely by $TrackStatus(x) = empty$ **or** $currtime + d_{open} < Deadline(x)$. Show:
   - If $WaitTime > d_{open}$, then $s(x)$ holds over every interval $[t_{3i}, t_{3i+1} + WaitTime - d_{open})$ and fails over every interval $[t_{3i+1} + WaitTime - d_{open}, t_{3i+3})$.
   - If $WaitTime \leq d_{open}$, then $s(x)$ holds over every interval $[t_{3i}, t_{3i+1}]$ and fails over every interval (t_{3i+1}, t_{3i+3}).
   - $s(x)$ changes from false to true at every t_{3i} with $i > 0$ (when $TrackStatus(x)$ has become empty).
5. $SignalOpen$ fires exactly when $SafeToOpen$ becomes true. If $SafeToOpen$ becomes true at t, then some $TrackStatus(x)$ has become empty at t.

6 Asynchronous Multi-Agent ASMs

In this chapter[1] the single-agent (basic or turbo) ASMs of Chaps. 3 and 4 and the multi-agent synchronous ASMs of Chap. 5 are extended to asynchronous multi-agent ASMs and shown to be useful for the design and the analysis of distributed systems. In Sect. 6.1 we define *async ASMs* and illustrate them by characteristic distributed network algorithms (for consensus, master–slave agreement, leader election, phase synchronization, load balance, broadcast acknowledgment) and a position-based routing protocol for mobile ad hoc networks. In Sect. 6.2 we show async ASMs at work in a requirements capture case study for a small embedded system (Light Control). In Sect. 6.3 we use async ASMs to model and analyze two time-constrained algorithms which support fault tolerance for a distributed computing service, namely in Sect. 6.3.1 the modem and network communication protocol *Kermit* for correct file transfer, well-known from TCP/IP installations, and in Sect. 6.3.2 a Processor Group Membership protocol. In Sect. 6.4 we use the ASM refinement method to show – adopting Lamport's famous mutual exclusion algorithm *Bakery* as an example – how time-constrained algorithms with "atomic actions" can naturally be turned in a provably correct way to reflect also the "real-time duration" of actions. We show that it suffices to refine the global state view of atomic non-overlapping reads and writes in shared registers to a local state view of single agents whose overlapping reads and writes to the same location are governed by the constraints that async ASM runs impose on controlled, monitored and shared locations. Section 6.5 deals with event-driven ASMs. As a concrete illustration we model in Sect. 6.5.1 event-driven UML activity diagrams by async ASMs with turbo components and apply them for a compact one-page definition of an interpreter for Occam programs.

This chapter can be read independently of the preceding Chaps. 3–5 and most of Chap. 2; it suffices to know the definition of basic ASMs, of the ASM refinement concept and, for the examples in Sect. 6.5.1, of the notion of turbo ASMs.

[1] Lecture slides can be found in `AsyncASM` (⇝ CD), `LightControl` (⇝ CD), `LightControlRequirements` (⇝ CD), `Kermit` (⇝ CD), `GroupMemberProtocol` (⇝ CD), `Bakery` (⇝ CD).

6.1 Async ASMs: Definition and Network Examples

The monitored and shared locations and functions in basic ASMs abstract from detailed modeling of the actions of the environment, thus supporting for basic ASMs the characteristic splitting of the dynamics of a system into a machine computation part and a part which describes, in a possibly declarative manner, the assumed environment properties. For a computation step of a basic ASM to happen, *all* locations are supposed to have well-defined values. In fact Definition 2.4.22 incorporates the *environment–controller separation principle* explained in Sect. 5.2.1, in the sense that the value changes for monitored locations are assumed to take place in such a way that the new value is stable each time the machine is going to perform a step. The changes of monitored locations can be viewed as resulting from "monitored" moves of "unknown" environment agents, made independently of the machine moves, which are "controlled" by the executing agent, but synchronized with the machine moves as happening either simultaneously with them or "between successive" ones.

The definition of *async ASMs* generalizes this situation to an arbitrary finite number of independent agents, each executing a basic or structured ASM[2] in its own local state. A problem to solve for runs of such asynchronously cooperating agents originates in the possible incomparability of their moves which may come with different data, clocks, moments and duration of execution. This makes it difficult to uniquely define a global state where moves are executed to locate changes of monitored functions in an ordering of moves. The coherence condition in the definition of asynchronous multi-agent ASM runs below postulates well-definedness for a relevant portion of state in which an agent is supposed to perform a step, thus providing a notion of "local" stable view of "the" state in which an agent makes a move. The underlying synchronization scheme is described using partial orders for moves of different agents which reflect causal dependencies, determining which move depends upon (and thus must come "before") which other move. This synchronization scheme is as liberal as it can be, restricted only by the consistency condition for the updates which is logically indispensable, and thus can be instantiated by any consistent synchronization mechanism.

Definition 6.1.1 (Asynchronous multi-agent ASM). An *async ASM* is given by a family of pairs $(a, ASM(a))$ of pairwise different agents, elements of a possibly dynamic finite set *Agent*, each executing its basic or structured ASM $ASM(a)$. A run of an async ASM, also called a *partially ordered run*,[3]

[2] For the definition of async ASMs it is convenient to consider sync ASMs as given by a one-agent ASM where the synchronized subagents are viewed as a team. This allows one to let the behaviorally relevant causal dependencies and the different clocks of independent agents stand out distinctly, separate from the modularity relevant distinction of the substates of each (asynchronous or synchronous) agent.

is a partially ordered set $(M, <)$ of *moves* m (read: rule applications) of its agents satisfying the following conditions:

finite history: each move has only finitely many predecessors, i.e. for each $m \in M$ the set $\{m' | m' < m\}$ is finite,

sequentiality of agents: the set of moves $\{m | m \in M, a \text{ performs } m\}$ of every agent $a \in \mathit{Agent}$ is linearly ordered by $<$,

coherence: each finite initial segment (downward closed subset) X of $(M, <)$ has an associated state $\sigma(X)$ – think of it as the result of all moves in X with m executed before m' if $m < m'$ – which for every maximal element $m \in X$ is the result of applying move m in state $\sigma(X - \{m\})$.

The coherence condition immediately implies the following lemma.

Lemma 6.1.1 (Linearization of partially ordered runs). Let X be a finite initial segment of a run of an async ASM. All linearizations of X yield runs with the same final state.

This definition provides no clue to how to construct partially ordered runs for an async ASM, but it makes explicit the freedom that one has in implementing the described causal dependencies of certain local actions of otherwise independent agents. This means the freedom to model independent actions by synchronous parallelism or interleaving or explicit scheduling, etc. Notably the definition also imposes no fairness condition on runs ("a move which is enabled infinitely often is chosen infinitely often"), giving the designer the freedom to formulate and rely upon the fairness concept which is appropriate for the system under investigation. This freedom to instantiate partially ordered runs to particular classes of asynchronous runs, going together with the freedom of abstraction explained already for basic ASMs in Chap. 2, partly explains the naturality with which other models for distributed computation could be embedded into ASMs, like co-design FSMs, UNITY, Petri nets, Message Sequence Charts (with their typical partial event order), etc., but not the other way round; see the detailed investigation in Sect. 7.1.

Agents define the locus of computation; the use of "global states" associated with run segments of async ASMs is an idealization which does not imply any "global control". Each agent is dynamically equipped with its own program[4] operating on its own state, determining a partial view of the system state as illustrated in Fig. 6.1 from [292, Annex F1]. Using agents, one

[3] In the literature also the term *distributed run* is used in this sense. We will try to stick to the term partially ordered run to avoid the connotations of the widespread understanding of distribution as placement of agents on hardware. The understanding of moves as moves of agents can be made explicit by introducing a function which to each move associates the agent which makes the move.

[4] We allow the program of an agent to change dynamically.

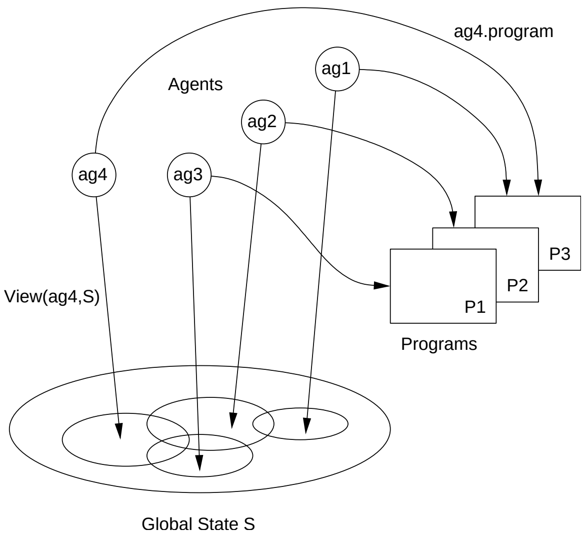
can easily provide the aspects of what in the literature are called *inter-object* descriptions, namely by multiple-agent ASMs, as opposed to *intra-object* descriptions, namely by single-agent ASMs. The relation between global and local states is supported by the use of the reserved name **self** in functions and rules to denote the agents which are executing the underlying "same" but differently instantiated basic, structured or sync ASM, similar to the use of *this* in object-oriented programming to denote the object for which the currently executed instance method has been invoked. For a function $f: X \times Y \to Z$, the expression $f(\mathbf{self})$ denotes the private version $\lambda y\ f(\mathbf{self}, y)$ – belonging to agent **self** – of a function from Y to Z. This self reference feature is rather useful for describing networks of agents which mostly run their own instances of one and the same program; e.g. one can associate with each agent its instance of a set of neighbor agents by parameterizing $neighb \subseteq Agent$ as $neighb(self)$. An often useful side effect is that *neighb* can also be viewed as an idealized global function $neighb: Agent \to PowerSet(Agent)$. Thus the use of **self** in async ASMs has the three typical object characteristics, namely to provide a unique identity and to encapsulate behavior and a persistent state, as illustrated for example by software architecture components. When it is clear from the context who is denoted by **self**, notationally **self** is omitted.

In the rest of this section we use async ASMs to define and analyze some small-size but characteristic distributed algorithms taken from [372], where they are treated in terms of Petri nets. In statements and proofs we often use without further statement the standard fairness condition that every enabled agent (read: agent with a true rule guard) will eventually make a move, meaning that there is no infinite order-respecting sequence of moves where an agent is enabled in each state without making a move. We conclude the section with an async ASM taken from [47] which models a routing layer protocol for mobile ad hoc networks. Larger examples appear in the following sections of this chapter.

6.1.1 Mutual Exclusion

The goal of mutual exclusion algorithms is to allow every interested process to eventually obtain for temporarily exclusive use some shared resource, thus preventing two users from using the shared resource simultaneously. So there is a set *Agent* of agents, each one with the right to access a *resource* from a set of *Resource*s, where different agents may have the same or overlapping *resource*s. A dynamic function $owner: Resource \to Agent$ records the current exclusive user of a given resource. A mutual exclusion algorithm has to manipulate updates of *owner* in such a way that every attempt by an agent to become an owner of his *resource* will eventually succeed. Ideally, every agent (denoted by **self**) would like to execute alternately the following two rules to get hold of and later to release the desired *resource* (where *none* stands for the default value *undef* of the set *Agent*):

Fig. 6.1 Global state and partial views in an async ASM

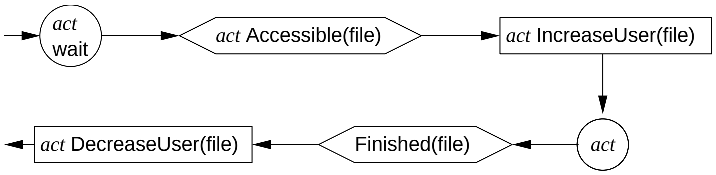


```
DiningPhilosopher =
  if $owner(resource) = none$ then $owner(resource) :=$ self
  if $owner(resource) =$ self then $owner(resource) := none$
```

Conflicts in using a *resource* which is shared among different agents are resolved by defining a partial order among possibly conflicting moves in an (appropriately initialized) run of an async ASM where each agent has the above two rules. Realizing such an order reflects appropriate scheduling and priority policies. It has become common to phrase this sort of problem as the problem of *dining philosophers* sitting around a table where the resource of every philosopher is a pair of a left fork and a right fork, both needed for eating but shared with the corresponding left or right neighbor philosopher, so that formally $resource(\mathbf{self}) = (leftFork(\mathbf{self}), rightFork(\mathbf{self}))$.

An example are Multiple-Read-One-Write algorithms allowing at each step one agent to start a read or a write operation in a given file, up to $maxRead > 0$ simultaneous reads, but only 1 write (not overlapping with any read). So let *Agent* be the set of agents which are allowed to access for read/write operations the files belonging to a set *File*, equipped with functions $user, maxRead, maxWrite\colon File \to \mathbb{N}$ indicating the number of agents which are currently reading or writing, respectively, and allowed to simultaneously read or write the given file, where $maxWrite = 1 \leq maxRead$. The basic ASM for file read/write access which is associated with each agent of the async MultipleReadOneWrite ASM is defined in Fig. 6.2. Assume that initially $user(file) = 0$. The function $finished\colon File \to Bool$ indicates whether

Fig. 6.2 Basic MultipleReadOneWrite ASM (act=Read,Write)



an agent has finished his current file operation. The macros are defined as follows:

$$
\begin{array}{l}
\textsc{MultipleReadOneWriteMacros} = \\
\quad \mathrm{act}\mathit{Accessible}(\mathit{file}) \equiv \mathit{user}(\mathit{file}) < \mathit{max}\mathrm{act}(\mathit{file}) \\
\quad\quad \textbf{where } \mathrm{act} \in \{\mathit{Read}, \mathit{Write}\} \\
\quad \mathit{Read}\mathrm{In}/\mathrm{De}\mathit{creaseUser}(\mathit{file}) \equiv \mathit{user}(\mathit{file}) := \mathit{user}(\mathit{file}) \pm 1 \\
\quad \mathit{Write}\mathrm{In}/\mathrm{De}\mathit{creaseUser}(\mathit{file}) \equiv \mathit{user}(\mathit{file}) := \mathit{user}(\mathit{file}) \pm \mathit{maxRead}(\mathit{file})
\end{array}
$$

If instead of 1 attempt one wants to allow multiple simultaneous file access attempts, *maxRead* becomes a cumulative counter to have the expected overall effect, i.e. $\mathit{ReadAccessible}(\mathit{file})$ becomes $\mathit{user}(\mathit{file}) + \mathit{newUsers} \leq \mathit{maxRead}(\mathit{file})$, where *newUsers* indicates the number of users attempting to access the file for reading.

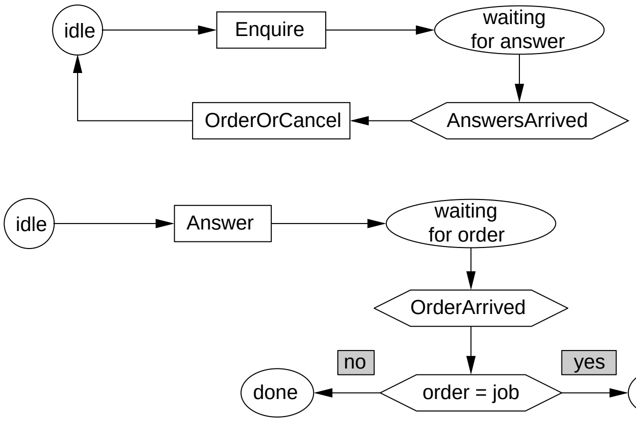
For the definition of the partial orders underlying the partially ordered runs of a concrete mutual exclusion algorithm see the investigation of the async Bakery ASM in Sect. 6.4.

6.1.2 Master–Slave Agreement

In this section we define an async ASM which can be proved to achieve the following goal: before a *master* process launches an *order* to *slave processes*, they are *asked* whether they are ready to *accept* or have to *refuse* the job. The master process *waits* for their confirmation and then definitely sends the *job* – for execution in case all slaves did accept, otherwise for cancellation.

The algorithmic idea consists in letting the initially *idle* master *Enquire* about a job to be launched, followed by *waitingForAnswer*s from the slaves and then – once the *AnswersArrived* – to *OrderOrCancel* and return *idle*. The slaves *Answer* the enquiry, followed by *waitingForOrder* until the *OrderArrived* in which case either all slaves become *busy* executing the ordered job or all slaves go into the control state *done*. Thus each agent of the async MasterSlaveAgreement ASM executes the corresponding basic ASM defined in Fig. 6.3 with the macros below, started with all agents *idle*, undefined *order, answer* and *asked* = *false*. We abstract from an explicit representation of the message passing by declaring the functions *asked, answer, order* to be shared among slave and master processes (see Exercise 6.1.1). For iterated use

Fig. 6.3 Basic ASM of MasterSlaveAgreement agents



the algorithm should be equipped with an additional *ClearOrder* command to reset *order* when the slaves return to *idle*.

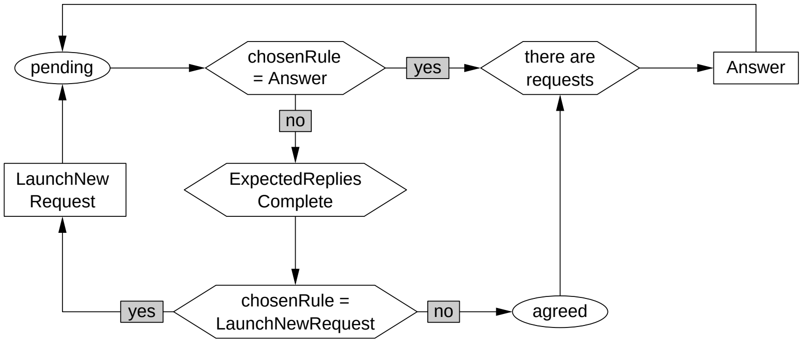
```
MasterMacros =
  $Enquire =$ forall $s \in Slave\ s.asked := true$
  $AnswersArrived =$ forall $s \in Slave\ s.answer \in \{accept, refuse\}$
  $OrderOrCancel =$
    if $\exists s \in Slave$ with $s.answer = refuse$ then $order := cancel$
      else $order := job$
    clear answer
  clear answer = forall $s \in Slave\ s.answer := undef$

SlaveMacros =
  $Answer =$ if $asked$ then choose $r \in \{accept, refuse\}$
    $answer := r$
    $asked := false$
  $OrderArrived = order \in \{job, cancel\}$
```

Proposition 6.1.1 (Correctness of MasterSlaveAgreement). In every properly initialized run of a master and its slaves, all equipped with the corresponding basic master/slave ASM, after the master has started an Enquiry, eventually the master becomes idle and either all slaves become done or all slaves become busy executing the job ordered by the master.

Proof. Follows by an induction on (initial segments of) runs. □

Fig. 6.4 Basic ASM of CONSENSUS agents



6.1.3 Network Consensus

In this section we define an async ASM which can be proved to achieve consensus among homogeneous agents, arranged as nodes of arbitrary finite connected networks, using only communication between neighbors, without broker or mediator. The algorithmic idea is that every agent may (a) launch a request to its neighbors and wait for the replies, (b) agree with the replies received from its neighbors, (c) reply to requests received from its neighbors – until all agents agree (maybe never). But in case they do agree, the consensus is not revoked.

As signature we have a finite connected set *Agent*, each equipped with an external neighborhood function $neighb \subseteq Agent$ and with mail boxes

$Request \subseteq \{RequestFrom(n) \mid n \in neighb\}$
$Reply \subseteq \{ReplyFrom(n) \mid n \in neighb\}$

and $ctl\_state \in \{pending, agreed\}$. For the sake of generality we work with abstract messages (requests and answers), simply *inserting* or *deleting* them into the corresponding mailbox. Initially all agents are *pending* with empty *Request* and full $Reply = \{ReplyFrom(n) \mid n \in neighb\}$. Since the above algorithmic specification leaves a non-deterministic choice whether to *Answer* a request, or to *LaunchNewRequest*, or to *Agree* with the replies received to a launched request, we reflect this choice by an auxiliary function *chosenRule*. Thus each agent of the async CONSENSUS ASM executes the properly initialized basic ASM defined in Fig. 6.4 with the macros below.

```
CONSENSUSMACROS =
  $ExpectedRepliesComplete = (Reply = full)$
  $LaunchNewRequest =$
    $BroadcastRequest$
    $Reply := empty$
  $BroadcastRequest =$ forall $n \in neighb$
```

```
    insert $RequestFrom(\textbf{self})$ into $Request(n)$
  $Answer = \textbf{forall } r \in Request$
    delete $r$ from $Request$
    send answer for $r$
  send answer for $RequestFrom(n) =$
    insert $ReplyFrom(\textbf{self})$ into $Reply(n)$
```

Proposition 6.1.2 (Correctness of CONSENSUS). In every properly initialized non-empty run of agents equipped with the basic consensus ASM, if the run terminates, then every agent is in $ctl\_state = agreed$ with full *Reply* and empty *Request* set.

Proof. Assuming that every enabled agent will eventually make a move, the claim follows from the definition of *LaunchNewRequest* and *Answer* by a run induction. Whenever $Reply = full$ for an agent a, then there is no $RequestFrom(a)$ in $Request(n)$ for any $n \in neighb(a)$. □

6.1.4 Load Balance

In this section we define an async ASM which can be proved to achieve a workload balance among the agents of a ring, using only communication between right/left neighbors. The algorithmic idea for determining the leader is that every agent (ring node) alternately sends a *workLoad* information message to his *rightNeighbor*, then a task transfer message to his *leftNeighbor*, possibly transferring a task to balance the workload with the left neighbor, and then updates his workload to balance it with his right neighbor. Eventually the difference between the workload of two nodes becomes at most 1. The ordering of these actions is important and reflected by the sequence of values of *ctl_state* in

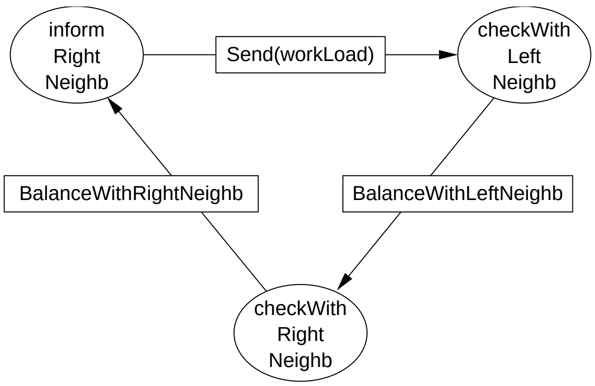
$$\{informRightNeighb, checkWithLeftNeighb, checkWithRightNeighb\}.$$

For the sake of generality we keep the message passing mechanism and the effective task transfer abstract; we thus update for each agent two mailboxes *neighbLoad*, *transferLoad*, which record the information received on the current *workLoad* of *leftNeighb* and on the workload transfered[5] by the *rightNeighb*. Initially, every agent is about to *informRightNeighb* with empty (i.e. undefined) *neighbLoad*, *transferLoad*. We assume here a fixed number of agents and constant total workload (see Exercise 6.1.4). Thus each agent of the async LOADBALANCE ASM executes the properly initialized basic ASM defined in Fig. 6.5 with the macros below.

```
LOADBALANCEMACROS =
  $Send(workLoad) = (rightNeighb.neighbLoad := workLoad)$
  $BalanceWithLeftNeighb = \textbf{if } arrived(neighbLoad) \textbf{ then}$
```

[5] In the definition below the workload transfer units are 0 or 1; see Exercise 6.1.5.

Fig. 6.5 Basic ASM of LoadBalance agents



```
  transfer task to $leftNeighb$
$arrived(l) = (l \neq undef)$
transfer task to $leftNeighb$ =
  $leftNeighb.transferLoad := transfer$
  $workLoad := workLoad - transfer$
  $neighbLoad := undef$
  **where** $transfer = \begin{cases} 1, & \text{if } workLoad > neighbLoad; \\ 0, & \text{else} \end{cases}$
$BalanceWithRightNeighb$ = **if** $arrived(transferLoad)$ **then**
  accept task from $rightNeighb$
accept task from $rightNeighb$ =
  $workLoad := workLoad + transferLoad$
  $transferLoad := undef$
```

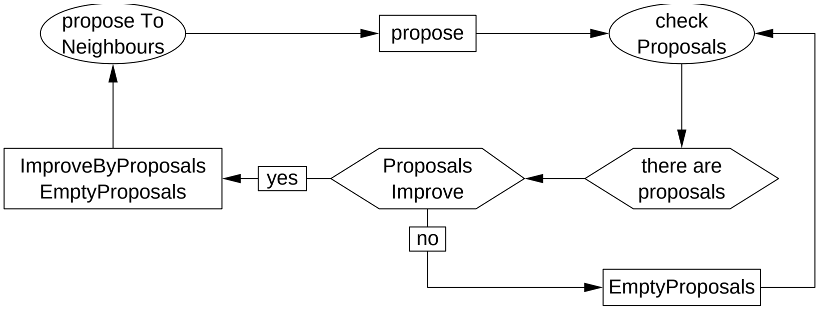
Proposition 6.1.3 (Correctness of LoadBalance). In every properly initialized run of agents equipped with the basic LoadBalance ASM, eventually the workload difference between two nodes becomes ≤ 1.

Proof. By induction on the weight of run workload differences. Let w be the sum of the *workLoad* of all nodes and let a be the number of agents. Case 1: w is divisible by a. Then eventually $workLoad(n) = w/a$ for every node n. Case 2: otherwise. Then eventually and permanently the workLoad of any pair of nodes will differ by at most 1. For proof details see [372, Chap. 82]. □

6.1.5 Leader Election and Shortest Path

In this section we define an async ASM which can be proved to achieve the election of a leader (and to be refined by the computation of a shortest path

Fig. 6.6 Basic ASM of LeaderElection agents



to the leader), established in terms of an order of homogeneous agents in finite connected networks, using only communication between neighbor nodes. Let $leader = max(Agent)$ with respect to a linear order $<$ of the set $Agent$. The algorithmic idea is that every agent proposes to his *neighb*ors his current leader *cand*idate, checks the leader *proposals* received from his *neighb*ors and upon detecting a proposal which improves his leader candidate, he improves his candidate for his next proposal. Initially every agent is without *proposals* from his neighbors and is supposed to *proposeToNeighbors* him*self* as *cand*idate . Eventually $cand = max(Agent)$ will hold for all agents. Thus each agent of the async LeaderElection ASM executes the properly initialized basic ASM defined in Fig. 6.6 with the macros below.

```
LeaderElectionMacros =
  $propose$ = forall $n \in neighb$ insert $cand$ to $proposals(n)$
  $proposals$ improve  = $max(proposals) > cand$
  improve by $proposals$ = $cand := max(proposals)$
  $EmptyProposals = (proposals := empty)$
  there are proposals  = $(proposals \neq empty)$
```

Proposition 6.1.4 (Correctness of LeaderElection). In every properly initialized run of agents equipped with the basic LeaderElection ASM, eventually every agent is in $ctl\_state = checkProposals$ with $cand = max(Agent)$ and empty *proposals*.

Proof. Assuming that every enabled agent will eventually make a move, the claim follows by an induction on runs and on $\sum\{leader - cand(n) \mid n \in Agent\}$ which measures the distances of candidates from the leader. □

We now refine the LeaderElection ASM by computing for each agent also a shortest path to the leader.[6] This is realized by providing for every agent, in addition to the leader candidate, also a neighbor (except for the leader) which is currently known to be closest to the leader, together with the minimal distance to the leader via that neighbor. The refinement is an example of a pure data refinement and consists in enriching *cand* and *proposals* by a neighbor with minimal distance to the leader. This is recorded in new dynamic functions *nearNeighb*: *Agent* and *distance*: *Distance* (e.g. $Distance = \mathbb{N} \cup \{\infty\}$), so that $proposals \subseteq Agent \times Agent \times Distance$ (triples of leader *cand*, *nearNeighbor* and *distance* to the leader). Initially $nearNeighbor =$ **self** and $distance = \infty$ except for the *leader* where $distance = 0$.

Thus each agent of the refined async MinPathToLeader ASM executes the properly initialized basic ASM defined in Fig. 6.6 with the refined macros below. Priority is given to determine the largest among the proposed neighbors (where *Max* over triples takes the *max* over the proposed neighbor agents), among the *proposalsFor* the current *cand* the one with *minimal distance* is chosen.

```
MinPathToLeaderMacros =
  $propose$ = forall $n \in neighb$
     insert $(cand, nearNeighb, distance)$ to $proposals(n)$
  $proposals$ improve = let $m = Max(proposals)$ in
    $m > cand$ or
      $(m = cand$ and $minDistance(proposalsFor\ m) + 1 < distance)$
  improve by $proposals$ =
    $cand := Max(proposals)$
    update PathInfo to $Max(proposals)$
  update PathInfo to $m$ = choose $(n, d)$ with
    $(m, n, d) \in proposals$ and $d = minDistance(proposalsFor\ m)$
      $nearNeighb := n$
      $distance := d + 1$
```

Proposition 6.1.5 (Correctness of MinPathToLeader). In every properly initialized run of agents equipped with the basic MinPathToLeader ASM, eventually every agent is in $ctl\_state = checkProposals$ with $cand = max(Agent)$, empty *proposals*, *distance*= the minimal distance of a path from agent to leader, and *nearNeighbor* a neighbor on a minimal path to the leader (except for leader where $nearNeighbor = leader$).

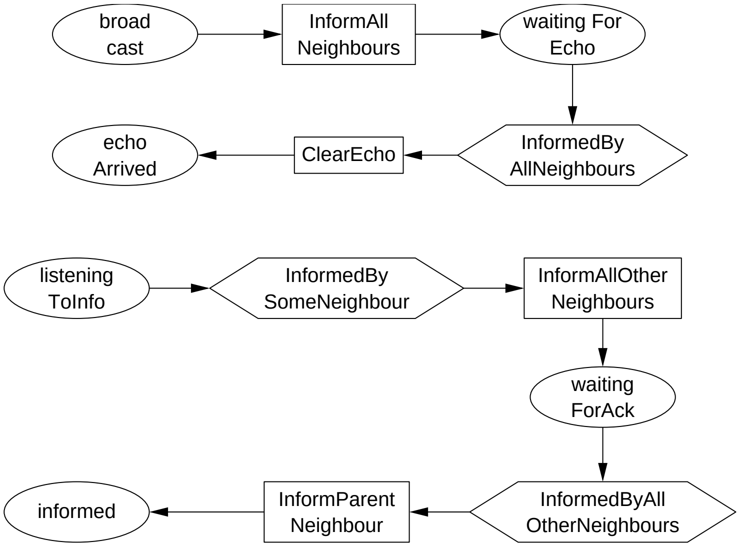
Proof. This follows with the same induction as for the LeaderElection with side induction on the minimal distances in *proposalsFor Max(proposals)*. □

6.1.6 Broadcast Acknowledgment (Echo)

In this section we define an async ASM which can be proved to guarantee an *initiator*'s message being broadcast and acknowledged ("echoed") through a finite connected network, using only communication between neighbors. The initiator (a distinguished node) broadcasts an info to all his neighbors,

[6] For a stepwise refined single-agent basic ShortestPath ASM see Sect. 3.2.2.

Fig. 6.7 Basic ASM of Echo agents (Initator/OtherAgent rules)



waits for their acknowledgments, and when these arrive terminates by clearing the echo, say for the next round. This triggers every node which has been *informedBySomeNeighbor* (its *parent* node) to *InformAllOtherNeighbors* and to wait in turn for their acknowledgments to come back, and to then forward his own acknowledgment to the parent node.

For the sake of generality we keep the message-passing mechanism abstract, providing for each agent a mailbox indicator *informedBy*: *Agent* → *Bool* recording whether a message (with information from the parent node or with an acknowledgment) from a neighbor agent has been sent (arrived); read $u.informedBy(v) = true$ as: u has a not-yet-read message from v. Initially the initiator is ready to *broadcast*, whereas all other agents are *listeningToInfo* with undefined *parent* and no message around (*informedBy* everywhere false). Thus the initiator and each other agent of the async Echo ASM executes the properly initialized basic ASM defined in Fig. 6.7 with the macros below.

```
EchoInitiatorMacros =
  $InformAllNeighbors = \textbf{forall}\ u \in neighb\ u.informedBy(self) := true$
  $InformedByAllNeighbors = \textbf{forall}\ u \in neighb\ informedBy(u) = true$
  $clearEcho = \textbf{forall}\ u \in neighb\ informedBy(u) := false$

EchoAgentMacros =
  $informedBySomeNeighbor = \exists u \in neighb\ informedBy(u) = true$
  $InformAllOtherNeighbors =$
    $\textbf{choose}\ u \in neighb\ \textbf{with}\ informedBy(u)$
```

```
        forall $v \in neighb - \{u\}$ $v.informedBy(self) := true$
        $informedBy(u) := false$
        $parent := u$
$informedByAllOtherNeighbors =$ forall $v \in neighb - \{parent\}$
    $informedBy(v) = true$
$InformParentNeighbor =$
    $parent.informedBy(self) := true$
    $clearAcknowledgment$
$clearAcknowledgment =$
    forall $u \in neighb - \{parent\}$ $informedBy(u) := false$
    $parent := undef$
```

Proposition 6.1.6 (Correctness of Echo). In every properly initialized run of agents equipped with the basic Echo ASMs for the initiator and the other agents, the initiator terminates (termination). He terminates only when all other agents have been informed about his originally sent message (correctness).

Proof. Follows from the following two lemmas. □

Lemma 6.1.2 (Downward Lemma). In every run of the async Echo ASM, each time an agent executes *InformAllOtherNeighbors*, in the spanning tree of agents *waitingForAck* the distance to the initiator grows until leafs are reached.

Proof. By downward induction on echo ASM runs. □

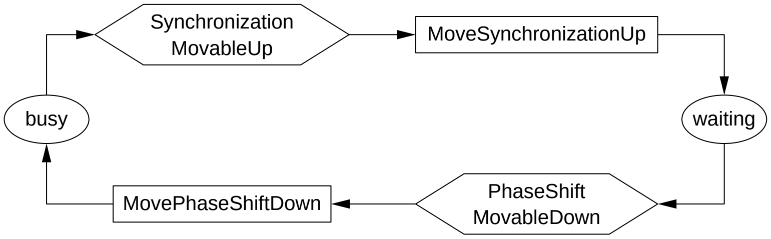
Lemma 6.1.3 (Upward Lemma). In every run of the async Echo ASM, each time an agent executes *InformParentNeighbor*, in the spanning tree the distance to the initiator of nodes with a subtree of informed agents shrinks, until the initiator is reached.

Proof. By upward induction on runs of the async echo ASM. □

6.1.7 Phase Synchronization

In this section we define an async ASM which can be proved to guarantee the synchronized execution of computations by phases, occurring at nodes of an undirected tree (connected acyclic network), using only communication between tree neighbors. The algorithmic idea is that every agent (tree node) before becoming *busy* in a new *phase* has to synchronize with its *neighb*or nodes (to make sure the busy ones are working in the same new phase) by (a) moving the synchronization up along the tree structure, (b) *waiting* until all agents are waiting for the phase increase to start, (c) increasing its phase, when the phase shift becomes movable down, and moving the phase shift

Fig. 6.8 Basic ASM of PhaseSync agents



further down along the tree structure – until all nodes have become busy in the new phase.

We keep the message-passing mechanism abstract, associating with every agent a *synchPartner* and a *waitPartner* which indicate the neighbor node an agent is synchronized with (in moving the synchronization through the tree) resp. waiting for (to reverse the previous upward synchronization downwards). Initially every agent is *busy* in *phase* = 0 with no *synchPartner* and no *waitPartner*. Each *MoveSynchronizationUp* of an agent **self** shifts the *synchPartner* up (read: by sending a message to him from **self**) and records it as *waitPartner* (e.g. to expect a message from him to **self**), to be used for the subsequent *MovePhaseShiftDown* which will be done in the reversed order. Therefore by the connectedness of the graph, for each phase p eventually pairs of *matching wait and synchronization partners* will be met making *PhaseShiftMovableDown*, determined by a node **self** in phase p which for this phase is the *synchPartner* of its *waitPartner* (read: has received a synchronization message from the expected node). More precisely a pair of nodes u and its wait partner $v = u.waitPartner(p)$, where u is the synchronization partner of v, i.e. $v.synchPartner(p) = u$ (read: u is waiting for a message from v and v has sent the expected message to u).

Each agent of the async PhaseSync ASM executes the properly initialized basic ASM defined in Fig. 6.8 with the macros below.

```
PhaseSyncMacros =
  $SynchronizationMovableUp = \exists y \in neighb$
    $SynchronizationMovableUpTo(y)$
  $SynchronizationMovableUpTo(y) = \textbf{forall}\ z \in neighb - \{y\}$
    $z.synchPartner(phase(\textbf{self})) = \textbf{self}$
  $MoveSynchronizationUp =$
    $\textbf{choose}\ y \in neighb\ \textbf{with}\ SynchronizationMovableUpTo(y)$
      $MoveSynchronizationUpTo(y)$
  $MoveSynchronizationUpTo(y) = \textbf{forall}\ z \in neighb - \{y\}$
    $z.synchPartner(phase(\textbf{self})) := undef$
    $\textbf{self}.synchPartner(phase(\textbf{self})) := y$
    $\mathbf{self}.waitPartner(phase(\mathbf{self})) := y$
  $PhaseShiftMovableDown =$
    $\mathbf{self}$ is $synchPartner$ of its $waitPartner$
  $\mathbf{self}$ is $synchPartner$ of its $waitPartner$ $=$
    $waitPartner(phase(\mathbf{self})).synchPartner(phase(\mathbf{self})) = \mathbf{self}$
  $MovePhaseShiftDown =$
    $phase := phase + 1$
    forall $z \in neighb - \{waitPartner\}$
      $\mathbf{self}.synchPartner(phase(\mathbf{self})) := z$
    $waitPartner(phase(\mathbf{self})).synchPartner(phase(\mathbf{self})) := undef$
```

Proposition 6.1.7 (Correctness of PhaseSync). In every properly initialized run of agents equipped with the basic PhaseSync ASM, in any state any two busy agents are in the same phase (correctness property). If every enabled agent will eventually make a move, then each agent will eventually reach each phase (liveness property).

Proof. Use the following lemma. □

Lemma 6.1.4. For every phase p, whenever in a run a state is reached where for the first time an agent $u = \mathbf{self}$ becomes *synchPartner* of its *waitPartner* say v (for $phase(\mathbf{self}) = p$), every element of $subtree(u, v) \cup subtree(v, u) \cup \{u, v\}$ is waiting in phase p, where $subtree(x, y) = \{n \mid n \text{ reachable by a path from } x \text{ without touching } y\}$.

The proof of the lemma follows from the following two claims.

Claim 1. When the synchronization is moved up in $phase(u) = p$ from *busy* u to $v = u.synchPartner(p)$, the elements of $subtree(u, v)$ are waiting, u becomes waiting, and they all remain so until the next application of a Shift rule *MovePhaseShiftDown*.

Proof. This follows by induction on the applications of the Synchronization rule. For $n = 1$ the claim is true since it holds at the leaves. For $n + 1$ the claim follows by the induction hypothesis and the Synchronization rule from

$$subtree(u, v) = \bigcup_{i<n} subtree(u_i, u) \cup \{u_0, \ldots, u_{n-1}\}$$

$neighb(u) = \{u_0, \ldots, u_n\}$ with $v = u_n$ (by connectedness). □

Claim 2. For every infinite run and every phase p, a state is reached in which all agents are waiting in phase p and some agent in phase p becomes *synchPartner* of its *waitPartner*.

Proof. By induction on p. □

6.1.8 Routing Layer Protocol for Mobile Ad Hoc Networks

Following [47] we illustrate async ASMs to model a distributed location service for mobile ad hoc networks. The service touches three layers of the communication architecture, namely the lower so-called Media Access layer (MAC), the higher Transport layer and the intermediate network layer. The network layer, constituted by mobile hosts which act as both communication endpoints and routers, is split into two sublayers: at the higher layer a Distributed Location Service (DLS) mediates between the transport layer and a lower layer Position-Based Routing (PBR) protocol, which in turn interacts with the MAC layer. The communication mechanism itself, typically using a wireless communication network, is kept abstract in terms of two types of appropriately parameterized primitives, *PacketToLayer* and *PacketFromLayer* for sending or receiving data packets to and from the *Layer*s. It is assumed that each $\mathit{PacketToLayer}(p)$ operation triggers a corresponding event $\mathit{PacketFromLayer}(p)$ through which the target *Layer* receives the sent packet p.

Sending and receiving are operated asynchronously at mobile hosts which are formalized by a finite set of *Node*s n with fixed (static) $\mathit{address}(n)$ and a dynamic (monitored) position $\mathit{pos}(n)$, belonging to an abstract set of geographical *Position*s. To each node n two independent agents *pbr* and *dls* are attached (formally via $\mathit{node}(\mathit{pbr}) = \mathit{node}(\mathit{dls}) = n$) to perform the position-based routing – executing a basic ASM PosBasedRouting – and the distributed location service – executing a basic ASM DistrLocationServ.

When the associated mobile host $\mathit{node}(\mathbf{self})$ is *switchedOn*, position-based routing agents run the cyclic detection of the nearest reachable neighbors, elaborate packets coming from MAC and forward to MAC the packets which arrive from DLS. When a *PacketFromDLS* arrives, the *position* of its *receiver* is retrieved where it is forwarded as *PacketToMAC*. The elaboration of packets from MAC comprises (a) the handling of neighbor detection packets, i.e. with $\mathit{type}(p) \in \mathit{Detection} = \{\mathit{NeighborRequest}, \mathit{NeighborReply}\}$, and (b) the delivery of data packets (with $\mathit{type}(p) \in \mathit{DataPacket}$) or of location discovery packets (of $\mathit{type}(p) \in \mathit{Discovery} = \{\mathit{locationRequest}, \mathit{locationReply}\}$ which are either to be delivered locally – if *address(receiver(p))* matches the address of the node of the routing agent – or to be forwarded to the position of the appropriate neighbor (if it exists) on the way to their remote destination. This neighbor is computed by a function *computeNeighbor* for the node of the routing agent relative to the packet receiver's *position*. Informative definitions for RunNearestNeighborDetectionCycle, $\mathit{Handle}(\mathit{NeighborRequest}, p)$ and $\mathit{Handle}(\mathit{NeighborReply}, p)$ involve timing issues which are given below.

```
PosBasedRouting = if $\mathit{switchedOn}(\mathit{node}(\mathbf{self}))$ then
    RunNearestNeighborDetectionCycle
    ElaborateMacPackets
    ForwardFromDlsToMac
  where
    ForwardFromDlsToMac = if $PacketFromDLS(p)$ then
      $PacketToMAC(p, position(receiver(p)))$
    ElaborateMacPackets = if $PacketFromMAC(p)$ then
      if $type(p) \in Detection$ then $Handle(type(p), p)$
      if $type(p) \in Discovery \cup DataPacket$ then
        if $localDelivery(p)$ then $PacketToDLS(p)$
          else $ForwardToNeighbor(p)$
    $localDelivery(p) = (address(receiver(p)) = address(node(\mathbf{self}))$
    $ForwardToNeighbor(p) =$
      let $neighb = computeNeighbor(node(\mathbf{self}), position(receiver(p)))$
        if $neighb \neq undef$ then $PacketToMAC(p, position(neighb))$
```

The task of distributed location service agents is to elaborate packets coming from the Transport or the PBR layer. When the Transport layer consecutively sends packets belonging to a sequence, the *dls* agent first has to determine the current *position* of the destination node where to send the packets in the sequence. A monitored predicate *FirstPacket* is used to distinguish when the subcomputation DiscoverDestinationPosition has to be started from the *address* of the *receiver* and then to send there this first packet, recorded at *firstpacket*(**self**). Subsequent packets are sent to PBR with the *position* of their *receiver* updated by the destination position discovered for the first packet.

```
  DistrLocationServ = if $switchedOn(node(\mathbf{self}))$ then
    ElaborateTransportPackets
    ElaboratePbrPackets
  where
  ElaborateTransportPackets = if $PacketFromTransport(p)$ then
    if $FirstPacket(p)$ then
      DiscoverDestinationPosition($address(receiver(p))$)
      $firstpacket(\mathbf{self}) := p$
    else
      $PacketToPBR(p)$
      $position(receiver(p)) := position(receiver(firstpacket(\mathbf{self})))$
  ElaboratePbrPackets = if $PacketFromPBR(p)$ then
    if $type(p) \in Discovery$ then HandleDiscoveryPacket($p$)
    else $PacketToTransport(p)$
```

Discovery packets arriving from PBR are handled as follows. In case of a *locationReply* the discovered destination position is recorded where to send the *firstpacket* and the remaining packets of the sequence. In case the location for a *locationRequest* has been found, one has to HandleLocationReply, i.e. to send back to the sender at PBR a *locationReply* packet containing the discovered destination position. Otherwise the *locationRequest* packet has to be forwarded to the appropriate neighbor, which is computed and assigned to the *receiver* of the packet using a function *nextHypercubicNeighbor* for the node of the *dls* agent relative to the packet's final destination *address*. This function reflects the hypercube underlying the network; see [47] for more details. We define DiscoverDestinationPosition, HandleLocationReply below, where further details about the representation of nodes in packets are provided.

```
HandleDiscoveryPacket($p$) =
  if $type(p) = locationReply$ then
    $position(receiver(firstpacket(\mathbf{self}))) := position(sender(p))$
    $PacketToPBR(firstpacket(\mathbf{self}))$
  if $type(p) = locationRequest$ then
    if $RequestedLocationFound$ then HandleLocationReply($p$)
    else $ForwardRequestToNeighbor$
  $RequestedLocationFound = (address(p) = address(node(\mathbf{self})))$
  $ForwardRequestToNeighbor$ =
    let $neighb = nextHypercubicNeighbor(node(\mathbf{self}), address(p))$
      $\{receiver(p) := neigh, PacketToPBR(p)\}$
```

To specify DiscoverDestinationPosition and HandleLocationReply implies detailing some attributes of packets, in particular how the relevant information about the address and position of the sender and receiver nodes of packets is represented. Including sender and receiver nodes directly ("unencoded") into packets would imply the unrealistic consequence that dynamic displacements of sender or receiver nodes, by the very magic of that idealization, become automatically known also to every sent packet. Therefore a substitute of nodes has to be introduced to play the role of sender and receiver with their static address and their position at the moment of sending. Let *NodeRef* be a set of node references with functions *address* and geographic *position* to represent *sender* and *receiver* information in packets, values of *nextHypercubicNeighbor*, etc. We use the following macros *decorate* and *pack* when new node references are created for new packets to be sent, as needed to discover a destination position or to handle a location reply:

$$decorate(nref, a, p) = \{address(nref) := a, position(nref) := p\}$$
$$pack(p, r, s, t) = \{receiver(p) := r, sender(p) := s, type(p) := t\}$$

```
DiscoverDestinationPosition($addr$) =
  let $neighb = nextHypercubicNeighbor(node(\mathbf{self}), addr)$
  let $r, s = new(NodeRef)$  let $q = new(Packet)$
    $decorate(r, address(neighb), position(neighb))$
    $decorate(s, address(node(\mathbf{self})), pos(node(\mathbf{self})))$
    $pack(q, r, s, locationRequest)$
    $address(q) := addr$
    $PacketToPBR(q)$[7]
```

```
HandleLocationReply$(p) =$
  let $r, s = \mathit{new}(\mathit{NodeRef})$ let $q = \mathit{new}(\mathit{Packet})$
    $\mathit{decorate}(r, \mathit{address}(\mathit{sender}(p)), \mathit{position}(\mathit{sender}(p)))$
    $\mathit{decorate}(s, \mathit{address}(\mathit{node}(\mathbf{self})), \mathit{pos}(\mathit{node}(\mathbf{self})))$
    $\mathit{pack}(q, r, s, \mathit{locationReply})$
    $\mathit{PacketToPBR}(q)$
```

Detection of nearest neighbors. Neighbors of nodes n are recorded by references $\mathit{neighbor}(n, s)$ to nodes whose geographic position is within a given sector s, which is an element of a static set *Sector* which is associated with each node and whose size depends on the network. The requirement that the computations of new nearest neighbors within the sectors do not interfere with each other is modeled below by using the **forall** construct together with a monitored predicate *UpdateNeighborEvent(s)* which indicates that a new neighbor detection cycle has to be started for the indicated sector (we suppress the parameter for the executing *pbr* agent). Each sector also has its own detection cycle timeout, when *neighbor* is updated by the found *NewNeighbor*. The timeout is represented by $\mathit{Timer}(s)$ which upon starting a neighbor detection cycle is set to the local current time $\mathit{now}(\mathit{node})$ at the mobile host increased by a network-wide cycle *duration*. *NeighborRequest* and *NeighborReply* packets p keep track of their sector and the timeout by appropriate dynamic functions $\mathit{sector}(p)$ and $\mathit{deadline}(p)$. This explains the machine RunNearestNeighborDetectionCycle to *IssueNeighborRequest* packets (addressed to any node with current $\mathit{pos}(\mathit{node}) \in s$) upon *UpdateNeighborEvent*, and to update upon detection cycle timeout *neighbor(n,s)* by the nearest $\mathit{NewNeighbor}(n, s)$ received in any *NeighborReply* within its deadline.

```
RunNearestNeighborDetectionCycle $=$
  let $\mathit{node} = \mathit{node}(\mathbf{self}), \mathit{pos} = \mathit{pos}(\mathit{node}), \mathit{now} = \mathit{now}(\mathit{node})$
  forall $s \in \mathit{Sector}$
    if $\mathit{UpdateNeighborEvent}(s)$ then StartNewDetectionCycle$(s)$
    if $\mathit{DetectionCycleTimeout}(s)$ then EndDetectionCycle$(s)$
  where
  StartNewDetectionCycle$(s) =$
    IssueNeighborRequest$(s, \mathit{pos}, \mathit{now})$
    $\{\mathit{ResetNewNeighbor}(s), \mathit{SetTimer}(s)\}$
  $\mathit{ResetNewNeighbor}(s) = (\mathit{NewNeighbor}(\mathit{node}, s) := \mathit{undef})$
  $\mathit{SetTimer}(s) = (\mathit{Timer}(\mathbf{self}, s) := \mathit{now} + \mathit{duration})$
  IssueNeighborRequest$(s, \mathit{pos}, \mathit{now}) =$
    let $r, t = \mathit{new}(\mathit{NodeRef})$ let $q = \mathit{new}(\mathit{Packet})$
      $\mathit{decorate}(r, \mathit{anyAddress}, \mathit{anyPosition} \in s)$[8]
```

[7] The idealized view of a global async ASM state allows us to abstract from the sequentialization an implementation needs for the decoration of node references, the packing and the sending of the packet.

```
        $decorate(t, address(node), pos)$
        $sector(q) := s$
        $deadline(q) := now + duration$
        $pack(q, r, t, NeighborRequest)$
        $PacketToMAC(q, s)$
$DetectionCycleTimeout(s) = (now > Timer(\mathbf{self}, s))$
$\textsc{EndDetectionCycle}(s) = \{UpdateNeighbor(s), ResetTimer(s)\}$
$UpdateNeighbor(s) =$ if $NewNeighbor(node, s) \in NodeRef$ then
  $neighbor(node, s) := NewNeighbor(node, s)$
$ResetTimer(s) = (Timer(\mathbf{self}, s) := \infty)$
```

To *Handle(NeighborRequest,p)* means to send a *NeighborReply* for the corresponding sector back to the sender's position, copying also the packet deadline which is checked against the local time of the sender when it comes to *Handle(NeighborReply,p)*. Upon receipt of a *NeighborReply* packet, *NewNeighbor* may be updated, namely when it is not nearer to the receiver – i.e. the sender of the corresponding original *NeighborRequest* packet – than the sender of the *NeighborReply* packet (assuming for the initialization $position(undef, s) = \infty$).

```
$Handle(NeighborRequest, p) =$
  let $r, s = new(NodeRef)$ let $q = new(Packet)$
    $decorate(r, address(sender(p)), position(sender(p)))$
    $decorate(s, address(node(\mathbf{self})), pos(node(\mathbf{self}))$
    $pack(q, r, s, NeighborReply)$
    $deadline(q) := deadline(p)$
    $sector(q) := sector(p)$
    $PacketToMAC(q, position(sender(p)))$
$Handle(NeighborReply, p) =$ if $ReplyComesWithinDeadline(p)$ then
  if $NewNeighbor(node(\mathbf{self}), sector(p)) = undef$ or
    $SenderNearerThanNewNeighbor(p)$ then
      $NewNeighbor(node(\mathbf{self}), sector(p)) := sender(p)$
$ReplyComesWithinDeadline(p) = (now(node(\mathbf{self})) \leq deadline(p))$
$SenderNearerThanNewNeighbor(p) =$
  $distance(pos(node(\mathbf{self})), position(sender(p))$
    $< distance(pos(node(\mathbf{self})), position(NewNeighbor(node(\mathbf{self}), s)))$
```

Problem 20 (Patterns of component hierarchies). Develop ASM models for run-time structures which provide useful patterns for hierarchical behavioral system descriptions in terms of interacting abstract components.

[8] Upon packet arrival the variable *anyAddress* is to be instantiated by (matched against) an address, and similarly for $anyPosition \in s$.

6.1.9 Exercises

Exercise 6.1.1. Data-refine the ASM MasterSlaveAgreement by replacing the shared functions *asked, answer, order* by messages.

Exercise 6.1.2. Show that in *Answer* of the ConsensusMacros, **forall** $r \in Request$ can be equivalently replaced by **choose** $r \in Request$.

Exercise 6.1.3. Refine the async LoadBalance ASM to an async ASM whose runs terminate when the *workLoad* difference of all node pairs is ≤ 1.

Exercise 6.1.4 (Dynamic workload). Refine the async LoadBalance ASM to an async ASM where new nodes with new workload can be introduced and where the total workload can also decrease.

Exercise 6.1.5 (Optimized workload transfer). Can one refine the async LoadBalance ASM to an async ASM where the load transfer among neighbors is not one-by-one but tries to locally balance the workload in one blow?

Exercise 6.1.6. ($\leadsto$ CD) Refine in the async LeaderElection ASM the CHECK submachine by a machine which checks proposals elementwise. Prove that the refinement is correct.

Exercise 6.1.7. Adapt the MinPathToLeader ASM with respect to a partial order instead of a total order.

Exercise 6.1.8. ($\leadsto$ CD) Reuse the LeaderElection ASM to define an algorithm which, *given the leader*, computes for each agent the distance (length of a shortest path) to the leader and a neighbor where to start a shortest path to the leader.

Exercise 6.1.9 (Echo to multiple initiators). ($\leadsto$ CD) Refine the async Echo ASM to the case of a network with more than one initiator.

Exercise 6.1.10 (Ring buffers [254]). ($\leadsto$ CD) Prove the async ASm RingBuffer defined below, consisting of two agents, one executing rule RingBufferInput and one RingBufferOutput, to correctly define a first-in, first-out ring buffer of fixed size N. The input at buffer position i cannot be taken until input has been taken at all previous buffer positions and either $i < N$ or the value at buffer position $i - N$ has been outputted; the value at buffer position i cannot be outputted until input at this buffer position has been taken and the values of all previous buffer positions have been outputted. *Buffer* is a function defined on $\{0, \dots, N-1\}$ (with a data range without need of further specification, except for the shared function *In* taking the input there). *First* and *Last* are counters indicating the next buffer slot to be used for input or output. (Turning *In* into a monitored function with appropriate assumptions on *In* would allow one to skip the guard and the update for *In*.)

```
RingBufferInput = if $First - Last \neq N$ and $In \neq undef$ then
    $Buffer(First) := In$
    $First := First + 1 \bmod N$
    $In := undef$
RingBufferOutput = if $First \neq Last$ then
    $Out := Buffer(Last)$
    $Last := Last + 1 \bmod N$
```

Define another async ASM where instead of one machine for inputting and one for outputting there are N machines, one for each buffer position, each of which alternates putting values to and getting them from its buffer position. Define a reasonable notion of equivalence and show that with respect to this notion the two async ASMs are equivalent.

Exercise 6.1.11. Describe the pattern of moves of PosBasedRouting involving the first and subsequent elements of a sequence of data packets newly arriving from the Transport layer, including the moves concerning the corresponding newly created detection and discovery packets. Formulate conditions from which it can be proved that packets of such a sequence are received in the right order.

6.2 Embedded System Case Study

In this section we construct an async ground model ASM coming with an executable refinement for an embedded control system *Light Control*, a reference case study in the literature for requirements capture methods [115, 113].[9] This section can be read independently from the other chapters; only the definition of async ASMs is needed. For this reason we briefly rephrase the *ground model problem*, which motivates this section, and refer the reader for a more detailed explanation of this concept to Sects. 2.1.1, 3.1.

6.2.1 Light Control Ground Model

In defining the async LightControl ASM we illustrate how to gently transform informal requirements into a succinct operational model of the to-be-implemented piece of "real world", transparent for both the customer and the software designer so that it can serve as a basis for the software contract. This implies removing from the given requirements their inconsistencies, ambiguities, incomplete or unnecessarily detailed parts, without adding details which belong to the subsequent software design, thus assigning to the requirements a sufficiently precise yet abstract, unambiguous meaning as a basis for their implementation-independent, application-oriented analysis, prior to coding.

[9] A stepwise refinement of an async ground model ASM to C++ code for the popular *Steam Boiler Control* case study [7, 8, 9] appeared in [43].

In fact this analysis must be two-fold to support a professional formulation of an unambiguous contract between the application-domain expert (standing for the customer) and the system designer. In the formulation of this contract the ground model represents for the customer the binding development goal and for the system designer a reliable (i.e. clear, stable and complete) starting point for the implementation. This means that one has to be able

– to analytically and experimentally check the correctness and completeness of the accurate specification with respect to the informal requirements (*faithfulness* and *adequacy*), which is the reason for which the model is (a) formulated in application-domain-oriented terms to make it inspectable for the domain expert, (b) refined to an executable version (in this case in AsmGofer [390]) serving for high-level simulation, test and debugging right at the beginning of the software project,
– to check by analytical means the *internal consistency* and the *intrinsic completeness* of the requirements, leaving as much space as possible for adapting the model to requirements changes which typically occur during the design.

This book is not the right place to illustrate the *process of ground model construction*, which is by no means linear, but iterative and typically comes with extensive simulation, high-level proving activities, layout of test plans, etc., as illustrated in Fig. 2.2. Instead we lead the reader to the final result of the analysis of the original specification reported in [125], thereby illustrating mainly the use of ground model ASMs for a good documentation which helps to keep the requirements traceable to the code – during refinement steps one has to make sure nothing is forgotten – and to enhance the maintainability and extendability of the working software system. We therefore explain how with the async LightControl ASM below one succeeds in disambiguating informally presented requirements, to structure them, to analyze them (with respect to internal consistency and correctness) and to complete them.[10] from the customer's, not the designer's point of view, in a way which makes them prototypically executable.

Starting from the original `LightControlRequirements` (⇝ CD),[11] refered to as Problem Description, we answer in the following subsections one by one the fundamental questions for requirements capture formulated in Sect. 3.1. Section 6.2.2 contains the formal counterparts of the Problem Description's objects and their properties. The Problem Description mentions three categories of needs, the user needs, the facility manager needs, and fault tolerance. For modularization purposes we parameterize the user and the facility manager needs so that they can be grouped in Sect. 6.2.3 as the possible manual interactions with the control system, separated from the automatic actions (Sect. 6.2.4) which are triggered by the control system. In Sect. 6.2.5

[10] Concrete examples of problems identified in the informal requirements are listed in [125, p. 601 and 619].

the required failure and service features are captured. In Sect. 6.2.6 we make the emerging system architecture (the component structure) explicit.

6.2.2 Signature (Agents and Their State)

The basic system objects are two sorts of *location*, namely *rooms* and (sections of) *hallways*, equipped with *light groups* (two groups of ceiling *lights* for rooms—one near the window and one near the wall, called window and wall ceiling lights—and ceiling lights for hallways) which come with operations of pushing various buttons (on the wall or a control panel) and of actuating dimmers. Locations are also associated with various motion detectors, light sensors and door-closing contacts. There are also status lines which report status values of the associated light groups. Also staircases are mentioned, but they enter the problem really only through their motion detector. Therefore we avoid the proliferation of irrelevant object types by including staircase motion detectors in the class of motion detectors which are related to the doors for entering a hallway from a staircase. These objects enter our model as parameters[12] for the various actions described in the following sections for the three types of agents: users, the facility manager and the control system.[13]

6.2.3 User Interaction (Manual Control)

The single manual actions which appear in the Problem Description are to push a RoomWallButton or a HallwayButton or the ControlPanel (user actions) and to ManuallySwitchOff lights in rooms and hallways (facility manager action). This leads to the four basic ASMs that we are going to describe in this section. How they work together as components of the async LightControl ASM is defined in Sect. 6.2.6.

RoomWallButton **(User action in rooms).** In every room both (wall and window ceiling) light groups have a wall switch which can be pushed by users. The switch behavior triggered by pushing the button is formulated as follows: (a) If the ceiling light group is completely on, it will be switched off, (b) otherwise it will be switched on completely. This is captured by the basic ASM RoomWallButton, where the external button-pressing event is reflected by an *event* function *lightgroup_wall_button_pressed* – a monitored

[11] We thank JUCS for the permission to reproduce here the requirements document which was published in [113].

[12] These parameters are naturally implementable as instances of appropriate classes.

[13] In a systematic documentation of the requirements elicitation one has to explicitly list the complete signature: the basic objects have to be defined through the lexicon, their properties and constraints have to be stated, making sure that the list is complete and correct with respect to the underlying application-domain information. We abstain from doing this here; any practical (computer supported) systematic method serves the purpose.

function supposed to become true when the corresponding button has been pressed and to become false when the rule fires whose guard contains the event function (PushButtonReq).[14] This interpretation resolves the incompleteness of the definition for *push button* in the dictionary of the Problem Description where it is not made clear when the light effect should take place, at the beginning or at the end of the possibly prolonged button-pushing action.

```
RoomWallButton($room, lightgroup$) =
  if $lightgroup\_wall\_button\_pressed(room, lightgroup)$ then
    if $lightgroup\_is\_completely\_on(room, lightgroup)$ then
      SwitchLightgroupOff($room, lightgroup$)
    else SwitchLightgroupCompletelyOn($room, lightgroup$)
```

The submachines for switching a light group off or completely on are defined as setting all lights in the corresponding room to *minDimValue* or *maxDimValue* respectively.[15] This definition resolves the apparent contradiction in the Problem Description where it is considered as safe to allow a person who wants to rest in a room to choose a light scene in which all the lights are switched off and the room is dark (U1Req). The function *mode* determines for each room whether the light was set by the user (*Manual*) or by the control system (*Ambient*).[16]

```
SwitchLightgroupOff($room, lightgroup$) =
  $mode(room) := Manual$
  forall $light \in lights\_in\_group(room, lightgroup)$
    SwitchLight($room, light, minDimValue$)
SwitchLightgroupCompletelyOn($room, lightgroup$) =
  $mode(room) := Manual$
  forall $light \in lights\_in\_group(room, lightgroup)$
    SwitchLight($room, light, maxDimValue$)
```

HallwayButton **(User action in hallways).** The switch buttons in hallway sections are linked in parallel. The light in any hallway section is required to be on if some of these buttons are defective (*any_hallway_button_defect*).[17] The event function *hallway_button_pressed* indicates that a switch button has been pressed and is supposed to become false by firing the rules in which the event appears in the guard.

[14] Such detailings of requirements have to be listed in a systematic documentation of all the decisions taken to interpret or complete the Problem Description.

[15] It is good practice to use symbolic names rather than constants.

[16] This interpretation of *Ambient* does not preclude letting the light from the sun be part of what is understood by the environmental light.

[17] See requirement `NF5`, where we interpret "not controllable manually" in view of the safety requirement `U1` as meaning that at least one hallway button is defective (NF5aReq). A "local" interpretation of "not controllable manually" in `NF5` is obtained by parameterizing *any_hallway_button_defective* with buttons.

```
HallwayButton(hallway) =
  if hallway_button_pressed(hallway) and
    not any_hallway_button_defect(hallway) then
      if light_is_on(hallway) then
        SwitchLightsOff(hallway)
      else SwitchLightsOn(hallway)
```

Switching on/off for a location (room or hallway) is defined as setting all lights of the location to *minDimValue* and *maxDimValue*, respectively. We group these lights for short as *lights_at*(*location*).[18] We use an abstract machine SwitchLight, not defined further here, which will be refined for the machine ServiceReport.

```
SwitchLightsOn(location) =
  forall light $\in$ lights_at(location)
    SwitchLight(location, light, maxDimValue)
SwitchLightsOff(location) =
  forall light $\in$ lights_at(location)
    SwitchLight(location, light, minDimValue)
  if location_is_room(location) then mode(location) := Manual
```

ManuallySwitchOff **(Manager action).** The possibility foreseen in requirement `FM6` for the facility manager to switch off the ceiling light in a location if it is not *occupied* is captured by the following basic ASM.

```
ManuallySwitchOff(location) =
  if manually_switch_off_pressed(location) and
    not occupied(location) then SwitchLightsOff(location)
```

The Problem Description takes the meaning of locations being *occupied* for granted. However, since the motion sensors of locations sense only motion, they will report "no motion" if somebody occupies a location without making any movement. Therefore[19] a better definition for a location to be not occupied is that there has been no motion for a period of *max_quiet_time* (RoomOccupationReq), to be measured starting from the time of the *last_motion*. To guarantee the consistency between user and facility manager light updates in rules RoomWallButton, HallwayButton, ManuallySwitchOff, ControlPanel we assume that the motion sensor detects when users push buttons (MotionDetectorReq). This is a semantic constraint which relates the notion of being occupied to the event functions *pressed* associated with buttons.

[18] Although for uniformity reasons we formulate SwitchLightsOn for locations, we will use it only for hallways, because by requirement `U5,U6,U9` and the dictionary entry "light scene", the light in a room is switched on only for a lightgroup as a whole.

[19] The question changes if other sensors are installed. This is a simple example of a hardware/software co-design issue.

```
$occupied(location) =$
  $current\_time - last\_motion(location) \leq max\_quiet\_time$
```

A basic ObserveMotionDetector ASM has the task of recording the time of the *last_motion*. It reads a monitored function *somebody_is_moving* which yields the value of the given motion detector, assumed to be true if the given motion detector is defective (NF5bReq) (to satisfy requirement `NF5`). To also reflect the malfunction requirement `NF4` we include the case that *somebody_is_moving* is true for a location if at least one of its motion detectors does not work correctly, so that in this case the light cannot be switched off by the facility manager.

```
ObserveMotionDetector($location$) =
  if $somebody\_is\_moving(location)$ then
    $last\_motion(location) := current\_time$
```

ControlPanel **(User room control panel action).** According to `U5`, `U6`, `U9` in the Problem Description, with the control panel the user can control the ceiling lights and the *light scene*, namely to switch on/off the ceiling light groups, to set a light scene or to activate the last-set light scene. This is formalized by the following basic ASM, which uses an event function *switch_value* to express the on/off position chosen by the user for the switch in question.

```
ControlPanel($room, switch$) =
  if $switch\_pressed(room, switch)$ then case $switch$ of
    $LightGroup(lg) \rightarrow$
      case $switch\_value(room, switch)$ of
        $On \rightarrow$ SwitchLightgroupCompletelyOn($room, lg$)
        $Off \rightarrow$ SwitchLightgroupOff($room, lg$)
    $SceneSelection \rightarrow$
      case $switch\_value(room, switch)$ of
        $Scene(s) \rightarrow$ SetLightScene($room, s$)
    $AmbientSelection \rightarrow$
      ActivateLightScene($room, last\_light\_scene(room)$)
```

Obviously one has to guarantee that simultaneous pushing on wall buttons and on the control panel does not produce effects which exclude each other. One can for example assume that the hardware solves this conflict, or one could establish a fixed priority (PushButtonReq).

The submachine for scene selection either activates the light scene to the one passed as parameter (in *ambient* mode) or (in *manual* mode) sets *last_light_scene* to the parameter (whereafter that scene can be activated by pressing *AmbientSelection*).

```
SetLightScene($room, scene$) =
if $mode(room) = Ambient$ then ActivateLightScene($room, scene$)
else $last\_light\_scene(room) := scene$
```

By requirement 2.10 (Paragraph 19) in the Problem Description a light scene contains an *ambient light level* and an ordered list of lights together with a dim value for each light. As the Problem Description dictionary indicates for "light scene", the control system has to switch on the lights in the given order with the corresponding dim value in order to achieve the specified ambient light level. Reflecting requirement `FM1` the control system must also take into account the ambient light from outside. We capture these requirements by introducing a function *lights_to_turn_on* which computes an ordered set containing all lights that should be switched on in this order, together with their dim values (LightSceneReq). Introducing an order makes the dictionary definition of "light scene" uniform with respect to the way light scenes and their light groups are built from components, achieving easy adaptability to changing requirements. The function depends for each room on the value of the outdoor light sensor and of the activated light scene. This specification still leaves much freedom for detailing the structure of light scenes.[20]

```
$\textsc{ActivateLightScene}(room, scene) =$
  $mode(room) := Ambient$
  $last\_light\_scene(room) := scene$
  **if** $scene = default\_light\_scene(room)$ **and**
    $outdoor\_light\_sensor\_defect(room)$ **then**
      $\textsc{SwitchLightsOn}(room)$
  **else let** $lights\_on =$
    $lights\_to\_turn\_on(room, outdoor\_sensor(room), scene)$
      **forall** $(light, value) \in lights\_on$
        $\textsc{SwitchLight}(room, light, value)$
      **forall** $light \in lights\_at(room) \setminus \{l \mid (l, v) \in lights\_on\}$
        $\textsc{SwitchLight}(room, light, minDimValue)$
```

The derived function *lights_to_turn_on* takes into account the information about malfunctioning lights, so that requirement `NF2` is correctly reflected, guaranteeing that "if any outdoor light sensor does not work correctly, the default light scene for all rooms is that both ceiling light groups are on". Also, the part of `NF1` which complements `NF2` is captured, namely by the assumption that the value of $outdoor\_sensor(room)$ remains constant if the sensor does not work correctly (OutdoorSensorReq). This assumption shows that `NF1` is not a requirement on the controller, but on the way the sensor values are transmitted as input to the controller. The definition of ActivateLightScene contains a decision about the interpretation of requirement `U10`. The Problem Description does not state what it means to

[20] The fact that we use this function only for rooms and not for hallways reflects that we consider the requirements `FM1` and `NF3` as useless for hallways (HallwayReq), since they have no windows, as is suggested by Fig. 1 in Paragraph 5 of the Problem Description, and if one does not want to consider light that may come into hallways through open room doors.

maintain the ceiling light group on a given light scene; we take it as requesting that the ceiling lights are set to *minDimValue* if they do not enter the lights to be turned on for the given light scene (U10Req).

6.2.4 Automatic Control

The automatic control system is required to be able to switch on/off any light in any location and to guarantee for rooms the use of the daylight.

AutoSwitchOn. Automatic switch-on is used to guarantee safe illumination at any time (`U1`, `U13`, `U14`). The lights in the hallway sections are not dimmable, so that switching on can be done there only completely. Switching on is triggered by the two events (1) motion in the hallway (*somebody_is_moving*) and (2) a door is open (*some_door_is_open*).

```
AutoSwitchOnInHallway($hallway$) =
  if ($somebody\_is\_moving(hallway)$ or $some\_door\_is\_open(hallway)$)
    and $light\_is\_off(hallway)$ then SwitchLightsOn($hallway$)
```

The analogous machine for rooms is more complicated. According to requirements `U3` and `U4` in the Problem Description one has to distinguish two cases:

`U3` If the room is reoccupied within T1 minutes after the last person has left the room, the *chosen light scene* has to be re-established.

`U4` If the room is reoccupied after more than T1 minutes since the last person has left the room, the *default light scene* has to be established.

In the first case, instead of establishing the *chosen light scene* we use the *last light scene* (U3Req) since otherwise the requirements would be incoherent (see Exercise 6.2.1). We do not commit here to any particular definition of *default_light_scene* (DefaultLightSceneReq).[21]

```
AutoSwitchOnInRoom($room$) =
  if ($somebody\_is\_moving(room)$ or $some\_door\_is\_open(room)$)
    and $light\_is\_off(room)$
  then ActivateLightScene($room, scene$)
  where
    $scene$ =if $recently\_occupied(room)$
      then $last\_light\_scene(room)$
      else $default\_light\_scene(room)$
    $recently\_occupied(room)$ =
      $current\_time - last\_motion(room) \leq t_1(room)$
```

[21] The definition in the dictionary is probably not reasonable because, with that definition, requirement `U4` makes no sense.

AutoSwitchOff. The control system switches off the light in a location which has been unoccupied for T3 or T2 minutes respectively (FM2, FM3). In accordance with requirement NF5 we do not switch off the light in a hallway section if one of its buttons is defective. To reflect the malfunction condition NF4, we stipulate that occurrence of a malfunction for a motion sensor is interpreted as the presence of motion, so that the location appears as occupied.

```
AutoSwitchOff($location$) =
if $location\_is\_hallway(location) \land any\_hallway\_button\_defect(location)$
  then $skip$
  elseif $\neg occupied(location) \land no\_motion\_for\_long\_time(location)$
    $\land \neg some\_door\_is\_open(location) \land \neg light\_is\_off(location)$
      then $Switch\_lights\_off(location)$
where $no\_motion\_for\_long\_time(location)$ =
  if $location\_is\_room(location)$ then
    $current\_time - last\_motion(location) > t_3(location)$
  else $current\_time - last\_motion(location) > t_2(location)$
```

UseDaylight. The control system should use daylight to achieve the desired *ambient light level* (FM1). We model this by reactivating the current light scene if the room is in ambient mode and there is no request for the ceiling lights. The following rule also reflects requirement U10 that “the ceiling light groups should be maintained depending on the current light scene”.

```
UseDaylight($room$) =
  if $no\_event\_for\_ceiling\_light(room)$ and $mode(room) = Ambient$
    then ActivateLightScene($room, last\_light\_scene(room)$)
```

Requirement U2 that “as long as a room is occupied, the chosen light scene has to be maintained” is fulfilled automatically because an ASM state remains unchanged unless a specific (user or control system) action triggers a change for the value of some specified functions for some specified arguments.

6.2.5 Failure and Service

Little is said in the Problem Description about the possibly complex failure-handling and the data reports from the normal system operation.

Malfunction. There are two actions to describe, namely identifying and handling malfunctions. Identifying malfunctions is a rather difficult application domain and less a software design problem. Not surprisingly the Problem Description does not provide any further details on this issue so that we assume having a function *malfunction_occurs* which tells whether a component works correctly or not; a component may be a hallway button, a light sensor, a motion sensor or any light. For building a concrete plant with its control software, this function has to be further specified by the domain expert, together with the support requested in FM8 for finding the reasons for occurring malfunctions. To reflect the malfunction requirement `FM1` we stipulate that this function can be updated also manually.

```
MALFUNCTION = forall $component \in all\_components$
  if $malfunction\_occurs(component)$ then
    $Handle\_malfunction(component)$
```

According to `U8`, `FM7` and `FM10`, the handling of malfunction logs the corresponding information. In the case in which a hallway button is defective we switch on the lights in that hallway (`NF5`). In case a hallway motion detector is defective, by assumption (NF5bReq) the function *somebody_is_moving* is true and we therefore switch on the lights by rule AUTOSWITCHONINHALLWAY. In the following rule we use i as index which has to match the name of the corresponding device (sensor, button).

```
HANDLEMALFUNCTION($component$) = case $component$ of
  $OutdoorLightSensor(i) \rightarrow$
    forall $room \in rooms\_under\_lightsensor(i)$
      $ReportAndRecord(LightSensorDefect(i))$
      $InformUser(room, LightSensorDefect(i))$
  $MotionSensor(location, i) \rightarrow$
    if $location\_is\_room(location)$ then
      $InformUser(location, MotionSensorDefect(location, i))$
      $ReportAndRecord(MotionSensorDefect(location, i))$
  $HallwayButton(hallway, i) \rightarrow$
    SWITCHLIGHTSON($hallway$)
    $ReportAndRecord(HallwayButtonDefect(hallway, i))$
  $Luminaire(location, light) \rightarrow$
    $ReportAndRecord(LightDefect(location, light))$
  where $ReportAndRecord(item)$ =
    $InformFacilityManager(item)$
    $WriteLogInDatabase(item)$
```

SERVICEREPORT. For the requirement (`FM9`) to report on energy consumption we data-refine SWITCHLIGHT by storing the current light dim value in a function *dim_value*. Switching the light then includes setting a dim value. As requested in the Problem Description, if the dim value is less than 10% of the maximum dim value, then the light is switched off.

```
SWITCHLIGHT($location, light, value$) =
  $dim\_value(location, light) := value$
  if $value < maxDimValue \div 10$ then
    $status\_of\_light(location, light) := Light\_Off$
  else $status\_of\_light(location, light) := Light\_On$
```

With *dim_value* one can compute the current *power_consumption* (derived function), taking into account the malfunctioning of lights) as follows:

```
$power\_consumption = \sum [p(l, dim\_value(l)) \mid l \in dom(dim\_value)]$
where $p(l, v) =$ case $light\_defect(l)$ of
  $NotDefect \rightarrow c * v$
  $DefectOn \rightarrow c * maxDimValue$
  $DefectOff \rightarrow c * minDimValue$
```

We use the constant c to adjust the dim value to the electrical power. The energy consumption is the integral of the power consumption over the time. Therefore we store the power consumption in each step in a dynamic function and define the energy consumption as the product of the interval t_e with the sum of the power consumptions. We assume that the following rule will be executed every t_e minutes.

```
ReportEnergyConsumption =
  $consumption(current\_time) := power\_consumption$
  $energy\_consumption(current\_time) := t_e * \sum_t \; consumption(t)$
```

Other features the Problem Description talks about without giving any indication as to what is required are setting parameters, detecting unreasonable input, etc., which can be defined in a routine way by basic ASMs.

6.2.6 Component Structure

In this section we capture the system architecture which emerges from the Problem Description and is constituted by three major components:

```
LightControl =
  ManualLightControl
  AutomaticLightControl
  FailureAndService

ManualLightControl =
  forall $location \in all\_locations$
    ManuallySwitchOff($location$)
    if $location\_is\_room(location)$ then
      RoomWallButton($location, LightGroupWall$)
      RoomWallButton($location, LightGroupWindow$)
      ControlPanel($location, switch(location)$)
    if not $location\_is\_room(location)$ then
      HallwayButton($location$)

AutomaticLightControl = forall $location \in all\_locations$
  AutoSwitchOff($location$)
  ObserveMotionDetector($location$)
  if $location\_is\_room(location)$ then
    AutoSwitchOnInRoom($location$)
    UseDaylight($location$)
  if not $location\_is\_room(location)$ then
    AutoSwitchOnInHallway($location$)

FailureAndService =
  Malfunction
  ReportEnergyConsumption
```

One can use different policies for the synchronization of the three submachines of LightControl to guarantee the consistency of their updates in the shared data area. One possibility is to make specific priority or scheduling assumptions on possibly conflicting actions, as we have indicated at various places during the formalization of the requirements. Another possibility is to impose a concrete scheduling on the coordination of the three submachines. Such a global policy relegates the consistency problem to the local levels of the single submachines. The Problem Description leaves these crucial issues completely open. In the ground model we could have reflected this freedom explicitly by introducing appropriate choice functions which determine at which time which submachine is running. For an executable version one has to take some concrete realistic decisions.

For the AsmGofer-executable version in [390] the manual and the automatic submachines alternate at a fixed rate – fast enough to guarantee the desired reaction time to user or environment input – and the failure and service submachines are executed in between with a certain frequency, again determined by the time requirements for failure handling and general services. In the initial state all locations are empty, rooms are in *Manual* mode with *default_light_scene* as *last_light_scene*, $last\_motion = 0$ and all lights are off.

6.2.7 Exercises

Exercise 6.2.1. (⤳ CD) Define for LightControl a scenario where a person upon entering a room twice, without changing the light scene and with nobody else in the room, for the first time gets the *default light scene* and, upon re-entering, gets the *chosen light scene* of another person who has been in the room before.

6.3 Time–Constrained Async ASMs

In this section we specify and verify two time-constrained algorithms which support fault tolerance for a distributed computing service, namely by elaborating the async ASM from [283] for the file transfer protocol Kermit [163] (Sect. 6.3.1) and the async ASM from [256] for a Processor-Group Membership protocol from [162] (Sect. 6.3.2).

6.3.1 Kermit Case Study (Alternating Bit/Sliding Window)

The goal of file transfer protocols is to guarantee that the files are transferred from sender to receiver correctly (without fail and in the right order), despite an unreliable network which may lose messages or deliver them in an order which is different from the sending order. The basic idea of the classical modem and network communication protocol we are going to analyze here is that every file is sent and retransmitted upon timeout until an acknowledgment of receipt arrives from the receiver, in which case the current file transfer is closed and the next one is started.

Two standard techniques are used for the identification of messages. One is the *alternating bit technique* where sender and receiver use a synchronization bit to identify the single file to be transferred currently: the sender sends every copy of the currently to be transferred file with attachment of its current synchronization bit (the current file number modulo 2), which upon message arrival is extracted and resent by the receiver as acknowledgment to the sender. Both sender and receiver check for every received bit for whether it matches their own current synchronization bit; in case of matching it is flipped – in reality the current file number is updated, which modulo 2 yields the new synchronization bit – and the current file transfer round is closed (which at the receiver's side includes accepting the current file). The second technique is an optimization which permits one to have finitely many files in transit simultaneously. As the file IDs of the files which are currently in transit the sender and receiver use, say, numbers within boundary values $low \leq high$, determining an integer interval called "window"; these numbers are checked upon message arrival to match the current interval boundaries, triggering the sliding of the boundaries in case of matching – hence the name *sliding window technique* – to provide room for the transfer of the next file.

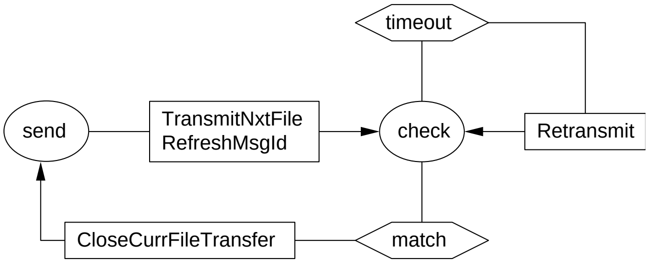
There are numerous papers in the literature analyzing with different techniques the communication protocols based upon these two message identification mechanisms. There are too many to list all of them; to mention a few, see [10], which uses the B method, and [372, Chaps. 27 and 28], [202, p. 390, Fig. 1], which use Petri nets. We show here that the two mechanisms can be captured and verified uniformly by the following sender and receiver rule templates – rules *Send, ReSend, Check* for the sender and rule *Receive* for the receiver – which we instantiate below to the alternating-bit protocol and then refine to the sliding-window optimization. In [283] it is shown how by appropriate reuse of verified alternating-bit and sliding-window protocol ASMs the complete Kermit protocol can be specified and verified, reflecting its four principal layers: (1) a session layer which controls sending and receiving files, (2) a transport layer which makes the message carrier sufficiently reliable, (3) a datalink layer which formats messages as strings to conform to communication network standards (on message start marking, its length, number, type, datum, checksum, end – features which are easily reflected by a pure data refinement of message data, types, numbers and related functions), and (4) a presentation layer which cuts strings into sequences of short strings, typically of printable ASCII characters, to be sent through the network (another pure data refinement step).

```
KermitTemplate =
  $Send$ = if $SendingTime$ then $StartNxtFileTransfer$
  $ReSend$ = if $RetransmitTime$ then $Retransmit$
  $Check$ = if $CheckingTime$ then
    if $Match$ then $CloseCurrFileTransfer$
    $ClearCurrMsg$
  $Receive$ = if $ArrivalTime$ then
    $ClearCurrMsg$
    $AcknowledgeReceipt$
    if $Match$ then $AcceptCurrFile$
```

Also, the signature of sender and receiver agents is largely uniform, reflected by the parameterization of functions by **self**. A function $\mathit{file}(\mathbf{self})\colon (\mathbb{N}\setminus\{0\}) \times \mathit{DATA}$ incorporates the sequence of files to be transferred or stored; it is monitored for the sender and controlled for the receiver. $\mathit{null}\colon \mathit{DATA}$ is a placeholder for "no-data" in acknowledgment messages. A controlled function $\mathit{currNum}(\mathbf{self})\colon \mathbb{N}$ yields the number of the file which is currently transferred or the next one to be stored. This function is refined below to denote an interval (the "sliding window"). We use $\mathit{send}(\mathbf{self})$ as abstract message sending action, which we assume to know its destination (whether as predefined or as retrievable from the message argument). $\mathit{queue}(\mathbf{self})\colon \mathit{MESSAGE}^*$ is a shared function where an input agent $a(\mathbf{self})$, delivering to **self** messages from an abstract message carrier, appends arriving input $in(a)$ at one end and where **self** fetches at the other end the next message to receive. Messages can be viewed here as consisting of a file together with an identifier, provided by projection functions $\mathit{data}, \mathit{msgId}$ defined on $\mathit{MESSAGE} = \mathit{DATA} \times \mathit{MsgID}$. For reasons of uniformity we denote the alternating bit message identifier by a derived function $\mathit{bit}(\mathbf{self}) = \mathit{currNum}(\mathbf{self}) \mathit{mod} 2$, called the alternating bit of the agent, and write also bit instead of msgId. A message $(\mathit{file}(i), \mathit{Id})$ is called an i-msg and a message $(\mathit{null}, \mathit{Id})$ acknowledging the receipt of $(\mathit{file}(i), \mathit{Id})$ is called an i-ack, where $\mathit{Id} = i$ or $\mathit{Id} = i \bmod 2$. The sender uses also a monitored function $\mathit{timeout}$ on which fairness constraints will be imposed below, without which the protocol would not be correct.

Alternating-bit instance. The agents of the async AlternatingBit ASM therefore are a sender and a receiver plus two input agents, representing the ends of the not completely reliable message-carrying medium – the network which remains implicit as environment. The input agents apply the rule KermitInput to deliver their input in, received from the network, by appending it to the queue which is shared with the input agent's client (sender or receiver). It will be in terms of this in function, shared between the input agent and the network, that we formulate the limit one has to impose on the

Fig. 6.9 Alternating bit sender ASM



forgetfulness of the communication mechanism to still be able to guarantee a correct file transfer (see the Message Carrier Reliability assumption).

```
KermitInput = **if** $in(\textbf{self}) \neq undef$ **then**
    $append\ in(\textbf{self})\ to\ queue(\textbf{self})$
    $in(\textbf{self}) := undef$
```

Once the sender has started to *send* a new file,[22] it switches to *check* mode to retransmit that file – until its receipt is acknowledged by a message from the receiver which matches the current *msgId*, in which case the sender closes the transfer of that file and starts to *send* the next file. This results in the following alternating-bit instance of the macros for the KermitTemplate sender rules and yields the AlternatingBitSender in Fig. 6.9.

```
AlternatingBitSenderMacros =
$SendingTime = (ctl\_state = send)$
$StartNxtFileTransfer =$
    $\{TransmitNxtFile, RefreshMsgId, ctl\_state := check\}$ **where**
        $TransmitNxtFile = send\ (file(currNum + 1), currNum + 1 \bmod 2)$
        $RefreshMsgId = currNum := currNum + 1$
$CheckingTime = (ctl\_state = check)$ **and** $queue \neq empty$
$Match = (msgId(fst(queue)) = bit)$
$CloseCurrFileTransfer = (ctl\_state := send)$
$ClearCurrMsg = delete\ fst(queue)\ from\ queue$
$RetransmitTime = (ctl\_state = check$ **and** $timeout)$
$Retransmit = \{send\ (file(currNum), bit), timeout := false\}$
```

Analogously we define the AlternatingBitReceiver with the same macro definitions for *ClearCurrMsg*, *Match*, *RefreshMsgId* as for the sender plus the following further macro instances in *Receive*.

[22] We use *send* without parameters for a control state and with parameters to denote an abstract message-sending action.

AlternatingBitReceiverMacros =
  $\mathit{ArrivalTime} = (\mathit{queue} \neq \mathit{empty})$
  $\mathit{AcknowledgeReceipt} = \mathit{send}\ (\mathit{null}, \mathit{msgId}(\mathit{fst}(\mathit{queue})))$
  $\mathit{AcceptCurrFile} = \{\mathit{file}(\mathit{currNum}) := \mathit{data}(\mathit{fst}(\mathit{queue})), \mathit{RefreshMsgId}\}$

To prove AlternatingBit correct and complete we have to formulate the AlternatingBit run assumptions, including the initialization. For the initial state we assume that at both sender and receiver there are no messages (i.e. $\mathit{queue} = \mathit{empty}$, $\mathit{in} = \mathit{undef}$, no message is in transit, say $\mathit{InTransit} = \mathit{empty}$ where by definition $\mathit{InTransit}(\mathbf{self})$ contains (in their sending order) all messages sent to **self** which are not lost and did not yet arrive, including $\mathit{in}(a)$ of the input agent $a(\mathbf{self})$ if $\mathit{in}(a) \neq \mathit{undef}$). For the sender we stipulate $\mathit{ctl\_state} = \mathit{send}$, so that *SendingTime* is true, $\mathit{timeout} = \mathit{false}$ and $\mathit{currNum} = 0$, so that $\mathit{bit}(\mathit{sender}) = 0$. The first file to be transferred is $\mathit{file}(1)$; when executing *Send* and thereby *StartNxtFileTransfer* for the first time, the sender sends $\mathit{file}(\mathit{currNum} + 1)$ and increases its current file number *currNum* by 1, so that it becomes the same as the current value of $\mathit{currNum}(\mathit{receiver})$. For the receiver we assume $\mathit{currNum} = 1$, so that $\mathit{bit}(\mathit{receiver}) = 1$, $\mathit{file}(x) = \mathit{undef}$ for every x. By $\mathit{InArrival}(\mathbf{self})$ we denote the concatenation of $\mathit{InTransit}(\mathbf{self})$ and $\mathit{queue}(\mathbf{self})$.

Async AlternatingBit runs are constrained to satisfy the following five assumptions. Not to distract the attention from the main issue for this protocol, we disregard an irrelevant technicality, namely the sender internal non-determinism resulting from the possibility that the guards of both *Check* and *ReSend* may become true simultaneously, depending on how the values of *timeout* are related to *CheckingTime*. It does not matter which scheduling is chosen as long as eventually each enabled rule will be applied.

Timeout: When for both sender and receiver *InArrival* is empty and it is not *SendingTime*, then *timeout* eventually becomes true.

Carrier Reliability: For every tail ρ of each infinite run, if in ρ an agent applies *send* infinitely often, some of the messages sent do not get lost and arrive as input value *in* at their target.

Message Order:[23] Messages which do not get lost during the transmission and are received as input at their target, arrive there one after the other in the order they have been sent.

Agent Fairness: In every tail ρ of each infinite run and for every agent other than the message carrier, if the agent is enabled infinitely often in ρ, then it will make a move in ρ.

Order of Moves: On instances of a non-lost message m the agents move in the following order: sender, Input(receiver), receiver; receiver, Input(sender), sender, i.e.:

- a *Send* or *ReSend* move with *send* m comes before the corresponding $\mathit{Input}(\mathit{receiver})$ move with $\mathit{in} = m$, which in turn comes before the corresponding *Receive* with $\mathit{fst}(\mathit{queue}) = m$.

- *Receive* with $\mathit{fst}(\mathit{queue}) = m$ comes before $\mathit{Input}(\mathit{sender})$ with the corresponding $\mathit{in} = m$, which in turn comes before the corresponding *Check* with $\mathit{fst}(\mathit{queue}) = m$.

Proposition 6.3.1 (AlternatingBit run-time verification). Every run of AlternatingBit transfers files correctly, transferring all files and in an order-preserving way in the following sense:

Correctness: In every reachable state of the run and for every file number x:
- $\mathit{file}(\mathit{receiver})(x) = \mathit{file}(\mathit{sender})(x)$, if they are defined,
- $\mathit{file}(\mathit{receiver})(x-1)$ is defined when $x = \mathit{currNum}(\mathit{receiver}) > 1$ and remains unchanged from then on.

Completeness: A specimen of every $(\mathit{data}, \mathit{id})$ sent by the sender or receiver will eventually enter and leave $\mathit{InArrival}(\mathit{receiver})$ or $\mathit{InArrival}(\mathit{sender})$, respectively. A specimen of every file which is sent by the sender is eventually received, and an acknowledgment of its receipt is eventually received by the sender.

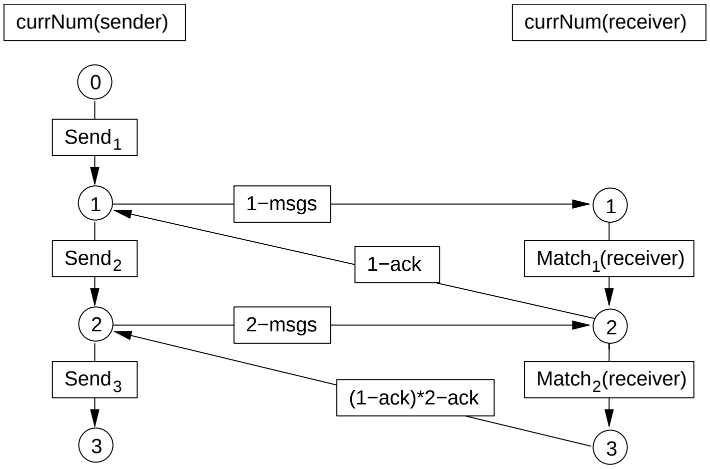
Proof. To easen the exposition we assume the moves to happen at real-time moments which are used to refer to the moves and respect the given partial order, i.e. we consider an arbitrary interleaving of moves in the run and take care not to use any property which holds only for the particular interleaving. The simple proof idea is illustrated in Fig. 6.10 and can be expressed as follows. Split the run into i-phases of all moves belonging to the transfer of $\mathit{file}(i)$. Let Send_i be the ith application of the *Send* rule, and similarly let $\mathit{Match}_i(\mathit{agent})$ be the ith application by *agent* of the rule with true premise *Match*. Into $\mathit{phase}_i(\mathit{sender})$ we put all the moves from Send_i included to Send_{i+1} excluded, i.e. all the moves which happen in states where either Send_i is enabled or ($\mathit{currNum}(\mathit{sender}) = i$ and Send_{i+1} is not enabled); we say that the receiver is in phase_i if $\mathit{currNum}(\mathit{receiver}) = i$. This reflects that the receiver enters phase $i+1$ by move $\mathit{Match}_i(\mathit{receiver})$ and then stays phase-ahead of the sender until move $\mathit{Match}_i(\mathit{sender})$ included. Therefore $[\mathit{Send}_i, \mathit{Send}_{i+1})$, where sender emits only i-msgs, can be split into $[\mathit{Send}_i, \mathit{Match}_i(\mathit{receiver}))$ (where the receiver emits only $(i-1)$-acks belonging to still-transiting retransmissions of an already-received file) and $[\mathit{Match}_i(\mathit{receiver}), \mathit{Match}_i(\mathit{sender})]$ (where the receiver emits only i-acks).

This idea is made precise in the following three lemmas which are used to verify AlternatingBit runs.

Lemma 6.3.1 (Phases of AlternatingBit). For AlternatingBit runs the following properties hold:

[23] The optimization by the sliding-window technique below avoids this rather unrealistic order assumption.

Fig. 6.10 Phases in AlternatingBit runs



- For every $i \geq 1$, when the sender reaches phase i to fire $Send_i$ increasing $currNum(sender)$ to i, the receiver has already moved to phase i. Thus during sender phase i, receiver is in phase i or $i+1$.
- During $(Send_i, Match_i(receiver)]$, bit and $currNum$ are the same at sender and receiver. $Match(receiver)$ fires only when $bit(sender) = bit(receiver)$.
- $currNum(receiver) = currNum(sender) + 1$ holds in

  $$(Match_i(receiver), Send_{i+1}],$$

  so that $Send$ fires only when $bit(sender) \neq bit(receiver)$.

Lemma 6.3.2 (InArrival message order). For AlternatingBit runs the following properties hold:

- During sender phase $i \geq 1$, $InArrival(sender)$ can contain only $(i-1)$-acks followed by i-acks.
- During receiver phase $i \geq 1$, $InArrival(receiver)$ can contain only $(i-1)$-msgs followed by i-msgs.
- In every state the concatenation $InArrival(sender)InArrival(receiver)$ has the form $(i-1-MSG)^*(i-MSG)^*$ (where MSG is either ack or msg) with bit-sequence $flip(bit(sender))^* bit(sender)^*$.

Corollary 6.3.1. When the sender has an ack with $bit(sender)$ $InArrival$, then $bit(sender) = flip(bit(receiver))$.

Lemma 6.3.3 (InArrival(receiver)-data). In every reachable state of AlternatingBit runs, $InArrival(receiver)$ messages contain the data of

– *file*(*currNum*(*sender*)), if their *msgId* is *bit*(*sender*)
– *file*(*currNum*(*sender*) − 1) otherwise.

We now prove the correctness claim from these lemmas. *file*(*receiver*) is updated only by *AcceptCurrFile* in *Match*(*receiver*), in a state where (by the phase lemma) sender and receiver have the same *currNum* and *bit* – which is the bit of the accepted message, so that the InArrival(receiver)-data lemma implies the first claim. The second claims follows since initially *currNum*(*receiver*) = 1; it is increased (by 1) only through *RefreshMsgId* in *AcceptCurrFile* as part of *Match*(*receiver*), when *file*(*receiver*) is defined for *currNum*(*receiver*).

The first claim of file transfer completeness follows from the constraints on the AlternatingBit run and from the rules *Retransmit*, *Input*, *ClearCurrMsg*. To establish the second claim let a file be sent by $Send_i$. Then the first time it leaves *queue*(*receiver*) its *msgId* matches *bit*(*receiver*) (by the phase lemma) so that *AcceptCurrFile* in $Match_i$(*receiver*) stores the file at the receiver and *AcknowledgeReceipt* sends the first i-ack. Therefore, when the first i-ack eventually leaves *queue*(*sender*), it is received by $Match_i$(*sender*) firing *CloseCurrFileTransfer*. □

Proof.[AlternatingBit Phases] Induction on i. Case $i = 1$: by initialization. $Send_{i+1}$ is preceded by $Match_i$(*sender*) whose i-ack must have been sent by a preceding *Receive* of an i-msg, which in turn can come only after or at $Match_i$(*receiver*): that move is the only one which can bring receiver into phase $i + 1$. From then until $Send_{i+1}$ holds *bit*(*sender*) ≠ *bit*(*receiver*). □

Proof.[InArrival message order] Simultaneous induction on phase changing moves $Send_i$, $Match_i$(*receiver*). Since j-msgs are sent in order $j = 1, 2, \ldots$, by the Message Order Assumption they can be received only in this order, so that also j-acks may be sent and arrive only in this order. Therefore, when the first i-ack is matched by the sender, only i-acks can be left *InArrival*(*sender*), and only i-msgs can still be *InArrival*(*receiver*), from where they will be discarded. The next sender move is $Send_{i+1}$, so that until $Send_{i+2}$ (excluded), messages which are newly transmitted to *InArrival*(*receiver*), yielding acknowledgments which may reach *InArrival*(*sender*), can be only $(i+1)$-msgs.

The second claim for i-msgs which are *InArrival*(*receiver*) and eventually get matched by $Match_i$(*receiver*) follows by a symmetric argument. The third claim is implied by the first two as follows. From the state (included) where $Send_i$ takes place until the state (included) where receiver move $Match_i$ takes place there can be no i-ack *InArrival*(*sender*); similarly from receiver move $Match_i$ – which switches its phase to $i+1$ – until $Send_{i+1}$ included, there can be no $(i + 1)$-msg *InArrival*(*receiver*). Therefore, during the sender phase i, *InArrival*(*sender*)*InArrival*(*receiver*) is of form $(i - 1 - MSG)^*(i - MSG)^*$ (where MSG is either ack or msg), and therefore its bit-sequence has the form *flip*(*bit*(*sender*))**bit*(*sender*)*. □

Proof.[Corollary] Let ack be an i-ack. Then the sender is in phase i or $i+1$ (InArrival message order lemma). Upon sending in $\mathit{Match}_i(\mathit{receiver})$ the first i-ack, by the phase lemma the receiver has flipped its bit from $\mathit{bit}(\mathit{sender})$ to the opposite bit and has passed from its and the sender's phase i to phase $i+1$. Therefore the claim follows in case the sender in the considered state is still in phase i or has just made move Send_{i+1}. Through Send_{i+1} the $\mathit{bit}(\mathit{sender})$ changes to the complement of i-ack. □

Proof.[InArrival(receiver)-data] If an i-msg m is $\mathit{InArrival}(\mathit{receiver})$ in state S, the receiver is in phase i or $i+1$ (InArrival message order lemma) and the phase of the sender is i or $i+1$ (phase lemma). Thus if $\mathit{msgId}(m) = \mathit{bit}(\mathit{sender})$, $\mathit{bit}(\mathit{sender})$ and therefore $\mathit{currNum}(\mathit{sender})$ do not change after Send_i until S, which by *Retransmit* implies the first claim. Otherwise between Send_i and reaching S, Send_{i+1} has fired as the only rule which changes $\mathit{currNum}(\mathit{sender})$. This proves the second claim. □

Refinement to sliding window. AlternatingBit can be refined to its "sliding window" optimization as follows. The idea that allows (multiple instances of) *different* files to be in transit simultaneously consists in grouping the files at both sender and receiver into an interval such that (a) a new round for the transmission of the next file is started by inserting (really a name of) that file at the upper end, and (b) retransmission upon timeout resends the file at the lower end. The receiver window remains synchronized with (i.e. never wider than) the sender window since only the sender can initiate a new round, namely upon receiving an acknowledgment for the file with number *low*. With this optimization one also gets rid of the unnatural message-order assumption made for the alternating-bit protocol verification.

To identify the different files which are in transit simultaneously the set *BOOL* does not suffice any more for *msgId*, which is therefore refined to, fir example, $\mathbb{N}$; for mnemonic reasons we then write *fileNum* for *msgId*. The role of $\mathit{currNum}(\mathbf{self})$ in *AcceptCurrFile* and in *CloseCurrFileTransfer* is refined by using $\mathit{fileNum}(\mathit{fst}(\mathit{queue}))$ instead. The roles of $\mathit{currNum}(\mathbf{self})$ in *Match*, as well as in *StartNxtFileTransfer* and *Retransmit* for determining which file to send next or to retransmit, are refined via a pair of integer interval boundaries identifying the "window" of files which are currently in transit: $\mathit{high}(\mathit{sender})$ refines the role of *currNum* in *Send*, $\mathit{low}(\mathit{sender})$ the one of *currNum* in *Retransmit*. Correspondingly *SendingTime* is refined by **not** *windowFull*, where $\mathit{windowFull} = (\mathit{high}(\mathit{sender}) - \mathit{low}(\mathit{sender}) + 1 = \mathit{winsize})$ for an implementation dependent constant $\mathit{winsize} = \mathit{maxWindowSize}$; the bit-test $\mathit{msgId}(\mathit{fst}(\mathit{queue})) = \mathit{bit}$ at sender and receiver is refined to the "window test" $\mathit{Match} = \mathit{low} \leq \mathit{fileNum}(\mathit{fst}(\mathit{queue})) \leq \mathit{high}$.

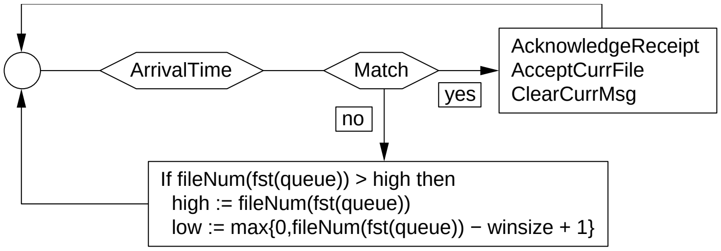
The refinement of *CloseCurrFileTransfer* has to take into account the policy adopted to guarantee the protocol to be complete, namely that only the message at the *low* window end is retransmitted upon *timeout*. The *Match*ing file may not be the one at $\mathit{low}(\mathit{sender})$ because for this protocol no message ordering assumption is made. Therefore, no relation is known between the order in which messages are sent and arrive, in particular not for sent i-msgs and received i-acks. As a consequence, to prevent future retransmissions of an acknowledged file, in case this file is not the one numbered *low*, to *CloseCurrFileTransfer* one has to store the acknowledgment, say by setting $receivedAck(fileNum(fst(queue)))$ to true. Only when $receivedAck(low)$ becomes true can the corresponding file transfer round be completely closed by sliding the window at the lower end via $low := low + 1$; see $SlideWindow(sender)$. Analogously a rule $SlideWindow(receiver)$ slides the receiver window upon arrival of a new message m whose $fileNum(m) \leq high(sender)$ does not match yet the upper receiver window bound $high(receiver)$. In this case $SlideWindow(receiver)$ can be executed – correctly so, since a new file can arrive only when the receiver window is not full and its number is never smaller than $low(receiver)$. After the sliding action $fileNum(m)$ matches the receiver window bounds and thus is accepted. The sliding may increase $high(receiver)$ by 1, as is always the case at the sender site, but only if starting the transmission of the new file by *StartNxtFileTransfer* has made the sender window full, due to the unknown order in which sent messages arrive (if they arrive at all).

This leads to the async SlidingWindow ASM with sender, receiver, input and hidden message carrier as independent agents. Their rules instantiate the KermitTemplate. We list below only the new rules *SlideWindow* and those macros which refine the homonymous ones of AlternatingBit. We “team” *SlideWindow* with the sender or receiver, respectively, instead of introducing independent agents, because we want to neglect, as already done for AlternatingBit, the possible though irrelevant non-determinism among receiver rules as well as among sender rules. For the sender it is here not only between *Send* and *ReSend*, but includes also *Check* (because the control state *check* disappeared) and $SlideWindow(sender)$, depending on the scheduling of *SendingTime*, *CheckingTime*, *RetransmitTime* and of $receivedAck(low)$ becoming true. The only relevant issue is that each enabled rule will eventually fire.

The AlternatingBit initialization and run constraints are carried over to SlidingWindow, deleting the message order assumption and refining the initialization by $low = 1$, $high = 0$ (for both sender and receiver) and by the sender condition $receivedAck(x) = false$ for all x.

```
SlidingWindowSenderMacros =
$SendingTime = \textbf{not}\ windowFull$
$StartNxtFileTransfer = \{TransmitNxtFile, RefreshMsgId\}$
  $TransmitNxtFile = send\ (file(high + 1), high + 1)$
  $RefreshMsgId = high := high + 1$
$CheckingTime = queue \neq empty$
$Match = low \leq fileNum(fst(queue)) \leq high$
$CloseCurrFileTransfer = receivedAck(fileNum(fst(queue))) := true$
```

Fig. 6.11 SlidingWindowReceiver ASM


ArrivalTime
Match
yes
no
AcknowledgeReceipt
AcceptCurrFile
ClearCurrMsg
If fileNum(fst(queue)) > high then
high := fileNum(fst(queue))
low := max{0,fileNum(fst(queue)) − winsize + 1}

$$\begin{aligned}
&\mathit{RetransmitTime} = \mathit{timeout}\\
&\mathit{Retransmit} = \{\mathit{send}\ (\mathit{file}(\mathit{low}), \mathit{low}), \mathit{timeout} := \mathit{false}\}\\
&\mathit{SlideWindow}(\mathit{sender}) = \textbf{if } \mathit{receivedAck}(\mathit{low}) \textbf{ then } \mathit{low} := \mathit{low} + 1
\end{aligned}$$

Including $\mathit{SlideWindow}(\mathit{receiver})$ into SlidingWindowReceiver yields the machine illustrated by Fig. 6.11 with its new macros defined below. The value $\mathit{high}(\mathit{receiver})$ is the highest number of any file sent and received so far, which is $\leq \mathit{high}(\mathit{sender})$. Therefore when a newly sent file arrives at receiver (for the first time) and the new file number exceeds $\mathit{winsize}$, then $\mathit{file}(\mathit{low}(\mathit{receiver}))$ has been already received (and stored), so that $\mathit{SlideWindow}(\mathit{receiver})$ can safely be fired sliding $\mathit{low}(\mathit{receiver})$.

```
SlidingWindowReceiver = {Receive, SlideWindow(receiver)}
where
  Receive = if ArrivalTime and Match then
    {AcknowledgeReceipt, AcceptCurrFile, ClearCurrMsg}
  AcceptCurrFile = (file(fileNum(fst(queue))) := data(fst(queue)))
  SlideWindow(receiver) =
    if ArrivalTime and fileNum(fst(queue)) > high then
      high := fileNum(fst(queue))
      low := max{0, fileNum(fst(queue)) − winsize + 1}
```

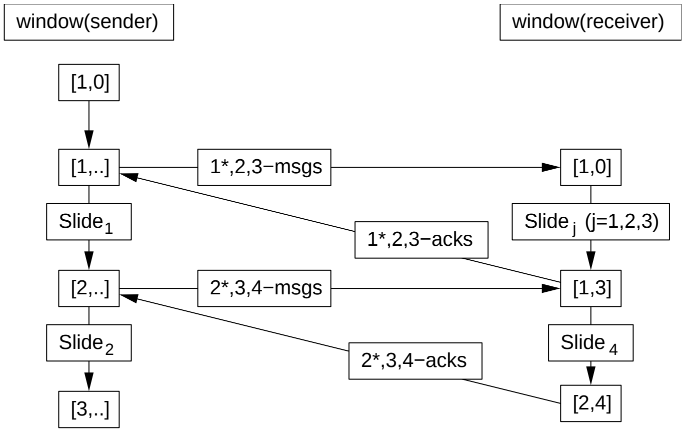
Proposition 6.3.2 **(**SlidingWindow **run-time verification).** Every run of SlidingWindow transfers files correctly, transferring all files and in an order preserving way in the following sense:

InArrival Completeness: For every $(\mathit{data}, \mathit{id})$ which is sent by the sender or receiver, an instance will eventually enter and leave $\mathit{InArrival}(\mathit{receiver})$ or $\mathit{InArrival}(\mathit{sender})$.

File Completeness: Every file which is sent by the sender is eventually received and stored by the receiver, and an acknowledgment of its receipt is eventually received by the sender.

Proof. The first claim follows from the run constraints and the refined rules $\mathit{Retransmit}$, Input, $\mathit{SlideWindow}(\mathit{receiver})$, $\mathit{ClearCurrMsg}$.

Fig. 6.12 Phases in SLIDINGWINDOW runs



For the second claim we use again a phase lemma. The critical moves which delimit the transfer of $\mathit{file}(i)$ are $\mathit{Slide}_i(\mathit{sender})$ – increasing low from i to $i+1$ – and $\mathit{Slide}_i(\mathit{receiver})$, starting $\mathit{file}(i)$-reception by increasing high from $i-1$ to i and directly followed by *Receive* of the first instance of an i-msg. The $\mathit{file}(i)$-transfer interval to be analyzed is therefore $(\mathit{Slide}_{i-1}, \mathit{Slide}_i]$ at the sender. We say that the receiver is in phase i if $\mathit{high}(\mathit{receiver}) = i$; thus this phase is $(\mathit{Slide}_i, \mathit{Slide}_{i+1}]$. The following lemma which is illustrated in Fig. 6.12 for $\mathit{winsize} = 3$, can be proved by an induction on runs.

Lemma 6.3.4 (SLIDINGWINDOW **run phases).** In phase $i \geq 1$, the sender can send finitely many i-msgs and for each $0 < j < \mathit{winsize}$ at most one $i+j$-msg. The sender can fire Slide_i to enter phase $i+1$ only after the receiver has already made a Slide_{i+j} move for some $0 \leq j < \mathit{winsize}$ followed by a *Receive* move where $\mathit{data}(i\text{-}\mathit{msg}) = \mathit{file}(\mathit{sender})(i)$ is stored in $\mathit{file}(\mathit{receiver})(i)$ and the first i-ack is sent.

Corollary 6.3.2. In every reachable state of SLIDINGWINDOW runs and for every file number i, $\mathit{file}(\mathit{receiver})(i) = \mathit{file}(\mathit{sender})(i)$ if both are defined. $\mathit{file}(\mathit{receiver})(i)$ is defined when $\mathit{phase}(\mathit{receiver}) \geq i$.

The second claim can be proved as follows. If an i-msg has been sent, by the phase lemma and the first claim at the latest when the sender is ready to fire Slide_i , the receiver has received a copy of an i-msg and stored $\mathit{file}(i)$. When the first i-ack eventually leaves $\mathit{queue}(\mathit{sender})$, the transfer phase for $\mathit{file}(i)$ terminates by applying *CloseCurrFileTransfer* which will be followed by an application of $\mathit{SlideWindow}(\mathit{sender})$ when $\mathit{low} = i$. □

6.3.2 Processor-Group-Membership Protocol Case Study

A classical approach to fault tolerance for distributed computing services consists in forming a server group whose members cooperate to provide ("identical versions" of) the intended service and to whose sites the relevant service state information is replicated. Group-membership protocols have to ensure that despite information propagation delays and server failures, the service state information which is stored at each group member remains up-to-date and is the same for all group members (in the steady state). In the case of processor-group membership, via message exchange a global agreement has to be achieved about who are the members of the group of all correctly functioning processors in the system; every time a processor in the system fails or starts working again a new agreement has to be achieved (read: a new group has to be formed). We take a real-time constrained processor-group membership protocol from [162] to illustrate how to make intricate timing and message-passing assumptions precise in terms of asynchronous runs of an accurate abstract model, for which one can formulate the correctness properties of interest explicitly, to make them verifiable by rigorous mathematical or machine-assisted proofs prior to the protocol implementation. We focus here on building an async ground model ASM GroupMember and defining its runs; see [256] for the correctness proof, which consists in an analysis of these GroupMember runs as illustrated above in detail for AlternatingBit runs.

Signature and state constraints. The objects of study are processors p, elements of a finite domain $PROCESSOR$, each equipped with its own real-time $Clock(p) \colon REAL \cup \{\infty\}$, a strictly increasing monitored function which assumes the value ∞ in a final state if there is one. The independence of these clocks is constrained only by the *delivery bound* formulated below for the travel time of messages between processors. On each alive processor p, one among the finitely many processes is scheduled to run as the current process $CurProc(p)$, among them the membership server $\mathrm{MS}(p)$ and the broadcast server $\mathrm{BS}(p)$ described below.[24] The scheduling is done by the Scheduler(p) described below. Processes execute various tasks from their deadline (i.e. the moment scheduled for their start) to completion. The task scheduling is supposed to be earliest-deadline-first; $Dline(x)$ denotes the minimum among the deadlines of the tasks waiting to be handled by x and is defined below for $x = \mathrm{MS}(p), \mathrm{BS}(p)$. Processors p can be interrupted between tasks, namely if one of their task-executing processes is scheduled after its deadline; a function $Status(p) \colon \{sober, crashed, recovered\}$ indicates for each processor its current failure status.

[24] In fact these two are the only processes we model explicitly. The passing of time abstractly reflects the moves of other processes, not specified further here, whose deadline misses or other performance failures are assumed to be detected and turned into processor crashes.

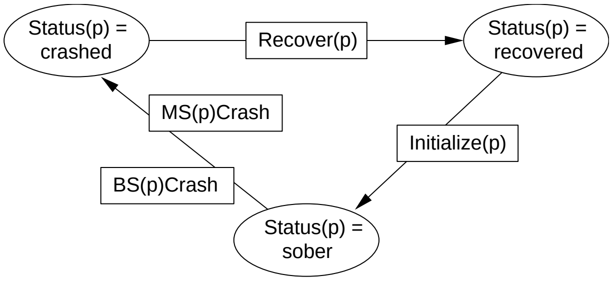
The idea of the protocol, made precise in Theorem 6.3.1 below, is to keep the correctly working processors in a group, all of whose members have the same membership view, which has to include each of them. A processor stays in its group until either some processor fails (in which case within a bounded *failure detection delay* a new group without that processor will be formed) or some processor recovers and attempts to rejoin (namely by creating a new group which – within a bounded *join delay* – will be joined by every other correctly working processor). The group and the local membership views are maintained by two processes of each group member, namely periodic broadcasting of messages attesting the presence in the group (via the broadcast server $\mathrm{BS}(p)$) and elaboration (via the membership server $\mathrm{MS}(p)$) of arrived or missing expected broadcast messages from group members or about recovery from a crash (with subsequent formation of a new group).

The fault tolerance life cycle of processors shown in Fig. 6.13 captures that the protocol distinguishes two ways to trigger and handle a crash, involving either the broadcast server $\mathrm{BS}(p)$ or the membership server $\mathrm{MS}(p)$. $\mathrm{BS}(p)$ periodically sends a broadcast message indicating to all processors that p is alive, wherefore the period d_h is called heartbeat; if $\mathrm{BS}(p)$ is scheduled too late – meaning that $\mathit{Clock}(p)$ exceeds the p-broadcast time $\mathit{BCastTime}(p) = \mathit{Dline}(\mathrm{BS}(p))$ in a state in which p is sober and the currently scheduled process is $\mathit{CurProc}(p) = \mathrm{BS}(p)$ – then p has missed the deadline of a broadcast and therefore is interrupted by $\mathrm{BS}(p)\mathit{Crash}$. The second reason for interruption of p (see $\mathrm{MS}(p)\mathit{Crash}$) is that $\mathrm{MS}(p)$ is scheduled too late – meaning that in a sober state with $\mathit{CurProc}(p) = \mathrm{MS}(p)$ processing a so-called new-group message as its current message $\mathit{CurMsg}(p)$, $\mathit{Clock}(p)$ exceeds the message deadline $\mathit{Timestamp}(\mathit{CurMsg}(p))$. In this case p misses the deadline of this new-group message, which is a broadcast every membership server sends out after its processor has *recovered* from a crash when handling in rule *Initialize* the processor's entry into a new processor-group. The membership server $\mathrm{MS}(p)$ is called (i.e. assigned by the SCHEDULER(p) agent to $\mathit{CurrProc}(p)$) for a processor to $\mathit{Recover}(p)$, but maintains also the processor group information, namely $\mathit{Group}(p)$ and the membership view $\mathit{Members}(p)$ (see below), while p is alive; this is done by handling incoming messages which either announce the constitution of a new group (type *newGp*) or report the processors present in a group (type *present*).

Elements of $\mathit{MESSAGE} = \mathit{MSGTYPE} \times \mathit{REAL} \times 2^{\mathit{PROCESSOR}}$ are kept abstract; their type (*newGp* or *present*), timestamp and view are accessible via functions $\mathit{Type}(m)$, $\mathit{Timestamp}(m)$ and $\mathit{View}(m)$. $\mathit{Timestamp}(m)$ records the time the group was created by $\mathit{Initialize}(p)$ and is used here as group Id. $\mathit{View}(m)$ represents the set of all members of processor group $\mathit{Timestamp}(m)$. Each message has a deadline which is defined as follows.

$$\mathit{Deadline}(m) = \begin{cases} \mathit{Timestamp}(m), & \text{if } \mathit{Type}(m) = \mathit{newGp}; \\ \mathit{Timestamp}(m) + d_n, & \text{if } \mathit{Type}(m) = \mathit{present}. \end{cases}$$

Fig. 6.13 Fault tolerance life cycle of processors



Here d_n is a delay which is supposed to be greater than the maximum network propagation delay, giving every correctly functioning processor a chance to see such messages; for *newGp*-messages this delay is already included in the definition of their timestamp when they are created by the *Initialize*(p) macro in rule $\mathrm{MS}(p)$.

Since the message *CurMsg*(p) currently seen by $\mathrm{MS}(p)$ should always store the one with the earliest deadline, the equation

$$\mathit{Dline}(\mathrm{MS}(p)) = \mathit{Deadline}(\mathit{CurMsg}(p))$$

has to hold.

Also, the message-passing procedure is kept abstract. An abstract action *Broadcast* m delivers messages m – say by setting $\mathit{InTransit}(m) := \mathit{true}$ – to the network, which acts as an implicit environment agent forwarding m to an agent executing MsgCarrier(p) which puts to the set *InBox*(p) every message *InMsg*(p) which arrives at p. From *InBox*(p) an agent executing Custodian(p) selects the message with the minimal timestamp (earliest deadline) and assigns it to $\mathrm{MS}(p)$ as *CurMsg*(p), from where an agent executing Scheduler(p) will assign it to *CurProc*(p) on an earliest-deadline-first basis applied to $\mathrm{MS}(p), \mathrm{BS}(p)$ (if enabled, see rule *Schedule*(p)). The restriction to *enabled* processes reflects the *scheduling uncertainty assumption* made for the system, meaning that a task with deadline d may not be scheduled until $d - d_u$, where d_u is a system-wide upper uncertainty bound. Therefore, processes on p are called enabled only when their deadline is $\geq \mathit{Clock}(p) - d_u$:

$$\begin{aligned}&\mathit{EnabledBS}(p) = (\mathit{DlineBS}(p) \geq \mathit{Clock}(p) - d_u)\\ &\mathit{EnabledMS}(p) =\\ &\quad \mathit{CurMsg}(p) \neq \mathit{undef} \ \textbf{and}\ \mathit{DlineMS}(p) \geq \mathit{Clock}(p) - d_u\end{aligned}$$

Two assumptions are made for the message-passing system. By the *reliability assumption* every message which has been broadcast by a processor will be delivered to every processor. The *delivery-bound* assumption relates different processor clocks via a message-delivery (carry-time) bound by stipulating that a broadcast by p at *Clock*(p)-time t_1 will be received by any processor q

at *Clock*(q)-time t_2 within a delay d_c, i.e. such that $0 < t_2 - t_1 \leq d_c$ holds (see below for a formulation in terms of GroupMember runs).

Before defining the async GroupMember ASM and its runs we state the correctness goal of the processor-group membership protocol.

Theorem 6.3.1 (Processor-group membership correctness). In runs of GroupMember the following properties hold.

Stability of local views: Once a processor joins a group, it stays there until either a processor fails or one recovers and attempts to rejoin.

Agreement on history: If two processors are joined to a common group g and none of them crashes between joining g and joining the next group, then that next group is the same for both processors.

Agreement on group membership: Alive processors in a group have the same membership view.

Reflexivity: The membership view of an alive processor in a group contains that processor.

Bounded join delay: There is a constant *join delay* d_j such that if a processor becomes sober at time t then, by time $t + d_j$, it will join a new group along with every other processor that stays correct from t to $t + d_j$.[25]

Bounded failure detection delay: There is a constant *failure delay* d_f such that if a processor belonging to a group g fails at time t then, by time $t + d_f$, all the members of g that stay correct in $[t, t + d_f]$ will join a new group not containing p.

Agents, rules and runs. From the architectural point of view it is natural to analyze the asynchronous membership-protocol interactions of processor group members at the level of the instances of the protocol code for every processor. This means to group the GroupMember rules related to one processor p into one basic ASM GroupMember(p), teaming together the agents which execute the machines Scheduler(p), MS(p), BS(p), MsgCarrier(p), Custodian(p) defined below.

$$\begin{aligned}&\textsc{GroupMember}(p) = \\ &\quad \{\textsc{Scheduler}(p), \mathrm{MS}(p), \mathrm{BS}(p), \textsc{MsgCarrier}(p), \textsc{Custodian}(p)\}\end{aligned}$$

The one-agent team resolves some possible update conflicts at the local processor level, namely for *CurMsg*(p) by MS(p) and Custodian(p) and for *InBox*(p) by MsgCarrier(p) and Custodian(p). Note that the guards of Scheduler(p), MS(p) and BS(p) as well as the guards of MsgCarrier(p) and Custodian(p) are pairwise disjoint. The machine GroupMember is defined as an async ASM which has one agent per processor $p \in$ *PROCESSOR*,

[25] Correctness for a processor means being without a pending task whose deadline has been exceeded. See below for a precise formulation in terms of GroupMember runs.

each executing the basic ASM GroupMember(p) whose monitored functions are its instances of *Clock* and *InMsg*, all other dynamic functions being controlled or derived.

For the partially ordered runs of GroupMember in addition to the assumptions listed above two further run constraints are made. First, a *recovery bound* requires that there is a minimum delay d_r on recovery time, i.e. the time elapsing between a processor crash and the next time it becomes sober again (see below for a more precise formulation). Second, the system constants are supposed to satisfy $d_h > d_u$ (the heartbeat leaves enough time for scheduling), $d_n > d_c + d_u$ (the deadline of new-group messages leaves enough time for the message to arrive at every destination and then to be scheduled), and $d_r > d_h + d_u$ (the recover time leaves enough time for at least one heartbeat to be broadcast and scheduled). For the initialization it is assumed that at every processor $Status = crashed$, $InBox(p) = \emptyset$, all of *CurMsg*, *Group*, *Members*, *StartUpTime*, *BCastTime* are undefined.

The Scheduler(p) rules assign values to $CurProc(p)$. $Recover(p)$ reflects that upon recovery of p, it is MS(p) which has to initialize p. $Schedule(p)$ schedules the protocol processes MS(p), BS(p) in a non-pre-emptive way on an earliest-deadline-first basis. Non-pre-emptiveness means that only processors can be interrupted, whereas tasks run until their completion; thus $Schedule(p)$ is executable only when p is sober and upon completing running its current task has set $CurProc(p)$ to undef. The earliest-deadline-first principle applies only to enabled processes – as defined above, those whose deadline is not yet missed, taking into account the possible delay due to the scheduling uncertainty. It has to be guaranteed also that scheduling takes place only when either an *apt* process has been assigned by Custodian(p) to $CurMsg(p)$ – i.e. a process with minimal deadline and timestamp issued not before the recovery of p from its last crash, called $StartUpTime(p)$ – or there is no such *AptMsg*, where

$$AptMsg(p) = \{m \in InBox(p) \mid Deadline(m) \geq Clock(p) - d_u \textbf{ and } Timestamp(m) \geq StartUpTime(p)\}$$

```
SCHEDULER(p) = {Recover(p), Schedule(p)} where
Recover(p) = if Status(p) = crashed then
    Status(p) := recovered
    CurProc(p) := MS(p)
Schedule(p) =
    if CurProc(p) = undef and Status(p) = sober
        and (CurMsg(p) ≠ undef or AptMsg(p) = ∅)
    then CurProc(p) := MinEnabled(p)
MinEnabled(p) =
    x if (Enabledx and not Enabledy)
        or (Enabledx and Enabledy and Dline(x) < Dline(y))
    y if (Enabledx and Enabledy and Dline(x) ≥ Dline(y))
        where (x, y) ∈ {(MS(p), BS(p)), (BS(p), MS(p))}
```

The membership server MS(p) has four rules. When MS(p) is scheduled to *Initialize* the recovered processor p, it cancels the information about the group p may have belonged to previously and broadcasts the information about the readiness of p to form a new group (with timestamp) $\mathit{Clock}(p) + d_n$, adding to the current processor time the delay $d_n > d_c + d_u$ to guarantee that all processors have a chance to receive that message within the maximum delivery time d_c, considering also the possible delay d_u due to uncertain scheduling.

$$\mathrm{MS}(p) = \{\mathit{Initialize}(p), \mathit{NewGroup}(p), \mathit{ChangeGroup}(p), \mathit{MS}(p)\mathit{Crash}\}$$

```
$\mathit{Initialize}(p) =$
  if $\mathit{Status}(p) = \mathit{recovered}$ and $\mathit{CurProc}(p) = \mathrm{MS}(p)$ then
    $\mathit{Reset}(p)$
    $\mathit{ResetGroupInfo}(p)$
    $\mathit{StartUpTime}(p) := \mathit{Clock}(p) + d_n$
    $\mathit{BroadcastNewGroup}(p)$
    $\mathit{Status}(p) := \mathit{sober}$
    $\mathit{CurProc\ done}$
where
$\mathit{Reset}(p) = \{\mathit{BCastTime}(p) := \mathit{undef}, \mathit{CurMsg}(p) := \mathit{undef}\}$
$\mathit{ResetGroupInfo}(p) = \{\mathit{Group}(p) := \mathit{undef}, \mathit{Members}(p) := \mathit{undef}\}$
$\mathit{BroadcastNewGroup}(p) = \mathit{Broadcast}(\mathit{newGp}, \mathit{Clock}(p) + d_n, \{p\})$
$\mathit{CurProc\ done} = (\mathit{CurProc}(p) := \mathit{undef})$
```

When MS(p) is scheduled to timely handle a new-group message, it reports p as present in the new group and synchronizes the heartbeat of p to that of the new group, canceling any pending heartbeat.

```
$\mathit{NewGroup}(p) =$
  if $\mathit{Status}(p) = \mathit{sober}$ and $\mathit{CurProc}(p) = \mathrm{MS}(p)$ and
    $\mathit{Type}(\mathit{CurMsg}(p)) = \mathit{newGp}$ and
    $\mathit{Clock}(p) \leq \mathit{Timestamp}(\mathit{CurMsg}(p))$ then
      $\mathit{BroadcastRegistration}(p)$
      $\mathit{SynchronizeWithNewGroup}(p)$
      $\mathit{CurMsg\ done}$
      $\mathit{CurProc\ done}$
  where
  $\mathit{BroadcastRegistration}(p) =$
    $\mathit{Broadcast}(\mathit{present}, \mathit{Timestamp}(\mathit{CurMsg}(p)), \{p\})$
  $\mathit{SynchronizeWithNewGroup}(p) =$
    $\mathit{BCastTime}(p) := \mathit{Timestamp}(\mathit{CurMsg}(p)) + d_h$
  $\mathit{CurMsg\ done} = (\mathit{CurMsg}(p) := \mathit{undef})$
```

When MS(p) is scheduled to handle a presence message, it checks whether p's group has changed – which is the case if the *Members*(p) do not correspond any more to the processors which are viewed as present in the group

Timestamp(*CurMsg*(p)). In this case the group is updated, including the reported membership information.

```
$ChangeGroup(p) =$ if $Status(p) = sober$ and $CurProc(p) = \mathrm{MS}(p)$
  and $Type(CurMsg(p)) = present$ then
    $CurMsg\ done$
    $CurProc\ done$
    if $Members(p) \neq View(CurMsg(p))$ then
      $UpdateGroupMembership(p)$
  where $UpdateGroupMembership(p) =$
    $Members(p) := View(CurMsg(p))$
    $Group(p) := Timestamp(CurMsg(p))$
```

When MS(p) is scheduled to handle the deadline miss of a new-group message, it interrupts p.

```
$MS(p)Crash =$ if $Status(p) = sober$ and $CurProc(p) = MS(p)$ and
  $Type(CurMsg(p)) = newGp$ then
    if $Clock(p) > Timestamp(CurMsg(p))$ then $Status(p) := crashed$
```

The broadcast server BS(p) has two rules. When it is scheduled and detects the deadline miss of a broadcast, it interrupts p. When the broadcast is within the deadline, p reports its presence in its group *BCastTime*(p) and updates its next heartbeat deadline.

```
$\mathrm{BS}(p) = \{BS(p)Crash, Presence(p)\}$ where
$BS(p)Crash =$ if $Status(p) = sober$ and $CurProc(p) = BS(p)$ then
  if $Clock(p) > BCastTime(p)$ then $Status(p) := crashed$
$Presence(p) =$ if $Status(p) = sober$ and $CurProc(p) = \mathrm{BS}(p)$ then
  if $Clock(p) \leq BCastTime(p)$ then
    $BroadcastPresence(p)$
    $BCastTime(p) := BCastTime(p) + d_h$
    $CurProc\ done$ where
  $BroadcastPresence(p) = Broadcast(present, BCastTime(p), p)$
```

The MsgCarrier(p) transfers every incoming message to *InBox*, bundling them into one per group (i.e. with the same timestamp).

```
MsgCarrier($p$) = if $InMsg(p) \neq undef$ then let $(a, b, c) = InMsg(p)$
  if $a = newGp$ or $InBox(p)$ has no message with $Timestamp\ b$ then
    $InBox(p) := InBox(p) \cup \{InMsg(p)\}$
  else let $m = \iota x(x \in InBox(p)$ and $Timestamp(x) = b)$
    $InBox(p) := (InBox(p) \setminus \{m\}) \cup \{(a, b, View(m) \cup \{c\})\}$
```

The Custodian(p) assigns the next message with minimal deadline to *CurMsg*(p), realizing the requirement that the membership servers see messages in their deadline order.

```
CUSTODIAN$(p)$ = if $Status(p) = sober$ and $CurMsg(p) = undef$ and
  $AptMsg(p)$ contains a message with minimal deadline then
    deliver $m$ from $InBox(p)$ to $MS(p)$
    where
      $m = \iota x(x$ has minimal deadline in $AptMsg(p))$
      deliver $m$ from $InBox(p)$ to $MS(p)$ =
        $InBox(p) := InBox(p) \setminus \{m\}$
        $CurMsg(p) := m$
```

We conclude this section by illustrating how to make intuitive protocol-related notions mathematically precise in terms of GROUPMEMBER runs, turning the processor-group membership correctness statement above into the genuinely mathematical Theorem 6.3.1, proved in [256]. Let $\Sigma(p)$ be the signature of GROUPMEMBER(p) without *InTransit*; $\Sigma^-(p)$ the signature $\Sigma(p)$ without the monitored functions $Clock(p)$, $InMsg(p)$; S^- the restriction of S to $\Sigma^-(p)$; S_n the nth state of a run and t_n the value of t in state S_n. The d_r-recovery bound run constraint then reads as follows: if p gets crashed in S_n and recovered in S_{n+k} for some $k > 0$, then $Clock(p)_{n+k} - Clock(p)_n \geq d_r$. The d_c-delivery bound reads: if p sends $msg = (type, time, view)$ to q (i.e. sets $InTransit(msg)$ to true) in S_n of p, then there is a unique state S_k of q such that $InMsg(q)_k = msg$, $InBox(q)_{k+1}$ contains some $(type, time, view')$ with $view' \supseteq view$, and $0 < Clock(q)_k - Clock(p)_n \leq d_c$. The correctness of p in state S_n means that $Clock(p)_n \geq StartUpTime(p)_n$ and $BCastTime(p)n \geq Clock(p)_n$ and that for all $m \in InBox(q)_n$ with $Timestamp(m) \geq StartUpTime(p)_n$ the condition $Deadline(m) \geq Clock(p)_n$ holds. Correctness of p in a real-time interval $[t, t']$ then means that p is correct in every state S_n with $k \leq n \leq k'$, where $Clock(p)_k \leq t < Clock(p)_{k+1}$ and $Clock(p)'_k - 1 < t' \leq Clock(p)'_k$.

6.3.3 Exercises

Exercise 6.3.1. (⇝ CD) Define the refinement relation between runs of ALTERNATINGBIT and SLIDINGWINDOW runs.

Exercise 6.3.2. Optimize $SlideWindow(sender)$ in rule SLIDINGWINDOW by sliding the window to the nearest position where *receivedAck* is false. Justify the correctness of the optimization.

Exercise 6.3.3. (⇝ CD) Refine SLIDINGWINDOW by a machine with bound $2 \times winsize$ for message numbers. Show by an example that $2 \times winsize$ message numbers are necessary.

Exercise 6.3.4. (⇝ CD) Phrase the correctness and completeness proofs in Sect. 6.3.1 in terms of the partial ordering of moves only, without labeling moves by real-time moments.

Exercise 6.3.5. ($\rightsquigarrow$ CD) Adapt the sliding-window correctness proof for an asynchronous ASM where the sender or receiver rules which could fire independently are not teamed.

Exercise 6.3.6. ($\rightsquigarrow$ CD) Show that in every reachable state of runs of the machine SlidingWindow the following window properties hold:

- Windows never exceed *winsize*: $high - low + 1 \leq winsize$ always holds at both sender and receiver.
- The receiver window is never ahead of the sender window: $bound(receiver) \leq bound(sender)$ for $bound = low, high$.
- Once the receiver window gets full, it remains full.
- For files with $fileNum < low(sender)$, the sender has received an acknowledgment: $0 \leq i < low(sender)$ implies $receivedAck(i)$.

Exercise 6.3.7. ($\rightsquigarrow$ CD) Define the partial-order relation for moves of agents in GroupMember runs.

Exercise 6.3.8. ($\rightsquigarrow$ CD) Prove the finite history property for runs of GroupMember.

Exercise 6.3.9. Prove the coherence property for GroupMember runs.

Exercise 6.3.10. Modify the rules of GroupMember(p) to include the bound d_r on recovery time and the bound d_c on message delivery time, so that the two constraints on these bounds can be deleted from the definition of GroupMember runs and be proved instead to always hold.

Exercise 6.3.11. ($\rightsquigarrow$ CD) Modify *Broadcast* m in GroupMember(p) to keep the size of *InTransit* bounded.

Exercise 6.3.12. ($\rightsquigarrow$ CD) Modify GroupMember(p) to keep the number of messages in $InBox(p)$ bounded, deleting unselectable messages, e.g. those arriving while p is crashed so that they may never be seen by $MS(p)$ and never be removed from $InBox(p)$.

6.4 Async ASMs with Durative Actions

In this section we elaborate the async ASM model developed in [114] for the mutual exclusion algorithm *Bakery* from [314, 315], to illustrate the ASM refinement method for turning in a provably correct way time-constrained atomic actions (read: execution of ASM rules) to machine executions taking time (real-time "durative actions"). We use a double refinement. First we prove the protocol correctness on the basis of atomic actions – atomic non-overlapping reads and writes to shared registers – for the concrete protocol BakeryGround by

(a) abstracting it into a high-level model BakeryHigh for which the correctness can be easily derived from natural axioms,
(b) proving the ground model BakeryGround to satisfy the axioms (i.e. to be a correct refinement of the abstract model).[26]

Then we refine atomic to durative actions – possibly overlapping reads and writes which take time – and show this to preserve the refinement relation between the abstract BakeryHigh and the concrete BakeryGround model together with the 2-step protocol correctness proof. The refinement of atomic to durative actions turns out to be essentially a refinement of the global to a local, agent-based, state view, which can be captured by a detailed analysis of which locations are monitored or controlled for which agent and what are the causal dependencies and the time relations among read/write accesses to locations shared by different agents.

Problem 21 (Framework for deployment structures). Develop ASM models for accessing resource structures which can be put to use to describe deployment structures.

6.4.1 Protocol Verification using Atomic Actions

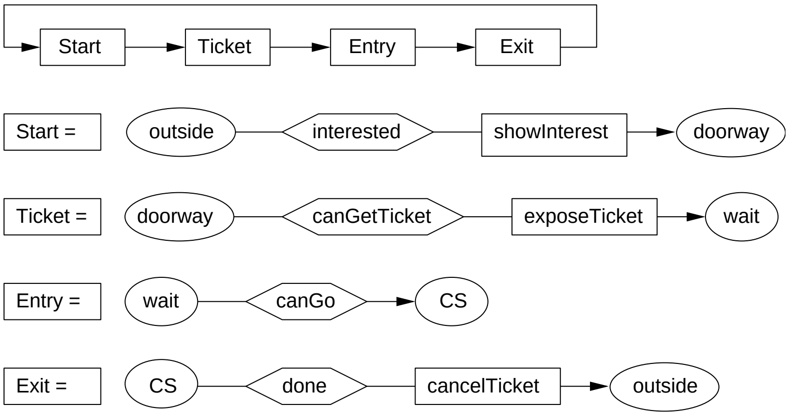
A mutual exclusion protocol is required to guarantee three properties when applied each time a process – among any finite number of processes – wants to enter a "critical code section":

Mutual exclusion: it never happens that two processes are simultaneously in the critical section.
Deadlock freedom: every process which attempts to enter the critical section will eventually enter it.
Fairness: the first-come-first-served principle holds (for a reasonable notion of coming-first, which has to be defined).

Bakery ground model ASM. In the Bakery protocol tickets are assigned to interested customer processes P_i ($1 \leq i \leq n$) and compared to let the one with the smallest ticket access the critical section *CS*. The customers *Start* to *showInterest* by entering a *doorway* (mode) and initializing their ticket. In the doorway, once they *canGetTicket* – namely the next available new ticket, say larger than the maximum ticket encountered in a *doorwayRead* (reading in *doorway* mode) of all issued tickets – they *exposeTicket*. Then they *wait*, each for its turn to access *CS*, performing a repeated *waitReadCheck* (reading and checking in *wait* mode) of all tickets – until one process *canGo* into *CS* when there is no other process with a smaller ticket. The *Entry* into *CS* is eventually followed by an *Exit* to *cancelTicket* and move *outside*, namely

[26] The reason for presenting the more detailed model first is not only historical, but also that the abstraction reflects Lamport's concrete algorithm and thus is easier to capture if the low-level algorithm has been understood.

Fig. 6.14 Control state ASM BAKERYCUSTOMERSCHEME

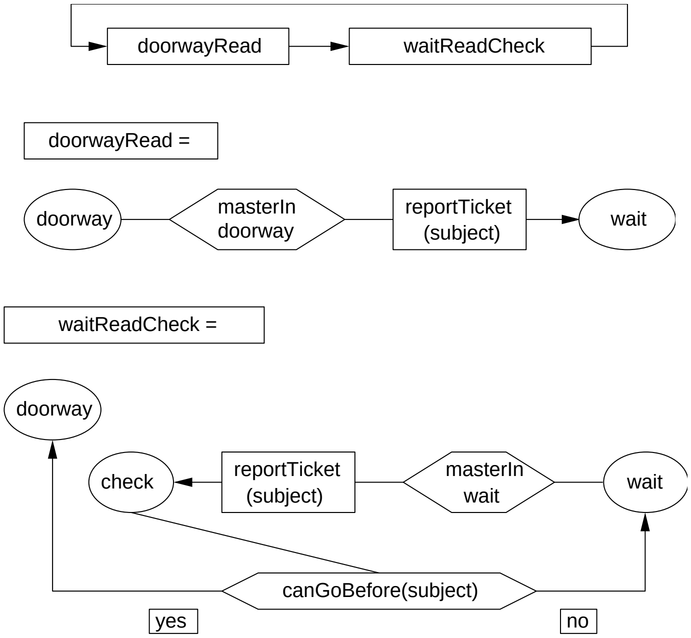


when the *CS* code is *done*. This sequence of four basic protocol moves of each customer is formalized by the control state ASM BAKERYCUSTOMERSCHEME in Fig. 6.14.

In Lamport's algorithm the ticket of P_i is written (exposed for reading by the other processes) only by P_i, namely in a register R_i. These registers are however shared for reading, to copy them into a private array P, recording for the array owner P_i what it has last received as the value read in the register of other customers. Reading and checking R_j and recording their values in $P(P_i, P_j)$ during the doorway or the wait phase of P_i are done independently for each i, j, namely by independent reader processes.

Consequently, the signature of the async ground model ASM has two finite (supposed to be static) sets, *CUSTOMER* and *READER*, of agents equipped with their instance of the following functions. For customer **self** a function $mode\colon \{doorway, wait, CS, outside\}$ indicates the control states of BAKERYCUSTOMERSCHEME in which the four basic protocol moves are executed. The monitored functions *interested* and *done* represent the events which trigger the *Start* or the *Exit* of a protocol execution. A register $R\colon \mathbb{N}$ is used to post tickets to the other customers; $R(\textbf{self})$ is controlled (for private writing), $R(Y)$ for $Y \neq$ **self** is monitored (for public reading). The private copies of other customers' tickets are held in a function $P\colon CUSTOMER \to \mathbb{N}$ with $P(\textbf{self})$ controlled and $P(Y)$ for $Y \neq$ **self** monitored for **self**. The above-mentioned auxiliary reader processes are captured by a static injective function $\textbf{self}.reader\colon CUSTOMER \setminus \{\textbf{self}\} \to READER$ yielding for each other customer the reader of **self** which reports the ticket value from the register of that customer. We also write $reader_X(Y)$ or $reader(X, Y)$ for $X.reader(Y)$.

Fig. 6.15 Control state ASM BakeryReader



Each reader **self**, one per ordered pair of customers to guarantee independent readings, serves its **self**.*master*: *CUSTOMER* concerning the ticket value of its **self**.*subject*: *CUSTOMER* when $master.mode \in \{doorway, wait\}$. In the macro *doorwayRead*, when the reader is in doorway *mode*, synchronized with its *masterIndoorway* mode, it reports the ticket value from the register of its subject customer and switches to **self**.$mode = wait$. When again mode-synchronized with *masterInwait*, it restarts to $reportTicket(subject)$, but now also *check*s whether the master $canGoBefore(subject)$ and in the latter case switches back to doorway mode. This is formalized by the basic ASM BakeryReader in Fig. 6.15.

The independence of readers implies that the *reader* function can be characterized as follows:

$$reader(c)(y) = \iota r(r \in READER \textbf{ and } r.master = c \textbf{ and } r.subject = y).$$

For reader **self** the locations $master.P(master)$, $master.mode$, $R(subject)$ are monitored, and $master.P(subject)$ with $subject \neq master$ is controlled. The locations $reader(c).mode$ are monitored for every customer c for the inspection of wait/doorway ticket values.

The ground model ASM BakeryGround for Lamport's algorithm consists of the BakeryCustomerScheme and the BakeryReader rules for each customer and each reader with the macros instantiated as follows.

Mod in the definition of the macro *masterInMod* is any *mode* value. The *canGoBefore*(c) relation reflects that only those customers are considered which currently have an interest to access *CS* and that for customers which happen to have got the same ticket a static decision is taken upon who will be served first (here that *master* comes before (is < than) the *subject*).

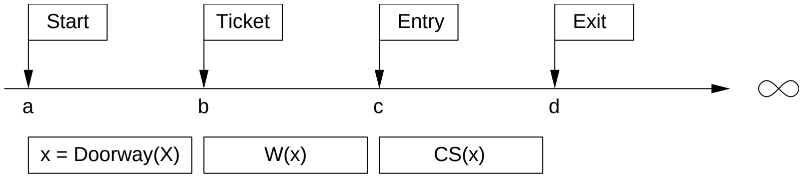
$$\begin{aligned}
&\mathit{showInterest} = \{R := 1, P(\mathbf{self}) := 1\}\\
&\mathit{exposeTicket} = \{R := \mathit{nextTicket}, P(\mathbf{self}) := \mathit{nextTicket}\}\\
&\mathit{nextTicket} = 1 + \mathit{max}\{P(X) \mid X \in \mathit{CUSTOMER}\}\\
&\mathit{cancelTicket} = (R := 0)\\
&\mathit{canGetTicket} =\\
&\quad \forall y \in \mathit{CUSTOMER} \setminus \{\mathbf{self}\}\, (\mathit{mode}(\mathbf{self}.\mathit{reader}(y)) = \mathit{wait})\\
&\mathit{canGo} = \forall y \in \mathit{CUSTOMER} \setminus \{\mathbf{self}\}\, (\mathit{mode}(\mathbf{self}.\mathit{reader}(y)) = \mathit{doorway})
\end{aligned}$$

$$\begin{aligned}
&\mathit{masterInMod} = (\mathit{master}.\mathit{mode} = \mathit{Mod})\\
&\mathit{reportTicket}(\mathit{subject}) = (\mathit{master}.P(\mathit{subject}) := R(\mathit{subject}))\\
&\mathit{canGoBefore}(\mathit{subject}) = \mathit{notInterested}(\mathit{subject}) \text{ }\mathbf{or}\text{ } \mathit{comesLater}(\mathit{subject})\\
&\mathit{notInterested}(\mathit{subject}) = (\mathit{master}.P(\mathit{subject}) = 0)\\
&\mathit{comesLater}(\mathit{subject}) = \mathit{master}.P(\mathit{master}) < \mathit{master}.P(\mathit{subject}) \text{ }\mathbf{or}\\
&\quad \mathit{master}.P(\mathit{master}) = \mathit{master}.P(\mathit{subject}) \text{ }\mathbf{and}\text{ } \mathit{master} < \mathit{subject}
\end{aligned}$$

Concerning the runs of BakeryGround we simplify the exposition by projecting the moves of customers and readers – atomic reads and writes to shared locations – to real-time moments ≥ 0. Since atomic moves take place in zero time, the state change due to a move at time t is effective at time $t+\epsilon$ for sufficiently small ϵ (for any positive ϵ such that $t+\epsilon$ is smaller than the time moment of the next move). By the sequentiality of agents two moves of one agent take place at different moments. Let therefore S_t denote the state at time t, resulting from all (finitely many) moves taking place before t. We make the diligence assumption that no agent can become continuously enabled without eventually making a move and that *done* will eventually become true each time the corresponding customer is in *CS*, whereas *interested* may eventually stay false forever. Initially for each customer c we assume that $\mathit{mode} = \mathit{outside}$ and $R = P(c) = 0$ hold, for each reader $\mathit{mode} = \mathit{doorway}$.

The mapping of BakeryGround moves to real-time moments yields for each customer protocol execution the real-time intervals between moves which are illustrated in Fig. 6.16 and have to be analyzed to prove the mutual exclusion properties. If customer X executes *Start* and then *Ticket* at moments a, b, the open interval $x = (a, b)$ is called a *doorway* of X; within this interval the readers of X do *doorwayRead*. By the assumption that no agent stalls forever, for every *Start* move at a there is a bound b defining the doorway. If X after *Ticket* executes *Entry* and *Exit* at moments c, d, $W(x) = (b, c)$ is called the *wait* section of the doorway x; during this interval the readers of X are busy with *waitReadCheck*. $CS(x) = (c, d)$ is called the *critical section* interval of x. If after b customer X makes no more move,

Fig. 6.16 Real-time intervals between atomic customer moves



we set $W(x) = (b, \infty)$, $CS(x) = undef$. It remains to prove that for each doorway x, $W(x)$ is bounded and $CS(x)$ is defined.

High-level bakery model. Since the real-time ordering of moves by the moments when they take place makes certain moves comparable which in Lamport's original formulation are intended to be independent, e.g. moves of the readers belonging to one master, we define a high-level model which abstracts completely from readers and turns out to be more appropriate for an analysis of independent reader and customer moves.

The async ASM BakeryHigh is defined as the instantiation of the BakeryCustomerScheme by the following macros, using a monitored ticket function T and a monitored *canGo* predicate *Go* constrained below by four natural axioms C0–C3 which suffice to establish the mutual exclusion properties. From the high-level proof we can then infer these properties for the ground model by showing that BakeryGround is a correct refinement of BakeryHigh since the axioms are easily shown to be true for BakeryGround runs.

$$
\begin{aligned}
&showInterest = (R := 1)\\
&exposeTicket = (R := nextTicket)\\
&nextTicket = T(\mathbf{self})\\
&cancelTicket = (R := 1)\\
&canGetTicket = true\\
&canGo = Go(\mathbf{self})
\end{aligned}
$$

We introduce an order for doorways in terms of which we formulate constraints on a) assigning $T(x)$ to customers X who leave a doorway x to enter $W(x)$[27] and b) on permitting customers to *Go* from a wait interval to *CS*, in such a way that the resulting BakeryHigh runs satisfy the three desired properties: mutual exclusion in accessing *CS*, fairness, deadlock freedom. To avoid repetitive case distinctions between processes which are or are not interested, we use the ticket value ∞ for $T-$values and R-values of customers who are not interested. Formally for a function $f \colon CUSTOMER \to \mathbb{N}$ we define $f'(X)$ as follows (and use it analogously also for values $f(x)$ on doorways of X):

$$f'(X) = \begin{cases} n \times f(X) + id(X), & \text{if } f(X) > 0;\\ \infty, & \text{else.} \end{cases}$$

Assuming $0 \leq id(X) < n$ this definition guarantees that $T'(x) \neq T'(y)$ if $X \neq Y$. We now order doorways x, y of customers $X \neq Y$ by the order of the issued tickets in case the doorways overlap, otherwise by the natural $<$-relation between real intervals:

$$x \lhd y = (x \cap y \neq \emptyset \ \textbf{and}\ T'(x) < T'(y))$$
$$x \prec y = (x < y \ \textbf{or}\ x \lhd y) \ \textbf{where}\ x < y = \forall a \in x \forall b \in y\ a < b$$

Lemma 6.4.1. For doorways x, y of different agents the condition $x \prec y$ **or** $y \prec x$ holds.

In the BakeryHigh run C0 guarantees that issued tickets are natural numbers > 1. C1 stipulates that if a doorway of Y comes before a doorway of X, then either Y has left the corresponding *CS* before X gets the ticket or the tickets of Y and X respect the temporal precedence of doorways with overlapping wait intervals. C1 requires that X can leave its wait interval only if during *waitReadCheck*, for every other customer the ticket of X at some moment turned out to be smaller than the ticket posted in the register of that other customer. C3 expresses an induction principle for wait intervals.

Definition 6.4.1. The BakeryHigh run constraints C0–C3 on the monitored functions T and *Go* are defined as follows for any doorways x, y:

C0: $T(x) \in \mathbb{N}$, $T(x) > 1$.
C1: If $y < x$, then either $CS(y) < sup(x)$ or $T'(y) < T'(x)$.
C2: If $Go(X)$ holds at any moment $t > sup(x)$, then for every $Y \neq X$ there is a moment $b \in W(x)$ such that $T'(x) < R'_b(Y)$.
C3: If $W(y)$ is bounded for all $y \prec x$, then $W(x)$ is bounded.

Refinement correctness. We prove that the abstraction of BakeryHigh from BakeryGround is correct, which means to prove that via the refinement relation, C0–C3 hold in BakeryGround runs. The correspondence of locations is given by the homonyms in the signatures. The corresponding states of interest are those in which the two instances of the same customer move in BakeryCustomerScheme are executed. The correspondence of computation segments is defined by the following mapping of move sequences. The definition provides an example for a refinement which uses type $(1,1)$, $(1,2)$, $(1, n+1)$ (where n is the number of processes whose tickets have to be inspected), and type $(1,*)$ where no concrete bound can be given on the number of concrete steps simulating an abstract step unless some information is made available on the execution time of agents.

- *Start* of X BakeryGround followed by one *doorwayRead* move of each reader of X – i.e. until $canGetTicket(X)$ becomes true – is mapped to *Start* of X in BakeryHigh.

[27] Thus the doorway determines the entry order into *CS*, a feature called Bakery Definability Property in [3, 2].

- *Ticket* moves are mapped to each other.
- *waitReadCheck* moves of every reader of $X = \mathit{master}$ – i.e. until $\mathit{canGo}(X)$ becomes true – followed by *Entry* of X in BakeryGround is mapped to *Entry* of X in BakeryHigh.
- *Exit* moves are mapped to each other.

Lemma 6.4.2 (Bakery refinement correctness). Constraints C0–C3 are satisfied by BakeryGround runs.

Proof. For C0. When $\mathit{nextTicket}(X)$ is taken at $\mathit{sup}(x)$, X contributes to the maximum with value $X.P(X) = 1$, which has been set by the *Start* move, so that $\mathit{nextTicket}(X) > 1$.

For C1. Let t be the time of the $\mathit{reader}(X, Y)$-move during x. Case 1. Y applies *Exit* from $\mathit{CS}(y)$ before this reading time t, i.e. during the interval $(\mathit{sup}(y), t)$. Then $\mathit{CS}(y) < t < \mathit{sup}(x)$. Case 2. Otherwise. Then $R_t(y) = T(y)$ and therefore $T(x) \geq 1 + R_t(y) > T(y) > 0$. Hence $T'(x) > T'(y)$.

For C2. $\mathit{canGo}(X)$ becomes true at $t > \mathit{sup}(x)$ when for every $Y \neq X$, $X.\mathit{reader}(Y)$ has finished its *waitReadCheck* moves in $W(x)$. Thus for each $Y \neq X$, the moment of the last *waitRead* move of $X.\mathit{reader}(Y)$ is a moment $b \in W(x)$ such that $T'(x) < R'_b(Y)$, as established by the following *waitCheck* move of $X.\mathit{reader}(Y)$.

ad C3. Indirect proof. Assume $W(y)$ is bounded for all $y \prec x$ but $W(x)$ is unbounded. Then we can derive the following two claims which provide a contradiction.

Claim 1: There is some $b \in W(x)$ which is later than any $\mathit{sup}(\mathit{CS}(y))$ (if $y \prec x$) and later than any $\mathit{sup}(y)$ (if $x \prec y$).

Claim 2: Claim 1 implies that every $X.\mathit{reader}(Y)$ finishes his *waitReadCheck* moves in $W(x)$, so that $W(x)$ is bounded.

For Claim 1. By the co-finiteness property of runs there are only finitely many $y \prec x$, all with $\mathit{sup}(\mathit{CS}(y)) < \infty$ (since $W(y)$ is bounded and by assumption no process stalls forever). Also for each Y there is at most one $y \succ x$, since $y \succ x$ implies $T'(x) < T'(y)$ (in case of $y > x$ by C1 and the unboundedness of $W(x)$). This implies the claim by the hypothesis that $W(x)$ is unbounded.

For Claim 2. If $X.\mathit{reader}(Y)$ finishes its *waitReadCheck* moves before $b \in W(x)$, it finishes them in $W(x)$. In fact, otherwise by the definition of b in Claim 1, no $Y \neq X$ has mode doorway at $t \geq b$. Therefore at $t \geq b$, $X.\mathit{reader}(Y)$ *waitReads* 0 or $T(y)$ for some $y \succ x$. In both cases the next *waitChec*g succeeds (in the second case by $T'(x) < T'(y)$, which is true by C1 if $x < y$, and true by definition of $\lhd$ if $x \lhd y$). □

High-level protocol verification. We derive here from the axioms C0–C3 for BakeryHigh runs the correctness of the protocol, namely that doorways are linearly ordered by $\prec$, that all waiting sections are finite and that the first-come-first-served property holds: $y \prec x$ implies $\mathit{CS}(y) < \mathit{CS}(x)$.

Lemma 6.4.3 (FCFS schedule and mutual exclusion). If $y \prec x$ and $W(x)$ is bounded, then $W(y)$ is bounded and $CS(y) < CS(x)$.

Proof. Indirect proof. Since $W(x)$ is bounded, by C2 there is for Y some $b \in W(x)$ such that $T'(x) < R'_b(Y)$. This implies $sup(y) < b$ (Claim 1) and under the hypothesis **not** $CS(y) < CS(x)$ also $T'(y) < T'(x)$ (Claim 2); see below. Then for some ϵ, $R_{sup(y)+\epsilon}(Y) = T'(y) < T'(x) < R'_b(Y)$ implies that Y must be writing to its register $R(Y)$ sometime in $(sup(y), b)$. But the first register write after $sup(y)$ takes place in an *Exit* move. Therefore Y has left $CS(y)$ during $W(x)$ – implying $CS(y) < CS(x)$ – and $W(y)$ is bounded, contradicting the hypothesis.

For Claim 1. $y \prec x$ implies $inf(y) \leq sup(x) < b$. But $b \in (inf(y), sup(y)]$ would imply $R_b(Y) = 1$, contradicting $1 < T(x)$ and $T'(x) < R'_b(Y)$. Therefore $b > sup(y)$.

For Claim 2. This follows for $y \lhd x$ from the definition of $\lhd$, for $y < x$ from C1 and the hypothesis **not** $CS(y) < CS(x)$. □

Lemma 6.4.4 (Liveness). Every W(x) is bounded.

Proof. This follows from the induction principle C3 for the (by the cofiniteness property well-founded) ordering of doorways by $\prec$, using the transitivity lemma below. □

Lemma 6.4.5. $\prec$ is transitive.

Proof. Indirect proof. Assume $x \prec y \prec z \prec x$ and let n be the number of occurrences of $<$ in this chain. In the case $n = 0$ or $n = 2, 3$ a contradiction follows from the transitivity and non-reflexivity of the ordering $<$ of reals and of real intervals. Case $n = 1$: without loss of generality assume $x \prec y \prec z < x$, so that $T'(x) < T'(y) < T'(z)$. By the FCFS lemma, $x \prec y \prec z \prec x$ implies that $W(x)$, $W(y)$, $W(z)$ are infinite, so that $CS(z)$ is undefined. C1 then implies $T'(z) < T'(x)$, a contradiction. □

6.4.2 Refining Atomic to Durative Actions

In an async ASM run with durative actions every move μ of any agent takes place during a non-empty open real-time interval, say $Time(\mu) = (a, b)$ with $0 \leq a < b$, where b is a real number or ∞. Such moves can be ordered by an ordering of their (possibly overlapping) time intervals. To capture the duration of rule executions it suffices to refine the notion of state and of run, keeping the rules of BakeryGround and BakeryHigh, so that we will speak of durative runs of these async ASMs as opposed to their atomic runs above. The environment–controller separation principle that every durative run has to satisfy for the moves of any agent concerning monitored and controlled locations, establishing which moves take place without changes of which monitored locations, has also to guarantee well-defined values of monitored locations during the intervals in which moves of agents take place, besides assuring the coherence condition of partially ordered runs.

To formulate the environment–controller separation principle for the Bakery protocol denote by $S_t(A)$ the local state of agent A at time t, i.e. the state S_t at time t restricted to the locations of A, more precisely to (loc, val) pairs with locations $loc \in Loc(T)^{S_t}$, where T is the set of terms in the signature of A. The following separation constraints which are additionally imposed on *runs with durative actions* (besides the above diligence constraint, the assumption on *interested* and *done* and the initialization) guarantee the coherence condition for customer and reader moves.

– No customer move and no reader *waitCheck* move does overlap with any monitored move for any of its monitored locations. This condition assures $S_b(A)$ to be the result of applying μ to $S_a(A)$ for these moves of agents A with $Time(\mu) = (a, b)$.
– No read move does overlap with a move concerning the mode of its master, but it may overlap with a write move concerning the register of its subject.
– Register regularity: let $Time(\mu) = (a, b)$ for a read move μ of $r = X.reader(Y)$. The read result stored in $X.P(Y)$ in state $S_b(r)$ is the value of $R(Y)$ at any of
  – either Y's last passive moment $\leq a$ before starting the read move
  – or a passive moment of Y in the read move interval (a, b).

Here A is called passive in an interval I if $I \cap Time(\mu) = \emptyset$ for every move μ of A. Denote by $S_c(A) \restriction Control(A)$ the restriction of $S_c(A)$ to the controlled locations of A.

Lemma 6.4.6 (State stability). If $[a, b]$ is a passive interval of an agent A in a durative BakeryGround run, then $S_b(A) \restriction control(A) = S_a(A) \restriction control(A)$.

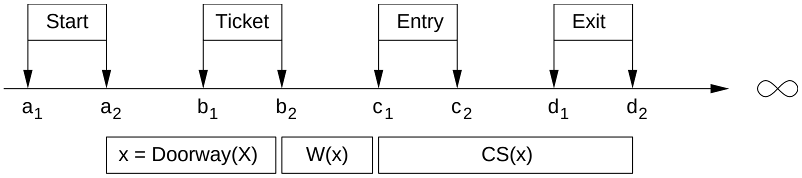
Proof. By definition only A can update its controlled locations though in the meantime the monitored locations may change their values. □

Lemma 6.4.7 (Sequentiality). For each agent A, the set of duration times of moves of A is linearly ordered by the $<$ relation on open real intervals. If A executes μ during (a, b) and then μ' during (c, d), the interval $[b, c]$ is passive for A.

Proof. By the sequentiality condition of partially ordered runs. □

The mapping of atomic moves to real-time moments in Fig. 6.16 is refined for durative moves by Fig. 6.17. If X executes *Start* during (a_1, a_2) and then *Ticket* during (b_1, b_2), the *doorway* of X starts when *Start* is finished and lasts until *Ticket* has finished, i.e. $x = (a_2, b_2)$. If X after *Ticket* executes *Entry* during (c_1, c_2) and *Exit* during (d_1, d_2), the *Wait* interval starts when *Ticket*

Fig. 6.17 Real-time intervals between durative customer moves



has finished and lasts until the beginning of *Entry*, i.e. $W(x) = (b_2, c_1)$. $CS(x) = (c_1, d_2)$ starts when *Entry* begins and lasts until *Exit* is finished. The BakeryHigh run constraints C0, C1, C3 remain in force for durative runs, but the constraint C2 on the monitored function *Go* is refined to regular register reading when checking the ticket values in a waiting interval against the values posted by competitors. With durative actions, X can leave its wait interval only if for every other customer Y its ticket at some passive moment for Y, satisfying the register regularity principle, has a smaller value than the ticket posted in the register of Y:

Definition 6.4.2. C2: If $Go(X)$ holds at any moment $t > sup(x)$, then for every $Y \neq X$ there is a passive moment b for Y such that $T'(x) < R'_b(Y)$ and either $b \in W(x)$ or b is Y's last passive moment $\leq inf(W(x))$.

Refinement correctness for durative runs. We have to show that C0–C3 hold for durative BakeryGround runs. The proofs for C0, C3 carry over from atomic runs without change. For C2 $Go(X)$ becomes true when every $X.reader(Y)$ finishes its *waitReadCheck* moves. For $Y \neq X$ consider the last *waitRead* move μ of $X.reader(Y)$ during $W(x)$. As the desired b take the moment chosen via the register regularity assumption for μ.

For C1 let μ be the *doorwayRead* move of $X.reader(Y)$ during x. Case 1. Y makes an *Exit* move ν from $CS(y)$ ending not later than μ, i.e. such that $supTime(\nu) \leq supTime(\mu)$. Then $CS(y) < supTime(\nu) \leq supTime(\mu) \leq sup(x)$. Case 2. Otherwise. Let $Time(\mu) = (a, b)$. If Y makes an *Exit* move ν from $CS(y)$, set $e = supTime(\nu) > supTime(\mu) = b$, otherwise set $e = \infty > b$. Then $R(Y) = T(y)$ holds over $[sup(y), e) \supseteq [sup(y), b)$, in particular $R_t(Y) = T(y)$ for the moment chosen via the register regularity principle for μ. (NB. Y is passive at $sup(y)$). Therefore $T(x) \geq 1 + R_t(Y) > T(y) > 0$. Hence $T'(x) > T'(y)$.

High-level protocol verification for durative runs. Only the proof of Lemma 6.4.3 has to be paraphrased, using the refined version of C2. In fact b is chosen *passive for* Y; thus the write move of Y to $R(Y)$ sometime in $(sup(y), b)$ by the refined C2 starts before $sup W(x)$. The proof for Claim 1 here reads as follows. First of all $inf(y) \leq b$ holds, since, by the regularity of register reads, $b < inf(y)$ forces b to be the last passive moment $\leq inf(W(x))$, whereas $inf(y)$ is passive for Y and $inf(y) < sup(x) =$

$\mathit{inf}W(x)$). Secondly $b \in [\mathit{inf}(y), \mathit{sup}(y))$ would imply $R_b(Y) = 1$, contradicting $1 < T(x)$ and $T'(x) < R'_b(Y)$. Finally $b \neq \mathit{sup}(y)$, since otherwise $R_b(Y) = T(y)$ contradicting $T'(y) < T'(x)$.

Problem 22 (Analysis techniques for async ASMs). Develop a practical method (conceptual framework and proof principles) for the analysis of concurrent algorithms in terms of truly concurrent runs, i.e. operating directly in terms of partially ordered async ASM runs and circumventing to map moves to linear time. A first step in this direction appears in [258] and is illustrated there by a variation of the above correctness proof for the Bakery algorithm. The relevant definitions are in the lecture slides `Bakery` (⇝ CD). See also the proof principles developed in [372, Parts C,D] for the verification of concurrent network algorithms.

6.4.3 Exercises

Exercise 6.4.1. (⇝ CD) In Lamport's bakery algorithm, instead of reading values from a global system clock, tickets (time stamps) are associated with each customer. Define an async Bakery ASM which uses global clock values instead of tickets.

Exercise 6.4.2. (⇝ CD) Show that the async Bakery ASMs with durative actions remain correct and fair if the register regularity condition is changed to Lamport's register regularity condition [315], where as the read-result stored in $P(X, Y)$ in state $S_b(r)$ also the value of $R(Y)$ at Y's first passive moment $\geq b$ can be taken.

Exercise 6.4.3. (⇝ CD) Show that in the bakery algorithm Lamport's register regularity condition allows overtaking of readers to happen.

Exercise 6.4.4. (⇝ CD) Show the correctness and fairness of the async Bakery ASMs with durative actions also for safe registers [315]: a read that is concurrent (overlapping) with a write may return any value, but otherwise returns the result of the last write preceding it.

6.5 Event–Driven ASMs

In this section we present event driven ASMs, basic as well as synchronous and asynchronous ones, in combination with turbo ASMs. We apply the concept to provide a precise model for the UML event mechanism, detailed for UML activity diagrams with and without multiplicities (cloning of processes) and asynchronous concurrent nodes. As an illustration we provide a succinct flow-diagram-based description of an interpreter for Occam programs.

Abstracting from the particularities of special event classes, which do distinguish different event handling models, one can interpret events as instantaneous *state changes*, whose effect can be described by predicates in terms of abstract states. The *atomicity* of events is reflected by the atomicity of state changes in ASM steps, obtained via firing sets of updates. The different possible ways an event may be originated – by a single agent (e.g. acting as environment) or by multiple agents which interact in a synchronous manner (e.g. to produce a set of simultaneous occurrences of signals forming the event) – are captured by monitored functions and the synchronous parallelism of ASMs.[28] Technically the general understanding of events leads us to split ASM rule conditions into an "event" and a "guard" part. This distinction comes out very clearly in *active databases* which can be modeled by rules of the following form:

$$\textsc{ActiveDatabaseRule} = \textbf{if } \mathit{event} \textbf{ and } \mathit{guard} \textbf{ then } \mathit{action}$$

An *event* in active databases represents the trigger which may (but not necessarily does) result in firing a rule. The *guard* which comes as an additional condition for a rule to get executed describes the relevant *state* part (also called the context) in which the event occurs. The *action* describes the task to be carried out by the database rule. Different active databases result from varying (a) the underlying notion of state, as constituted by the syntax and semantics of events, guards and actions, and of their relation to the underlying database states, (b) the scheduling of the evaluation of guard and action components relative to the occurrence of events (coupling modes, priority declarations, etc.), (c) the rule ordering (if any). See [160].

Many FSMs and reactive control systems present this feature of event-triggered rules, typically with *signals* or *passage of time* as events. The general pattern of such rules is the following:

$$\textbf{upon } \mathit{event} \textbf{ do } \mathit{action}$$

standing for **if** *event* **then** *action* (where as usual we interpret *action* as an ASM rule). The frequently imposed constraint that each *event* is instantaneously perceived by all involved processes (underlying for example the semantics of synchronous programming languages) is directly captured by the synchronous parallelism of basic or turbo ASMs. We illustrate this by using instances of the machine Sustain from p. 38 and of $\textsc{Switch}(\mathit{cond}_i, \mathit{ctl}_i)_i$ from Fig. 2.9 as components to build a stopwatch:

[28] This general understanding of events does not abstract from states and does not relegate their description to a few variables which appear as parameters, differently from declarative process algebra systems like CSP [281], where computations are abstracted into "agreement protocols", machines into "patterns of possible behavior" which are defined entirely in terms of the occurrence and availability of events, with events understood as instantaneous, atomic, possibly parameterized interactions ("agreements") between such "processes".

> The basic stopwatch receives an input signal START_STOP that alternatively puts it in “running” and “stopped” states. Initially the stopwatch is stopped. It also receives a signal HS each 1/100 second. The stopwatch computes an integer TIME, whole value is the total amount of time (counted in 1/100 second) spent in the “running” state. [266, p. 25]

We instantiate the monitored function *ClockTick* of SUSTAIN(*signal*) by the environment-controlled time pulse HS (with its desired frequency) and use *CountTicks* as name for the TIME *signal*. We use a user-controlled monitored function *StartStop* for the event which triggers switching between the two control states *stopped* and *running*. We use the SWITCH instance

$$\{\textsc{Switch}(\mathit{StartStop}, \mathit{running}), \textsc{Switch}(\mathit{StartStop}, \mathit{stopped})\}$$

This leads to the following control state ASM (see the flowchart definition in `Clock` (⇝ CD)) to compute and display the number of time intervals spent *running*, started in $\mathit{ctl\_state} = \mathit{stopped}$ with $\mathit{CountTicks} = 0$. We leave it as Exercise 6.5.1 to explain what could go wrong with SUSTAIN(*CountTicks*) instead of SUSTAIN($\mathit{CountTicks} + 1$).

```
BasicStopWatch = {ClockRun, ClockReStart} where
  ClockRun = if ($ctl\_state = running$) then
    upon $ClockTick$ do $CountTicks := CountTicks + 1$
    Sustain($CountTicks + 1$)
    Switch($StartStop, stopped$)
  ClockReStart =
    if $ctl\_state = stopped$ then Switch($StartStop, running$)
```

The basic stopwatch works properly only if correctly initialized. To extend it to a STOPWATCHWITHRESET it suffices to add a RESET rule, which brings the machine upon a user-controlled *Reset* signal from *stopped* to a new control state *start*,[29] where a new rule CLOCKSTART upon a *StartStop* signal initializes *CountTicks*. A natural assumption for using stopwatches is that *StartStop* and *Reset* are disjoint so that they never happen together (avoiding an inconsistent update of *ctl_state*).

```
ReSet = Switch($Reset, start$)
ClockStart = if ($ctl\_state = start$) then
  upon $StartStop$ do $CountTicks := 0$
  Switch($StartStop, running$)
```

An example for a data flow machine with *input* events – value carrying signals – is the *Neural Net Machine* in Fig. 5.8, which in $\mathit{ctl\_state} = \mathit{input}$ is triggered by the arrival of *newInputToBeConsumed*. Another machine with input events is the widely used parallel virtual machine (PVM) [214] which

[29] What has to be changed to allow resetting also from running state?

realizes a distributed computation model characterized by the *reactive behavior* of concurrently operating PVM daemon processes, each residing on one of several host computers. The daemons are triggered by the environment; they carry out the PVM instructions of the local tasks they have to manage and interact with each other through asynchronous message-passing. No daemon can influence when, from where and which request or message will reach him; rather he has to wait for the next such event to come. This intuition can be faithfully modeled by introducing a monitored *event* function which for a given daemon might yield a PVM instruction or a message as value. If *event*(*pvmd*) is defined and has the value *instr/mssg*, then the daemon *pvmd* is going to execute/read *instr/mssg*, formalized by the following rule scheme taken from [107, 108]:

```
if $event(pvmd) = instr/mssg$ then $execute\_instr/read\_mssg$
```

for each individual PVM instruction *instr* or PVM message *mssg*, where *execute_instr/read_mssg* represents the corresponding updates. A typical integrity constraint on the function *event* is that a defined value of *event*(*pvmd*) remains stable until the PVM daemon *pvmd* has evaluated the function and that *event*(*pvmd*) is reset or indicates the next event as soon as the *pvmd* has read the current value. "Consuming an event" can be modeled also locally, per reacting agent and without granting access to the updates of the global state which characterize the event, namely by an update, upon reacting to the event, of a local copy of a synchronization bit which is flipped each time the event "happens". Assume that *EventBit*(*event*) is flipped each time the *event* "happens". Every agent who is supposed to react to the *event* by an *action*(*event*) is equipped with an instance **self**.*EventBit* of this event bit function, to be switched upon reacting to the event.

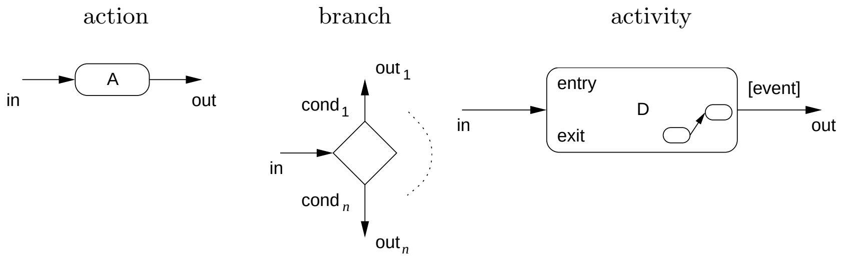
```
ConsumeEvent =
  if $event$ and self.$EventBit(event) = EventBit(event)$ then
    $action(event)$
    self.$EventBit(event) := 1 -$ self.$EventBit(event)$
```

The most traditional form of event is a procedure or operation *call*. Typically such "input" or "communication" events (monitored or shared functions) come with varying mechanisms or assumptions for the agents which update (produce and/or delete) events and for the (synchronous or asynchronous) scheduling schemes used by them for the purpose. To provide a concrete example, in the following section we present a model for a UML variant of event-triggered FSMs.

6.5.1 UML Diagrams for Dynamics

In UML four types of diagrams are proposed for the description of system dynamics, namely interaction (sequence and collaboration) diagrams, state

Fig. 6.18 Sequential UML nodes



machine and activity diagrams. For all four of them a precise semantical foundation can be given by identifying them as a particular subclass of sync or async ASMs. We show this here for activity diagrams which can be classified into mono-agent basic and multi-agent synchronous and asynchronous diagram classes, extending networks of hierarchical FSMs. We first describe the signature and then the ASM rules of activity diagrams.

An activity diagram is a finite directed graph whose nodes are classified into atomic and composite ones. Each *atomic* node represents an atomic (not decomposable) guarded action whose execution is considered as not interruptible and performed in zero ("insignificant") time in case the guard is true. Multiple invocations of action nodes are allowed to be executed concurrently. This is modeled by basic ASM steps, so that we identify atomic UML actions with basic ASM rules. A *composite* node represents a structured (decomposable) computation which is considered to take some time to complete and to be interruptible by the occurrence of (external or internal) events. The distinction between sequential and concurrent composition nodes is captured by defining sync and async activity ASMs. Arcs between nodes specify transitions from one action or activity to the next and can be labeled by an *event* (so-called triggered transitions) and by a *guard* which together condition the firing of the transition.

Atomic nodes are action or branching nodes. *Action nodes* are of form $node(in, \textbf{if } guard \textbf{ then } act, out, isDynamic, dynArgs, dynMult)$ with in denoting the unique incoming arc, out the unique outgoing arc, act the associated atomic action conditioned by $guard$; $isDynamic$ denotes whether the action has to be executed in $dynMult$ many parallel instances with parameters L_i from the set $dynArgs$ of sequences of objects $\{L_1, \ldots, L_n\}$. We denote these node parameters by homonymous functions $in(n)$, $action(n)$, $out(n)$ etc.; see Fig. 6.18.

There are two special parameterless action nodes, namely exactly one *initial node* per diagram indicating the default starting place, and *final nodes* indicating where the control flow for the diagram terminates. By $initArc(diagram)$ we denote the arc exiting the initial node of the given *diagram*, by *finalArc*(*diagram*) any arc entering one of its final nodes. The notation is justified since without loss of generality one can assume that these arcs are "fused" into *one* arc entering one distinguished final node.

Basic activity diagrams. Activity diagrams with only action and branching nodes are a special case of basic control state ASMs. In fact the view that being positioned on the arc $in(n)$ ingoing a node n results in performing $action(n)$ and moving to an arc $out(n)$ can be rendered as executing $action(n)$ in control state $ctl\_state = n$ and being led by $out(n)$ to a node $next(n)$ as next ctl_state. Therefore the semantics of an action node n is a rule $\mathrm{FSM}(n, action, next(n))$ as described below, where in preparation of a further refinement coming with the semantics of activity nodes we add $ctl\_state = n$ as an additional predicate $active(n)$ to the guard. For a compact formulation of multiple action instances, $isDynamic(n) = false$ is required to imply that $dynArgs(n)$ yields the empty parameter list, so that $correctMultiplicity(n)$ means that either $\neg isDynamic(n)$ or $dynArgs(n)$ contains exactly $dynMult(n)$ parameter lists. Thus the meaning of action nodes is captured by the following basic ASM rule where, as required in the UML manuals, no action is performed if $\mid dynArgs(n) \mid \neq dynMult(n)$.[30]

```
$\textsc{ActionNode}(n) = \mathrm{FSM}(n, action, next(n))$ where
  $action =$ if $active(n)$ and $guard$ and $correctMultiplicity(n)$ then
    $act$
    forall $L \in dynArgs(n)$ with $guard(L) = true$ do $act(L)$
```

Branching nodes $node(in, (cond_i)_{i \leq k}, (out_i)_{i \leq k})$ express alternative control state flow with alternatives checked in sequential order. As for action nodes we include also here being *active* into the formulation of the rule guard.

```
$\textsc{BranchNode}(n) = \mathrm{FSM}(n, test, next(n, min\{i \leq k \mid cond_i\}))$
  where $test =$ if $active(n)$ then skip
```

Synchronous activity diagrams. Basic activity diagrams are extended by so-called sequential composite or *activity nodes*. These nodes represent an entire activity subdiagram, which comes with entry and exit actions (to be performed on entering/leaving the node) and with events which interrupt the subcomputations in a context of multiple synchronous agents (clones). The form of activity nodes is

$$node(in, entry, exit, diagram, out, event, isDynamic, dynArgs, dynMult)$$

[30] We reflect that the parameterization of guards for multiple instances of actions may influence their truth value. It happens frequently in UML documents that such semantically relevant issues are not even contemplated. A benefit of attempts to capture the intended generality by precise models consists in revealing whether what UML calls "semantic variation points" represent a real freedom for implementations or simply an omission of semantically relevant issues. See for example [98, 99].

extending that of action nodes by the entry/exit actions *entry, exit*, the associated activity sub*diagram* and an optional *event* which labels the outgoing arc to determine possible interrupts of the sub*diagram* execution. Due to the view of atomic actions as not interruptable, events are never associated with arcs leaving action nodes. Activities instead are intended to have duration and to be interruptable, so that multiple instances of a subcomputation cannot be captured by the simple parallelism of a basic ASM but have to be associated with subagents (clones which are in synchrony, i.e. are executed in parallel with the master agent of the activity diagram).[31] Being $active(n)$ then has to be refined to include as guard also the condition that currently the executing agent hears $noInterrupt(\mathbf{self})$; in preparation of the extension to the asynchronous case of subdiagrams which are executed by multiple principal (non-clone) agents we also include here already a *running* mode (with a function $mode(\mathbf{self})$ which will change to $mode = interrupted$ when interrupt events for subagents occur).

$$active(n) \equiv ctl\_state = n \textbf{ and } noInterrupt \textbf{ and } mode = running$$

The parameterization of the durative subcomputations is captured by an *env*ironment function associated with agents.

```
$\textsc{EnterActivity}(n) = \textbf{if } active(n) \textbf{ and } correctMultiplicity(n) \textbf{ then}$
  $entry(n)$
  $ctl\_state := initArc(diagram(n))$
  $\textbf{if } isDynamic(n) \textbf{ then } clone(dynArg(n), diagram(n))$
$\textbf{where}$
$correctMultiplicity(n) =$
  $\neg isDynamic(n) \textbf{ or } dynMult(n) =\mid dynArgs(n) \mid$
$clone(Arg, Dgm) = \textbf{forall } L \in Arg \textbf{ let } a = new(Agent) \textbf{ in}$
  $isClone(a, Dgm) := true$
  $activate(a, Dgm, L)$
$activate(a, Dgm, L) =$
  $ctl\_state(a) := initArc(Dgm)$
  $mode(a) := running$
  $env(a) := insert(L, env(\mathbf{self}))$
```

There are two ways to exit an activity diagram, either normally or due to an interrupt. The normal exit takes place when the subcomputation of an agent has reached a *final* node of an activity diagram whose *out*going arc is not labeled by an event. If the agent is a clone or the principal agent leaving the *Main* diagram, it terminates its computation (formally by being deleted

[31] Since here we deal only with UML activity diagrams, we assume that every element put into the set *Agent* is automatically equipped with the ASM rules we are defining to execute UML activity diagrams. This saves the update $ASM(a) := ASM(\mathbf{self})$ in the *activate* macro below. In the case that *dynArg* is empty, EnterActivity lets the main agent proceed without creating any clone.

from *Agent*), otherwise the control of the principal agent is led to the next node by the *out*going arc and the exit action is performed, determined by the nearest activity node *actRoot*(n) to whose diagram n belongs.[32]

```
NormalExitActivity($n$) = if $FinalActDgmNode(n)$ and
  $active(n)$ and $event(actRoot(n)) = undef$ then
    if not $isClone(\mathbf{self}, diagram(actRoot(n)))$ and
      not $diagram(actRoot(n)) = Main$ then
        $exit(actRoot(n))$
        $ctl\_state := next(actRoot(n))$
    else $delete(\mathbf{self}, Agent)$
```

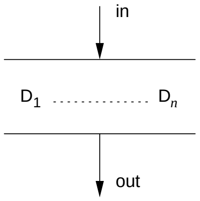
Exit from a diagram whose *out*going arc is labeled by an *ev*ent depends on the event to have *occurred*(ev), a notion which is kept abstract in UML to be adaptable to different event-handling mechanisms. The UML semantics of event processing is based on the *run to completion* assumption: events are processed one at a time and when the machine is in a stable configuration, i.e. a new event is processed only when all the consequences of the previous event have been exhausted. The first condition can be translated by a corresponding assumption on the monitored predicate *occurred*. The second condition is more difficult to capture, given the requirement that the exit actions of interrupted nested diagrams are performed following the subdiagram relation, from the bottom to the nearest event occurrence on an arc *out*going an enclosing activity diagram. We have to use turbo ASMs to capture this phenomenon.

A static function $upDgm\colon DIAGRAM \rightarrow DIAGRAM$ determines the subdiagram relation, assigning to a diagram its immediately enclosing diagram (if any), either the main diagram or the diagram of an activity or a concurrent node (see below). Let *scope*(e) denote the set of nodes which are in the scope of exactly the given occurrence of event e on an arc *out*going an activity diagram, following the subdiagram relation to avoid conflicts between different occurrences of events. When a principal agent **self** *hears event* e (implying that *noInterrupt* has become false), its subcomputation has to be interrupted to transfer the control to $n = next(actRoot(e, ctl\_state(\mathbf{self})))$ where $actRoot(e, node)$ yields the nearest activity node whose outgoing arc is labeled by the occurrence of e with *scope*(e) comprising *node*. The interrupt also triggers the sequence of exit actions and of clone interruption on the way from $currDgm = diagram(ctl\_state(\mathbf{self}))$ to $diagram(n)$.

```
InterruptExit($e$) = if self hears $e$ and not $isClone(\mathbf{self})$ then
  let $n = next(actRoot(e, ctl\_state(\mathbf{self})))$
    $exitFromTo(currDgm, diagram(n))$
```

[32] This would be the place to include the condition that also all clone computations of the diagram have reached a final node. The UML texts leave it unclear whether this condition is intended.

Fig. 6.19 UML concurrent nodes



```
    $ctl\_state(\mathbf{self}) := n$
  **where** **self** hears $e =$
    $occurred(e)$ **and** $ctl\_state(\mathbf{self}) \in scope(e)$
```

Here $exitFromTo(D, D')$ denotes the turbo ASM which iterates along the diagram nesting to perform $exit(T)$ and to $deleteClones(T)$[33] for each activity diagram T encountered between D and D'. We leave the definition as Exercise 6.5.3.

Asynchronous activity diagrams. Adding composite concurrent nodes, which permit us to split a single flow of control into multiple concurrent flows of control to synchronize concurrent processes, results in asynchronous activity diagrams. *Concurrent nodes* are of form $node(in, D, out)$ (see Fig. 6.19) with a sequence $D = D_1, \ldots, D_m$ of subdiagrams not consisting of only initial or final nodes. The upper synchronization bar is called *fork*, the lower one *join*.[34]

Entering a concurrent node means to activate all of its subdiagrams so that at each moment in each of these subdiagrams at least one agent is active, until exiting the corresponding join becomes possible when and only if all subdiagram computations have been terminated. This is easily described by the splitting rule UmlFork, which is analogous to the rule OccamParSpawn (p. 43) and goes together with the synchronization rule UmlJoin. Here it becomes relevant that $active(n)$ is refined to include also $mode(\mathbf{self}) = running$. The set of *Subagents* is defined by the *parent* relation.

```
UmlFork($n$) = **if** $ConcurNode(n)$ **and** $active(n)$ **then**
  **let** $a_1, \ldots, a_m = new(Agent)$ **forall** $1 \leq i \leq m$
    $activate(a_i, D_i)$
    $entry(D_i)$
    $parent(d_i) := d$
```

[33] $deleteClones(T)$ deletes from *Agent* every clone for any activity subdiagram T' of T.

[34] The initial (resp. final) node of the subdiagrams D_i is considered to be represented by the fork (resp. join) bar. The balancing property is assumed whereby the number of flows that leave a fork must match the number of flows that enter its corresponding join.

```
    $mode(\textbf{self}) := waiting$

$\textsc{UmlJoin}(n) =$ if $ConcurNode(n)$ and $ctl\_state = n$ and
    $mode = waiting$ and forall $a_i \in Subagent(\textbf{self})$
        $mode(a_i) = running$ and $ctl\_state(a_i) = finalArc(D_i)$
    then
        $ctl\_state(\textbf{self}) := next(n)$
        $mode(\textbf{self}) := running$
        forall $a \in Subagent(\textbf{self})$ do $delete(a, Agent)$
```

To define a reasonable refinement of the InterruptExit rule for asynchronous activity diagram ASMs is more complicated. It requires us to refine the *exitFromTo* macro to a *ExitSubDgm* machine which takes into account also the nesting in concurrent subdiagrams. The UML manuals say (almost) nothing about this; in [98, 99] a scheme is described which, however, leaves many critical points unmentioned. Here too we restrict ourselves to delineate a rather generic scheme which a standardization effort may instantiate to a reasonable concrete definition.

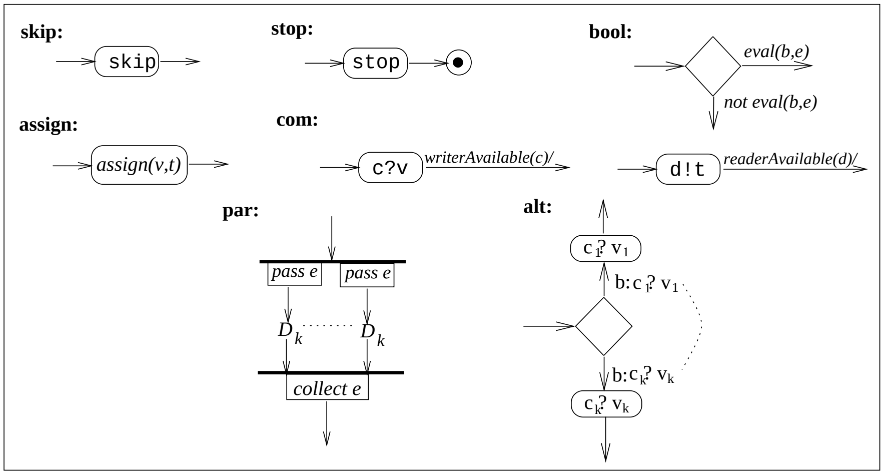
```
$\textsc{AsyncInterruptExit}(e) =$
    if self hears $e$ and not $isClone(\textbf{self})$ then
        forall $EvNode \in EvCtl(e, currDgm)$ do $ExitSubDgm(EvNode)$
```

Occam interpreter. To conclude this section we exploit the combination of graphical UML notation with the semantical rigor of activity diagram ASMs for a compact formulation of an ASM interpreter for Occam programs [105]. The description starts from the usual graphical layout of a program as a flowchart, flattening the structured program. As a result the standard sequential and syntax-directed part of control is already incorporated into the underlying graph structure and thus semantically defined by the ASM rules for activity diagrams with atomic and concurrent nodes.[35] It only remains to define the semantical meaning of the atomic actions – basic Occam instructions – which are executed by the respective agents upon passing through the action nodes of the diagram. This is defined by Fig. 6.20 (taken from [98]) with the specific Occam action and guard macros defined in OccamActions below. We use Occam notation, function e for the environment (i.e. the association of values to local variables) of each agent a, *eval* for expression evaluation under a given environment, $s[v/t]$ for the result of substituting v in s by t, $c!t$ for offering the value of t at channel c and $c?v$ for requesting through c a value for v.

```
$\textsc{OccamActions} =$
    $writerAvailable(c) \equiv \exists writer \in AGENT\ \exists n \in NODE$
        $active(writer) = in(n)$ and $action(n) = d!t$ and
```

[35] As a consequence of flattening, the description in Fig. 6.20 uses no activity nodes. Using them one could reflect the code structure directly.

Fig. 6.20 Semantics of Occam: activity diagram ASM



$$\begin{array}{l}
\qquad eval(c, e(a)) = eval(d, e(writer)) \\
readerAvailable(c) \equiv \exists reader \in AGENT\ \exists n \in NODE \\
\quad active(reader) = in(n) \textbf{ and } action(n) = c?v \textbf{ and} \\
\qquad eval(d, e(a)) = eval(c, e(reader)) \\
assign(v, t) \equiv e := e[v/eval(t, e)] \\
c?v \equiv e(a) := e(a)[v/eval(t, e(writer\_for\_c))] \\
d!t \equiv skip \\
b : c?v \equiv writerAvailable(c) \textbf{ and } eval(b, e(a)) = true \\
\text{pass } e \equiv e(a) := e(parent(a)) \\
\text{collect } e \equiv e(\textbf{self}) := \cup_{1 \le i \le k} e(a_i)
\end{array}$$

In the Fork rule *pass e* is added to the updates of $activate(a, D)$, in the Join rule *collect e* to the updates of **self**. Occam syntax conditions guarantee that the definition of reader and writer for a channel is consistent, since for each channel at each moment at most one pair of reader/writer exists which are available for each other.[36] From this definition of the semantics of Occam one can derive its implementation to Transputer code by a series of stepwise ASM refinements, as done in [104] where also the correctness of the transformation is proved.

[36] The ALT construct as described in Fig. 6.20 really provides the deterministic Transputer implementation of this construct, due to the underlying UML assumption that the guards of branching nodes have to be evaluated in sequential order and to be disjoint and complete. One can reflect the non-deterministic Occam interpretation of this construct by interpreting the guards g_i as driven by monitored choice functions.

Problem 23 (Modeling and analyzing a real-life operating system). Use structured async ASMs to model a real-life operating system of your choice as a layered event-based system. See the small MINIX case study for the X86 architecture in [186].

6.5.2 Exercises

Exercise 6.5.1. Explain what could go wrong with SUSTAIN($CountTicks$) instead of SUSTAIN($CountTicks + 1$) in BASICSTOPWATCH.

Exercise 6.5.2. (⇝ CD) Extend STOPWATCHWITHRESET to a stopwatch with intermediate time handling: "A new input signal LAP now allows us to record an intermediate time (for instance, the time spent by one runner for one trap lap) while continuing to measure the global time. One occurrence of LAP freezes the time on display, while the internal stopwatch time continues to be computed as before. The next occurrence of LAP puts the stopwatch back in a state displaying the running time."[266, p. 26]

Exercise 6.5.3. Write a turbo ASM for the $exitFromTo(D, D')$ macro which is used in INTERRUPTEXIT.

Sources and Historical Remarks

The extension of basic ASMs to async (there called distributed) ASMs appeared in [248], together with the async Bakery ASMs [114]. This definition builds upon and supersedes earlier more restricted definitions of concurrency for ASMs, defined to capture the parallelism of Occam [257, 105], of various parallel forms of Prolog [375, 122, 123, 374] and of the Chemical Abstract Machine and the π-calculus [227]. The first combination of async ASMs with turbo ASMs appeared in [98, 99], which is the basis for the UML activity diagram model in Sect. 6.5.1. See Sect. 9.2 for details.

7 Universal Design and Computation Model

This chapter is devoted to an investigation of the universality properties of ASMs[1]. ASMs are put into relation with other well-known system design and analysis approaches and models of computation (which are assumed to be known, though references are given so that the details we refer to can be checked). The chapter can be read independently from the others; for an understanding only the definition of ASMs is needed.

In Sect. 7.1 we show how ASMs capture the principal models of computation and specification in the literature, including major UML concepts. Our main goal is to illustrate the possibility that ASMs offer to smoothly integrate useful concepts and techniques which have been tailored and are successfully applied for specific goals or application domains. As a by-product we obtain a uniform set of transparent model descriptions which – starting from scratch – provide the conceptual basis for a comparative investigation of specification methods and constitute a useful definitional framework for teaching computation theory. Sect. 7.2 is primarily of logical interest. It contains Gurevich's mathematical analysis of the epistemological universality claim, which became known as the "ASM thesis", together with the proof for its sequential version from a small number of postulates.

7.1 Integrating Computation and Specification Models

In this section we show how widely used current models of specification and computation can be naturally mapped into the ASM framework, including basic UML concepts for which our models provide an abstract precise meaning. We model them by ASMs, starting from scratch and providing a uniform set of transparent easily understandable descriptions which are faithful to the basic intuitions and concepts of each investigated system.[2] Our main goal is to provide a mathematical basis for technical comparison of established models of computation, contributing to rationalizing the scientific evaluation of different system specification approaches in the literature by clarifying in detail their merits and drawbacks. As a side effect we obtain a small set of defi-

[1] Lecture slides can be found in `UniversalCompModel` (⇝ CD).

nitions which unravel the basic common structure of the myriad of different machine concepts which are studied in computation theory.

Starting from a subclass of UML diagrams for system dynamics – namely control state ASMs as defined in Fig. 2.5 – we investigate in Sect. 7.1.1 Turing-machine-like *classical models of computation* (automata, substitution systems, structured programming) and in Sect. 7.1.2 the major currently used *system design models* (*executable* high-level design languages like UNITY or COLD, *state-based* specification languages like Petri nets, B, UML state machines and activity diagrams, *stateless* modeling approaches like functional programming or axiomatic logico-algebraic design systems).

The ASM models in this section are different from the ones which come out of the proofs for the two special versions of the ASM thesis in Sect. 7.2 and in [61], where a small number of postulates is exhibited from which every synchronous parallel computational device can be proved to be simulatable in lock-step by an appropriate ASM. The construction there depends on the way the abstract postulates capture the amount of computation (by every single agent) and of the communication (between the synchronized agents) allowed in a synchronous parallel computation step. The desire to *prove* computational universality from abstract postulates implies the necessity to first capture the huge class of data structures and the many ways they can be used in a basic computation step and then to unfold every concrete basic parallel communication and computation step from the postulates; this unavoidably yields some "encoding" and "decoding" overhead to guarantee, for *every* computational system which possibly could be proposed, a representation by the abstract concepts of the postulates. As a side effect of this – epistemologically significant – generality of the postulates, the application of the Blass and Gurevich proof scheme to established models of computation may yield "abstract" machine models which are more involved than necessary and may blur features which really distinguish different concrete systems. Furthermore, *postulating by an existential statement* that "states" are appropriate equivalence classes of structures of a fixed signature (in the sense of logic), that the evolution happens as the iteration of single "steps" and that the single-step "exploration space" is bounded (i.e. that there is a uniform bound on the memory locations that the basic computation steps depend upon, up to isomorphism) does not by itself provide, for a given computation or specification model, a standard reference description of its characteristic states, of the objects entering a basic computation step, and of its next-step function.

The goal in this section is that of explicitly and *naturally* modeling systems of specification and computation – *closely and faithfully*, *at their level of abstraction* as the spirit of the ASM thesis requires – based upon an analysis

[2] The particularly natural way ASMs capture other computation models and thus turn out to be "universal" contrasts with the difficulties one encounters when trying to reverse the simulation, defining ASMs in other computational frameworks.

of the characteristic conceptual features of each of them. In other words we look for ground model ASMs[3] for each established model of computation or of high-level system design which include asynchronous multi-agent systems, for which no proof of the ASM thesis is known, and which

- for every framework directly reflect the basic intuitions and concepts, by gently and explicitly capturing the basic data structures and single computation steps which characterize the investigated system,
- are formulated in a way which is uniform enough to allow explicit comparisons between the considered classical system models.

By deliberately keeping the ASM ground model for each proposed system as close as possible to the original usual description of the system, so that it can be recognized straightforwardly to be simulated correctly and step by step by its ground model, we provide for the full ASM thesis, i.e. including distributed systems, a strong theoretical argument which

- avoids a sophisticated existence proof for the ASM models from abstract postulates,
- avoids decoding of concrete concepts from abstract postulates,
- avoids a sophisticated proof to establish the correctness of the ASM models.

7.1.1 Classical Computation Models

In this section we use the control state ASMs of Fig. 2.5 to construct mostly basic ASMs for classical automata and substitution systems (see any textbook on computation theory, e.g. [70]): FSMs (finite state machines à la Moore-Mealy and their more recent extensions by stream-processing or by timing conditions or by co-design control features), pushdown automata and computation universal automata (à la Turing, Scott, Eilenberg, Minsky, Wegner), and replacement systems (à la Thue, Markov, Post). Conceptually, modeling structured and functional programming concepts and tree computations (including language-generating grammars like context-free, attribute and tree adjoining grammars) also belong here, which have already been analyzed in Sections 4.1, 3.2.

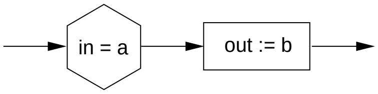
Variations of Mealy/Moore automata. By the very definition of control state ASMs given in Fig. 2.5, deterministic Mealy and Moore automata are ASMs where every rule has the following form (see Fig. 7.1), with *skip* instead of the output assignment in the case of Moore automata.

$$\textsc{Fsm}(i, \textbf{if } \mathit{in} = a \textbf{ then } \mathit{out} := b, j)$$

Writing programs in the usual tabular form, with one entry (i, a, j, b) for every instruction "in state i reading input a, go to state j and print output b",

[3] For the concept of a ground model see Sect. 2.1.1.

Fig. 7.1 MEALY automata rules



yields the following guard-free FSM rule scheme for updating (ctl_state, out), where the parameters $Nxtctl, Nxtout$ are the two projection functions which define the program table, mapping "configurations" (i, a) of control state and input to the next control state j and output b.

```
MealyFsm($Nxtctl, Nxtout$) =
  $ctl\_state := Nxtctl(ctl\_state, in)$
  $out := Nxtout(ctl\_state, in)$
```

Since the input function *in* is monitored, it is not updated in the rule scheme, though one could certainly make it shared to formalize an input tape which is scanned letterwise from, say, left to right (see as an example StreamProcessingFsm below). The question of 1-way or 2-way automata is a question of whether one also includes into the instructions *Moves* of the input head (say on the input tape), yielding additional updates of the *head* position and a refinement of *in* to *in*(*head*) (the input portion seen by the new reading head):

```
TwoWayFsm($Nxtctl, Nxtout, Move$) =
  $ctl\_state := Nxtctl(ctl\_state, in(head))$
  $out := Nxtout(ctl\_state, in(head))$
  $head := head + Move(ctl\_state, in(head))$
```

Non-deterministic versions of FSMs, as well as of all the machines we consider below.[4] are obtained by placing the above rules under a **choose** operator to allow choices among different $R \in Rules$, obtaining rules of the form **choose** $R \in Rule$ **in** R.

We illustrate an instance of this scheme for the extension of FSMs to machines which compute stream functions $S^m \rightarrow S^n$ over a data set S (typically the set $S = A^*$ of finite or $S = A^{\mathbb{N}}$ of infinite words over a given alphabet A), yielding an output stream *out* resulting from consumption of the input stream *in*. Non-deterministically in each step these automata

- read (consume) at every input port a prefix of the input stream *in*,
- produce at each output port a part of the output stream *out*,
- proceed to the next control state *ctl_state*.

[4] Therefore below we will only mention deterministic machine versions. See also Sect. 4.2.

To extend the MealyFsm machines to a model of these stream processing FSMs it suffices to introduce two choice-supporting functions $\mathit{Prefix}\colon \mathit{Ctl} \times S^m \rightarrow \mathit{PowerSet}(S^m_{\mathit{fin}})$, yielding sets of finite prefixes among which to choose for a given control state and input stream, and $\mathit{Transition}\colon \mathit{Ctl} \times (S^m_{\mathit{fin}}) \rightarrow \mathit{PowerSet}(\mathit{Ctl} \times S^n_{\mathit{fin}})$ describing the possible choices for the next control state and the next finite bit of output. The rule extension for stream processing FSMs is then as follows, where input consumption is formalized by deletion of the chosen prefix from the shared function *in*:[5]

```
StreamProcessingFsm($\mathit{Prefix}, \mathit{Transition}$) =
  choose $\mathit{pref} \in \mathit{Prefix}(\mathit{ctl\_state}, \mathit{in})$
    choose $(c, o) \in \mathit{Transition}(\mathit{ctl\_state}, \mathit{pref})$
      $\mathit{ctl\_state} := c$
      $\mathit{out} := \mathit{concatenate}(o, \mathit{out})$
      $\mathit{in} := \mathit{delete}(\mathit{pref}, \mathit{in})$
```

Mealy/Moore automata give rise to *Mealy/Moore ASMs* (defined in [86]), a subclass of control state ASMs where the emission of output is generalized to arbitrary ASM rules:

$$\textsc{MealyAsm} = \textsc{Fsm}(i, \textbf{if } \mathit{in} = a \textbf{ then } \mathit{rule}, j)$$

An instance of MealyAsms are Leveson's Requirements State Machines (RSM), introduced to relate a general set of completeness criteria for requirements specifications to process-control systems [320]. In an RSM, each rule guard is an input predicate conditioning the value and the timing of the sensor input, and the to-be-generated output to the actuators is described by an output predicate. The special-purpose specification language SpecTRM-RL which has been developed to overlay the low-level RSM model offers a particular form of abstraction, supported by a set of tools [321]. MealyAsms also appear as components of *co-design FSMs*, where turbo ASM component rules are needed to compute arbitrary combinational (external and instantaneous) functions. Co-design FSMs are used in [319] for high-level architecture design and specification and for a precise comparison of current models of computation. Usually co-design FSMs come together with a global agent scheduler or with timing conditions for agents which perform durative (not only atomic) actions.[6] We illustrate the inclusion of timing conditions by an extension of Mealy–ASMs to *timed automata* [13]. In these automata letter input comes at a real-valued occurrence time which

[5] In [293] these machines are used to enrich the classical networks of stream-processing FSMs (stream-processing components communicating among each other via input/output ports) by ASM state transformations of individual components. They are applied to provide a uniform semantics to common visual notations for discrete event systems.

[6] For modeling durative actions see Sect. 6.4.

is used in the transitions where clocks record the time difference of the current *in*put with respect to the *previousIn*put, derived from those two input functions and their (typically external) *occurrenceTime* by the equation $time_\Delta = occurrenceTime(in) - occurrenceTime(previousIn)$. Firing of transitions may be subject to clock constraints and includes clock updates (resetting a clock or adding to it the last input time difference). Typically the constraints are about input to occur within $(<, \leq)$ or after $(>, \geq)$ a given (constant) time interval, leaving some freedom for timing runs, i.e. choosing sequences of $occurrenceTime(in)$ to satisfy the constraints. Thus timed automata can be modeled as control state ASMs where all rules have the following form (the parameters *Constraint* and *Reset* are allowed to change with the control state or the input):

```
$\textsc{TimedAutomaton}(i, a, Constraint, Reset) =$
  $\textsc{Fsm}(i, \textbf{if } TimedIn(a) \textbf{ then } ClockUpdate(Reset), j)$ where
    $TimedIn(a) = (in = a \textbf{ and } Constraint(time_\Delta) = true)$
    $ClockUpdate(Reset) = \textbf{forall } c \in Clock \textbf{ do}$
      $\textbf{if } c \in Reset \textbf{ then } clock(c) := 0$
      $\textbf{else } clock(c) := clock(c) + time_\Delta$
```

In pushdown automata the Mealy automaton "reading from the input tape" and "writing to the output tape" is extended to reading from input and/or a *stack* and writing on the *stack*. Since these machines may have control states with no input-reading or no stack-reading, pushdown automata can be defined as control state ASMs where all rules have one of the following forms with the usual meaning of the *stack* operations $push, pop$ (optional items are enclosed in []):

```
$\textsc{PushDownAutomaton} =$
  $\textsc{Fsm}(i, \textbf{if } Reading(a, b) \textbf{ then } StackUpdate(w), j)$ where
    $Reading(a, b) = [in = a] \textbf{ and } [top(stack) = b]$
    $StackUpdate(w) \ = \ stack := push(w, [pop](stack))$
```

Turing-like automata. Writing pushdown transitions in tabular form (using an auxiliary function defined by $Pop\&Push(s, w, 1) = push(w, pop(s)$ and $Pop\&Push(s, w, 0) = push(w, s)$

```
$\textsc{PushDownAutomaton}(Nxtctl, Write, Pop) =$
  $ctl\_state := Nxtctl(ctl\_state, in, top(stack))$
  $stack := Pop\&Push(stack, w, b)$ where
    $w = Write(ctl\_state, in, top(stack))$
    $b = Pop(ctl\_state, in, top(stack))$
```

identifies the "memory refinement" of FSM *in*put and *out*put tape to *in*put and *stack* memory. The general scheme becomes explicit with Turing machines which combine *in*put and *out*put into one *tape* memory with moving *head*. All the *Turing-like machines* we mention below are control state ASMs which in each step, placed in a certain *pos*ition of their *mem*ory, read this *mem*ory in the *env*ironment of that *pos*ition and react by updating *mem* and *pos*. Variations of these machines are due to variations of mem, pos, env, whereas their rules are all of the following form:

```
$\textsc{TuringLikeMachine}(mem, pos, env) =$
$\textsc{Fsm}(i, \textbf{if } Cond(mem(env(pos)) \textbf{ then } \text{update } (mem(env(pos)), pos), j)$
```

For the original *Turing* machines this scheme is instantiated by $mem = tape$ containing words, integer positions $pos\colon Z$ where single letters are retrieved, $env = identity$, *Write*s in the position of the *tape head*. This leads to extending the rules of TwoWayFsm as follows (replacing *in* by *tape* and *Nxtout* by *Write*):

```
$\textsc{TuringMachine}(Nxtctl, Write, Move) =$
  $ctl\_state := Nxtctl(ctl\_state, tape(head))$
  $tape(head) := Write(ctl\_state, tape(head))$
  $head := head + Move(ctl\_state, tape(head))$
```

The alternating variation of Turing machines can be obtained by extending TuringMachine with a data refinement of OccamParSpawn on p. 43 to spawn subprocesses. An alternating TM-computation is focussed to either accept or reject the initial input tape, whereto it is permitted to also invoke TM-subcomputations and to explore whether some or all of them accept or reject their input. For this purpose to the traditional control states, which are termed *normal* and in which the given TuringMachine is executed, four new *type*s are added: control states which (a) simply *accept* or which (b) simply *reject* or which (c) accept if some subcomputation accepts and reject if every subcomputation rejects (*existential* type) or which (d) accept if every subcomputation accepts and reject if some subcomputation rejects (*universal* type). When in an existential or universal control state the subcomputations are created and put into *running mode* (rule AltTmSpawn below), the invoking computation turns to *idle mode* to observe whether the *yield* of the subcomputations switches from *undef* to either *accept* or *reject* and to define its own *yield* correspondingly (rules TmYieldExistential, TmYieldUniversal below). Different subcomputations of an alternating Turing machine, whose program is defined by the given functions $Nxtctl$,$Write$,$Move$, are distinguished by parameterizing the machine instances by their executing agents a, obtaining TuringMachine($Nxtctl$, $Write$, $Move$)(a) from the ASM TuringMachine defined above by replacing the dynamic functions ctl_state, $tape$, $head$ with their instances $a.ctl\_state$, $a.tape$, $a.head$. This leads us to the following definition, combining TuringMachine with a data refinement of OccamParSpawn. For simplicity of exposition but without loss of generality we assume that in an existential or universal state, the alternating Turing machine does not print or move its head and *NxtCtl* yields the set of possible next control states where the subcomputations are started. We use the derived function $children(a) = \{c \mid parent(c) = a\}$.

```
AlternatingTm($Nxtctl, Write, Move$) =
  if $type(\mathbf{self}.ctl\_state) = normal$ then
    TuringMachine($Nxtctl, Write, Move$)($\mathbf{self}$)
  if $type(\mathbf{self}.ctl\_state) \in \{existential, universal\}$ then
    AltTmSpawn($\mathbf{self}$)
    TmYieldExistential($\mathbf{self}$)
    TmYieldUniversal($\mathbf{self}$)
  if $type(\mathbf{self}.ctl\_state) \in \{accept, reject\}$ then
    $yield(\mathbf{self}) := type(\mathbf{self}.ctl\_state)$
where
AltTmSpawn($a$) = if $a.mode = running$ then
  forall $j \in Nxtctl(a.ctl\_state, a.tape(a.head))$ do
    let $b = new(Agent)$ in
      Activate($b, a, j$)[7]
      $parent(b) := a$
  $a.mode := idle$
Activate($b, a, j$) =
  $b.mode := running$
  $b.yield := undef$
  $b.ctl\_state := j$
  forall $pos \in domain(a.tape)$ do $b.tape(pos) := a.tape(pos)$
  $b.head := a.head$
TmYieldExistential($a$) =
  if $a.mode = idle$ and $type(a.ctl\_state) = existential$ then
    if $\forall c \in children(a)\ yield(c) = reject$ then $yield(a) := reject$
    if $\exists c \in children(a)\ yield(c) = accept$ then $yield(a) := accept$
TmYieldUniversal($a$) =
  if $a.mode = idle$ and $type(a.ctl\_state) = universal$ then
    if $\forall c \in children(a)\ yield(c) = accept$ then $yield(a) := accept$
    if $\exists c \in children(a)\ yield(c) = reject$ then $yield(a) := reject$
```

In contrast to Turing and register machines, their generalizations introduced by Scott [398] and Eilenberg [192] instead of read/write operations on words stored in a tape provide data processing for arbitrary data, residing in abstract *mem*ory, by arbitrarily complex *mem*-transforming functions. Looking back, the incorporation of abstract arbitrary data processing lets these

[7] The fact that the program of Occam subprocesses is static is reflected in OccamParSpawn (p. 43) by positioning the agents upon their creation at their start position in that program. Similarly in AlternatingTm every subcomputation is supposed to execute the same program, namely the one defined by *Nxtctl*, *Write*, *Move* and *type*. Therefore we skip mentioning in Activate the fixed association of *program*(b) to b.

two machine concepts appear as precursors of ASMs, even before Abstract Data Types and the notion of states as structures became fashionable.[8] However, the unfortunate "generalization" of the *local* Turing machine operations on the tape – deemed to be not abstract enough and too close to an implementation view – by global functional memory tests/updates exposed both of them to the frame problem and made them irrelevant for practical system design. Eilenberg's *X-machines* can be modeled as instances of Mealy ASMs whose rules in addition to yielding *out*put also update *mem* via global memory functions f (one for each input and control state):

$$\text{XMachine} = \text{Fsm}(i, \textbf{if } \mathit{in} = a \textbf{ then } \{\mathit{out} := b, \mathit{mem} := f(\mathit{mem})\}, j)$$

The global *mem*ory *Action*s of *Scott machines* come with the standard control flow directed by global memory *Test* predicates, formally described by a function *IfThenElse* defined by the given program. This yields control state ASMs consisting of rules of the following form:

```
ScottMachine($\mathit{Action}$, $\mathit{Test}$) =
  $\mathit{ctl\_state} := \mathit{IfThenElse}(\mathit{ctl\_state}, \mathit{Test}(\mathit{ctl\_state})(\mathit{mem}))$
  $\mathit{mem} := \mathit{Action}(\mathit{ctl\_state})(\mathit{mem})$
```

Wegner's *interactive Turing machines* [423] can in each step receive some input from the environment and yield output to the environment. Thus they simply extend the TuringMachine by an additional *input* parameter and an *output* action:[9]

```
TuringInteractive($\mathit{Nxtctl}$, $\mathit{Write}$, $\mathit{Move}$) =
  $\mathit{ctl\_state} := \mathit{Nxtctl}(\mathit{ctl\_state}, \mathit{tape}(\mathit{head}), \mathit{input})$
  $\mathit{tape}(\mathit{head}) := \mathit{Write}(\mathit{ctl\_state}, \mathit{tape}(\mathit{head}), \mathit{input})$
  $\mathit{head} := \mathit{head} + \mathit{Move}(\mathit{ctl\_state}, \mathit{tape}(\mathit{head}), \mathit{input})$
  $\mathit{output}(\mathit{ctl\_state}, \mathit{tape}(\mathit{head}), \mathit{input})$
```

Considering the output as written on an in-out tape results in defining $\mathit{output} := \mathit{concatenate}(\mathit{input}, \mathit{Out}(\mathit{ctl\_state}, \mathit{tape}(\mathit{head}), \mathit{input}))$ as the output action using a function *Out* defined by the program. Viewing the input as a combination of preceding inputs/outputs with the new user input results in defining *input* as a derived function $\mathit{input} = \mathit{combine}(\mathit{output}, \mathit{user\_input})$

[8] In [282] X-machines are even suggested as a software engineering framework for "building correct business process solutions".

[9] This description clarifies the limitations of Wegner's rather particular model for systems of interacting machines, compared with the concept of asynchronous multi-agent ASMs in Chap. 6. See the Neural Abstraction Machine in Fig. 5.8 with an analogous interaction scheme between machine and environment, but a more realistic internal machine computation. An abstract form of TuringInteractive is used in [298] under the name of interactive ASM to compute sequence functions (from finite sequences of multi-sets to sets of sequences of multi-sets).

depending on the current *output* and *user_input*. The question of single-stream versus multiple-stream interacting Turing machines (SIM/MIM) is only a question of instantiating input to a stream vector $(inp_1, \ldots, inp_n)$.

Substitution systems. The substitution systems à la Thue, Markov, Post are Turing-like machines operating over *mem*: A^* for some finite alphabet A with a finite set of word pairs (v_i, w_i), where in each step one occurrence of a "premise" v_i in *mem* is replaced by the corresponding "conclusion" w_i. The difference between *Thue systems* and *Markov algorithms* is that Markov algorithms have a fixed scheduling mechanism for choosing the replacement pair and for choosing the occurrence of the to be replaced v_i. In the Thue ASM rule below[10] we use $mem([p, q])$ to denote the subword of *mem* between the pth and the qth letter of *mem*, which *match*es v if it is identical to v. By $mem(w/[p, q])$ we denote the result of substituting w in *mem* for $mem([p, q])$. The non-determinism of Thue systems is captured by two selection functions. For the Markov version we show how one can include the condition on *match*ing already into the specification of these selection functions.

```
ThueSystem($ReplacePair$) =
  let $(v, w) = select_{rule}(ReplacePair, mem)$
  let $(p, q) = select_{sub}(mem)$
    if $match(mem([p, q]), v)$ then $mem := mem(w/[p, q])$
```

The Markov ASM is obtained from the Thue ASM by a pure data refinement, instantiating $select_{rule}(ReplacePair, mem)$ to yield the first $(v, w) \in ReplacePair$ with a premise occurring in *mem*, and $select_{sub}(mem, v)$ to determine the leftmost occurrence of v in *mem*. Similarly the ASM for *Post normal systems* is obtained by instantiating $select_{rule}(ReplacePair, mem)$ to yield a pair $(v, w) \in ReplacePair$ with a premise occurring as an initial subword of *mem*, $select_{sub}(mem)$ to determine this initial subword of *mem*, and by updates of *mem* which delete the initial subword v and copy w at the end of *mem*.

Concluding the ASM modeling of classical notions of computation in this section one can say that, *with hindsight*, it comes as no surprise that the numerous definitions of the notion of algorithms found in the 1930s and 1940s, in an attempt to mathematically capture the intuitive notion of *computable function*, all turned out to be equivalent. They are all variations (mostly data refinements) of the TuringLikeMachine model, which has been made explicit above. This positions also the Church-Turing thesis (e.g. see [70, Chap. AI]) with respect to the more general ASM thesis in Sect. 7.2.

[10] To be precise: the rule we state below defines what in the literature goes under the name of semi-Thue systems. Thue systems are semi-Thue systems which for each replacement pair (v, w) contain also the inverse pair (w, v).

7.1.2 System Design Models

In this section we show how to model by ASMs the basic semantical concepts of the executable high-level design languages UNITY and COLD, of widely used sequential and distributed state-based specification languages (illustrated for sequential systems by Parnas tables and B machines, and for distributed systems by Petri Nets), of dedicated virtual machines (e.g. data flow machines), and of stateless modeling systems (axiomatic logic-based systems, like denotational semantics and process algebraic systems). Conceptually, sequential functional programming concepts also belong here, which have already been analyzed in terms of turbo ASMs in Sect. 4.1.2. For modeling dynamic UML diagrams by basic, sync and async ASMs see Sect. 6.5.1.

UNITY [329]. *Unity* computations are sequences of state transitions where each step comprises the simultaneous execution of multiple conditional variable assignments, including quantified array variable assignments of form **forall** $0 \leq i < N$ **do** $a(i) := b(i)$. States are formed by variables (0-ary dynamic functions which may be shared, respecting some naming conventions), conditions are typically formulated in terms of $<, =$, and steps are executions of program statements which correspond in an obvious way to basic ASM rules. The steps are scheduled using a global clock (Unity system time) which synchronizes the system components for an interleaving semantics: per step one statement of one component program in the system is scheduled using non-deterministic schedulers (required to respect a certain fairness condition on infinite runs).[11] As in basic ASMs, there is no further control flow. Identifying components with basic ASMs and systems with sets of components leads therefore to the following computational model for Unity systems (which come with a particular proof system, geared to extract proofs from the program text):

```
UnitySystem($S$) =
  choose $com \in Component(S)$, choose $rule \in Rule(com)$
    $rule$
```

Problem 24 (Mobile ASM framework). In [331] Unity is extended to *Mobile Unity*, adding to programs location parameters and rules for transient interactions (engagement and disengagement of variable assignments, based upon a *react-to* construct which guarantees an assignment to be executed each time a condition is true), together with a distinction between transactions (statements composed by **seq** which cannot be interrupted by non-reactive statements) and reactive statements (which are executed only at **seq** composition points of transactions). Starting from this model define general concepts for mobile ASMs and use them for an investigation of concepts of mobile code in the literature.

[11] Dijkstra's guarded commands come with the same type of non-deterministic choice of one command per step.

COLD [197]. In the *C*ommon *O*bject-oriented *L*anguage for *D*esign states are realized as structures, including abstract data types (ADT) linked to an underlying dynamic logic proof system which is geared to provide proofs for algebraic specifications of states and their dynamics (à la Z and VDM). Computations are sequences of state transitions (due to the execution of procedure calls, built from statements viewed as expressions with side effects) allowing synchronous parallelism of simultaneous multiple conditional variable assignments (but no explicit **forall** construct) and non-deterministic choices among variable assignments and rules (procedure invocations). Thus a Cold class (with a set of states, one initial state, and a set of transition relations) corresponds in a standard way to a control state ASM, except that different states of a same class are allowed to have different signatures. The black-box view offered for sequencing and iteration is directly reflected by the corresponding turbo ASM constructs, taking into account that Cold provides a separate *guard statement* for blocking evaluation of guards which is executed only (with *skip* effect) when the guard becomes true.[12]

See the machine ColdModify(*Var*) on p. 40 for the idiomatic high-level construct *Mod* of Cold which supports non-determinism in choosing subsets of variables to be updated by chosen values. A similar construct *Use* permits one to choose procedures from a set *Proc* to be called in sequence.

```
ColdUse($Proc$) = choose $n \in \mathbb{N}$, choose $p_1, \ldots, p_n \in Proc$
  $p_1$ seq ... seq $p_n$
```

Parnas tables. An elaborate definition has been given in [360] for the semantics of a complex classification of Parnas tables which underlies the SCR method [276].[13] Their semantical meaning as a special matrix notation – a 2-dimensional layout of the CASE construct – for sequential systems with finitely many (controlled or monitored) state variables can be succinctly expressed by basic ASMs, providing an easily accessible foundation for the systematic use of such tables in system engineering. *Normal tables* (see Fig. 7.2) are used to express the assignment of a value $t_{i,j}$ to a dynamic function $f(x, y)$ under the ith row condition r_i and the jth column condition c_j (where it is assumed that for each x, y at most one pair of row and column condition is true); formally:

```
NormalTable = forall $i \leq m, j \leq n$
  if $RowCond_i$ and $ColumnCond_j$ then $f(x, y) := t_{i,j}$
```

Inverted tables (see Fig. 7.2) are used to assign a value t_j to $f(x, y)$ under a leading row condition and a side condition (assumed to be sufficiently

[12] See Sect. 4.2 for modeling blocking guards.

[13] The frame problem enters Parnas-table-based methods through the declarative x/x'-notation for values of variables x in a given state and their value x' in the next state, yielding the quickly overwhelming so-called NC-clauses (No Change).

Fig. 7.2 Normal and inverted Parnas tables

$N(f)$	c_1	$\cdots$	c_j	$\cdots$	c_n
r_1	$t_{1,1}$		$\ldots$		$t_{1,n}$
$\vdots$					
r_i			$t_{i,j}$		
$\vdots$					
r_m	$t_{m,1}$		$\ldots$		$t_{m,n}$

$I(f)$	t_1	$\cdots$	t_j	$\cdots$	t_n
r_1	$c_{1,1}$		$\ldots$		$c_{1,n}$
$\vdots$					
r_i			$c_{i,j}$		
$\vdots$					
r_m	$c_{m,1}$		$\ldots$		$c_{m,n}$

Fig. 7.3 Parnas decision tables

$D(f)$	t_1	$\cdots$	t_j	$\cdots$	t_n
s_1	$r_{1,1}$	$\ldots$	$r_{1,j}$	$\ldots$	$r_{1,n}$
$\vdots$	$\vdots$		$\vdots$		$\vdots$
s_m	$r_{m,1}$	$\ldots$	$r_{m,j}$	$\ldots$	$r_{m,n}$

disjoint as for normal tables, to prevent inconsistent ("ambiguous") function updates), formally described by the following rules (for all $i \leq n, j \leq m$):[14]

```
InvertedTable(i, j) = if RowCond_i(x, y) then
  if SideCond_{i,j} then f(x, y) := t_j
```

Decision tables (see Fig. 7.3) trigger a column action t_j under a parameterized column condition, formally expressed by the following set of rules (for all $j \leq n$, where disjoint properties avoid column actions conflicts):

```
DecisionTable(j) = if ∀i ≤ m RowCond_{i,j}(s_i) then trigger t_j
```

VDM, Z, B [199, 431, 5]. These three high-level design languages share the notion of computation as a sequence of state transitions given by a before-after relation, where states are formed by variables taking values in certain sets (in VDM built up from basic types by constructors) with explicitly or implicitly defined auxiliary functions and predicates. The single (in basic B sequencing-free and loop-free) transitions can be modeled in a canonical way by basic ASM rules which capture also the "unbounded" as well as the "bounded" choice and the parallelism B offers in terms of simultaneous ("multiple generalized") substitution. The basic scheme is determined by what Abrial calls the "pocket calculator model", which views a machine (program) as offering a set of operations (in VDM procedures with side effects) which are callable one at a time, e.g. in the non-deterministic form **choose** $R \in$ *Operation* **in** R or harnessed by a scheduler **let** $R =$

[14] Where useful it would probably be allowed to let $\mathit{SideCond}_{i,j}$ also depend on (x, y). A similar remark applies to other Parnas tables.

scheduled(*Operation*) **in** R;[15] similarly for events which in event-B are allowed to happen only one per time unit. The structuring mechanisms for large and refined B machines are captured by turbo ASMs, including also the mechanism operations typically come with to hide the machine state: it is allowed to activate (call) an operation for certain parameters, which results in an invariant-preserving state modification, but besides calling the operation and taking its result no other direct access to the state is granted. Historically, this view has led to a certain bias to functional modeling that one can observe in the use of VDM.

By the logical nature of Z specifications, their before-after expressions define the entire system dynamics. In B, as in the ASM method, the formulation of the system dynamics – in B by operations (in event-based B by events [6, 10, 11]), in ASMs by rules – is separated from the formulation of the static state invariants and of the dynamic run constraints, which express the desired system properties that one has to prove to hold through every possible state evolution. However for carrying out these proofs, in contrast to the ASM method,[16] there is a fixed link between B and a computer-assisted proof system[17] relating syntactical program constructs to proof rules which are used to establish program invariants and dynamic constraints along with the program construction. Thus, defining modules becomes intimately related to inventing lemmas. This fits also the basically axiomatic foundation of B as of Z and VDM, whose logical complexity tends to turn aside the attention of practitioners: VDM by a denotational semantics; Z by axiom systems formulated in (mainly first-order) logic; B by Dijkstra's weakest precondition theory, interpreted in set-theoretic models and based upon the syntactic global concept of substitution (from which local assignment $x := t$ and parallel composition are derived). In contrast to Z, which due to the purely axiomatic character of Z descriptions has intrinsic problems in turning specifications into executable code (see [267]), VDM and B are geared to obtain software modules from abstract specifications via refinements which are tai-

[15] This view points to a methodological difference between the forces which drove the development of the B method compared to that of the ASM method. Abrial's B method is the result of an engineer's bottom-up analysis: "The ideas behind the concept of abstract machine have all been borrowed from those ideas that are behind some well-known programming features such as modules, packages, abstract data types or classes" [6, p. 175]. Also the event-B notion of basic events, which corresponds to the guarded update rules of basic ASMs, came out of the concern to "separate assignments from scheduling". Gurevich's concept of ASMs is the result of a logician's top-down analysis, brought to light by a mathematical reflection on the ASM thesis (and turned by an extensive experimentation with practical applications of the concept into the basis of the system engineering method explained in this book; see Chap. 9 for the historical details).

[16] See Sect. 2.1 for an explanation of the methodological reasons. As to a pragmatic reason to consider, see the remark in [316, p. 8.1]: "A spec should be written to be as clear as possible to the clients, not to make it easy to prove the correctness of an implementation. The reason for these priorities is that we expect to have more clients for the spec than implementers."

lored to the proof rules used for proving that the refined operations satisfy "unchanged" properties of their abstract counterparts. For example when refining B machines, at most one operation of an included machine is allowed to be called from within an operation of the including machine [5, p. 317] (see Exercise 7.1.6 for the reason).

Petri Nets [372]. The general view of Petri nets is that of distributed transition systems transforming objects under given conditions. In Petri's classical version the objects are marks on *places* ("passive net components" where objects are stored), and the *transitions* ("active net components") modify objects by adding and deleting marks on the places or, more generally speaking, generating and consuming data, constituting the put-and-get pattern of Petri net computation steps. In modern instances (e.g. the predicate/transition nets) places are locations for objects belonging to abstract data types (read: variables taking values of given type, so that a marking becomes a variable interpretation), and transitions update variables and extend domains under conditions which are described by arbitrary first-order formulae. The distributed nature of Petri nets is captured by modeling them as async ASMs, as defined in Chap. 6, associating with each transition one *agent* to execute the transition. Each single transition is modeled by a basic ASM rule of the following form, where pre/post-places are sequences or sets of places which participate in the "information flow relation" (the local state change) due to the transition and *Cond* is an arbitrary first-order formula. By modeling Petri net states as ASM states we include the abstract Petri net view proposed in [372], where states are interpreted as logical predicates which are associated with places and transformed by actions.

$$\begin{array}{l}\textsc{PetriTransition} = \textbf{if } \mathit{Cond}(\mathit{prePlaces}) \textbf{ then } \mathit{Updates}(\mathit{postPlaces}) \\ \textbf{where } \mathit{Updates}(\mathit{postPlaces}) = \text{ a set of function updates}\end{array}$$

For ASM models of specific Petri nets which exploit the distributed character of such nets see Chap. 6.1.

Virtual machines. IBM's *Virtual Machine* [305] and Dijkstra's *Abstract Machine* [183] concept originated in the 1960s as a high-level operating system abstraction, but quickly spread to hierarchical system design in general, ranging from programming language platforms to layered software architectures, and nowadays has become ubiquitous in high-level system design. The definition of ASMs provides an explicit mathematical description of the class of machines covered by this concept and thus not surprisingly found quickly numerous applications for modeling complex virtual machines (e.g. Warren's

[17] The above-mentioned pocket-calculator view of abstract machines has a counterpart in Abrial's interesting concept of a "pocket prover", coming out of the concern about simple and natural moves for the human interaction with the prover via the screen, the mouse and the keyboard.

Abstract Machine [132] and its extensions [40, 133, 42, 41, 39], the Transputer [104], the RISC machine DLX [119], the Java Virtual Machine architecture [406], the Neural Net (abstract data flow) Machine [142], and the UPnP architecture [224]).

A small-size example of such a virtual machine is illustrated in Fig. 5.8 by the top level of the Neural Abstract Machine model defined in [142]. It reflects the view of a *Neural Net* as a black-box yielding output to the environment, as result of a hidden internal computation of the *Neural Kernel* submachine which is triggered by an input taken from the environment. The internal computation consists of an iteration of atomic actions, until there are no *moreUnitsToBeComputed*, performed by the basic computing units, which are the nodes of a directed data-flow graph which are scheduled for execution. In each unit computation step, the to-be-*sched*uled *Units* for the next execution step are determined by an external scheduler *nextExecUnits*, depending on the *inputType* which decides whether the net works in forward or in backward *mode*. In forward-propagation mode, the network input is transmitted by the *input units* to the *internal units*, which *update* their *local state* and *propagate* their result *value* through the graph until the *output units* are reached; similarly for the backward mode. Results are propagated to all destination or source units and thus eventually will reach the output or input units (depending on the forward or backward mode). This is summarized by the following refinement of the submachines in Fig. 5.8, assuming that *activateNeuralKernel* provides the correct initialization by the current *newInputToBeConsumed*.

```
$\textsc{NeuralKernelStep}(nextExecUnits) = \textbf{forall } u \in schedUnits$
    $computeUnit(u)$
    $schedUnits := nextExecUnits(schedUnits, inputType)$
$\textbf{where}$
$computeUnit(u) = \textbf{let } dir = inputType,\ result = Value(u, dir)$
    $updateLocalState(u, result, dir)$
    $propagate(u, result, dir)$
$propagate(u, data, forward) = \textbf{forall } d \in dest(u)$
    $inForward^{internal}(d, u) := internalValueForw(d, u, data)$
    $\textbf{if } u \in outputUnits \textbf{ then } output(u) := externalValueForw(u, data)$
$propagate(u, data, backward) = \textbf{forall } s \in source(u)$
    $inBackward^{internal}(s, u) := internalValueBack(s, u, data)$
    $\textbf{if } u \in inputUnits \textbf{ then } outputBack(u) := externalValueBack(u, data)$
```

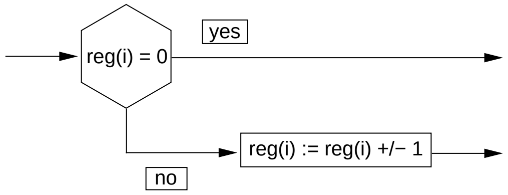
Logico-algebraic design systems. The fascinating idea of writing specifications as logical formulae with computations corresponding to logical deductions, furthermore in such a way that logical conjunction corresponds to system composition and logical implication to refinement, has led to a myriad of logic and algebraic specification and “declarative programming” languages and calculi. Examples are algebraic specification systems, Prolog and its numerous variants, VDM, Z.[18] the equational and rewriting logic system Maude (see http://maude.csl.sri.com), and innumerable "logics of programs" offering proof calculi to support verification of program properties.

An authoritative reference we disagree with is [280, p. 576] where it is maintained that "all the properties of a program and all the consequences of executing it can, in principle, be found out from the text of the program itself by means of purely deductive reasoning". It is certainly an advantage that once a complete logical specification is in place, a (possibly machine supported) proof for the desired program properties provides a rather high degree of reliability (assuming the implementation of the prover and its handling by the operating system are correct). Unfortunately this advantage has to be paid for at such a high price that, despite the longstanding worldwide research effort in this direction, logical specifications are simply not part of standard industrial software engineering practice (though they are used with success in certain well-delineated areas, among which are the design and verification of medium-size control systems, protocols and hardware [215, 159, 156, 307, 303, 196, 65]). This is so not only for the extraordinary cost of logical formalizations of real-life software projects, due to the considerable technico-mathematical skill and the time needed to carry to the end a large scale logico-algebraic design and verification project, but also for intrinsic reasons, three of which are to be mentioned here. All declarative specifications by their very logical nature are subject to the frame problem of having to describe not only the local changes, but also everything that is supposed not to change. Every logic system implies a fixed level of abstraction for design and verification. Most logic-based verification methods and tools "in dealing primarily with syntactic and structural aspects of software ... fail to address major issues of software quality having to do with *semantic* aspects of software", as one of the originators of the successful logic-based PVS system states [419, p. 345]. These three features lead to the rightly criticized "formal specification explosion phenomenon" that formal specifications which come in the form of a huge logical formula or system of algebraic equations tend to become orders of magnitude larger than the executable code, making it difficult (if possible at all) to fully understand them and to derive an efficient program from them. Minor but not negligible disadvantages derive from the typical external non-determinism in inference rule applications, which does not necessarily reflect the computationally intended scheduling,[19] and from

[18] The view expressed in [267, p. 89], that "the most important characteristic of Z, which singles it out from every other formal method, is that it is completely independent of any idea of computation", explains why Z specifications typically lead to abstract data types. The ASM method deliberately starts with a notion of computing state changes by destructive (though abstract) assignments.

[19] This is illustrated by the so-called SOS specifications of non-deterministic expression evaluation schemes, where the non-determinism of the describing and of the described systems match perfectly. However, when it comes to specifying

Fig. 7.4 Register machine rules



the natural drive of logical descriptions to lead to the rather special case of purely functional specifications (so-called "big-step semantics" with exclusive consideration of relations between initial and final states).[20]

The ASM method allows one to use such logic-based design and verification techniques *where appropriate* – desired, technically feasible and cost-effective – , integrating them into the high-level but state-based, genuinely semantical and computation-oriented specification and analysis techniques which are possible with ASMs. Successful projects in this direction have been reported using theorem proving systems (KIV, PVS, Isabelle) and model checkers, see e.g. [369, 229, 388, 439, 188, 187, 389, 386, 207], [424, 425, 428] and Chaps. 8.1 and 9 for details. This confirms the fallacious nature of the controversy which purports a dichotomy between using declarative methods (from logic or algebra) and operational methods (using state-based transition rules or code) to model and investigate complex systems. The most important methodological issue which decides the practicality of a design and analysis method is that of the levels of abstraction the method allows one to target and of the refinement schemes it offers to relate the models at different levels of abstraction.

7.1.3 Exercises

Exercise 7.1.1 (Variations of Turing machines). Extend the 1-tape Turing machine (TM) to a k-tape and to an n-dimensional TM by data refining the 1-tape Turing memory and the related operations and functions.

Exercise 7.1.2 (Register machines [70, Chap. AI1]). Formulate register machines (i.e. with rules as shown in Fig. 7.4) as instances of Turing-like

the semantics of expression evaluation in real-life languages with assignment expressions, like C or C++, the so-called natural semantics descriptions if provided at all tend to become unacceptably complex, due to the difficulties of describing by purely structural and deductive means the underlying order for the evaluation of subexpressions. See the analysis in [440].

[20] In [4] such "definite limitations" of the classical denotational paradigm are recognized: "... the appropriateness of modeling programs by functions is increasingly open to question. Neither concurrency nor 'advanced' imperative features have been captured denotationally in a fully convincing fashion".

machines. What has to be changed in your definition for the variant where the registers instead of numbers contain words?

Exercise 7.1.3 (Stream X-machines). (⇝ CD) Define a stream-processing version of Eilenberg's X-machines.

Exercise 7.1.4 (Parallel random access machine). Formulate PRAM models with exclusive and with concurrent writing. For the case of concurrency distinguish the purely non-deterministic case from the case where all processors seeking to write to a memory location must agree on the value to be written, from the priority model.

Exercise 7.1.5 (Schönhage storage modification machines). (⇝ CD) A Schönhage storage modification machine (SMM) has as memory a dynamic graph whose nodes n are named (not necessarily uniquely) by sequences of labels for edges, forming a path from a distinguished center node to n. Besides the usual instructions for control (Goto s, if input $= i$ goto s_i (for i=0,1), if $n = n'$ then s else s' conditioned by an equality test for node names) and instructions to write output symbols on an output tape, there are two characteristic instructions to create new nodes and to redirect edges between nodes: (1) new (n, e) redirects edge e from (the node named by) n to a new node which is linked (by an edge) to the same nodes that n is linked to, and (2) set e to n' redirects e to n$'$. Describe SMMs as ASMs using only 0-ary or unary dynamic functions, no static or shared function and only input as monitored function. (In [177] it is also shown that every ASM restricted in this way is lock-step equivalent to an SMM.)

Exercise 7.1.6 (Refinement restrictions in B). Consider the following rules which both satisfy the invariant $v \leq w$: $\textsc{Increment} \equiv$ **if** $v < w$ **then** $v := v + 1$, $\textsc{Decrement} \equiv$ **if** $v < w$ **then** $w := w - 1$. Show that the invariant will be broken at $w = v + 1$ by every machine which contains the rule **if** $v < w$ **then** $\{\textsc{Increment}, \textsc{Decrement}\}$.

Exercise 7.1.7. Define an ASM for one of the earliest virtual machines, Landin's SECD (Stack-Environment-Control-Dump) machine [317] for evaluating LISP constructs, where the stack is used to record intermediate expression evaluation results, the environment for the variable values, the control for the program counter, and the dump as a stack of states which are to be restored after the completion of the function applications.

7.2 Sequential ASM Thesis (A Proof from Postulates)

What became known as the ASM thesis was first formulated in 1985 by Gurevich in a note to the American Mathematical Society [241] where it reads as follows (dynamic structures stand for what nowadays are called ASMs):

> Every computational device can be simulated by an appropriate dynamic structure – of appropriately the same size – in real time.

In this section we derive the thesis from a few postulates for the case of so-called sequential algorithms:

> Every sequential algorithm can be step-for-step simulated by an appropriate sequential ASM.

By a sequential algorithm we mean a sequential-time, bounded parallel algorithm. The property *sequential-time* means that the computation steps of the algorithm are linearly ordered and that the algorithm is not multithreaded. By *bounded parallel* we mean that the algorithm is allowed to do operations in parallel. There exists, however, a bound on the amount of work the algorithm can do in parallel in one computation step. The bound depends on the algorithm only and not on the input.

7.2.1 Gurevich's Postulates for Sequential Algorithms

We now formulate three postulates and show that any algorithm (or computer system) that satisfies the three postulates can be step-for-step simulated by an ASM. Let A be an algorithm.

Postulate 1 (Sequential time). The algorithm A is associated with

– a set $\mathcal{S}(A)$, the set of *states* of A,
– a subset $\mathcal{I}(A) \subseteq \mathcal{S}(A)$, the set of *initial states* of A,
– a map $\tau_A \colon \mathcal{S}(A) \to \mathcal{S}(A)$, the *one-step transformation* of A.

The sequential-time postulate can be explained as follows. First, the states of an algorithm are considered to be the full instantaneous descriptions of the algorithm and not the internal *control states* or *modes* of the algorithm. A *run* (or *computation*) of an algorithm is a finite or infinite sequence $S_0, S_1, S_2, \ldots$ of states, where S_0 is an initial state, the transition from S_0 to S_1 is the first computation step, the transition from S_1 to S_2 is the second computation step, and so on. The computation steps are linearly ordered. The behavior of the algorithm is fully described by its one-step transformation τ_A.

Notice that the one-step transformation may consist of several distinct actions. For example, a Turing machine can change its control state, print a symbol at the current tape cell, and move its head, all in one step.

We call a state reachable, if it occurs in a run of A. The set of *reachable* states is a subset of $\mathcal{S}(A)$. The set of reachable states is fully determined by $\mathcal{I}(A)$ and τ_A. We do not assume that $\mathcal{S}(A)$ is the set of reachable states. Intuitively, it is the set of *a priori* states of the algorithm A, which is often much simpler than the set of reachable states.

One could associate a set $\mathcal{T}(A)$ of final states with the algorithm A. It would be natural then to restrict τ_A to $\mathcal{S}(A) \setminus \mathcal{T}(A)$. Final states are not needed in the main theorem below and therefore we assume that τ_A is total and $\tau_A(S) = S$ if the algorithm A terminates in state S.

Postulate 2 (Abstract state).

- The states of A are algebraic first-order structures.
- All states of A have the same finite signature.
- The one-step transformation τ_A does not change the base set of the state.
- $\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphisms.
- If $\mathfrak{A}, \mathfrak{B} \in \mathcal{S}(A)$ and α is an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$, then α is also an isomorphism from $\tau_A(\mathfrak{A})$ to $\tau_A(\mathfrak{B})$.

We discuss the abstract-state postulate as follows. The huge experience of mathematical logic and its applications indicates that any static mathematical situation can be faithfully described as a first-order structure. It is convenient to identify the states with the corresponding structures. The mathematical notion of structure is explained in detail in Sect. 2.4.1. The structures are called first-order, since they are used to give semantics of first-order logic (see Sect. 2.4.2 for details). When structures represent states of algorithms, they may contain "higher-order" objects like sets of elements, functions mapping elements to elements, etc. Such structures can be seen as special first-order structures.

It is assumed that the signatures of the states are finite. This reflects the assumption that the program of an algorithm A can be given by a finite text. The choice of the signature is dictated by the chosen abstraction level.

While the base set can change from one initial state to another, it does not change during the computation. All states of a given run have the same base set. If an algorithm needs new elements (new memory) at run-time, then the operating system usually provides new space. We assume that new elements come from a *reserve* which is already present in the initial state and that a special external function (which is part of the signature) fishes the new elements out of the reserve.

The last part of the abstract-state postulate reflects the fact that we are working on a fixed level of abstraction. An algorithm is not allowed to depend on the internal representations of a state. The details on which the algorithm depends must be present in the isomorphism type of the state. If the algorithm depends on the internal structure of the base set of a state, then the signature and the basic functions have to be readjusted.

Since the one-step transformation of an algorithm does not change either the base set of a state or its signature, the difference between a state $\mathfrak{A}$ and its successor state $\tau_A(\mathfrak{A})$ can be described as a set of non-trivial updates (for the definition of the difference between two structures see Def. 2.4.7).

Definition 7.2.1 (Delta). $\Delta(A, \mathfrak{A}) = \tau_A(\mathfrak{A}) - \mathfrak{A}$.

The difference $\Delta(A, \mathfrak{A})$ is always a consistent set of updates. When it is fired in state $\mathfrak{A}$, the successor state $\tau_A(\mathfrak{A})$ is obtained, i.e., $\mathfrak{A} + \Delta(A, \mathfrak{A}) = \tau_A(\mathfrak{A})$.

According to the abstract-state postulate an algorithm does not distinguish between isomorphic states. A state is just a particular implementation of its isomorphism type. An algorithm can only access elements of the base set via ground terms that contain functions from the signature of the state. Two states coincide over a set T of ground terms, if every term in T evaluates to the same value in both states.

Definition 7.2.2 (Coincide). Let T be a set of ground terms. Two states $\mathfrak{A}$ and $\mathfrak{B}$ coincide over T, if $[\![s]\!]^{\mathfrak{A}} = [\![s]\!]^{\mathfrak{B}}$ for each term $s \in T$.

The uniformly-bounded-exploration postulates says that an algorithm examines only a bounded number of elements in any state. There must exist a finite set T of ground terms – depending on the algorithm only and not on the initial state – such that the next computation step of the algorithm depends only on that part of the state which can be accessed via terms in T.

Postulate 3 (Uniformly bounded exploration). There exists a finite set T of ground terms such that whenever two states $\mathfrak{A}$ and $\mathfrak{B}$ coincide over T, then $\Delta(A, \mathfrak{A}) = \Delta(A, \mathfrak{B})$. The terms in T are called *critical* terms of A.

Example 7.2.1. For the Turing machines on p. 289 the critical terms are: $head$, $Nxtctl(ctl\_state, tape(head))$, $Write(ctl\_state, tape(head))$, $head + Move(ctl\_state, tape(head))$.

A consequence of the uniformly-bounded-exploration postulate is that the changes that have to be made to state $\mathfrak{A}$ in order to obtain the successor state $\tau_A(\mathfrak{A})$ can be described by critical terms. The arguments of locations that have to be updated as well as their new content are values of critical terms.

Lemma 7.2.1 (Critical terms). Let A be an algorithm that satisfies the three postulates. If $\mathfrak{A} \in \mathcal{S}(\mathfrak{A})$ and $((f, (a_1, \dots, a_n)), v)$ is an update in $\Delta(A, \mathfrak{A})$, then $a_1, \dots, a_n$ as well as v are values of critical terms of A.

Proof. By contradiction. Suppose that one of the arguments $a_1, \dots, a_n$ of the update or v is not the value of a critical term. Let α be the function that replaces this element in the base set of $\mathfrak{A}$ with a fresh element b. Let $\mathfrak{B}$ be the isomorphic image of $\mathfrak{A}$ under α. By the abstract-state postulate, it follows that $\mathfrak{B} \in \mathcal{S}(A)$ and that α is also an isomorphism from $\tau_A(\mathfrak{A})$ to $\tau_A(\mathfrak{B})$; hence $((f, (\alpha(a_1), \dots, \alpha(a_n)), \alpha(v))$ is an update in $\Delta(A, \mathfrak{B})$.

We claim that $\mathfrak{A}$ and $\mathfrak{B}$ coincide over T. Let $s \in T$. By Lemma 2.4.5, it follows that $\alpha([\![s]\!]^{\mathfrak{A}}) = [\![s]\!]^{\mathfrak{B}}$. Since $[\![s]\!]^{\mathfrak{A}}$ is not mapped to b by α, we have $[\![s]\!]^{\mathfrak{A}} = \alpha([\![s]\!]^{\mathfrak{A}}) = [\![s]\!]^{\mathfrak{B}}$.

By the uniformly bounded exploration postulate, it follows that $\Delta(A, \mathfrak{A})$ equals $\Delta(A, \mathfrak{B})$. Since one of the elements in $((f, (\alpha(a_1), \dots, \alpha(a_n)), \alpha(v))$ is the fresh element b, it follows that b occurs also in an update of $\Delta(A, \mathfrak{A})$. This gives the desired contradiction. □

Thus every update in $\Delta(A, \mathfrak{A})$ can be programmed by an update rule $f(s_1, \ldots, s_n) := t$, where the terms $s_1, \ldots, s_n$ and t are critical terms of A. Let $P_A^{\mathfrak{A}}$ be the parallel combination of all these update rules:

$$P_A^{\mathfrak{A}} = \{f(s_1, \ldots, s_n) := t \mid s_1, \ldots, s_n, t \in T, \\ ((f, [\![s_1]\!]^{\mathfrak{A}}, \ldots, [\![s_n]\!]^{\mathfrak{A}}), [\![t]\!]^{\mathfrak{A}}) \in \Delta(A, \mathfrak{A})\}$$

Since T is finite and the signature of $\mathfrak{A}$ is finite, the set $P_A^{\mathfrak{A}}$ is finite, too. By Lemma 7.2.1, we obtain that the update set that is computed by $P_A^{\mathfrak{A}}$ in state $\mathfrak{A}$ is exactly the update set of the one-step transformation τ_A.

Corollary 7.2.1. $P_A^{\mathfrak{A}}$ yields the update set $\Delta(A, \mathfrak{A})$ in each state of A.

For two different states $\mathfrak{A}$ and $\mathfrak{B}$ the programs $P_A^{\mathfrak{A}}$ and $P_A^{\mathfrak{B}}$ can be quite different. However, if $\mathfrak{A}$ and $\mathfrak{B}$ are *similar* with respect to the values of critical terms, then the two programs are equal, as we will see below in Corollary 7.2.2. Two states are called T-similar, if they satisfy the same equations between terms in T.

Definition 7.2.3 (Similar). Let T be a set of ground terms. A state $\mathfrak{A}$ is T-similar to $\mathfrak{B}$ iff for all terms $s, t \in T$: $\mathfrak{A} \models s = t \iff \mathfrak{B} \models s = t$.

If two states coincide over T, then they are T-similar. The converse is in general not true. If two states are T-similar, they can have disjoint universes and therefore they may not coincide over T. However, there always exists an isomorphic state which coincides over T with the other state.

Lemma 7.2.2 (Similarity). Let T be a set of ground terms. If $\mathfrak{A}$ and $\mathfrak{B}$ are T-similar, then there exists a state $\mathfrak{C}$ which is isomorphic to $\mathfrak{A}$ and coincides with $\mathfrak{B}$ over T.

Proof. We can assume that the base set of $\mathfrak{A}$ is disjoint from the base set of $\mathfrak{B}$. Otherwise we take an isomorphic copy of $\mathfrak{A}$ with no elements from $\mathfrak{B}$. Let α be the function that replaces in the base set of $\mathfrak{A}$ the values of terms of T with their corresponding value in $\mathfrak{B}$, i.e., $\alpha([\![s]\!]^{\mathfrak{A}}) = [\![s]\!]^{\mathfrak{B}}$ for each term s in T, and $\alpha(a) = a$ if a is not the value of a term of T in $\mathfrak{A}$. Since $\mathfrak{A}$ and $\mathfrak{B}$ are T-similar and $|\mathfrak{A}|$ is disjoint from $|\mathfrak{B}|$, the function α is well-defined and a bijection. Let $\mathfrak{C}$ be the isomorphic image of $\mathfrak{A}$ under α. We claim that $\mathfrak{C}$ coincides with $\mathfrak{B}$ over T. Let $s \in T$. By Lemma 2.4.5, we have $[\![s]\!]^{\mathfrak{C}} = \alpha([\![s]\!]^{\mathfrak{A}})$. Since $\alpha([\![s]\!]^{\mathfrak{A}}) = [\![s]\!]^{\mathfrak{B}}$ by the definition of α, it follows that $[\![s]\!]^{\mathfrak{C}} = [\![s]\!]^{\mathfrak{B}}$. □

Corollary 7.2.2. If $\mathfrak{A}$ and $\mathfrak{B}$ are T-similar, where T is the set of critical terms of A, then $P_A^{\mathfrak{A}} = P_A^{\mathfrak{B}}$.

Proof. By Lemma 7.2.2 there exists a state $\mathfrak{C} \in \mathcal{S}(A)$ that is isomorphic to $\mathfrak{A}$ and coincides with $\mathfrak{B}$ over T. Let α be an isomorphism from $\mathfrak{A}$ to $\mathfrak{C}$. By the abstract-state postulate it follows that $\alpha(\Delta(A, \mathfrak{A})) = \Delta(A, \mathfrak{C})$. Hence, $P_A^{\mathfrak{A}} = P_A^{\mathfrak{C}}$. Since the states $\mathfrak{C}$ and $\mathfrak{B}$ coincide over T, by the uniformly-bounded-exploration postulate, $\Delta(A, \mathfrak{C}) = \Delta(A, \mathfrak{B})$. Hence, $P_A^{\mathfrak{C}} = P_A^{\mathfrak{B}}$. □

The main theorem now uses the fact that the set T of critical terms is finite and that there exist only finitely many different similarity types with respect to T. Moreover, the similarity type of a state can be described by a finite boolean combination of equations betweenthe terms of T.

Theorem 7.2.1 (Sequential ASM thesis [249]). If an algorithm A satisfies the sequential-time postulate, the abstract-state postulate and the uniformly-bounded-exploration postulate, then the one-step transformation of A can be programmed by an ASM of the following kind:

```
**if** $\varphi_1$ **then** $f_1(s_{1,1},\ldots,s_{1,n_1}) := t_1$
$\vdots$
**if** $\varphi_k$ **then** $f_k(s_{k,1},\ldots,s_{k,n_k}) := t_k$
```

The guards φ_i are boolean combinations of equations between terms.

Proof. Let T be the set of critical terms of A. For a state $\mathfrak{B}$ let $\varphi_{\mathfrak{B}}$ be the boolean combination of equations and negated equations between critical terms that characterizes its similarity type:

$$\varphi_{\mathfrak{B}} = \bigwedge_{\substack{s,t\in T\\ \mathfrak{B}\models s=t}} s = t \quad \wedge \bigwedge_{\substack{s,t\in T\\ \mathfrak{B}\not\models s=t}} \neg(s = t)$$

Since T is finite, there exists a finite set $\{\mathfrak{A}_1,\ldots,\mathfrak{A}_n\}$ of states of A such that every state of A is T-similar to one of the states $\mathfrak{A}_i$. The ASM that computes the one-step transformation τ_A of A is the following parallel combination of guarded updates:

```
**if** $\varphi_{\mathfrak{A}_1}$ **then** $P_A^{\mathfrak{A}_1}$
$\vdots$
**if** $\varphi_{\mathfrak{A}_k}$ **then** $P_A^{\mathfrak{A}_k}$
```

Let $\mathfrak{B}$ be a state of A. There exists a state $\mathfrak{A}_i$ such that $\mathfrak{B}$ is T-similar to $\mathfrak{A}_i$. Hence, $\mathfrak{B}$ satisfies the guard $\varphi_{\mathfrak{A}_i}$. By Corollary 7.2.1, it follows that $P_A^{\mathfrak{B}}$ yields the update set $\Delta(A,\mathfrak{B})$ in state $\mathfrak{B}$. By Corollary 7.2.2, it follows that $P_A^{\mathfrak{B}} = P_A^{\mathfrak{A}_i}$. Therefore, the ASM computes the update set $\Delta(A,\mathfrak{B})$ which is defined as $\tau_A(\mathfrak{B}) - \mathfrak{B}$. In other words, the ASM can make a move from $\mathfrak{B}$ to $\tau_A(\mathfrak{B})$ for each $\mathfrak{B} \in \mathcal{S}(A)$. Hence, the ASM simulates the algorithm A step-for-step. □

So far, the algorithm A has been deterministic. The postulates and the main theorem, however, can be extended to non-deterministic algorithms.

7.2.2 Bounded-Choice Non-Determinism

Non-deterministic algorithms are useful, for example, as higher-level descriptions of complicated deterministic algorithms. There are two possibilities for adding non-determinism to algorithms. In the first approach, the choices are made by the environment via monitored functions that can be viewed as external oracles. This approach is already covered by Theorem 7.2.1.

Sometimes it is convenient to pretend that the choices are made by the algorithms themselves rather than by the environment. In that case the one-step transformation τ_A of an algorithm is no longer a function from $\mathcal{S}(A)$ to $\mathcal{S}(A)$ but a binary relation on $\mathcal{S}(A)$, i.e. $\tau_A \subseteq \mathcal{S}(A) \times \mathcal{S}(A)$. The abstract-state postulate has to be changed as follows:

- If $(\mathfrak{A}, \mathfrak{B}) \in \tau_A$, then the base set of $\mathfrak{B}$ is that of $\mathfrak{A}$.
- If $\mathfrak{A}, \mathfrak{B} \in \mathcal{S}(A)$ and α is an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$, then for every state $\mathfrak{A}'$ with $(\mathfrak{A}, \mathfrak{A}') \in \tau_A$ there exists a state $\mathfrak{B}'$ with $(\mathfrak{B}, \mathfrak{B}') \in \tau_A$ such that α is an isomorphism from $\mathfrak{A}'$ to $\mathfrak{B}'$.

The $\Delta(A, \mathfrak{A})$ is no longer a single update set but a set of possible update sets. For each state $\mathfrak{B}$ that is a possible successor state of $\mathfrak{A}$ according to τ_A, the difference $\mathfrak{B} - \mathfrak{A}$ is added to $\Delta(A, \mathfrak{A})$:

$$\Delta(A, \mathfrak{A}) = \{\mathfrak{B} - \mathfrak{A} \mid (\mathfrak{A}, \mathfrak{B}) \in \tau_A\}$$

The uniformly-bounded-exploration postulate can then be used unchanged.

If a non-deterministic algorithm satisfies the non-deterministic versions of the sequential-time, the abstract-state and the uniformly-bounded-exploration postulates, then there exists a natural number k such that for each state $\mathfrak{A} \in \mathcal{S}(A)$ the set $\{\mathfrak{B} \mid (\mathfrak{A}, \mathfrak{B}) \in \tau_A\}$ has at most k members. This can be seen as follows. As in the deterministic case, each element that is used in an update set of $\Delta(A, \mathfrak{A})$ can be described by a critical term (cf. Lemma 7.2.1). Since the set of critical terms is finite, there exists a bound for the number of update sets in $\Delta(A, \mathfrak{A})$ that depends on the algorithm A only and not on the state $\mathfrak{A}$. The main theorem 7.2.1 remains valid except that the ASMs $P_A^{\mathfrak{A}}$ now have the form

```
choose $i \in \{1, 2, \ldots, k\}$ do
  if $i = 1$ then $R_1$
  ⋮
  if $i = k$ then $R_k$
```

where the transition rules R_i consist of a finite set of parallel updates.

7.2.3 Critical Terms for ASMs

An ASM transition rule P of the signature Σ can be seen as an algorithm. The states of P are arbitrary structures for Σ and all states are initial. The one-step transformation of P is defined by $\tau_P(\mathfrak{A}) = \mathfrak{A} + [\![P]\!]^{\mathfrak{A}}$ (we assume that P does not contain **choose** and hence P yields a unique update set $[\![P]\!]^{\mathfrak{A}}$ in state $\mathfrak{A}$). It is obvious that P satisfies the sequential-time and the abstract-state postulate. The last part of the abstract-state postulate follows from Lemma 2.4.2 and 2.4.12.

The transition rule P, however, may not satisfy the uniformly-bounded-exploration postulate, e.g. if P contains **forall** or P contains quantifiers. For example, the ShortestPath algorithm on p. 82 is highly parallel and not sequential. The work performed by the algorithm in one step cannot be bounded by a finite set of ground terms:

$$\textbf{forall } x, y \textbf{ with } E(x, y) \land \mathit{visited}(x) \land \neg \mathit{visited}(y) \textbf{ do } \mathit{visited}(y) := \mathit{true}$$

If we allow quantifiers in the guards of ASM rules, we can check in one step whether a finite graph is a clique (complete graph):

$$\textbf{if } \forall x\, \forall y\, E(x, y) \textbf{ then } \mathit{clique} := \mathit{true} \textbf{ else } \mathit{clique} := \mathit{false}$$

The algorithm changes only one boolean 0-ary function *clique*, but it explores the whole graph in one step and it is not uniformly-bounded.

ASM rules without **forall** and without quantifiers satisfy the uniformly-bounded-exploration postulate. The proof of the following lemma shows how we can obtain the set of critical terms. Together with Theorem 7.2.1 on the sequential ASM thesis we can even use the lemma to eliminate **let** and **seq** and to obtain normal forms of transition rules. Notice that in the presence of **seq**, the rule **let** $x = t$ **in** P is in general not equivalent to $P\frac{t}{x}$, since the term t may contain dynamic functions that have different interpretations at different occurrences of x in P.

Since in the presence of **let** not all terms are ground, we have to extend the notion "two states coincide over T" to sets T of terms that may contain variables.

Definition 7.2.4. Let T be a set of terms and ζ be a variable assignment. Two states $\mathfrak{A}$ and $\mathfrak{B}$ coincide over T under ζ, if $\zeta(x) \in |\mathfrak{A}| \cap |\mathfrak{B}|$ for every variable x occurring in a term of T and $[\![s]\!]^{\mathfrak{A}}_{\zeta} = [\![s]\!]^{\mathfrak{B}}_{\zeta}$ for each term $s \in T$.

Lemma 7.2.3 (Critical terms for ASMs). If an ASM rule is composed of updates and **skip** using **par**, **if-then-else** (with quantifier-free guards), **let** and **seq**, then it satisfies the uniformly-bounded-exploration postulate.

Proof. We show that for each such transition rule P, there exists a finite set T of terms such that the variables of T are free variables of P and the following two conditions hold:

1. If P yields the update set U in state $\mathfrak{A}$ under ζ and $\mathfrak{A}$ coincides with $\mathfrak{B}$ over T under ζ , then P yields U in $\mathfrak{B}$ under ζ.
2. If P yields U in $\mathfrak{A}$ under ζ, then every element occurring in an update of U is a value of a term from T in $\mathfrak{A}$ under ζ, i.e. $\mathit{El}(U) \subseteq \{[\![s]\!]^{\mathfrak{A}}_{\zeta} \mid s \in T\}$.

The proof is by induction on the size of the transition rule P.

Case 1. $f(s_1, \dots, s_n) := t$: We choose $T = \{s_1, \dots, s_n, t\}$.

Case 2. P **par** Q: By the induction hypothesis, we obtain sets T_P for P and T_Q for Q. We choose $T = T_P \cup T_Q$.

Case 3. **if** φ **then** P **else** Q: By the induction hypothesis, we obtain sets T_P for P and T_Q for Q. We choose

$$T = T_P \cup T_Q \cup \{s, t \mid s = t \text{ is an equation in } \varphi\}.$$

If $\mathfrak{A}$ and $\mathfrak{B}$ coincide over T under ζ, then $[\![\varphi]\!]_\zeta^{\mathfrak{A}} = [\![\varphi]\!]_\zeta^{\mathfrak{B}}$, since φ is quantifier-free and all terms occurring in atomic equations of φ have the same values in $\mathfrak{A}$ and $\mathfrak{B}$ under ζ.

Case 4. **let** $x = t$ **in** P: by the induction hypothesis, we obtain a term set T_P for P. We choose $T = \{s[t/x] \mid s \in T_P\} \cup \{t\}$, i.e. we substitute the term t for x in every term of T_P and finally add t to the set. Assume that $\mathfrak{A}$ and $\mathfrak{B}$ coincide over T under ζ. Let $a = [\![t]\!]_\zeta^{\mathfrak{A}} = [\![t]\!]_\zeta^{\mathfrak{B}}$. Then $\mathfrak{A}$ and $\mathfrak{B}$ coincide over T_P under $\zeta[x \mapsto a]$ (Lemma 2.4.6). The rule **let** $x = t$ **in** P yields U in $\mathfrak{A}$ under ζ iff P yields U in $\mathfrak{A}$ under $\zeta[x \mapsto a]$.

Case 5. P **seq** Q: By the induction hypothesis, we obtain term sets T_P for P and T_Q for Q. We choose $T = T_P \cup T_Q^{T_P}$ (the exponentiation is defined below). Assume that P yields a consistent update set U in $\mathfrak{A}$ under ζ and Q yields V in $\mathfrak{A} + U$ under ζ. Hence, P **seq** Q yields $U \oplus V$ in $\mathfrak{A}$ under ζ. Assume that $\mathfrak{A}$ and $\mathfrak{B}$ coincide over T under ζ. By the induction hypothesis, we know that P yields U in $\mathfrak{B}$ under ζ and $El(U) \subseteq \{[\![s]\!]_\zeta^{\mathfrak{A}} \mid s \in T\}$. Hence we can apply Lemma 7.2.4 below and obtain that $\mathfrak{A} + U$ and $\mathfrak{B} + U$ coincide over T_Q under ζ. By the induction hypothesis, it follows that Q yields V in $\mathfrak{B} + U$ under ζ. Hence, P **seq** Q yields $U \oplus V$ in $\mathfrak{B}$ under ζ. □

What remains to be done is to define the exponentiation S^T for sets of terms S and T. First we denote by s^T the set of terms that are obtained by replacing some subterms in s by terms of T. The set s^T can be defined by induction on the size of s:

$$x^T = \{x\}$$
$$c^T = \{c\} \cup T$$
$$f(s_1, \dots, s_n)^T = \{f(r_1, \dots, r_n) \mid r_i \in s_i^T \text{ for } i = 1, \dots, n\} \cup T$$

The term set s^T is finite, if T is finite, and $s \in s^T$. For a set S of terms we define $S^T = \{s^T \mid s \text{ is subterm of a term of } S\}$. Finally we have to prove the following technical lemma that yields the properties of S^T that are needed to compute the critical terms for the sequential composition of transition rules.

Lemma 7.2.4. If $\mathfrak{A}$ and $\mathfrak{B}$ coincide over $T \cup S^T$ under ζ and U is a consistent update set such that all elements occurring in updates of U are values of terms from T in $\mathfrak{A}$ under ζ, then for each term $s \in S$ there exists a term $t \in s^T$ such that $[\![s]\!]_\zeta^{\mathfrak{A}+U} = [\![t]\!]_\zeta^{\mathfrak{A}} = [\![t]\!]_\zeta^{\mathfrak{B}} = [\![s]\!]_\zeta^{\mathfrak{B}+U}$.

Proof. We show by induction on the length of a subterm s of S that there exists a term $r \in s^T$ such that $[\![s]\!]_\zeta^{\mathfrak{A}+U} = [\![r]\!]_\zeta^{\mathfrak{A}} = [\![r]\!]_\zeta^{\mathfrak{B}} = [\![s]\!]_\zeta^{\mathfrak{B}+U}$.

Let $f(s_1, \dots, s_n)$ be a subterm (of a term) of S.

By the induction hypothesis, we obtain terms $r_i \in s_i^T$ with

$$[\![s_i]\!]_\zeta^{\mathfrak{A}+U} = [\![r_i]\!]_\zeta^{\mathfrak{A}} = [\![r_i]\!]_\zeta^{\mathfrak{B}} = [\![s_i]\!]_\zeta^{\mathfrak{B}+U} \quad \text{for } i = 1, \dots, n.$$

Let l be the location $(f, ([\![s_1]\!]_\zeta^{\mathfrak{A}+U}, \dots, [\![s_n]\!]_\zeta^{\mathfrak{A}+U}))$.

Case 1. There exists an update for l in U:

By assumption, there exists a term $t \in T$ such that $(l, [\![t]\!]_\zeta^{\mathfrak{A}}) \in U$. We have:

$$[\![f(s_1, \dots, s_n)]\!]_\zeta^{\mathfrak{A}+U} = [\![t]\!]_\zeta^{\mathfrak{A}} = [\![t]\!]_\zeta^{\mathfrak{B}} = [\![f(s_1, \dots, s_n)]\!]_\zeta^{\mathfrak{B}+U}$$

By definition, the term t belongs to $f(s_1, \dots, s_n)^T$.

Case 2. There is no update for l in U: Then we have

$$\begin{array}{rcl}
[\![f(s_1, \dots, s_n)]\!]_\zeta^{\mathfrak{A}+U} & = & f^{\mathfrak{A}+U}([\![s_1]\!]_\zeta^{\mathfrak{A}+U}, \dots, [\![s_n]\!]_\zeta^{\mathfrak{A}+U}) \\
& = & f^{\mathfrak{A}}([\![r_1]\!]_\zeta^{\mathfrak{A}}, \dots, [\![r_n]\!]_\zeta^{\mathfrak{A}}) \\
& = & [\![f(r_1, \dots, r_n)]\!]_\zeta^{\mathfrak{A}} \\
& = & [\![f(r_1, \dots, r_n)]\!]_\zeta^{\mathfrak{B}} \\
& = & f^{\mathfrak{B}}([\![r_1]\!]_\zeta^{\mathfrak{B}}, \dots, [\![r_n]\!]_\zeta^{\mathfrak{B}}) \\
& = & f^{\mathfrak{B}+U}([\![s_1]\!]_\zeta^{\mathfrak{B}+U}, \dots, [\![s_n]\!]_\zeta^{\mathfrak{B}+U}) \\
& = & [\![f(s_1, \dots, s_n)]\!]_\zeta^{\mathfrak{B}+U}
\end{array}$$

Since the term $f(r_1, \dots, r_n)$ belongs to $f(s_1, \dots, s_n)^T$, the proof is completed. □

Problem 25 (Linear time lower bounds). In [56] it is shown that there exists a lock-step universal sequential ASM U, i.e. for a constant c and under honest time counting, U simulates every other sequential ASM in lock-step with log factor c. The proof method yields linear-time hierarchy theorems for random-access machines and ASMs. Use this linear-time hierarchy method to establish linear lower bounds for interesting linear time problems.

Problem 26 (Computational complexity with respect to abstract data types). Use ASMs to analyze the complexity of algorithms directly in terms of their underlying data structure, to make the relation explicit between the purely conceptual ("algorithmic") complexity and the complexity due to encodings into specific data structures.

Problem 27 (Framework for synchronous languages). In [61] an axiomatic characterization of synchronous parallel computations is proposed. It uses a notion of "sequential subprocesses" called *proclets* ("subprocesses so small that they no longer involve unbounded parallelism") which work in parallel, communicating with each other by messages, to compute one overall synchronous parallel step. The axiomatic description, resulting from a general epistemological analysis, is illustrated by classic examples, notably parallel random-access machines, alternating Turing machines, and Boolean circuits. These machines are easily modeled directly without going through proclets. Compare for example the verbal description of the alternating Turing machine in [61, Sect. 8.3] with the explicit composition of AlternatingTm (p. 290) out of the ordinary TuringMachine and of AltTmSpawn – a simple refinement of the standard subprocess creating machine OccamParSpawn (p. 43) – as components. It would be interesting to try out proclets (or maybe use sync ASMs?) to model in a real-time context the synchronous parallelism of synchronous languages [266], e.g. Esterel [49] where an overall computation step involving substeps of communicating parallel processes is described by a fixpoint computation (to be guaranteed by program properties which are checked by the compiler). Evaluate the ASM-based and the fixpoint-based analysis, in particular concerning a transparent model for the semantics-preserving implementation of synchronous parallel languages.

7.2.4 Exercises

Exercise 7.2.1. (⇝ CD) Let T be the set of critical terms of an algorithm A that satisfies the sequential-time, the abstract-state and the uniformly-bounded-exploration postulate. Show that the following is *not* true: if two states $\mathfrak{A}$ and $\mathfrak{B}$ of A coincide over T, then $\tau_A(\mathfrak{A})$ and $\tau_A(\mathfrak{B})$ also coincide over T.

Sources and Historical Remarks

The ASM models in Sect. 7.1 are taken from [86, 91, 93]. Section 7.2 is based on [249]. For a detailed historical account and a discussion of the methodological role of the ASM thesis see Chap. 9.

8 Tool Support for ASMs

In this chapter we discuss the various forms of tool support for the analysis of ASMs, namely by mechanical verification systems and by environments to refine ASMs into executable programs one can use for validation purposes.

8.1 Verification of ASMs

In the preceding chapters numerous proofs have been given to illustrate how one can verify dynamic properties for given ASMs by exploiting the fact that ASMs are not logical formulae, but machines coming with a notion of run which lends itself to traditional inductive arguments. The proofs range from simple to more challenging ones and are of a mathematical nature, providing for *human experts*, of the considered application domain or of software design, arguments which are deemed to be rigorous and complete enough to represent a verification of the claims of interest. Certainly such proofs provide *no absolute guarantee*, they may be incomplete or contain flaws. If the complexity is not prohibitive, the proofs may be detailed further to become still more trustworthy, e.g. by formalizations in appropriate logical calculi or for mechanical machine-supported verification. This will entail a considerably higher effort and cost, which the development manager should be aware of; see the evaluation in [65] and the positive experience with the industrial use of the B method [37].

There is no obstacle of principle to applying mechanical theorem proving and model checking systems to verify the properties of ASMs. In fact KIV, Isabelle, PVS and model checkers have been successfully used in this way (for verifying correctness properties for compilers, architectures, protocols, control programs etc., as has been pointed out in the preceding chapters) and numerous logics have been developed to deal with specific features of ASM verification (see Sect. 9.4.3 for detailed references). In Sect. 8.1.1 we present the unifying logic developed in [405], which is tailored for ASMs in terms of an atomic predicate for function updates (together with a definedness predicate for the termination of the evaluation of turbo ASMs). This chapter can be read independently of the others except for the definition of ASMs in Sect. 2.4, but the reader is supposed to have a basic knowledge of first-order logic. In Sect. 8.2 we briefly explain the basic transformation of ASMs to FSMs which underlies the applications of model checking to ASMs.

8.1.1 Logic for ASMs

The logic for ASMs that we describe in this section differs from other logics that have been proposed for ASMs in two points: (i) it is complete for the class of ASMs that do not contain cycles in the dependency graph of rule declarations (so-called hierarchical ASMs); (ii) it can be applied also to turbo ASMs where the evaluation of transition rules might not terminate.

The reader may wonder why a logic for a computationally universal mechanism like hierarchical ASMs can be complete at all? The answer is that the logic is not able to talk about full ASM runs. Instead, the logic talks about single steps of an ASM and therefore the logic is complete for statements about the single steps of an ASM like the invariants of rules, the consistency conditions for rules, or the step-for-step equivalence of rules. Moreover, the completeness theorem holds for the uninterpreted logic, where the static functions do not have a fixed standard interpretation. Hence, the logic is complete in the same sense as first-order logic (FOL) is complete. In fact, it turns out that the logic for ASMs is a *definitional extension* of FOL. This means that the formulas of the rich language of the logic for ASMs (including the modal operators and a special predicate for function updates) can be translated into pure first-order formulas.

If we allow cycles in the dependency graph of rule declarations (as it is allowed for turbo ASMs), then the logic cannot be complete, since iteration and while-loops can be recursively defined and therefore it is possible to talk about the outcome of full ASMs runs. The same arguments, that are used to show that the dynamic logic is not complete, can be applied in this case.

Since ASMs are special instances of transition systems, the logic contains modal operators. We do not, however, stick to modal logic, since in some situations it can be more convenient and even more economical to use functions with an additional argument that specifies the nth state of the run of an ASM (explicit time versus implicit time). This has been shown to be useful in correctness proofs of ASM refinements (see Sect. 3.2.2).

What comes closest to the logic is known as *dynamic logic with array assignments* [268, 269]. In the dynamic logic with array assignments, the programs are sequential while-programs that manipulate arrays. There is no notion of parallel execution. Hence, the dynamic logic with array assignments is not concerned with the parallel execution of assignments and therefore does not need a notion of consistency for programs. The substitution principle which is used in its axiomatization is derivable (see Lemma 8.1.11).

For reasons that we explain below in Sect. 8.1.4 we exclude the **choose** construct from the logic. This means that the transition rules that are allowed in ASMs of the logic are deterministic. It does not mean that everything has to be deterministic. The logic can still be used to prove the properties of interactive ASMs where the environment updates certain locations, as long as the assumptions about the environment can be formulated in FOL.

8.1.2 Formalizing the Consistency of ASMs

We let the letters P, Q, R range over the transition rules of ASMs as defined in detail in Sect. 2.4.3. The transition rules are not allowed to contain the **choose** construct. They may, however, contain a version of **try-else**. The meaning of **try** P **else** Q is: execute P; if it yields a consistent update set, return the update set and do not execute Q at all; otherwise, execute Q and the result of the whole transition rule is the evaluation of Q. In terms of the calculus for the “yields” predicate in Table 2.2:

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U)}{\text{yields}(\textbf{try } P \textbf{ else } Q, \mathfrak{A}, \zeta, U)} \qquad \text{if } U \text{ is consistent}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(\textbf{try } P \textbf{ else } Q, \mathfrak{A}, \zeta, V)} \qquad \text{if } U \text{ is inconsistent}$$

We extend the language of first-order predicate logic (see Sect. 2.4.2) by a modal operator $[R]$ for each transition rule R. The intended meaning of a formula $[R]\varphi$ is that the formula φ is true after firing R. More precisely, the formula $[R]\varphi$ is true iff one of the following conditions is satisfied:

1. R is not defined or the update set of R is inconsistent, or
2. R is defined, the update set of R is consistent and φ is true in the next state after firing the update set of R.

Equivalently we can say that the formula $[R]\varphi$ is true in state $\mathfrak{A}$ under the variable assignment ζ iff for each update set U such that R yields U in $\mathfrak{A}$ under ζ and U is consistent, the formula φ is true in the state $\mathfrak{A} + U$ under ζ (see Table 8.1).

In order to express the definedness and the consistency of the transition rules we extend the set of formulas by the atomic formulas $\mathsf{def}(R)$ and $\mathsf{upd}(R, f, x, y)$. The semantics of these formulas is defined in Table 8.1. The formula $\mathsf{def}(R)$ asserts that the rule R is defined. The rule R is defined in a state $\mathfrak{A}$ under a variable assignment α, if there exists an update set U (consistent or contradictory) such that R yields U in $\mathfrak{A}$ under ζ according to Table 2.2. The formula $\mathsf{upd}(R, f, x, y)$ asserts that rule R is defined and yields an update set which contains an update for f at x to y. The argument x of the function f has to be read as a finite vector of variables.

The basic properties of def and upd are listed in Table 8.2 and Table 8.3. Note that the equivalences in Table 8.2 and Table 8.3 are not the *definitions* of def and upd. The equivalences are just properties which are true under the interpretation of def and upd given in Table 8.1. The equivalences cannot be considered as definitions, since D9 and U9 depend on the rule declarations of the given ASM and the call graph of the rule definitions may contain cycles.

Table 8.1 The semantics of modal formulas and basic predicates

$$[\![\,[R]\varphi\,]\!]_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } [\![\varphi]\!]_\zeta^{\mathfrak{A}+U} = \mathit{true} \text{ for each consistent update set } U \\ & \text{such that yields}(R,\mathfrak{A},\zeta,U) \text{ is derivable in Table 2.2;} \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$[\![\mathsf{def}(R)]\!]_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if there exists an update set } U \text{ such that} \\ & \text{yields}(R,\mathfrak{A},\zeta,U) \text{ is derivable in Table 2.2;} \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$[\![\mathsf{upd}(R,f,s,t)]\!]_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if there exists an update set } U \text{ such that} \\ & \text{yields}(R,\mathfrak{A},\zeta,U) \text{ and } ((f,[\![s]\!]_\zeta^{\mathfrak{A}}),[\![t]\!]_\zeta^{\mathfrak{A}}) \in U; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

Table 8.2 Axioms for definedness

D1. $\mathsf{def}(\textbf{skip})$

D2. $\mathsf{def}(f(s) := t)$

D3. $\mathsf{def}(P \textbf{ par } Q) \leftrightarrow \mathsf{def}(P) \land \mathsf{def}(Q)$

D4. $\mathsf{def}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q) \leftrightarrow (\varphi \land \mathsf{def}(P)) \lor (\neg\varphi \land \mathsf{def}(Q))$

D5. $\mathsf{def}(\textbf{let } x = t \textbf{ in } P) \leftrightarrow \exists x\,(x = t \land \mathsf{def}(P))$ if $x \notin \mathrm{FV}(t)$

D6. $\mathsf{def}(\textbf{forall } x \textbf{ with } \varphi \textbf{ do } P) \leftrightarrow \forall x\,(\varphi \to \mathsf{def}(P))$

D7. $\mathsf{def}(P \textbf{ seq } Q) \leftrightarrow \mathsf{def}(P) \land [P]\mathsf{def}(Q)$

D8. $\mathsf{def}(\textbf{try } P \textbf{ else } Q) \leftrightarrow \mathsf{def}(P) \land (\mathsf{Con}(P) \lor \mathsf{def}(Q))$

D9. $\mathsf{def}(r(t)) \leftrightarrow \mathsf{def}(P\frac{t}{x})$ if $r(x) = P$ is a rule declaration of M

Table 8.3 Axioms for updates

U1. $\neg\mathsf{upd}(\textbf{skip}, f, x, y)$

U2. $\mathsf{upd}(f(s) := t, f, x, y) \leftrightarrow s = x \land t = y$, $\neg\mathsf{upd}(f(s) := t, g, x, y)$ if $f \neq g$

U3. $\mathsf{upd}(P \textbf{ par } Q, f, x, y) \leftrightarrow \mathsf{def}(P \textbf{ par } Q) \land (\mathsf{upd}(P, f, x, y) \lor \mathsf{upd}(Q, f, x, y))$

U4. $\mathsf{upd}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, f, x, y) \leftrightarrow (\varphi \land \mathsf{upd}(P, f, x, y)) \lor (\neg\varphi \land \mathsf{upd}(Q, f, x, y))$

U5. $\mathsf{upd}(\textbf{let } z = t \textbf{ in } P, f, x, y) \leftrightarrow \exists z\,(z = t \land \mathsf{upd}(P, f, x, y))$ if $z \notin \mathrm{FV}(t)$

U6. $\mathsf{upd}(\textbf{forall } z \textbf{ with } \varphi \textbf{ do } P, f, x, y) \leftrightarrow$
$\mathsf{def}(\textbf{forall } z \textbf{ with } \varphi \textbf{ do } P) \land \exists z\,(\varphi \land \mathsf{upd}(P, f, x, y))$

U7. $\mathsf{upd}(P \textbf{ seq } Q, f, x, y) \leftrightarrow$
$(\mathsf{upd}(P, f, x, y) \land [P]\mathsf{inv}(Q, f, x)) \lor (\mathsf{Con}(P) \land [P]\mathsf{upd}(Q, f, x, y))$

U8. $\mathsf{upd}(\textbf{try } P \textbf{ else } Q, f, x, y) \leftrightarrow$
$(\mathsf{Con}(P) \land \mathsf{upd}(P, f, x, y)) \lor (\mathsf{def}(P) \land \neg\mathsf{Con}(P) \land \mathsf{upd}(Q, f, x, y))$

U9. $\mathsf{upd}(r(t), f, x, y) \leftrightarrow \mathsf{upd}(P\frac{t}{z}, f, x, y)$ if $r(z) = P$ is a rule declaration of M

The formula $\mathsf{Con}(R)$ used in D8, U7 and U8 asserts that R is defined and consistent. It is an abbreviation defined as follows:

$$\mathsf{Con}(R) = \mathsf{def}(R) \wedge \bigwedge_{f \text{ dyn.}} \forall x, y, z\, (\mathsf{upd}(R, f, x, y) \wedge \mathsf{upd}(R, f, x, z) \to y = z)$$

The formula is true in a state if, and only if, the rule R is defined in the state and yields a consistent update set:

$$[\![\mathsf{Con}(R)]\!]^{\mathfrak{A}}_{\zeta} = \mathit{true} \iff \text{there exists a consistent } U \text{ with } \mathrm{yields}(R, \mathfrak{A}, \zeta, U).$$

The formula $\mathsf{inv}(R, f, x)$ in U7 asserts that the rule R is defined and does not update the function f at the argument x. It is a simple abbreviation defined as follows:

$$\mathsf{inv}(R, f, x) = \mathsf{def}(R) \wedge \forall y\, \neg \mathsf{upd}(R, f, x, y)$$

Note that it would be wrong to define the predicate $\mathsf{upd}(R, f, x, y)$ by saying that $f(x)$ is different from y in the present state but equal to y in the next state after the firing of rule R:

$$\mathsf{upd}(R, f, x, y) \iff f(x) \neq y \wedge [R]f(x) = y \qquad \text{(wrong definition)}$$

Using this definition, the predicate $\mathsf{upd}(f(0) := 1, f, 0, 1)$ would be false in a state where $f(0)$ is equal to 1, although the rule $f(0) := 1$ does update the function f at the argument 0 to 1.

8.1.3 Basic Axioms and Proof Rules of the Logic

The set of formulas (Def. 2.4.14) is extended such that it includes also formulas $[R]\varphi$ containing the modal operator $[R]$ and formulas $\mathsf{def}(R)$ and $\mathsf{upd}(R, f, s, t)$ containing the new basic predicates. The formulas of the logic for abstract state machines are generated by the following grammar:

$$\begin{aligned} \varphi, \psi \quad ::= \quad & s = t \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi \mid \forall x\, \varphi \mid \exists x\, \varphi \mid \\ & \mathsf{def}(R) \mid \mathsf{upd}(R, f, s, t) \mid [R]\varphi \end{aligned}$$

A formula is called *pure* (or *first-order*), if it contains neither the predicate def nor upd nor the modal operator $[R]$. A formula is called *static*, if it does not contain dynamic function names. The class of formulas that are used as guards in **if-then-else** or to specify ranges in **forall** is not extended. It still consists of the first-order formulas as in Sect. 2.4.3.

The semantics of formulas is given by the definitions in Table 2.1 and Table 8.1. Since formulas now contain rule names and the interpretation of rule names depends on a given abstract state machine M consisting of a set of rule declarations, we have to make M explicit when defining the notion of logical consequence:

Definition 8.1.1 (Logical consequence). Let M be an ASM. A formula φ is a *logical consequence* of a set Ψ of formulas with respect to M (written $\Psi \models_M \varphi$), if $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{true}$ for all states $\mathfrak{A}$ and variable assignments ζ such that $[\![\psi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{true}$ for every $\psi \in \Psi$.

The substitution of a term t for a variable x in a formula φ is denoted by $\varphi\frac{t}{x}$ and is defined as usual. Variables bound by a quantifier, a **let** or a **forall** have to be renamed when necessary. The substitution is also performed inside the transition rules that occur in formulas. The following substitution property holds (cf. Lemmas 2.4.6, 2.4.8, 2.4.11).

Lemma 8.1.1 (Substitution for modal formulas). Let t be a static term and $a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$. Then $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x\mapsto a]} = [\![\varphi\frac{t}{x}]\!]^{\mathfrak{A}}_{\zeta}$.

We define two transition rules P and Q to be *equivalent*, if they are defined and consistent in the same states and produce the same next state when they are fired. An observer from outside cannot distinguish two equivalent transition rules by just looking at the runs generated by them.

Definition 8.1.2 (Equivalence). The formula $P \simeq Q$ is defined as follows:

$$\begin{aligned} P \simeq Q \quad = \quad & (\mathsf{Con}(P) \lor \mathsf{Con}(Q)) \to (\mathsf{Con}(P) \land \mathsf{Con}(Q) \land \\ & \bigwedge\!\!\!\bigwedge_{f \text{ dyn.}} \forall x, y\, (\mathsf{upd}(P, f, x, y) \to (\mathsf{upd}(Q, f, x, y) \lor f(x) = y)) \land \\ & \bigwedge\!\!\!\bigwedge_{f \text{ dyn.}} \forall x, y\, (\mathsf{upd}(Q, f, x, y) \to (\mathsf{upd}(P, f, x, y) \lor f(x) = y))) \end{aligned}$$

The formula $P \simeq Q$ has the intended meaning:

Lemma 8.1.2. The formula $P \simeq Q$ is true in $\mathfrak{A}$ under ζ iff the following two conditions are true:

1. $[\![\mathsf{Con}(P)]\!]^{\mathfrak{A}}_{\zeta} = [\![\mathsf{Con}(Q)]\!]^{\mathfrak{A}}_{\zeta}$
2. If P yields U in $\mathfrak{A}$ under ζ and Q yields V in $\mathfrak{A}$ under ζ, and U and V are consistent, then $\mathfrak{A} + U = \mathfrak{A} + V$.

We already know that the axioms D1–D9 and U1–U9 are valid for a given abstract state machine M. Together with the following principles they will be the basic axioms and proof rules of the logic $\mathcal{L}(M)$. We start with the standard axioms and rules of the classical predicate calculus with equality. The quantifier axioms 1 and 2 as well as the substitution scheme, however, have to be restricted to static terms which do not contain dynamic function names. The reason for the restriction is that, if we substitute a term t for a variable x, then t can be evaluated in a different states due to the modal operators.

Axiom 3 is the so-called axiom K of modal logic. Together with the necessitation rule 4 it allows us to derive all the modal principles that are valid in arbitrary Kripke frames. Axiom 5 uses the fact that a rule R which is not defined or yields an inconsistent update set cannot be fired in a state. Since there is no successor state in this case, the necessity operator $[R]$ is trivial. Axiom 6 can be applied because the transition rules are deterministic. In modal logic the axiom for deterministic accessibility relations is written as $\Diamond\varphi \to \Box\varphi$ or $\neg\Box\neg\varphi \to \Box\varphi$.

The Barcan axiom 7 is true, since the universe does not change during the run of an ASM. Hence the quantifiers range over the same set in every state of a computation. Axioms 8 and 9 assert that the meaning of pure, static first-order formulas (which do not contain dynamic function names) is the same in all states of a computation.

Axiom 12 asserts that if a rule updates a function, then the rule is defined. Axiom 13 says that, if a rule updates a function f at the argument x to the value y, then in the next state the value of f at x is equal to y. If the rule does not update f at the argument x, then the value of f in the next state is the same as in the present state (Axiom 14).

The extensionality axiom 15 asserts that the modal operators of two equivalent transition rules are the same. Axioms 16 and 17 are wellknown from dynamic logic. They express the property that the empty rule has no effect on a state and that the sequential composition of transition rules corresponds to their sequential execution.

I. Classical logic with equality: We use the axioms and proof rules of the classical predicate calculus with equality. The quantifier axioms, however, are restricted.

II. Restricted quantifier axioms:

1. $\forall x\, \varphi \to \varphi\frac{t}{x}$ if t is static or φ is pure
2. $\varphi\frac{t}{x} \to \exists x\, \varphi$ if t is static or φ is pure

III. Modal axioms and proof rules:

3. $[R](\varphi \to \psi) \land [R]\varphi \to [R]\psi$
4. $\dfrac{\varphi}{[R]\varphi}$
5. $\neg\mathsf{Con}(R) \to [R]\varphi$
6. $\neg[R]\varphi \to [R]\neg\varphi$

IV. The Barcan axiom:

7. $\forall x[R]\varphi \to [R]\forall x\varphi$ if $x \notin \mathrm{FV}(R)$.

V. Axioms for pure static formulas:

8. $\varphi \to [R]\varphi$ if φ is pure and static
9. $\mathsf{Con}(R) \land [R]\varphi \to \varphi$ if φ is pure and static

VI. Axioms for def and upd:

10. D1–D9 in Table 8.2
11. U1–U9 in Table 8.3

VII. Update axioms for transition rules:

12. $\mathsf{upd}(R, f, x, y) \to \mathsf{def}(R)$

13. $\mathsf{upd}(R, f, x, y) \to [R]f(x) = y$
14. $\mathsf{inv}(R, f, x) \land f(x) = y \to [R]f(x) = y$

VIII. Extensionality axiom for transition rules:

15. $P \simeq Q \to ([P]\varphi \leftrightarrow [Q]\varphi)$

IX. Axioms from dynamic logic:

16. $[\mathbf{skip}]\varphi \leftrightarrow \varphi$
17. $[P \ \mathbf{seq}\ Q]\varphi \leftrightarrow [P][Q]\varphi$

The notion of derivability is defined as usual.

Definition 8.1.3 (Derivablility in the logic). Let M be an ASM. We write $\Psi \vdash_M \varphi$, if there exists a finite subset $\Theta \subseteq \Psi$ such that the formula $\bigwedge\!\!\!\bigwedge \Theta \to \varphi$ is derivable using the axioms and proof rules I–IX.

Note that axioms D9 and U9 depend on the rule declarations of the given abstract state machine M. Therefore, M has to be added as a parameter in $\Psi \vdash_M \varphi$. Since the principles I–IX are valid, the logic is sound.

Theorem 8.1.1 (Soundness of the logic). If $\Psi \vdash_M \varphi$, then $\Psi \models_M \varphi$.

Remark 8.1.1. The formula $\forall x\, \varphi \to \varphi\frac{t}{x}$ is not valid for non-static terms t. Consider the following tautology:

$$\forall x\, (x = 0 \to [f(0) := 1]x = 0).$$

If we substitute the term $f(0)$ for x, then we obtain the formula

$$f(0) = 0 \to [f(0) := 1]f(0) = 0.$$

This formula is not valid. Hence, the quantifier axioms must be restricted.

Several axioms use the formula $\mathsf{Con}(R)$, which asserts the consistency of the transition rule R. For example, $\mathsf{Con}(R)$ is used in Axioms 5 and 9 as well as in the extensionality Axiom 15. Since the notion of consistency is fundamental, we mention several equivalences which express the consistency of a compound transition rule in terms of the consistency of its components.

Lemma 8.1.3. The following consistency properties are derivable:

18. $\mathsf{Con}(\mathbf{skip})$
19. $\mathsf{Con}(f(s) := t)$
20. $\mathsf{Con}(P\ \mathbf{par}\ Q) \leftrightarrow \mathsf{Con}(P) \land \mathsf{Con}(Q) \land \mathsf{joinable}(P, Q)$
21. $\mathsf{Con}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q) \leftrightarrow (\varphi \land \mathsf{Con}(P)) \lor (\neg\varphi \land \mathsf{Con}(Q))$
22. $\mathsf{Con}(\mathbf{let}\ x = t\ \mathbf{in}\ P) \leftrightarrow \exists x\, (x = t \land \mathsf{Con}(P))$ if $x \notin \mathrm{FV}(t)$
23. $\mathsf{Con}(\mathbf{forall}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P) \leftrightarrow$
    $\forall x\, (\varphi \to \mathsf{Con}(P) \land \forall y(\varphi\frac{y}{x} \to \mathsf{joinable}(P, P\frac{y}{x}))$
24. $\mathsf{Con}(P\ \mathbf{seq}\ Q) \leftrightarrow \mathsf{Con}(P) \land [P]\mathsf{Con}(Q)$

25. $\mathsf{Con}(\mathbf{try}\ P\ \mathbf{else}\ Q) \leftrightarrow \mathsf{Con}(P) \lor (\mathsf{def}(P) \land \mathsf{Con}(Q))$
26. $\mathsf{Con}(r(t)) \leftrightarrow \mathsf{Con}(P\frac{t}{x})$ if $r(x) = P$ is a rule declaration of M

The predicate $\mathsf{joinable}(P, Q)$ which is used in property 20 above to reduce the consistency of a parallel composition $P\ \mathbf{par}\ Q$ into consistency properties of P and Q is defined as follows (where x, y, z are not free in P):

$$\mathsf{joinable}(P, Q) = \bigwedge_{f \text{ dyn.}} \forall x, y, z\, (\mathsf{upd}(P, f, x, y) \land \mathsf{upd}(Q, f, x, z) \to y = z)$$

It expresses the property that the update sets of P and Q do not conflict. This means that whenever P and Q both update a function f at the same argument x, then the new values of f at x are the same.

If a function $f(x)$ is equal to y in the next state after firing a consistent rule R, then either R does update the function f at the argument x to y or R does not update the function f at the argument x and $f(x)$ is equal to y in the present state. This principle is not a basic axiom of the logic, since it is derivable.

Lemma 8.1.4. The following principles are derivable:

27. $\mathsf{Con}(R) \land [R]f(x) = y \to \mathsf{upd}(R, f, x, y) \lor (\mathsf{inv}(R, f, x) \land f(x) = y)$
28. $\mathsf{Con}(R) \land [R]\varphi \to \neg[R]\neg\varphi$
29. $[R]\exists x\, \varphi \leftrightarrow \exists x\, [R]\varphi,$ if $x \notin \mathrm{FV}(R)$.

Axiom U7 and Axiom 14 use the predicate $\mathsf{inv}(R, f, x)$ that expresses the property that the rule R is defined and does not update the function f at the argument x. The following properties of inv are useful if one has to show that a compound transition rule does not update certain functions at certain arguments.

Lemma 8.1.5. The following properties of inv are derivable:

30. $\mathsf{inv}(\mathbf{skip}, f, x)$
31. $\mathsf{inv}(f(s) := t, f, x) \leftrightarrow x \neq s$
32. $\mathsf{inv}(P\ \mathbf{par}\ Q, f, x) \leftrightarrow \mathsf{inv}(P, f, x) \land \mathsf{inv}(Q, f, x)$
33. $\mathsf{inv}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q) \leftrightarrow (\varphi \land \mathsf{inv}(P, f, x)) \lor (\neg\varphi \land \mathsf{inv}(Q, f, x))$
34. $\mathsf{inv}(\mathbf{let}\ z = t\ \mathbf{in}\ P, f, x) \leftrightarrow \exists z\, (z = t \land \mathsf{inv}(P, f, x))$ if $z \notin \mathrm{FV}(t)$
35. $\mathsf{inv}(\mathbf{forall}\ z\ \mathbf{with}\ \varphi\ \mathbf{do}\ P, f, x) \leftrightarrow \forall z\, (\varphi \to \mathsf{inv}(P, f, x)))$
36. $\mathsf{inv}(P\ \mathbf{seq}\ Q, f, x) \leftrightarrow \mathsf{inv}(P, f, x) \land [P]\mathsf{inv}(Q, f, x)$
37. $\mathsf{inv}(\mathbf{try}\ P\ \mathbf{else}\ Q, f, x) \leftrightarrow$
    $(\mathsf{Con}(P) \land \mathsf{inv}(P, f, x)) \lor (\mathsf{def}(P) \land \neg\mathsf{Con}(P) \land \mathsf{inv}(Q, f, x))$
38. $\mathsf{inv}(r(t), f, x) \leftrightarrow \mathsf{inv}(P\frac{t}{z}, f, x)$ if $r(z) = P$ is a rule declaration of M

The system for formal reasoning about abstract state machines of [238] contains also for every rule R a modal operator $[R]$. The logic contains besides *true* and *false* a third truth-value which stands for *undefined*. The basic axioms FM1, FM2, AX1, AX2 of [238] are derivable in our system using the update Axioms 13 and 14.

Lemma 8.1.6. The following principles of [238] are derivable:

39. $s = x \rightarrow (y = t \leftrightarrow [f(s) := t]f(x) = y)$
40. $s \neq x \rightarrow (y = f(x) \leftrightarrow [f(s) := t]f(x) = y)$
41. $[P]f(x) = y \wedge [Q]f(x) = y \rightarrow [P\ \mathbf{par}\ Q]f(x) = y$
42. $f(x) \neq y \wedge ([P]f(x) = y \vee [Q]f(x) = y) \rightarrow [P\ \mathbf{par}\ Q]f(x) = y.$

The following inverse implication of principles 41 and 42 is not mentioned in [238] (maybe because of the lack of a consistency notion), but is derivable here:

$$\begin{aligned}&\mathsf{Con}(P\ \mathbf{par}\ Q) \wedge [P\ \mathbf{par}\ Q]f(x) = y \rightarrow\\ &\quad ([P]f(x) = y \wedge [Q]f(x) = y) \vee (f(x) \neq y \wedge ([P]f(x) = y \vee [Q]f(x) = y))\end{aligned}$$

Several principles known from dynamic logic are derivable using the extensionality Axiom 15.

Lemma 8.1.7. The following principles are derivable:

43. $[\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q]\psi \leftrightarrow (\varphi \wedge [P]\psi) \vee (\neg\varphi \wedge [Q]\psi)$
44. $[\mathbf{let}\ x = t\ \mathbf{in}\ P]\varphi \leftrightarrow \exists x\,(x = t \wedge [P]\varphi),$ if $x \notin \mathrm{FV}(t) \cup \mathrm{FV}(\varphi).$
45. $[\mathbf{try}\ P\ \mathbf{else}\ Q]\varphi \leftrightarrow [P]\varphi \wedge ((\mathsf{def}(P) \wedge \neg\mathsf{Con}(P)) \rightarrow [Q]\varphi)$
46. $[r(t)]\varphi \leftrightarrow [P\frac{t}{x}]\varphi,$ if $r(x) = P$ is a rule declaration of M.

A more intensional equivalence of rules can be defined as follows:

$$P \mathrel{\dot{\simeq}} Q \quad = \quad \begin{aligned}[t]&(\mathsf{def}(P) \vee \mathsf{def}(Q)) \rightarrow (\mathsf{def}(P) \wedge \mathsf{def}(Q) \wedge\\ &\bigwedge_{f\ \text{dyn.}} \forall x, y\,(\mathsf{upd}(P, f, x, y) \leftrightarrow \mathsf{upd}(Q, f, x, y)))\end{aligned}$$

The formula $P \mathrel{\dot{\simeq}} Q$ asserts that the transition rules P and Q are defined in the same states and yield the same update sets. The so-obtained notion of equivalence is stronger than the one used in the extensionality Axiom 15.

Lemma 8.1.8. $P \mathrel{\dot{\simeq}} Q \rightarrow P \simeq Q$ is derivable.

In the sequent calculus for an extended dynamic logic proposed in [394] to express the properties of ASMs, new rules are introduced that express the commutativity, the associativity and similar properties of the parallel combination of the transition rules. In our system, these properties are derivable (cf. Exercise 2.4.10).

Lemma 8.1.9. The following principles of [394] are derivable:

47. $(P\ \mathbf{par}\ \mathbf{skip}) \mathrel{\dot{\simeq}} P$
48. $(P\ \mathbf{par}\ Q) \mathrel{\dot{\simeq}} (Q\ \mathbf{par}\ P)$
49. $((P\ \mathbf{par}\ Q)\ \mathbf{par}\ R) \mathrel{\dot{\simeq}} (P\ \mathbf{par}\ (Q\ \mathbf{par}\ R))$
50. $(P\ \mathbf{par}\ P) \mathrel{\dot{\simeq}} P$
51. $(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q)\ \mathbf{par}\ R \mathrel{\dot{\simeq}} \mathbf{if}\ \varphi\ \mathbf{then}\ (P\ \mathbf{par}\ R)\ \mathbf{else}\ (Q\ \mathbf{par}\ R)$
52. $R\ \mathbf{par}\ (\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q) \mathrel{\dot{\simeq}} \mathbf{if}\ \varphi\ \mathbf{then}\ (R\ \mathbf{par}\ P)\ \mathbf{else}\ (R\ \mathbf{par}\ Q)$

If we can derive $P \mathrel{\dot{\simeq}} Q$ in the logic, then we immediately obtain the principle $[P]\varphi \leftrightarrow [Q]\varphi$ using Lemma 8.1.8 and the extensionality Axiom 15. For example, the commutativity of the parallel composition yields:

53. $[P \textbf{ par } Q]\varphi \leftrightarrow [Q \textbf{ par } P]\varphi$

Lemma 8.1.10. The following properties of the sequential composition are derivable:

54. $(P \textbf{ seq skip}) \mathrel{\dot{\simeq}} P$
55. $(\textbf{skip seq } P) \mathrel{\dot{\simeq}} P$
56. $((P \textbf{ seq } Q) \textbf{ seq } R) \mathrel{\dot{\simeq}} (P \textbf{ seq } (Q \textbf{ seq } R))$
57. $(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q) \textbf{ seq } R \mathrel{\dot{\simeq}} \textbf{if } \varphi \textbf{ then } (P \textbf{ seq } R) \textbf{ else } (Q \textbf{ seq } R)$

The dynamic logic with array assignments (see [268]) uses a substitution principle which is derivable in our system. Let φ be a quantifier-free, pure (first-order) formula. Then by $\varphi\frac{t}{f(s)}$ we denote the formula which is obtained in the following way. First, φ is transformed into an equivalent formula

$$\varphi \leftrightarrow \exists x\, \exists y \Big(\bigwedge_{i=1}^{n} f(x_i) = y_i \wedge \psi\Big),$$

where $x = x_1, \ldots, x_n$, $y = y_1, \ldots, y_n$ and ψ does not contain f. Then we define:

$$\varphi\frac{t}{f(s)} = \exists x\, \exists y \Big(\bigwedge_{i=1}^{n} ((x_i = s \wedge y_i = t) \vee (x_i \neq s \wedge f(x_i) = y_i)) \wedge \psi\Big)$$

The substitution of t for $f(s)$ can be generalized to arbitrary first-order formulas by first bringing them into prenex form and then applying the transformation to the quantifier-free kernel.

Lemma 8.1.11. For any first-order formula φ, the following substitution principle is derivable: $\varphi\frac{t}{f(s)} \leftrightarrow [f(s) := t]\varphi$.

An ASM is called *simple*, if it is defined by a single rule R, which has the following form:

$$\left.\begin{array}{l}
\textbf{if } \varphi_1 \textbf{ then } f(s_1) := t_1 \\
\textbf{if } \varphi_2 \textbf{ then } f(s_2) := t_2 \\
\qquad\qquad \vdots \\
\textbf{if } \varphi_n \textbf{ then } f(s_n) := t_n
\end{array}\right\} R$$

Simple ASMs have the obvious properties formulated in the following lemma. Property 61 is a variant of the basic axiom that is used in [365] to prove the partial correctness of imperative programs. It can easily be extended to disjoint **if-then** rules with simultaneous function updates.

Lemma 8.1.12. Let R be the rule of a simple ASM. Then,

58. $\mathsf{Con}(R) \leftrightarrow \bigwedge\!\!\!\bigwedge_{i<j} (\varphi_i \wedge \varphi_j \wedge s_i = s_j \rightarrow t_i = t_j)$

59. $\mathsf{upd}(R, f, x, y) \leftrightarrow \bigvee\!\!\!\bigvee_{i=1}^{n} (\varphi_i \wedge x = s_i \wedge y = t_i)$

60. $\mathsf{inv}(R, f, x) \leftrightarrow \bigwedge\!\!\!\bigwedge_{i=1}^{n} (\varphi_i \rightarrow x \neq s_i)$

61. $\bigvee\!\!\!\bigvee_{i=1}^{n} \varphi_i \wedge \bigwedge\!\!\!\bigwedge_{i<j} \neg(\varphi_i \wedge \varphi_j) \wedge \bigwedge\!\!\!\bigwedge_{i=1}^{n} (\varphi_i \rightarrow \psi\frac{t_i}{f(s_i)}) \rightarrow [R]\psi, \qquad$ if ψ is first-order.

Iteration can be reduced to recursion. We can define the **while** rule recursively, as follows (cf. Example 4.1.2):

$$\textbf{while } \varphi \textbf{ do } P = \textbf{if } \varphi \textbf{ then } (P \textbf{ seq while } \varphi \textbf{ do } P)$$

The expression **while** φ **do** P has to be read as a rule call $r(x)$, where x are the free variables of φ and P. So the above equation stands for the following rule declaration:

$$r(x) = \textbf{if } \varphi \textbf{ then } (P \textbf{ seq } r(x))$$

Lemma 8.1.13. The following properties of the **while** rule are derivable:

62. $\mathsf{Con}(\textbf{while } \varphi \textbf{ do } P) \leftrightarrow (\varphi \rightarrow \mathsf{Con}(P) \wedge [P]\mathsf{Con}(\textbf{while } \varphi \textbf{ do } P))$
63. $[\textbf{while } \varphi \textbf{ do } P]\psi \leftrightarrow (\varphi \wedge [P][\textbf{while } \varphi \textbf{ do } P]\psi) \vee (\neg\varphi \wedge \psi)$

Several properties of ASMs can be expressed in the basic logic. Let M be the distinguished rule name of the ASM and φ_{init} a formula characterizing the initial states of M. Then we can express:

– ψ is an invariant of M by $(\varphi_{\text{init}} \rightarrow \psi) \wedge (\psi \rightarrow [M]\psi)$.
– ψ ensures the consistency of M by $(\varphi_{\text{init}} \rightarrow \psi) \wedge (\psi \rightarrow \mathsf{Con}(M) \wedge [M]\psi)$.

Compiler correctness. The statement in [406] for the correctness of the compiler from Java to the JVM can be formulated as follows:

$$\begin{array}{l}(\varphi_{\text{init}} \rightarrow \varphi_{\text{eqv}}) \wedge \\ (\varphi_{\text{eqv}} \rightarrow [J](\varphi_{\text{eqv}} \vee [V]\varphi_{\text{eqv}} \vee [V][V]\varphi_{\text{eqv}} \vee [V][V][V]\varphi_{\text{eqv}}))\end{array}$$

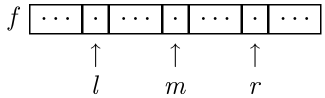
Here, J is an ASM that specifies the semantics of a Java source level program according to the *Java Language Specification*; V is an ASM that specifies the *Java Virtual Machine*. The two ASMs have disjoint dynamic function names and use the same static functions. The formula φ_{eqv} expresses that two dynamic states of the two ASMs are equivalent for a given Java program and its compiled bytecode program. The above formula says, that if two states are equivalent, then for each step of the Java source code interpreter J the bytecode interpreter V has to make zero, one, two or three steps to reach an equivalent state again. The proof in [406], which comprises 83 cases, could be mechanically checked in the basic system with appropriate structural induction principles for lists and abstract syntax trees (which are encoded using static functions).

Problem 28 (Mechanical verification of Java-to-JVM compilation and bytecode verification). Use your favorite theorem prover (PVS, Isabelle, KIV, . . .) to mechanically verify the correctness and completeness proofs given in [406] on the basis of ASM models for the Java-to-JVM compilation and JVM bytecode verification.

Verification of MergeSort. For an application of the logic to turbo ASMs consider the following specification of the MergeSort algorithm:

```
$\textsc{Msort}(l, r) =$
  if $l < r$ then
    let $m = \lfloor (l + r)/2 \rfloor$ in
      $(\textsc{Msort}(l, m)$ par $\textsc{Msort}(m + 1, r))$ seq $\textsc{Merge}(l, m, r)$
```

The ASM uses a unary dynamic function f that represents an array. The parameters l and r in $\textsc{Msort}(l, r)$ are the left and the right bound of the subarray $f(l \mathinner{..} r)$ that has to be sorted.



The middle point m between l and r is computed and the subarrays $f(l \mathinner{..} m)$ and $f(m + 1 \mathinner{..} r)$ are recursively sorted in parallel by the calls $\textsc{Msort}(l, m)$ and $\textsc{Msort}(m + 1, r)$. After that, the two sorted halves are merged together using $\textsc{Merge}(l, m, r)$:

```
$\textsc{Merge}(l, m, r) =$
  (forall $i$ with $l \leq i \leq r$ do $g(i) := f(i)$) seq
  $\textsc{MergeCopy}(l, m, m + 1, r, l)$

$\textsc{MergeCopy}(i, m, j, r, k) =$
  if $k \leq r$ then
    if $(i \leq m \land j \leq r \land g(i) \leq g(j)) \lor (r < j)$ then
      $f(k) := g(i)$ par $\textsc{MergeCopy}(i + 1, m, j, r, k + 1)$
    else
      $f(k) := g(j)$ par $\textsc{MergeCopy}(i, m, j + 1, r, k + 1)$
```

A second unary function g is used in which a copy of f is stored before the merging starts. $\textsc{MergeCopy}(i, m, j, r, k)$ merges the subarrays $g(i \mathinner{..} m)$ and $g(j \mathinner{..} r)$ into $f(k \mathinner{..} r)$. The parameter i runs over the first half of g from l to m; the parameter j runs over the second half of g from $m + 1$ to r; the parameter k runs over the whole array f from l to r.

The following statements about the $\textsc{Msort}$ algorithm can then be derived in the logic using additional axioms for induction on natural numbers:

– $\textsc{Msort}$ always terminates and computes a consistent set of updates:

$$\forall l, r \in \mathbb{N}\, \mathsf{Con}(\textsc{Msort}(l, r))$$

- The result of executing MSORT is an ordered array:

$$\forall l, r \in \mathbb{N}\,[\textsc{Msort}(l,r)]\forall i\,(l \le i < r \rightarrow f(i) \le f(i+1))$$

- The result of MSORT is a permutation of the original array:

$$\forall l, r \in \mathbb{N}\,\exists \pi \in \mathit{Perm}([l\,..\,r])\,\forall i \in [l\,..\,r]\,\mathsf{upd}(\textsc{Msort}(l,r), f, i, f(\pi(i)))$$

- There are no updates outside of the subarray $f(l\,..\,r)$:

$$\forall l, r \in \mathbb{N}\,\forall x, y\,(\mathsf{upd}(\textsc{Msort}(l,r), f, x, y) \rightarrow x \in [l\,..\,r])$$

The machines MSORT, MERGE and MERGECOPY can then be refined to sequential programs MSORT′, MERGE′ and MERGECOPY′ by replacing every occurrence of the parallel composition **par** by the **seq** operator. The formula

$$\forall l, r \in \mathbb{N}\,(\textsc{Msort}(l,r) \doteq \textsc{Msort}'(l,r))$$

expresses that the refinement is correct.

8.1.4 Why Deterministic Transition Rules?

If a transition rule contains **choose** then it is no longer deterministic. For a given transition rule R there can be several different update sets U such that yields$(R, \mathfrak{A}, \zeta, U)$ is derivable in Table 2.2. In the definition of a run of an ASM (Def. 2.4.22) one has to choose an update set in each step to obtain the next state of the run. As a consequence, there can be several different runs for a given initial state of a machine.

Unfortunately, the formalization of consistency cannot be applied directly to non-deterministic ASMs. The formula $\mathsf{Con}(R)$ (as defined in Sect. 8.1.2) expresses the property that the *union of all possible* update sets of R in a given state is consistent. This is clearly not what is meant by consistency. Therefore, in a logic for ASMs with **choose** one had to add $\mathsf{Con}(R)$ as an atomic formula to the logic.

The first question would be, what is the semantics of $\mathsf{Con}(R)$ for possibly non-deterministic rules? One possibility would be to define

$$[\![\mathsf{Con}(R)]\!]^{\mathfrak{A}}_{\zeta} = \mathit{true} \iff \begin{array}{l}\text{for each update set } U, \text{ if yields}(R, \mathfrak{A}, \zeta, U),\\ \text{then } U \text{ is consistent.}\end{array}$$

Hence, R is consistent, if each update set for R is consistent. (If R is not defined in state $\mathfrak{A}$, it is considered as consistent, too.) But then, the property 20 of Lemma 8.1.3 is no longer true:

$$\mathsf{Con}(P \textbf{ par } Q) \rightarrow \mathsf{Con}(P)$$

For example, if Q is not defined in a state, then the parallel composition P **par** Q is also not defined and therefore, by definition, consistent. The rule P, however, could be inconsistent. Property 20 could be re-written as follows:

$$\mathsf{Con}(P \textbf{ par } Q) \leftrightarrow (\mathsf{def}(P \textbf{ par } Q) \rightarrow \mathsf{Con}(P) \wedge \mathsf{Con}(Q) \wedge \mathsf{joinable}(P, Q))$$

Another possibility would be to define the consistency of rules as *weak consistency*:

$$[\![\mathsf{Con}(R)]\!]^{\mathfrak{A}}_{\zeta} = \mathit{true} \iff \begin{array}{l}\text{there exists a consistent update set } U\\ \text{such that yields}(R, \mathfrak{A}, \zeta, U).\end{array}$$

For this weak notion of consistency, however, we could not use the relation $\mathsf{joinable}(P, Q)$ in property 20 in Lemma 8.1.3. It is not clear what we should use instead:

$$\mathsf{Con}(P \ \mathbf{par}\ Q) \leftrightarrow \mathsf{Con}(P) \land \mathsf{Con}(Q) \land ?$$

A third possibility is to combine definedness and strong consistency as follows:

$$[\![\mathsf{Con}(R)]\!]^{\mathfrak{A}}_{\zeta} = \mathit{true} \iff \begin{array}{l}\text{there exists an update set } U\\ \text{such that yields}(R, \mathfrak{A}, \zeta, U)\text{, and}\\ \text{for each update set } U\text{, if yields}(R, \mathfrak{A}, \zeta, U)\text{,}\\ \text{then } U \text{ is consistent.}\end{array}$$

But what about property 24 in Lemma 8.1.3? Is the following implication still true under the new interpretation of consistency?

$$\mathsf{Con}(P \ \mathbf{seq}\ Q) \rightarrow [P]\mathsf{Con}(Q)$$

Assume that P yields U in $\mathfrak{A}$ under ζ and U is consistent. How do we know that the rule Q is defined in state $\mathfrak{A} + U$? How do we know that there exists a V such that Q yields V in $\mathfrak{A} + U$ under ζ?

One problem remains in all three cases, namely how to characterize the update predicate for sequential compositions (Axiom U7 in Table 8.3). What should be on the right-hand side of the following equivalence?

$$\mathsf{upd}(P \ \mathbf{seq}\ Q, f, x, y) \leftrightarrow ?$$

Consider the case, where P yields U in $\mathfrak{A}$ under ζ and U is consistent, $((f, a), b) \in U$ and the rule Q does not update the function f at the argument a in the state $\mathfrak{A} + U$ after firing the rule P with the update set U in $\mathfrak{A}$. This case cannot be expressed by the formula

$$\mathsf{upd}(P, f, a, b) \land [P]\mathsf{inv}(Q, f, a),$$

since the rule P could have several different consistent update sets and the rule Q could update the function f at the argument a after firing one of them, although it does not after firing the given update set U of P.

Because of all the problems, it seems impossible to find natural and simple axioms for the **choose** rule together with sequential composition and recursive rule declarations. Therefore we follow the approach that non-determinism is modeled from outside by choice functions such that, in the view of the transition rules, everything is deterministic (cf. Remark 2.4.1).

8.1.5 Completeness for Hierarchical ASMs

The main technical result of this section is that the logic is complete for so-called *hierarchical* ASMs.

Definition 8.1.4 (Hierarchical ASM). An ASM is called *hierarchical*, if the call graph of the rule declarations does not contain cycles.

An ASM is hierarchical iff it is possible to assign levels to the rule names of the machine such that in each rule declaration $r(x) = P$ the levels of rule names in P are less than the level of r.

In an earlier version of [405] the completeness of the logic for hierarchical ASMs was obtained via an extension of the Henkin model construction. Later, G. R. Renardel de Lavalette observed that, in the case of hierarchical ASMs, the logic for ASMs is a *definitional extension* of first-order logic (FOL). This means that there exists a translation of formulas φ of the logic for a hierarchical ASM M into first-order formulas φ^* with the following properties:

1. The equivalence $\varphi \leftrightarrow \varphi^*$ is derivable in $\mathcal{L}(M)$.
2. If φ is derivable in $\mathcal{L}(M)$, then φ^* is derivable in FOL.

Hence, for hierarchical ASMs, it is possible to eliminate the modal operator $[R]$ as well as the atomic formulas $\mathsf{def}(R)$ and $\mathsf{upd}(R, f, s, t)$. Due to parallel updates, however, the translation of modal formulas into FOL is more complicated than the comparable embedding of the logic of modification and creation in [373].

We first observe that the transition rules of hierarchical ASMs are always defined. If R is a transition rule which uses rules from a hierarchical machine M, then the formula $\mathsf{def}(R)$ is derivable in $\mathcal{L}(M)$. Therefore we can identify the formula $\mathsf{def}(R)$ with the constant $\top$ (true).

Moreover, we can assume that atomic formulas are restricted to simple formulas $x = y$, $f(x) = y$, $\mathsf{upd}(R, f, x, y)$. To bring general atomic formulas into this form, one can apply the following principles of $\mathcal{L}(M)$:

$$\begin{array}{lcl}
s = t & \leftrightarrow & \exists x\,(s = x \wedge t = x) \\
\mathsf{upd}(R, f, s, t) & \leftrightarrow & \exists x, y\,(s = x \wedge t = y \wedge \mathsf{upd}(R, f, x, y)) \\
f(s) = y & \leftrightarrow & \exists x\,(s = x \wedge f(x) = y)
\end{array}$$

The translation of modal formulas into FOL distributes over negation, boolean connectives and quantifiers. For eliminating $\mathsf{upd}(R, f, x, y)$ we use the axioms U1–U9 in Table 8.3. For eliminating the modal operator $[R]$ in $[R]\varphi$ we first translate φ into a first-order formula and use then the following equivalences of $\mathcal{L}(M)$:

$$\begin{array}{lll}
[R]x = y & \leftrightarrow & (\mathsf{Con}(R) \to x = y) \\
[R]f(x) = y & \leftrightarrow & (\mathsf{Con}(R) \to \mathsf{upd}(R, f, x, y) \lor (\mathsf{inv}(R, f, x) \land f(x) = y)) \\
[R]\neg\varphi & \leftrightarrow & (\mathsf{Con}(R) \to \neg[R]\varphi) \\
[R](\varphi \land \psi) & \leftrightarrow & ([R]\varphi \land [R]\psi) \\
[R](\varphi \lor \psi) & \leftrightarrow & ([R]\varphi \lor [R]\psi) \\
[R](\varphi \to \psi) & \leftrightarrow & ([R]\varphi \to [R]\psi) \\
[R]\forall x\, \varphi & \leftrightarrow & \forall x\, [R]\varphi \\
[R]\exists x\, \varphi & \leftrightarrow & \exists x\, [R]\varphi
\end{array}$$

In order to see that the translation is well-defined we define a rank for formulas and transition rules. The rank is also used to show that the translation into FOL has the above property 1.

$$\begin{array}{l}
|s = t| = 0 \\
|\neg\varphi| = |\varphi| + 1 \\
|\varphi \land \psi| = |\varphi \lor \psi| = |\varphi \to \psi| = \max(|\varphi|, |\psi|) + 1 \\
|\forall x\, \varphi| = |\exists x\, \varphi| = |\varphi| + 1 \\
|\mathsf{upd}(R, f, x, x)| = |\psi| + 1, \text{ if } \mathsf{upd}(R, f, x, x) \leftrightarrow \psi \text{ is an instance of U1–U9} \\
|[R]\varphi| = |R| + |\varphi| + 1 \\
|R| = \max(|\mathsf{Con}(R)|, |\mathsf{upd}(R, f, x, y)|, |\mathsf{inv}(R, f, x)|) + 1
\end{array}$$

That the rank is defined for each formula φ and transition rule R can be seen as follows. First, one assigns levels to the rule names of the hierarchical ASM such that the levels in the rule body are less than the level of the rule name for each rule declaration. Then one shows by main induction on the maximum level of a rule name occurring in φ or R and side induction on the size (number of symbols) of φ or R that $|\mathsf{upd}(R, f, x, y)|$ and $|\varphi|$ is defined.

The completeness and compactness theorems then follow from the corresponding results for FOL.

Theorem 8.1.2 (Completeness). Let M be a hierarchical ASM and Ψ be a set of sentences. If $\Psi \models_M \varphi$, then $\Psi \vdash_M \varphi$.

Theorem 8.1.3 (Compactness). If each finite subset of a set of formulas Ψ is satisfiable, then Ψ is satisfiable.

Remark 8.1.2 (Incompleteness for turbo ASMs). For ASMs with recursive rule declarations the logic is incomplete. Consider an ASM over the vocabulary of arithmetic with the following recursive rule declaration (there are no dynamic function names):

$$r(x, y) = \textbf{if } x = y \textbf{ then skip else } r(x, y + 1)$$

Let φ_{N} be the conjunction of the following seven formulas:

$$\begin{array}{ll}
\forall x\, (x + 1 \neq 0) & \forall x, y\, (x + 1 = y + 1 \to x = y) \\
\forall x\, (x + 0 = x) & \forall x, y\, (x + (y + 1) = (x + y) + 1) \\
\forall x\, (x * 0 = x) & \forall x, y\, (x * (y + 1) = (x * y) + x) \\
\forall x\, \mathsf{def}(r(x, 0)) &
\end{array}$$

Note that yields$(r(x,t), \mathfrak{A}, \zeta, U)$ is derivable in Table 2.2 iff U is the empty set and $\zeta(x) = [\![t+1+\ldots+1]\!]_\zeta^\mathfrak{A}$. Hence, $[\![\mathsf{def}(x,0)]\!]_\zeta^\mathfrak{A} = \mathit{true}$ iff $\zeta(x)$ is reachable from 0 by a finite number of successor steps. Therefore a structure $\mathfrak{A}$ is a model of φ_N iff $\mathfrak{A}$ is isomorphic to the structure of natural numbers. An arithmetical sentence ψ is true in the structure of natural numbers iff the formula $\varphi_\mathrm{N} \to \psi$ is valid. Since the set of true arithmetical sentences is not recursively enumerable, there cannot be a finitary, sound and complete formal system for the logic.

Remark 8.1.3. The extensionality Axiom 15, Axiom 16 for **skip** and Axiom 17 for the sequential compositions are not used for the completeness Theorem 8.1.2. Since the axioms are valid, they must be derivable for hierarchical ASMs (as a consequence of the completeness theorem).

8.1.6 The Henkin Model Construction

In this section we present a different completeness proof that follows the traditional Henkin-style completeness proof for classical first-order predicate logic. We do not expand the language by so-called witnessing constants (Henkin constants) but use variables for that purpose. This is possible because we consider countable signatures only. We start with some standard definitions.

Definition 8.1.5. A set Φ of formulas is *satisfiable* iff there exists a structure $\mathfrak{A}$ and a variable assignment ζ such that $[\![\varphi]\!]_\zeta^\mathfrak{A} = \mathit{true}$ for each $\varphi \in \Phi$.

Definition 8.1.6. A set Φ of formulas is *inconsistent* iff there exists a finite subset $\Psi \subseteq \Phi$ such that $\bigwedge\!\!\!\bigwedge \Psi \to \bot$ is derivable.

Definition 8.1.7. A set Φ of formulas is *maximal consistent* iff the following two conditions are true:

1. Φ is consistent.
2. If $\Phi \cup \{\varphi\}$ is consistent, then $\varphi \in \Phi$.

Definition 8.1.8. A set Φ of formulas *contains witnesses* iff for each formula $\exists x\, \varphi \in \Phi$ there exists a variable y such that $\varphi\frac{y}{x} \in \Phi$.

Proposition 8.1.1. If Φ is a consistent set of sentences, then there exists a maximal consistent set Ψ such that $\Phi \subseteq \Psi$ and Ψ contains witnesses.

Lemma 8.1.14. Let Φ be a maximal consistent set of formulas.

1. For each formula φ, either $\varphi \in \Phi$ or $\neg\varphi \in \Phi$.
2. If $\varphi \in \Phi$ and $\varphi \to \psi$ is derivable, then $\psi \in \Phi$.

For a set Φ and a transition rule R the set Φ_R is defined to be the set of formulas φ such that $[R]\varphi$ is in Φ:

$$\Phi_R = \{\varphi \mid [R]\varphi \in \Phi\}$$

If Φ is the set of formulas which are true in a state $\mathfrak{A}$ and if R is consistent in $\mathfrak{A}$, then Φ_R is the set of formulas which are true after firing the rule R in state $\mathfrak{A}$.

Lemma 8.1.15. Let Φ be a maximal consistent set of formulas that contains witnesses. If $\mathsf{Con}(R) \in \Phi$, then Φ_R is maximal consistent and contains witnesses.

Proof. We first show that Φ_R is consistent. Suppose that Φ_R is inconsistent and Ψ is a finite subset of Φ_R such that $\bigwedge\!\!\!\bigwedge \Psi \to \bot$ is derivable. By Axiom 3 and Rule 4, it follows that $\bigwedge\!\!\!\bigwedge\{[R]\varphi \mid \varphi \in \Psi\} \to [R]\bot$ is derivable. According to the definition of Φ_R, the formula $[R]\varphi$ is in Φ for each $\varphi \in \Psi$. Since $\mathsf{Con}(R) \wedge [R]\bot \to \bot$ is an instance of Axiom 9 and $\mathsf{Con}(R) \in \Phi$, it follows that Φ is inconsistent. Contradiction. Therefore, Φ_R is consistent.

Assume that $\Phi_R \cup \{\varphi\}$ is consistent. We have to show that φ belongs to Φ_R, i.e. that the formula $[R]\varphi$ is in Φ. Since Φ is maximal consistent, by Lemma 8.1.14 either $[R]\varphi$ or $\neg[R]\varphi$ belongs to Φ. If $[R]\varphi \in \Phi$, we have completed the proof. Otherwise, the formula $\neg[R]\varphi$ is in Φ. Since $\neg[R]\varphi \to [R]\neg\varphi$ is an instance of Axiom 6 and Φ is deductively closed (Lemma 8.1.14), it follows that $[R]\neg\varphi \in \Phi$ and thus $\neg\varphi$ in Φ_R. Since $\Phi_R \cup \{\varphi\}$ is consistent, the second case is not possible. Hence, Φ_R is maximal consistent.

Finally we show that Φ_R contains witnesses. Assume that $\exists x\, \varphi \in \Phi_R$. This means that $[R]\exists x\, \varphi \in \Phi$. We can assume that the variable x is not free in R. Since $[R]\exists x\, \varphi \to \exists x\, [R]\varphi$ is derivable (Lemma 8.1.4) and Φ is deductively closed (Lemma 8.1.14), the formula $\exists x\, [R]\varphi$ belongs to Φ. Since Φ contains witnesses, there exists a variable y such that $[R]\varphi\frac{y}{x} \in \Phi$. By the definition of the set Φ_R, it follows that $\varphi\frac{y}{x} \in \Phi_R$. □

For each maximal consistent set Φ which contains witnesses we define a structure $\mathfrak{A}_\Phi$. We start by defining an equivalence relation $\sim_\Phi$ on the set of variables:

$$x \sim_\Phi y \iff (x = y) \in \Phi$$

The equivalence class of a variable x is denoted by $[x]_\Phi$ and the set of all equivalence classes by $|\Phi|$. The universe $|\mathfrak{A}_\Phi|$ of the structure $\mathfrak{A}_\Phi$ is the set $|\Phi|$. For equivalence classes X, $Y \in |\Phi|$ we define

$$f^{\mathfrak{A}_\Phi}(X) = Y \iff \text{there exist } x \in X \text{ and } y \in Y \text{ such that } (f(x) = y) \in \Phi.$$

The function $f^{\mathfrak{A}_\Phi}$ is well-defined, since the formula

$$x_1 = x_2 \wedge f(x_1) = y_1 \wedge f(x_2) = y_2 \to y_1 = y_2$$

is derivable and Φ is deductively closed. For the totality of $f^{\mathfrak{A}_\Phi}$ we use the fact that Φ contains witnesses and therefore, for each variable x there exists a variable y such that the equation $f(x) = y$ belongs to Φ.

Instead of $[\![\varphi]\!]_\zeta^{\mathfrak{A}_\Phi}$ and $[\![t]\!]_\zeta^{\mathfrak{A}_\Phi}$ we just write $[\![\varphi]\!]_\zeta^{\Phi}$ and $[\![t]\!]_\zeta^{\Phi}$. By ε we denote the variable assignment that assigns the equivalence class $[x]_\Phi$ to each variable x.

It is easy to see that the interpretation of terms in the structure $\mathfrak{A}_\Phi$ under ε has the following property:

$$[\![t]\!]_\varepsilon^\Phi = [x]_\Phi \iff (t = x) \in \Phi$$

We could as well define the relation $\sim_\Phi$ between arbitrary terms of the language as it is usually done in completeness proofs for first-order predicate logic. Since for each term t there exists a variable x such that the equation $t = x$ belongs to Φ, the set of equivalence classes of terms would be isomorphic to the set of equivalence classes of variables. The reason that we define the relation $\sim_\Phi$ between variables only is that, if a transition rule is fired, then the interpretation of terms (which contain dynamic function names) can change, whereas the values assigned to variables do not change. In this way it is easier to see that the universe does not change when a rule is fired (in the proof of the following lemma).

Lemma 8.1.16. Let φ be a formula. Then for any maximal consistent set Φ of formulas which contains witnesses the following two statements are true:

F1. If $\varphi \in \Phi$, then $[\![\varphi]\!]_\varepsilon^\Phi = \mathit{true}$.
F2. If $(\neg\varphi) \in \Phi$, then $[\![\varphi]\!]_\varepsilon^\Phi = \mathit{false}$.

Proof. By induction on the rank $|\varphi|$ of a formula φ. The critical case is that of a modal formula $[R]\varphi$. We assume that F1 and F2 are true for all formulas with rank less than $|[R]\varphi|$ and prove the following statement:

(∗) If $\mathsf{Con}(R) \in \Phi$ and R yields U in Φ under ε, then U is consistent and $\mathfrak{A}_\Phi + U = \mathfrak{A}_{\Phi_R}$.

Statement (∗) means that if the formula $\mathsf{Con}(R)$ is in Φ and the update set of R in the structure $\mathfrak{A}_\Phi$ under ε is U, then U is consistent and the result of firing U in state $\mathfrak{A}_\Phi$ is equal to the structure associated with the set Φ_R (which is maximal consistent and contains witnesses according to Lemma 8.1.15).

Assume that $\mathsf{Con}(R)$ belongs to Φ and R yields U in Φ under ε. Since $|\mathsf{Con}(R)| < |[R]\varphi|$, by the induction hypothesis F1 for $\mathsf{Con}(R)$ and Φ, it follows that $[\![\mathsf{Con}(R)]\!]_\varepsilon^\Phi = \mathit{true}$, which means by definition, that U is consistent. Since Φ is deductively closed, by Axioms 8 and 9 it follows that

$$(x = y) \in \Phi \iff [R](x = y) \in \Phi \iff (x = y) \in \Phi_R.$$

Hence, the equivalence relation $\sim_\Phi$ is the same as $\sim_{\Phi_R}$ and the universe of the structure $\mathfrak{A}_\Phi$ is identical to the universe of $\mathfrak{A}_{\Phi_R}$, i.e., $|\Phi| = |\Phi_R|$.

Let x be a variable. By Lemma 8.1.14, $\mathsf{inv}(R, f, x) \in \Phi$ or $\neg\mathsf{inv}(R, f, x) \in \Phi$. Assume that $\mathsf{inv}(R, f, x) \in \Phi$. Since $|\mathsf{inv}(R, f, x)| < |[R]\varphi|$, by the induction hypothesis F1 for $\mathsf{inv}(R, f, x)$ and Φ, it follows that $[\![\mathsf{inv}(R, f, x)]\!]_\varepsilon^\Phi = \mathit{true}$. This means that there is no update for f at the argument $[x]_\Phi$ in U. Hence, $f^{\mathfrak{A}_\Phi + U}([x]_\Phi) = [y]_\Phi$, where y is a variable such that the equation $f(x) = y$ is in Φ. By Axiom 14, it follows that $[R]f(x) = y \in \Phi$ and thus $f(x) = y$ belongs to Φ_R. Hence, $f^{\mathfrak{A}_{\Phi_R}}([x]_\Phi) = [y]_\Phi$ and f has the same value at $[x]_\Phi$ in the structures $\mathfrak{A}_\Phi + U$ and $\mathfrak{A}_{\Phi_R}$. If $\neg\mathsf{inv}(R, f, x) \in \Phi$, then the formula $\exists y\, \mathsf{upd}(R, f, x, y)$ is in Φ. Since Φ contains witnesses, there exists a variable z such that $\mathsf{upd}(R, f, x, z) \in \Phi$. Since $|\mathsf{upd}(R, f, x, z)| < |[R]\varphi|$, by the induction hypothesis F1 for $\mathsf{upd}(R, f, x, z)$ and Φ, it follows that $[\![\mathsf{upd}(R, f, x, z)]\!]^{\Phi}_{\varepsilon} = \mathit{true}$. Hence, $((f, [x]_\Phi), [z]_\Phi) \in U$ and therefore $f^{\mathfrak{A}_\Phi + U}([x]_\Phi) = [z]_\Phi$. By Axiom 13, it follows that $[R]f(x) = z \in \Phi$ and thus the equation $f(x) = z$ is in Φ_R. Hence, $f^{\mathfrak{A}_{\Phi_R}}([x]_\Phi) = [z]_\Phi$ and f has the same value at $[x]_\Phi$ in the structures $\mathfrak{A}_\Phi + U$ and $\mathfrak{A}_{\Phi_R}$, too. Since the variable x was chosen arbitrarily, the proof of statement $(*)$ is completed.

F1: Assume that $[R]\varphi \in \Phi$. By Lemma 8.1.14, we know that either $\mathsf{Con}(R) \in \Phi$ or $\neg\mathsf{Con}(R) \in \Phi$. If $\mathsf{Con}(R) \in \Phi$, then Φ_R is maximal consistent and contains witnesses (Lemma 8.1.15). Since $\varphi \in \Phi_R$ and $|\varphi| < |[R]\varphi|$, by the induction hypothesis F1 for φ and Φ_R, it follows that $[\![\varphi]\!]^{\Phi_R}_{\varepsilon} = \mathit{true}$. By $(*)$, it follows that $[\![[R]\varphi]\!]^{\Phi}_{\varepsilon} = \mathit{true}$. In the second case, if $\neg\mathsf{Con}(R) \in \Phi$, then by the induction hypothesis F2 for $\mathsf{Con}(R)$ and Φ, if follows that $[\![\mathsf{Con}(R)]\!]^{\Phi}_{\varepsilon} = \mathit{false}$ and thus $[\![[R]\varphi]\!]^{\Phi}_{\varepsilon} = \mathit{true}$.

F2: Assume that $\neg[R]\varphi \in \Phi$. Since Φ is deductively closed, by Axioms 5 and 6, we obtain that $\mathsf{Con}(R) \in \Phi$ and $[R]\neg\varphi \in \Phi$. Since $\mathsf{Con}(R) \in \Phi$, we can apply Lemma 8.1.15 and obtain that Φ_R is maximal consistent and contains witnesses. By the induction hypothesis F2 for φ and Φ_R, it follows that $[\![\varphi]\!]^{\Phi_R}_{\varepsilon} = \mathit{false}$. By $(*)$, we obtain $[\![[R]\varphi]\!]^{\Phi}_{\varepsilon} = \mathit{false}$.

Another important case is that of a formula $\mathsf{upd}(R, f, s, t)$. Since Φ contains witnesses, there exist variables x and y such that the equations $s = x$ and $t = y$ belong to Φ. Moreover, there exists a ψ such that $\mathsf{upd}(R, f, x, y) \leftrightarrow \psi$ is an instance of U1–U9. By the definition of the rank of formulas, it follows that $|\psi| < |\mathsf{upd}(R, f, x, y)| = |\mathsf{upd}(R, f, s, t)|$.

F1: Assume that $\mathsf{upd}(R, f, s, t) \in \Phi$. Since Φ is deductively closed, the formulas $\mathsf{upd}(R, f, x, y)$ and ψ are also in Φ. By the induction hypothesis for ψ and Φ, it follows that $[\![\psi]\!]^{\Phi}_{\varepsilon} = \mathit{true}$. Since $\mathsf{upd}(R, f, x, y) \leftrightarrow \psi$ is valid in any structure, $[\![\mathsf{upd}(R, f, x, y)]\!]^{\Phi}_{\varepsilon} = \mathit{true}$. Since the equations $s = x$ and $t = y$ belong to Φ, $[\![\mathsf{upd}(R, f, s, t)]\!]^{\Phi}_{\varepsilon} = \mathit{true}$.

F2: Assume that $\neg\mathsf{upd}(R, f, s, t) \in \Phi$. Since Φ is deductively closed, the formulas $\neg\mathsf{upd}(R, f, x, y)$ and $\neg\psi$ are also in Φ. By the induction hypothesis for ψ and Φ, it follows that $[\![\psi]\!]^{\Phi}_{\varepsilon} = \mathit{false}$. Hence, $[\![\mathsf{upd}(R, f, s, t)]\!]^{\Phi}_{\varepsilon} = \mathit{false}$.

□

The completeness of the logic for hierarchical ASMs is a consequence of the preceding lemma.

Theorem 8.1.4 (Completeness). Let M be a hierarchical ASM and Ψ be a set of sentences. If $\Psi \models_M \varphi$, then $\Psi \vdash_M \varphi$.

Proof. Assume that φ is not derivable from Ψ. Then the set $\Psi \cup \{\neg\varphi\}$ is consistent and can be extended to a maximal consistent set Θ that contains witnesses (Proposition 8.1.1). By Lemma 8.1.16, it follows that the structure $\mathfrak{A}_\Theta$ is a model of Ψ in which the formula φ is false. Hence φ is not a consequence of Ψ. □

8.1.7 An Extension with Explicit Step Information

For proving properties of ASMs it is often convenient to use "the function f in the nth state of the run". For that purpose we extend the signature Σ of the given (deterministic) ASM for each dynamic function name f by a new function name with one additional argument and write $f_n(t)$ instead of $f(n,t)$. We write $\mathfrak{A}_n$ for the nth state of the run of the ASM started in state $\mathfrak{A}$. (If the run is finite and n exceeds the length of the run, then $\mathfrak{A}_n$ is defined to be equal to the last state of the run.) The idea is that f_n is the function f in the nth state of the run.

We now work in two-sorted predicate logic. Each structure $\mathfrak{A}$ is extended by a copy of the structure of the natural numbers. The quantifiers $\forall x \in \mathbb{N}$ and $\exists x \in \mathbb{N}$ range over the set of natural numbers. Terms of the sort of natural numbers are denoted by ν. The grammar for formulas is extended by the following new atomic formulas:

$$\varphi, \psi \ ::= \ \ldots \mid \mathsf{def}_\nu(R) \mid \mathsf{upd}_\nu(R, f, s, t) \mid [R]_\nu \varphi$$

The semantics of the new formulas is defined as follows, where $n = [\![\nu]\!]^{\mathfrak{A}}_\zeta$:

$$\begin{array}{lcl} [\![\mathsf{def}_\nu(R)]\!]^{\mathfrak{A}}_\zeta & = & [\![\mathsf{def}(R)]\!]^{\mathfrak{A}_n}_\zeta \\ [\![\mathsf{upd}_\nu(R,f,s,t)]\!]^{\mathfrak{A}}_\zeta & = & [\![\mathsf{upd}(R,f,s,t)]\!]^{\mathfrak{A}_n}_\zeta \\ [\![[R]_\nu\varphi]\!]^{\mathfrak{A}}_\zeta & = & [\![[R]\varphi]\!]^{\mathfrak{A}_n}_\zeta \end{array}$$

For example, $\mathsf{def}_\nu(R)$ is true in the present state $\mathfrak{A}$, if the rule R is defined in the nth state of the run of the ASM starting in state $\mathfrak{A}$, where n is the value of the term ν. The new function names are interpreted in the obvious way:

$$f^{\mathfrak{A}}(n,a) = f^{\mathfrak{A}_n}(a), \quad \text{for each } n \in \mathbb{N} \text{ and } a \in |\mathfrak{A}|.$$

For formulas φ, not containing the new symbols with subscripts, we define a transformation $\varphi^{(\nu)}$ as follows:

$$\begin{array}{lcllcl} (s = t)^{(\nu)} & = & s^{(\nu)} = t^{(\nu)} & (\forall x\, \varphi)^{(\nu)} & = & \forall x\, \varphi^{(\nu)} \\ (\neg\varphi)^{(\nu)} & = & \neg\varphi^{(\nu)} & (\exists x\, \varphi)^{(\nu)} & = & \exists x\, \varphi^{(\nu)} \\ (\varphi \wedge \psi)^{(\nu)} & = & \varphi^{(\nu)} \wedge \psi^{(\nu)} & \mathsf{upd}(R,f,s,t)^{(\nu)} & = & \mathsf{upd}_\nu(R,f,s,t) \\ (\varphi \vee \psi)^{(\nu)} & = & \varphi^{(\nu)} \vee \psi^{(\nu)} & ([R]\varphi)^{(\nu)} & = & [R]_\nu\varphi \\ (\varphi \to \psi)^{(\nu)} & = & \varphi^{(\nu)} \to \psi^{(\nu)} & \mathsf{def}(R)^{(\nu)} & = & \mathsf{def}_\nu(R) \end{array}$$

Hence, $\varphi^{(\nu)}$ is obtained from φ by subscripting dynamic function names as well as the modal operators and the predicates def and upd in φ with ν. The subscript is neither applied inside modal formulas $[R]\varphi$ nor inside the atomic formulas except in equations. Inside the terms of equations it is applied everywhere:

$$x^{(\nu)} = x \qquad f(t)^{(\nu)} = \begin{cases} f_\nu(t^{(\nu)}), & \text{if } f \text{ is dynamic;} \\ f(t^{(\nu)}), & \text{otherwise.} \end{cases}$$

As expected, the value of a term $t^{(\nu)}$ is the value of t in the nth state of the run of the ASM and the formula $\varphi^{(\nu)}$ is equivalent to the formula φ in the nth state, where n is the value of the term ν.

Lemma 8.1.17. If $n = [\![\nu]\!]^{\mathfrak{A}}_{\zeta}$, then $[\![t^{(\nu)}]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}_n}_{\zeta}$ and $[\![\varphi^{(\nu)}]\!]^{\mathfrak{A}}_{\zeta} = [\![\varphi]\!]^{\mathfrak{A}_n}_{\zeta}$.

We write $\mathsf{Con}_\nu(R)$ for $\mathsf{Con}(R)^{(\nu)}$ and $\mathsf{inv}_\nu(R, f, x)$ for $\mathsf{inv}(R, f, x)^{(\nu)}$.

The basic logic of Sect. 8.1.3 is extended by the following principles, where P_M is the main rule of the ASM. First we add appropriate axioms for the structure of natural numbers including the induction scheme 64. Axiom 65 asserts that $\varphi^{(0)}$ is equivalent to φ, since the functions f_0 are, by definition, the same as the functions in the initial state. Axioms 66 and 67 characterize $\varphi^{(\nu+1)}$ depending on whether the main rule of the ASM is consistent or not in state ν. If P_M is not consistent in state ν, then by definition the state $\nu+1$ is the same as state ν. The proof rule 68 allows us to transfer axioms and theorems of the basic system into statements about the νth state of a run. Hence, we do not need new axioms for $\mathsf{def}_\nu(R)$ and $\mathsf{upd}_\nu(R, f, x, y)$, since we can transfer the axioms D1–D9 and U1–U9 using Rule 68.

X. Axioms for the structure of natural numbers:
For example, the Peano Axioms including the induction scheme:

64. $\varphi(0) \land \forall x \in \mathbb{N}\,(\varphi(x) \to \varphi(x+1)) \to \forall x \in \mathbb{N}\,\varphi(x)$

XI. Axioms and proof rules for formulas with step information:

65. $\varphi^{(0)} \leftrightarrow \varphi$
66. $\mathsf{Con}_\nu(P_M) \to (\varphi^{(\nu+1)} \leftrightarrow [P_M]_\nu\varphi)$
67. $\neg\mathsf{Con}_\nu(P_M) \to (\varphi^{(\nu+1)} \leftrightarrow \varphi^{(\nu)})$
68. $\dfrac{\varphi}{\varphi^{(\nu)}}$

Why is Rule 68 sound? Assume that φ is derivable. We want to show that $\varphi^{(\nu)}$ is valid. Let $\mathfrak{A}$ be a state and n be the value of ν in $\mathfrak{A}$. Since φ is valid, it is true in the nth state $\mathfrak{A}_n$ of the run starting with $\mathfrak{A}$. By Lemma 8.1.17, it follows that $\varphi^{(\nu)}$ is true in $\mathfrak{A}$.

The following proof rules are derivable using the induction scheme 64 and 65–68. The rule on the left-hand side says that, if φ is an invariant of P_M and if it is true in the initial state, then it is true in all states of the run. The rule on the right-hand side says that, if φ is an invariant which ensures the consistency of P_M and φ is true in the initial state, then P_M is consistent in all states of the run.

$$\frac{\varphi \to [P_M]\varphi}{\varphi \to \forall x \in \mathbb{N}\, \varphi^{(x)}} \qquad \frac{\varphi \to \mathsf{Con}(P_M) \land [P_M]\varphi}{\varphi \to \forall x \in \mathbb{N}\, \mathsf{Con}_x(P_M)}$$

The following characterizing formulas for f_ν are derivable as well:

69. $\forall x\, (f_0(x) = f(x))$
70. $\mathsf{Con}_\nu(P_M) \to$
    $\forall x, y (f_{\nu+1}(x) = y \leftrightarrow \mathsf{upd}_\nu(P_M, f, x, y) \lor (\mathsf{inv}_\nu(P_M, f, x) \land f_\nu(x) = y))$
71. $\neg\mathsf{Con}_\nu(P_M) \to \forall x\, (f_{\nu+1}(x) = f_\nu(x))$

In the extended system, several properties of the ASM can be expressed in a simple way (where φ_{init} and φ_{stop} are formulas characterizing the initial and halting states of the ASM):

- The ASM is consistent: $\varphi_{\text{init}} \to \forall x \in \mathbb{N}\, \mathsf{Con}_x(P_M)$.
- The ASM terminates: $\varphi_{\text{init}} \to \exists x \in \mathbb{N}\, \varphi_{\text{stop}}^{(x)}$.
- The formula ψ is an invariant of the ASM: $\varphi_{\text{init}} \to \forall x \in \mathbb{N}\, \psi^{(x)}$.

8.1.8 Exercises

Exercise 8.1.1. (⤳ CD) Show that the properties D1–D9 in Table 8.2 and U1–U9 in Table 8.3 are valid under the interpretations given in Table 8.1.

Exercise 8.1.2. (⤳ CD) Prove Lemma 8.1.1 (Substitution for modal formulas). Show that it is in general not true for non-static terms that contain dynamic functions.

Exercise 8.1.3. (⤳ CD) Prove Lemma 8.1.2 (cf. Exercise 2.4.12).

Exercise 8.1.4. (⤳ CD) Prove Theorem 8.1.1 (Soundness of the logic).

Exercise 8.1.5. (⤳ CD) Show that the following equivalence is derivable:

72. $(\mathsf{Con}(R) \to [R]\varphi) \leftrightarrow [R]\varphi$

Exercise 8.1.6. (⤳ CD) Show that the Axiom 8, $\varphi \to [R]\varphi$, for pure, static first-order formulas φ can be replaced by the following two axioms:

73. $x = y \to [R]x = y$
74. $x \neq y \to [R]x \neq y$

Exercise 8.1.7. (⤳ CD) Show that the Axiom 9, $\mathsf{Con}(R) \land [R]\varphi \to \varphi$, for pure, static first-order formulas φ can be replaced by the following axiom:

75. $\mathsf{Con}(R) \land [R]\bot \to \bot$

Exercise 8.1.8. (⤳ CD) Show that the properties of Con in Lemma 8.1.3 are derivable in the logic.

Exercise 8.1.9. (⤳ CD) Prove Lemma 8.1.4.

Exercise 8.1.10. ($\leadsto$ CD) Show that the properties of inv in Lemma 8.1.5 are derivable in the logic.

Exercise 8.1.11. ($\leadsto$ CD) Show that the axioms FM1, FM2, AX1, AX2 of [238] in Lemma 8.1.6 are derivable.

Exercise 8.1.12. ($\leadsto$ CD) Consider the following generalization of formula 41 in Lemma 8.1.6:

$$[P]\varphi \land [Q]\varphi \to [P \mathbf{\ par\ } Q]\varphi$$

Is this principle derivable for arbitrary first-order formulas φ?

Exercise 8.1.13. ($\leadsto$ CD) Show that the formulas in Lemma 8.1.7, 8.1.8, 8.1.9, 8.1.10, 8.1.11, 8.1.12, and 8.1.13 are derivable in the logic.

Exercise 8.1.14. ($\leadsto$ CD) Show that the following equivalence is derivable:

76. $P \simeq Q \quad \leftrightarrow \quad ((\mathsf{Con}(P) \lor \mathsf{Con}(Q)) \to (\mathsf{Con}(P) \land \mathsf{Con}(Q) \land \bigwedge_{f \text{ dyn.}} \forall x, y\, ([P]f(x) = y \leftrightarrow [Q]f(x) = y)))$

Exercise 8.1.15. ($\leadsto$ CD) Show that the following substitution principle for transition rules is derivable (cf. Exercise 2.4.11):

77. $P \mathrel{\dot{\simeq}} Q \to R[P] \mathrel{\dot{\simeq}} R[Q]$

Exercise 8.1.16. ($\leadsto$ CD) Define a weak consistency predicate $\mathsf{con}(R)$ by

$$\mathsf{con}(R) = \bigwedge_{f \text{ dyn.}} \forall x, y, z(\mathsf{upd}(R, f, x, y) \land \mathsf{upd}(R, f, x, z) \to y = z)$$

Show that the following relations between $\mathsf{con}(R)$ and $\mathsf{Con}(R)$ are derivable:

78. $\mathsf{Con}(R) \leftrightarrow (\mathsf{def}(R) \land \mathsf{con}(R))$
79. $\neg\mathsf{def}(R) \to \mathsf{con}(R)$
80. $\mathsf{con}(R) \leftrightarrow (\mathsf{def}(R) \to \mathsf{Con}(R))$

Show that the following properties of con are derivable (cf. Lemma 8.1.3):

81. $\mathsf{con}(\mathbf{skip})$
82. $\mathsf{con}(f(s) := t)$
83. $\mathsf{con}(P \mathbf{\ par\ } Q) \leftrightarrow (\mathsf{def}(P \mathbf{\ par\ } Q) \to \mathsf{con}(P) \land \mathsf{con}(Q) \land \mathsf{joinable}(P, Q))$
84. $\mathsf{con}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q) \leftrightarrow (\varphi \land \mathsf{con}(P)) \lor (\neg\varphi \land \mathsf{con}(Q))$
85. $\mathsf{con}(\mathbf{let}\ x = t\ \mathbf{in}\ P) \leftrightarrow \exists x\, (x = t \land \mathsf{con}(P))$ if $x \notin \mathrm{FV}(t)$
86. $\mathsf{con}(\mathbf{forall}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P) \leftrightarrow$
    $(\mathsf{def}(\mathbf{forall}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P) \to$
    $\forall x\, (\varphi \to \mathsf{con}(P) \land \forall y\, (\varphi\frac{y}{x} \to \mathsf{joinable}(P, P\frac{y}{x}))))$
87. $\mathsf{con}(P \mathbf{\ seq\ } Q) \leftrightarrow \mathsf{con}(P) \land [P]\mathsf{con}(Q)$
88. $\mathsf{con}(\mathbf{try}\ P\ \mathbf{else}\ Q) \leftrightarrow (\mathsf{def}(\mathbf{try}\ P\ \mathbf{else}\ Q) \to \mathsf{con}(P) \lor \mathsf{con}(Q))$
89. $\mathsf{con}(r(t)) \leftrightarrow \mathsf{con}(P\frac{t}{x})$, if $r(x) = P$ is a rule declaration of M.

Exercise 8.1.17. ($\rightsquigarrow$ CD) Extend the logic to the **try-catch** construct of Def. 4.1.6 in Sect. 4.1.2.

Exercise 8.1.18. ($\rightsquigarrow$ CD) Show that $[P \textbf{ seq } Q]\varphi \leftrightarrow [P][Q]\varphi$ (Axiom 17) is derivable for hierarchical ASMs.

Exercise 8.1.19. ($\rightsquigarrow$ CD) Show that $P \simeq Q \rightarrow ([P]\varphi \leftrightarrow [Q]\varphi)$ (Axiom 15) is derivable for hierarchical ASMs.

Exercise 8.1.20. ($\rightsquigarrow$ CD) Show that $[\textbf{skip}]\varphi \leftrightarrow \varphi$ (Axiom 16) is derivable for hierarchical ASMs.

8.2 Model Checking of ASMs

In a model checking environment like SMV, a system is specified as a finite state machine. The states of the FSM are given by a finite number of state variables, which do not necessarily have to be binary but can take values from an arbitrary user-defined finite range. The state variables correspond to nullary dynamic functions of an ASM. Hence, if one wants to use the SMV model checker to explore the state space of a given ASM, one has to eliminate all dynamic function symbols of arity greater than zero in the given ASM. In addition, the model checker can only be used if the universe of the initial state of the ASM is finite and the static functions are known to the model checker. One must also be aware that in SMV a variable without update is supposed to take any possible value of its range in the next state (non-determinism by under-specification), whereas in an ASM a nullary dynamic function without update does not change its value in the next computation step.

In [426] an elimination procedure for dynamic functions of arity greater than zero is presented, which we briefly summarize here. We start with a signature Σ, a finite state $\mathfrak{A}$ for Σ and an ASM rule P. The goal is to transform P into an equivalent transition rule that does not use dynamic functions of arity greater than zero. Of course, this is only possible, if we extend the signature and add new nullary functions:

- For each element of the universe of $\mathfrak{A}$ a new static constant is added to Σ. The new constants are called *values*. The finite set of values is denoted by V, i.e. $V = \{c_1, \ldots, c_m\}$.
- For each dynamic function name f in Σ of arity $n > 0$ and each tuple $(v_1, \ldots, v_n)$ of values, a new nullary dynamic function $\ell_{f,v_1,\ldots,v_n}$ is added to Σ (called a *location*).

The new names have the obvious interpretation. A value c_i is identified with the element of $\mathfrak{A}$ it denotes, and hence $|\mathfrak{A}| = V$. A location $\ell_{f,v_1,\ldots,v_n}$ is interpreted as the content of the location $(f, (v_1, \ldots, v_n))$ in $\mathfrak{A}$.

By $R[s]$ we denote a transition rule with several occurrences of a term s, and by $R[t]$ we denote the result of replacing the indicated occurrences of s

by the term t. An occurrence of a term is called *critical*, if it is inside the argument of a dynamic function. For example, assuming that $+$ is static, f is dynamic, and $\ell_1 + f(\ell_2 + 3)$ is a top-level term, the occurrence of ℓ_1 is not critical whereas the occurrence of ℓ_2 is.

We assume that the given transition rule P is a simple transition rule built from updates using if-then-else and parallel composition. Moreover, the rule guards in P are quantifier-free formulas (boolean combinations of equations between terms). Hence, P does not contain variables and all terms in P are ground terms. The following transformations are applied to P until all dynamic functions of arity greater than zero have disappeared:

$$\begin{array}{lcll}
R[f(v_1,\dots,v_n)] & \Longrightarrow & R[\ell_{f,v_1,\dots,v_n}] & \text{if } f \text{ is dynamic and} \\
 & & & v_1,\dots,v_n \text{ are values} \\[1ex]
R[f(v_1,\dots,v_n)] & \Longrightarrow & R[c] & \text{if the occurrences of } f \\
 & & & \text{are critical, } f \text{ is static,} \\
 & & & \text{and } f^{\mathfrak{A}}(v_1,\dots,v_n) = c \\[1ex]
R[\ell] & \Longrightarrow & \textbf{if } \ell = c_1 \textbf{ then } R[c_1] & \text{if the occurrences of } \ell \\
 & & \vdots & \text{are critical} \\
 & & \textbf{if } \ell = c_m \textbf{ then } R[c_m] &
\end{array}$$

In the first transformation a dynamic function is replaced by a location provided that the arguments of the function are all values. The occurrence of the dynamic function can be on the left-hand side of an update or anywhere inside a term or a rule guard.

The arguments of functions become values by applying repeatedly transformations 2 and 3. In transformation 2, a static function is evaluated (computed) at given arguments. In transformation 3, a location is replaced by a case distinction on the possible values of the location.

Transformations 1 and 2 are called *simplifications*, whereas transformation 3 is called an *unfolding*. Unfoldings have to be applied with care, since they can lead to an explosion of the size of the resulting ASM. Such an explosion is bad from the viewpoint of model checking, since in model checking the explosion is mostly caused by the number of state variables and the resulting huge state space, whereas the description of the transition system in the language of the model checker is in general small.

For a different approach to generating for testing purposes an FSM from an ASM see the algorithm AsmGenFsm on p. 123.

Problem 29 (Implement a model checker for ASMs). The model checker should directly support ASMs using efficient data structures for dynamic functions and update sets. It should also support symbolic model checking of ASMs, where regions of the state space are represented by first-order formulas and pre- and post-images of regions can be computed efficiently.

8.3 Execution of ASMs

We briefly survey here various methods and systems which support the execution of ASMs for simulation and testing purposes. For more detailed information on the systems we refer to the indicated code and documentation sources, which are mostly publicly accessible. For a description of the historical development of ASM tools see Sect. 9.4.3.

By their very definition as machines, ASMs are executable for mental simulation. As the previous chapters show, the run-based understanding and analyzability of ASMs are rather helpful for supporting the practitioner's daily work, independently from the possibility of developing automatic execution mechanisms. In the same vein the notion of ground models is more general than that of mechanically executable specifications. Ground models as *software contracts*, as explained in Chap. 2, are not limited to models at a high but fixed level of abstraction of code of some programming language or for machine-executable specifications. Ground models are specifications which come with a notion of run, but their *raison d'être* precedes that of machine-executable code to which they may be linked more or less directly, possibly through a hierarchy of stepwise refined models which bridge the gap between the abstraction level of the ground model – the software contract – and its executable version. There are different ways to make ASM models executable, which can be classified into four major groups.

user-scenario simulation of concrete ASMs by embedding the abstract machines into a run-time environment where each time the value for an argument of a domain-specific external function is required during execution, this value is provided as a user-scenario determined input. This "coding-free" method of testing user-scenarios for high-level ASMs has been successfully used in the industrial re-engineering project Falko [121]; the Workbench [170] provided the basic execution engine for ASM rules.

coding concrete ASMs directly in some programming language. This method is illustrated by C++ programs refining ground model ASMs for the small case studies known under the names of Steam-Boiler and Production-Cell; see [43, 332]. It has been applied also to a ground model ASM for the operating system MINIX on the X86-architecture, which has been refined in [186] to code written in C and in Java. In [150] the method is reported to be used with Java for a complex industrial telecommunication software project.

interpreting concrete ASMs by an interpreter for a class of ASMs. The first example of such an interpreter is Kappel's ASM interpreter in [301], written in Prolog to make the Prolog models in [71, 72, 74] executable. An industrial counterpart was developed as early as 1991 in [143] and was used in the context of the ISO standardization process of Prolog [101]. An elegant 9-line Prolog interpreter for basic ASMs appeared in [36]. Later numerous other interpreters were written in various languages and for various purposes. We refer the reader to Chap. 9 for the complete details and mention here only two such interpreters which have been successfully used in industrial projects. The ASM Workbench [170] provides not only an interpreter for a large class of ASMs (namely supporting basic ASMs with external functions written in ML), but also a tool environment including basic functionalities such as parsing, abstract syntax trees, type checking, pretty printing, and in particular a transformation of ASMs into FSMs which can be model-checked using SMV, see [175]. The ASM Workbench has been successfully used for testing purposes in the FALKO project [121]. In [355] it is used to define an executable semantics for UML which covers real-time aspects. The machine AsmGofer [391, 390] is equipped with a graphical user interface to execute turbo ASMs, thus supporting the structuring and composition concepts defined in [134]. It has been illustrated by the Light Control Case Study [125] and has been applied in an industrial ASIC design and verification project (including a compiler from ASM to VHDL) and for testing the Java and the JVM models in [406] against current implementations of Java/JVM. In [152] it has been extended by a simulator for UML state machines.

compiling ASMs to executable code. Three systems are to be mentioned under this heading. In [391] a scheme is developed for compiling ASMs from the syntax of the ASM Workbench [170] to C++, coding algebraic types, pattern matching, functional expressions, dynamic functions and simultaneous updates in such a way that efficient C++ code is obtained without losing the structure of the original ASM specification. The proprietary compiler has been successfully applied in a middle sized software development project at Siemens [121, 391]; the ASM-to-VHDL compiler developed in [392] has been used at Siemens in an ASIC design and verification project.

In 2001/02 the ASM-based language AsmL running now on the .NET platform has been made accessible at Microsoft Research [201] and has been applied within Microsoft (see [27, 28, 224, 30, 31]), including test-case generation from AsmL specifications (see the algorithm AsmGenFsm on p. 123). This language exploits the abstraction potential of ASMs to offer object-oriented and component-based structuring principles. Like Eiffel [333], AsmL supports the formulation of pre-conditions and post-conditions in the specification of functions: if the pre-condition is not true, or if after the execution of the function the post-condition is not true, then an exception is thrown. Currently Microsoft Research supports the development of an SDL-to-AsmL compiler at the Humboldt University of Berlin to make the ASM definition of the SDL'2000 semantics (see [194]) executable.[1]

[1] Personal communication (e-mail of October 8, 2002) from A. Prinz and M. v. Löwis to E. Börger.

> The Open Source project XASM [15, 17] at present comes with an XASM-compiler to C, a run-time system and a graphical debugging and animation interface. It offers an interface to C allowing (a) C-functions to be used as static or monitored ASM-functions and (b) ASMs to be called from within C-programs. It has been applied for Montages and Teich's architecture and compiler co-generation project, as described in Sect. 9.4.2. In [299] it has been used to execute tests of a compiler for the extension mpC of ANSI C by arrays and parallelism. In [51] it is used as target for translating specifications written in *SiteLang*, a language which extends basic ASM constructs by constructs tailored for stepwise refinable specifications of information services. For the treatment of recursion in XASM see the end of Sect. 4.1.2.

Unfortunately none of these systems comes with a manual which describes by a hierarchy of stepwise refined ASM models the design decisions which led from the underlying ASM concept to the implementation (except for the submachine concept of AsmGofer which is documented by the turbo ASM definition in [134] and is realized also in AsmL).[2] More importantly none of these systems supports the powerful ASM refinement techniques other than the submachine concepts realized in AsmGofer, AsmL and XASM. Also none of these systems supports async ASMs.

Problem 30 (Implement ASM refinement techniques). Enhance an ASM execution engine to support *provably correct* ASM *refinement* techniques which allow us to make the intended equivalence of the specification and its implementation during the design process checkable. Link this to existing (a) system documentation and versioning techniques, (b) theorem provers.

Problem 31 (Implement asynchronous ASMs). Extend an ASM execution engine to one which also implements async ASMs and offers for experimentation with different schedulers a facility to plug in different scheduling policies which realize the partial order underlying async ASM runs.

[2] In AsmL **seq** is written **step**, **iterate** is written **step until fixpoint**, **do** *R* **until** *Cond*, and **while** *Cond* **do** *R* is written **step until/while** *Cond R*. In addition sequential stepwise iteration is offered through sequences and sets, and the construct **try** *P* **catch** *T Q* is simplified to **try** *P* **catch** *Q* (deleting from the construct the possibility to explicitly program exception patterns *T*). Submachine calls appear as method calls for which, besides the general version called "update procedures", also the specialization to functions as used in programming is offered.

9 History and Survey of ASM Research

The ASM method for the high-level design and analysis of computing systems and their stepwise refinement to executable code grew naturally out of the foundational concern which led to the discovery of the notion of ASMs, although it took some time for the concept to sink in. Indeed, as often happens with ideas which change the way we look at things, its "real"ization – through becoming the basis for an intellectual, human-centric though machine supported instrument for practical system design and analysis – encountered significant resistance in the scientific community. In this chapter, which can be read independently of the rest of the book, we survey the rich ASM literature and the salient steps of the development of the ASM method from its epistemological origins. The survey covers the period from 1984 to 2002.[1]

From 1984 to today one can distinguish four phases which we are going to survey in the following sections:

- **Idea of a sharpened Church–Turing thesis** for "an alternative computation model which explicitly recognizes finiteness of computers" [240].
- **Recognition of the practical potential** of the abstract machine concept for building and analyzing reliable ground models and their refinements to executable code, an insight which came through the experience gained at the beginning of the 1990s by extensive modeling of the semantics of various programming languages and their implementation [76].
- **Practicability test** for ASM-based concepts in real-life applications, a broadband experimental effort which shaped the ASM design and analysis method through numerous modeling and verification projects for real-life architectures, virtual machines, protocols, and controller software. Carried out 1993–1995 [80], it influenced the final definition of ASMs in [248].
- **Integration of the ASM method** into established software development environments which created the practical ASM approach to high-level system design and analysis, as one sees it nowadays, ready to be deployed in industrial settings [86]. In 2000/01 also the first part of the original foundational goal was completed by the discovery of a proof for the se-

[1] This chapter is an elaboration of [92]. We gratefully acknowledge the permission of the J. of Universal Computer Science to reuse the paper which is freely available at the JUCS web site.

quential ASM thesis from three basic postulates [249] and of its extension to synchronous parallel algorithms [61].

9.1 The Idea of Sharpening Turing's Thesis

It was an epistemological concern which led Gurevich to the idea of abstract state machines, namely the goal to sharpen the Church–Turing thesis by an "alternative to the notion of a potentially infinite machine", named a "uniform family of finite machines" and described as a notion which "is to remain informal" [240, p. 1]. In 1985 the program was made more precise in a note to the American Mathematical Society [241], from which we quote the central part:

> First, we adapt Turing's thesis to the case when only devices with bounded resources are considered. Second, we define a more general kind of abstract computational device, called dynamic structures, and put forward the following new thesis: Every computational device can be simulated by an appropriate dynamic structure – of appropriately the same size – in real time; a uniform family of computational devices can be uniformly simulated by an appropriate family of dynamic structures in real time. In particular, every sequential computational device can be simulated by an appropriate sequential dynamic structure.

In 1988 a more detailed exposition of this primarily complexity theoretic program appeared [242], but again without any concrete definition of the abstract machine concept. Apparently at that time there was still some hesitation on "different classes of dynamic structures" which "may be defined by imposing syntactical restrictions on transition rules, by allowing or forbidding the evolution of the signature (the language) of the current configuration, by allowing or forbidding the creation of new universes (sorts, types) and the elimination of old ones, and so on" ([242, p. 413]). In this period there was also the formulation of the Kolmogorov–Uspenskii thesis [306] as stating that "every computation, performing only one restricted local action at a time, can be viewed as the computation of an appropriate Kolmogorov–Uspenskii machine" [243], a subclass of what became known as Schönhage's storage modification machines [396], which later could be characterized as a class of unary sequential ASMs [177].

In a series of lectures entitled "Toward an alternative computation theory: algorithms and dynamic model semantics", delivered to the computer science PhD program in Pisa from May 26 to June 6, 1986[2] Gurevich ex-

[2] In previous publications, e.g. [92], erroneously only the year 1987 was named. As Börger found out from his notes, he had invited Gurevich to lecture for the first time in the Spring of 1986, then again in May 1987 (where the title of the course was "Dynamic algebra semantics").

plained the concept of dynamic structures – later called evolving algebras and eventually ASMs – by examples, namely Turing machines, stack machines and some Pascal programs. Börger learnt ASMs from these lectures and suggested adding the course material to [244, Sect. 10, 11].[3] There the proposal appears to use "dynamic structures" for an operational semantics of imperative programming languages, realized for the core of Modula-2 in Morris' PhD thesis [338]. During the winter of 1988/89 ASMs were tried out to define the semantics of Prolog by an execution-oriented yet abstract model, which was intended to become complete and precise, but nevertheless of manageable size and reflecting the logical content of Prolog programs in a transparent way, to be useful to programmers [71, 72, 74]. The goal was achieved by introducing the notion of a ground model and the stepwise refinement method into modeling with ASMs (see Sects. 2.1.2, 3.2), exploiting the possibilities for abstraction that are inherent in the ASM concept. Stepwise refinements allowed us to separate orthogonal language features by modules of rule sets (*horizontal refinement*) and to deal with them at different levels of detail (*vertical refinement*), supported by an appropriate classification of functions into static and dynamic, external and internal (later more precisely distinguished into basic and derived, monitored (or input), output, controlled, and shared functions or locations [80, 86]; see Fig. 2.4).

9.2 Recognizing the Practical Relevance of ASMs

Through his sabbatical work at IBM Germany in 1989/1990 Börger realized that his ASM model for Prolog solved some central problems that the ISO Prolog standardization committee had faced for years [101], in particular the database update view problem [103, 126] and the semantical problems related to Prolog's solution-collecting predicates [130]. In fact a version of this ground model became the ISO standard definition of the dynamic semantics of Prolog [291], after many unsuccessful attempts documented in the literature to provide such a definition using traditional approaches (see the detailed discussion in [71, Sect. 4]). Through the work with the software engineers from IBM, Quintus, Bim, Interface and Siemens, who at the time were developing commercial implementations of Prolog, the usefulness of the ASM concept became apparent for *supporting changing designs* in an industrial standardization and development process. It was recognized that flexibility is gained by using ASMs for modeling and for prototypical (mental or machine supported) simulations. This showed up through the ease with which ASMs allowed the practitioners to perform the following three of their daily duties:

- to rigorously model and document design decisions, i.e. **building ground models**[4] in a faithful and objectively checkable manner. This means turn-

[3] The stack and Turing machine ASMs went into [245, Sect. 4, 6].

ing descriptions expressed in application domain (typically natural language) terms into precise abstract definitions (see Sects. 2.1.1, 3.1), which the software engineers were comfortable to manipulate as a *semantically well-founded form of pseudo-code over abstract data*,

- to *adapt abstract models to changing requirements*, and to refine them in successive steps – in a controllable and well-documentable way – to their implementation, thus providing **practical forms of refinement** for linking ground models by hierarchies of intermediate models (modules) to executable code (see Sects. 2.1, 3.2 and `RefinemtMeth` (⇝ CD)),
- to turn such precise abstract pseudo-code models into prototypical *executable versions* which can be used for **simulations prior to coding** of the system under development (see Section 8.3).

Unfolding the potential of the concept of ASMs for such a *method of modeling-for-change*, at the desired level of abstraction, to be used together with an appropriately chosen mathematical verification and experimental validation technique, was the result of extensive experimentation during the years 1990–1992. It was focussed on the following three issues:

1. **Adaptations and extensions of models** via *horizontal refinements*, realized by refining the basic ASM model for Prolog to some of the major extensions of the language and their implementations, to be precise the following seven ones (see Backtrack on p. 114):
   - Colmerauer's Prolog III, obtained by adding to the unifiability check of the Prolog model a solvability test for general constraints [135],
   - IBM's Protos-L, developed and implemented on the Protos Abstract Machine at IBM Germany, obtained from the Prolog model by adding to it type constraints and a solvability predicate [40, 42, 41],
   - the functional-logic language Babel and its implementation on the Narrowing Machine [118], obtained by adding to the backtracking rules of the Prolog model rules for the reduction of functional expressions to normal form,
   - Lloyd's and Hill's logic programming language Gödel, obtained by abstracting in the Prolog model from the deterministic and sequential execution strategy of ISO Prolog [124],
   - B. Müller's object-oriented Prolog [348, 347], obtained by enriching the four ASM rules for the user-defined core of Prolog [75] with rules for object creation and deletion, data encapsulation, inheritance, messages, polymorphism and dynamic binding,
   - Sauer's adaption of the Prolog model to an ASM defining the semantics of the domain-specific language HERA, tailored for programming scheduling algorithms for business processes on the basis of given heuristics [385, Chap. 3.3],
   - the main parallel extensions of Prolog (see below).

[4] *Ground models* [80] were originally called *primary models* [76, Sect. 3].

2. **Stepwise detailing of models** by *vertical refinements*, realized for the WAM implementation of Prolog by defining a chain of 12 proven-to-be-correct refinement steps which link the high-level Prolog model (in fact its streamlined version in [131]) to its implementation by Warren Abstract Machine code [128, 129, 132]. It turned out that the models and the proofs could be reused and extended to derive the correctness also for the following four implementations:
   - the implementation of a high-level CLP(R) model on the Constrained Logic Arithmetical Machine, developed at IBM Yorktown Heights [133],
   - the implementation of a high-level Protos-L model on the Protos Abstract Machine [42, 41],[5]
   - the parallel execution of Prolog on distributed memory [21] (see also the related later work [351]),
   - the implementation of scoping of procedure definitions in a sublanguage of Lambda-Prolog where implications are allowed in the goals [313].
3. **Making abstract models executable** for their experimental validation, realized during the academic year 1989/90 in Kappel's Diplom thesis at the University of Dortmund [301, 302], and a year later at Quintus [143], allowing one to simulate the ASM Prolog models defined in [71, 72].

The method of successive refinements of ASMs was applied in [251] to modeling the dynamic semantics of C, structuring an earlier flat version of this model, which had followed the spirit of the early unstructured Modula-2 ASM [338] and had inspired a similar project for Cobol (started in [417], though not continued). Before that, in Blakley's PhD thesis [53] an unstructured ASM model for a subset of Smalltalk had been defined. Inspired by [338], ASMs are used in [232] to provide a succinct operational description of typical object-oriented features like object creation, overriding, dynamic binding and inheritance in the context of data models. In [397] an ASM rule was added to define cooperative message handling, by describing the run-time search of the most specific cooperation contract in the inheritance hierarchy which implements a cooperative message, i.e. a message which involves several objects on the basis of cooperation contracts. Later the modeling of object-oriented programming language features was taken up once more in Ann Arbor, using once more the refinement method to extend the ASM model for C to one for C++ [420].

In 1991 Gurevich wrote for his column on Logic in Computer Science in the Bulletin of the EATCS the so-called ASM tutorial [245], which is based on lecture notes from his Fall 1990 course on Principles of Programming

[5] Beierle [39] turned this construction into a general implementation scheme for CLP(X) over an unspecified constraint domain X, by designing a generic extension WAM(X) of the Warren Abstract Machine and a corresponding generic compilation scheme of CLP(X) programs to WAM(X) code.

Languages at the University of Michigan, containing most notably the first definition of a subclass of basic ASMs.[6] In the same year a textbook introduction to ASMs was written [73]. This was illustrated with machines which operate on standard data structures and with the tree-based core Prolog machine [75], drawn from the notes of Börger's lectures on the Semantics of Programming Languages in a summer school held in 1989 in Cortona/Italy which triggered the first European PhD project on ASMs [375]. Gurevich completed the tutorial definition in the so-called Lipari guide [248], which are the lecture notes of a course delivered in 1993 at the Lipari/Sicily summer school on the *Specification and Validation Methods for Programming Languages and Systems* [79]. The definition has essentially remained stable since then,[7] in fact it constitutes the basis for the proof established five years later for the sequential version of the ASM thesis from three fundamental postulates [249]. As observed by Blass [54, p. 11], this subclass of so-called *sequential* ASMs is natural from a logical point of view, corresponding to the class of quantifier-free interpretations in logic.

The tutorial and the Lipari guide incorporate the experience which had been gained through the early applications of ASMs,[8] those described above and those reported in the contributions to the first international ASM workshop which was organized as part of the 13th World Computer Congress in Hamburg in 1994 [361]; see below. Besides the introduction of the **choose** and **forall** operators, the major addition of the Lipari guide to the tutorial concerns the notion of async (there called distributed) ASM runs. In [257] an ASM model had been developed for the parallelism of Occam. It was presented by Gurevich in May 1990 in another series of lectures in Pisa, which inspired the concrete theme for Riccobene's PhD thesis [375] to refine the ASM model for Prolog by the different forms of parallelism encountered in Parlog, Concurrent Prolog, Guarded Horn Clauses, and Pandora [122, 123, 374]. Later another instance of refinement and parallelization of Prolog to a semi-ring based constraint system appeared [52], replacing the Call and Select Rules of [71] by a Reduction Rule which activates a child process simultaneously for each alternative of the current process. The notion of parallelism in these models was generalized in [227] where ASM models appear for the Chemical Abstract Machine and the π-calculus. Eventually in

[6] The operators **choose** and **forall** do not appear – non-deterministic choice is handled by external choice functions. The semantics of conflicting rules is different from the definition adopted later: in the case of a conflict among rules a consistent rule subset is chosen for execution.

[7] The initially present construct to shrink domains, which was motivated by concerns about resource bounds, was abandoned because it belongs to garbage collection rather than to high-level specification. Concerning variations concerning non-determinism and inconsistent update sets see the previous footnote and [262].

[8] See the acknowledgment in [248, p. 11].

1995, the Lipari guide definition of distributed ASM runs superseded these more restricted definitions of concurrency for ASMs (see Def. 6.1.1).

9.3 Testing the Practicability of ASMs

Once the practical potential of the ASM notion had been understood, a natural next step was to test its practical impact by trying out ASMs for the modeling and rigorous mathematical and experimental analysis of a variety of complex real-life computing devices, looking for relevant problems beyond those of the semantics of programming languages. In the Fall of 1992 Börger defined this program and started it with his students by systematically extending the application of ASMs to the specification and analysis of virtual machines, processor architectures, protocols, embedded controller software and requirements capture. As part of this effort the 1993 Lipari Summer School on *Specification and Validation Methods* [79] was organized, which triggered the fundamental Lipari Guide [248] and a series of papers applying ASMs [133, 420, 114, 283, 256].[9] The endeavor attracted many researchers and resulted in the elaboration of a fully fledged system design and analysis method built upon the notion of ASMs, pragmatically confirming the definition of ASMs in the Lipari guide.

9.3.1 Architecture Design and Virtual Machines

For computer architectures, the work on program started during the winter term 1992/93 with a reverse engineering study, commissioned by a group of physicists in Pisa and Rome for the massively parallel APE100 architecture, a rather successful dedicated machine which had been developed for floating point intensive numerical simulations in Lattice Gauge Theory [32, 33]. As part of the work for Del Castillo's Tesi di Laurea, a programmer's view ground model has been defined in [97] and refined in [102] to a provably correct decomposition of the control unit processor zCPU, a VLSI-implemented microprocessor with pipelining and VLIW parallelism, built from formally specified basic architectural components. The intermediate models, obtained by stepwise refinement, correspond to views of the architecture provided by different languages used within the APE100 compilation chain, a crucial part of the software environment of the machine. A companion Tesi di Laurea [119] had the goal of isolating the underlying pipelining scheme and of proving its correctness via stepwise refined models. Hennessy and Patterson's RISC machine DLX was chosen as the reference architecture to treat the standard

[9] A companion Lipari Summer School on *Architecture Design and Validation Methods* followed in 1997 [88]. Another one was organized in 2002 on *Software Engineering*, with three of the seven courses based on ASMs (held by Gurevich, Börger and Riccobene).

pipelining methods which handle structural hazards, data hazards and control hazards, respectively. Starting from the one-instruction-at-a-time view of the processor, each hazard is dealt with in one dedicated refinement step which concentrates on the corresponding machine property (see Sect. 3.3). This ASM based architecture design and verification method was taken up by other research projects; see below.

For virtual machines, the program to extend the WAM work [132] beyond issues concerning the implementation of programming languages started with modeling the well-known Oak National Laboratory public domain software system PVM [214], a general-purpose environment for heterogeneous distributed computing. The virtual machine appears there logically as a single distributed-memory computer, "created" by PVM out of a dynamic heterogeneous set of physically interconnected and concurrently operating machines, namely host computers which can be dynamically added to or deleted from the virtual machine and may belong to a variety of architectures (including serial, parallel and vector computers). Glässer suggested trying out the notion of async ASMs for modeling PVM. In [107, 108] a ground model for the Parallel Virtual Machine is defined at the C-interface level, where it appears as an async ASM with a characteristic event handling mechanism and message-passing interface (reflecting the uniform access that the PVM agents ("daemons") have to daemons on other host machines, whether multicast or point-to-point between single tasks).

The cooperation on modeling the PVM triggered the project to provide a ground model for the, at the time, new IEEE standard VHDL'93 [288] of the hardware design language VHDL. The models defined in [111, 112] come as an async ASM and cover the entire language with the new features of the 1993 standard, in particular the complete signal behavior and the time model, including pulse rejection limits and the various wait and signal assignment statements involved in the subtle issues related to postponed processes. Later, these ASM models were used in W. Müller's PhD thesis at the University of Paderborn [349] for defining the semantics of a pictorial extension PHDL of VHDL'93, by a group of Toshiba engineers for an extension to analog VHDL and Verilog [383, 384, 380, 382, 381], and recently for an adaptation to SystemC [345] and to SpecC [344].

Conversations with Langmaack since 1991 on the relations between the ASM-based method used for proving the correctness of Prolog-to-Wam compilation [132] and the European ProCoS project on provably correct systems [318], which aimed in particular at a correctness proof for executing compiled Occam programs [350] on the Transputer architecture [236], have led to the Transputer-verification ASM case study. In [104] the Transputer instruction set architecture has been modeled by a hierarchy of stepwise refined ASMs to support the correctness and completeness proof for the general compilation scheme of Occam programs to Transputer code proposed in [289, 290]. As the basis for the semantics of truly concurrent and non-deterministic Occam programs an appropriate ground model ASM has been used, which was defined in [105]. It leads from the programmer level (see Fig. 6.20) by various proven-to-be-correct refinement steps to the starting point of the hierarchy of Transputer models, namely a machine which appears as an abstract processor running a high- and a low-priority queue of Occam processes. This ASM-based method for a mathematical verification of real-life compilation schemes with respect to a rigorous semantics of source and target languages has been taken up again in the Verifix project discussed below, based upon the recognition in the ProCoS project that to establish the correctness of (modulo hardware correctness) reliable initial compilers, in addition to verifying the compiling function the verification of a compiler implementation is also needed.

9.3.2 Protocols

Using ASMs for modeling and verifying protocols has been started in three papers which were published in [79]. The work in [114] is an answer to one of the, at the time, frequent public challenges of ASMs: Abraham and Magidor at a Dagstuhl seminar in June 1993 expressed doubt that the atomic character of function updates in ASM transitions would prevent these machines from naturally reflecting complex combinations of low level durative actions. The discussion in the seminar focussed on the concrete example of Lamport's mutual exclusion protocol known as the bakery algorithm [314, 315], for which Abraham had presented a new proof method, relying on a distinction between a lower and a higher view of the algorithm [2, 3]. In [114] three ASMs are built. The first one serves as a ground model to faithfully reflect Lamport's protocol. By abstracting from the low-level read and write operations of the ground model, a high-level model with atomic actions (non-overlapping reads and writes) is defined and then

– proven to have the desired correctness and liveness properties (under four natural assumptions about the abstract functions that were used),
– proven to be correctly refined by the ground model (proving that the assumptions made for the abstract model hold for the implementation).

In the third ASM, the state of the abstract machine is refined by replacing atomic actions with durative ones, allowing overlapping of reads and writes to shared registers. It is proved that this refined notion of state satisfies the corresponding assumptions made for the machine with atomic actions. The resulting correctness and liveness proofs for the bakery algorithm considerably shorten numerous other proofs in the literature. The arguments are expressed using a mapping of ASM moves to moments of continuous and linear real time (see Sect. 6.4), but they can and have been rephrased in the more general terms of partial orders [258] which characterize the notion of async ASM runs. A further analysis of the role of timing constraints on async ASM runs for proving the correctness of refinements of asynchronous algorithms with continuous time appears in [157].

Reference [283] contains another illustration of the technique of stepwise ASM refinements for a real-world protocol, providing a faithful readable specification together with an understandable correctness proof (see Sect. 6.3.1). The Kermit file-transfer protocol chosen as the object of the study had been presented by Knuth in his foreword to the Kermit book [163] by expressing the

> hope that many readers of this book will be challenged to find high-level concepts and invariant relations by which various versions of the Kermit protocol can be proved correct in a mathematical sense.

Huggins deals with a complete version of Kermit, showing how this protocol combines the underlying alternating bit protocol and its sliding windows extensions, thus differing from numerous earlier verification studies in the literature which had focussed on these two simple protocols in isolation. Later, two other ASM formalizations of the alternating bit protocol alone appear, one in [328, Chap. 5] as an illustration of a modularization and communication concept implemented on top of ASMs, the other one in [239] to illustrate the application of algebraic-categorical composition schemes to ASMs.

In [256] a processor group membership protocol is modeled as an async ASM (see Sect. 6.3.2), a typical protocol of the kind used to achieve fault tolerance for distributed computing services. The underlying assumptions for the well-functioning of the protocol are made explicit, such as the reliability of the message passing mechanism, lower bounds for processor recovery, upper bounds for message exchange time, etc. These assumptions are then proved to imply the correctness of the protocol, namely that despite delays in message passing and server failures, the protocol achieves a global agreement about the set of all correctly functioning processors in a synchronous system.

This debut of ASMs for protocol verification triggered numerous other ASM projects in the area, which are discussed below.

9.3.3 Why use ASMs for Hw/Sw Engineering?

In the first half of the 1990s, the concept of ASMs encountered considerable scepticism and not seldom strong opposition in the scientific community, even in Europe. Interestingly enough the criticism came from two directions, on the one side from researchers whose longstanding trust in purely declarative logico-algebraic methods made them view ASMs as nothing else than yet another form of an old fashioned low-level operational method, on the other side from researchers who claimed earlier fathership for the notion in a variety of forms.[10] Both objections contain a grain of truth. They motivated the attempt, made in 1995 [80] and again in 1998 [86], to understand better the relation between ASMs and established formal methods and to formulate the stringent scientific reasons why, after decades of intensive research in the area, an apparently new concept was proposed as the basis for a practical high-level system design and analysis method. The articles explain that although a logician discovered the concept of ASMs, as an outsider driven by a foundational concern, the *notion* triggered the development of a *method* which allows one to really "complete the longstanding structural programming endeavour (see [164]) by lifting it from particular machine or programming notation to truly abstract programming on arbitrary structures" [86, Sect. 3.1]. As a matter of fact the notion is made up of three ingredients which had been there already for a long time but with nobody to bring them together, namely the notion of pseudo-code, IBM's concept of virtual machines [305] together with Dijkstra's related concept of abstract machines [183] and the fundamental idea of using Tarski structures as the most general notion of *abstract states*, an idea which pervaded the area of abstract data types and algebraic specifications [325, 265, 359, 210] without becoming relevant for practical system design:

$$\text{ASM} = \textbf{A}\text{bstract } \textbf{S}\text{tate} + \text{Abstract } \textbf{M}\text{achine}.$$

[10] Among the earlier definitions we know, the one which comes closest to basic ASMs is that of *data space* defined in [161]. It proposes to mathematically capture the notion of a virtual machine as a "mathematical machine with structured states", which "emphasizes the interaction between data and control structures", to be used for implementations of abstract models and their testing at high abstraction levels. The definition given is that of an arbitrary partial function (called "processor") which transforms elements of a set X of abstract states and for whose description an unrestricted access is allowed to what is called an "information structure", namely an arbitrary set of static functions with domain X and arbitrary data types as range. The class of allowed processors is not specified; judging from the provided examples and later case studies it seems to have been intended to permit for their definition standard programming constructs plus some bounded parallelism (read: simultaneous application of different functions). These partial functions roughly correspond to the *next state* function of basic ASMs, but without providing a concrete universal set of machine operations – à la basic ASM transition rules of Sect. 2.2.2 – which compute these functions. To characterize states it is said that each information structure function "describes some aspects of the states, and for any given state x, we expect the combination or 'sum' of all aspects to determine the 'meaning' of x." Two properties (of completeness and orthogonality of information structures) are defined which permit states to be viewed as the Cartesian product of the ranges of the functions in the information structure. This stands for the concrete definition of ASM states as algebras of the underlying signature. Data spaces are exposed to the frame problem by the decision to use total functions to determine next states in a functional manner, instead of going via updates of finitely many locations of dynamic functions (in fact no classification of functions à la Sect. 2.2.3 is given). The definition of "implementation" realizes that "for the successful refinement of a given level of abstraction, it is necessary to consider the control structure together with the collection of data types whose interaction is defined by that control structure"; however, it restricts implementations to $(1, m)$-refinements with $m > 0$ (see Sect. 2.1.2).

9.4 Making ASMs Fit for their Industrial Deployment

The positive experience gained through the multitude and diversity of successful ASM-based modeling and verification projects convinced Börger of the potential of ASMs for industrial system development and brought him to the decision, in the Spring of 1994, to apply ASMs to down-to-earth software engineering problems and to pave the way for their industrial deployment. At the time it was hard to find support for projects directed towards this goal, the effort was judged even by some leading peer in the ASM community to be a waste of time. Nevertheless it attracted more and more researchers and eventually led to an industrially viable, theoretically well-founded *system development method built around the concept of an ASM*, an approach which supports practical system design and analysis by application-tailored high-level modeling that is seamlessly linked to executable code, going through mathematically verifiable, experimentally validatable and objectively documentable refinement steps.

The program was articulated through the following three major themes surveyed below:

- investigation of *practically relevant case studies* and system development problems, to identify the strengths and weaknesses of ASMs for software development, and to compare ASMs with established formal methods,
- application of ASMs in challenging *industrial software engineering projects*,
- *integration of tools* for simulation, verification, documentation and maintenance of ASMs during the software development process.

9.4.1 Practical Case Studies

In the Spring of 1994 the preparation of a research competition among methods for semantics and specification was started with the declared goal to "contribute to a realistic comparison, from the point of view of practicality for applications under industrial constraints, of the major techniques which are currently available for formally supported specification, design, and verification of large programs and complex systems" [9, p. 1]. This developed into the Steam Boiler Dagstuhl Seminar [7] – by the name of the industrial case study that the participants were asked to solve – and into the Steam Boiler Case Study Book [8], which came out of a subsequent international call for participation. Although only a quarter of the numerous solutions came up with a validatable executable model, to provide a complete ASM solution turned out to be a relatively easy task. In [43] first a ground model ASM is defined – a rigorous form of the given problem description which is phrased in such a way that it can be checked by the domain expert to be faithful to the intended requirements. Then the ground model is stepwise refined to C++ code, each intermediate model reflecting some design decision.[11] Proofs for some of the required system properties are reported, and during a demo to the seminar Durdanovic showed his C++ program to successfully control the simulator [326] that Lötzbeyer had built at FZI in Karlsruhe for an experimental evaluation of the problem solutions.

Mearelli's Tesi di Laurea, started at the beginning of 1995, had the goal of extending the positive experience made with the development of the steam-boiler control software by a test of the integratability of the ASM method into the various phases of the software development cycle. As an experimental guide an ASM ground model for the, at the time, freshly published Production Cell Case Study [323] was developed and stepwise refined to C++ code (see [120, 332] and Sect. 5.1), with particular attention to the following two features:

- the modularity of the specification and the code and their structural similarity (to support code inspection), together with their complete documentation as support for inexpensive changes and easy maintenance,
- the applicability of standard verification and validation methods to prove the desired properties stated in the requirements.

In fact in [120] all the required correctness, safety, performance and liveness conditions are proved by mathematical argument – typically for the high-level model under appropriate assumptions, proving these assumptions to hold at the refined level. The C++ code produced by implementing the final refined ASM model [332], taking care that the specification can be traced through the structure of the code, has been validated by extensive experimentation with the simulator built at the FZI in Karlsruhe. It has also been submitted for an inspection process to another software engineering Dagstuhl Seminar, organized in 1997 and focussed on "Practical Methods for Code Documentation and Inspection" [117]. To test the integratability of mechanical verification methods into the software development with ASMs, the Production Cell models were used for model checking experiments [424, 362] and for theorem proving with PVS [207]. As one of the first test examples for his code generator from ASM specifications (see below), Schmid has used the Production Cell ASM for generating efficient C++ code whose structure allows one to trace the specification to support the reliability of code inspection [391].

In 1999, through another software engineering Dagstuhl seminar, it has been tried to bring together once more researchers from academia and practitioners from the software industry to evaluate the contribution of so-called formal and informal methods for solving practical system engineering problems. The seminar was focussed on problems encountered in industrial software development processes to capture, document and validate requirements in a principled manner [115]. This seminar, too, was centered around a practical case study which triggered some complete solutions published in [113].

[11] One can view it also the other way round as lifting the C++ code to a more abstract level with simultaneous updates, access to historical function values, etc., a methodological view which was held by Durdanovic and has been further elaborated in [171].

The ASM solution of this Light Control Case Study (see [125] and Sect. 6.2) showed that building and simulating ASM ground models is an efficient method to capture, validate and document requirements for a precise reason: it allows one to document in a traceable way the desired mix of rigorous, explicit ("formal") elements of description and of others intended to remain vague and implicit ("informal"). Such a mix is needed to bridge the gap between the views of the application-domain expert and of the system designer, persons who speak different languages but nevertheless have to understand each other to be able to agree on the definite characteristics of the system to be developed. This observation has led to Cavarra's PhD project [152], where an attempt is made to link ASMs to so-called semi-formal specification techniques as they are used in industrial practice. The formal versus semi-formal issue is an instance of the more general need for an encompassing framework to combine heterogeneous specification elements, which is discussed in [219, 16, 172]. The 1999 Dagstuhl Seminar [115] brought out this requirements engineering aspect of the systematic *separation of different concerns*, which has been advocated in [80, Sect. 4] as a characteristic and major distinctive feature of the ASM method compared to other approaches to system design (see also [77, 247]). It was pointed out again in [87, Sect. 1] that such a separation of orthogonal system features, and of different methods to model and analyze them, is necessary for a successful combination of multiple ways to construct and relate different system views – by modeling, simulating and verifying the system with different degrees of precision. Indeed it is one of the reasons for the success of the ASM method that the mathematical "openness" of the basic ASM concept allows fine-tuning this *separation-for-integration strategy* – a classical divide-et-impera approach – to the needs of the system to be developed or investigated (see Sect. 2.1).

Also, other case studies which had been presented during this period as challenges to the scientific community have been elaborated using ASMs. Examples are the ASM solution [284] to the Broy–Lamport specification problem [144], which had been formulated in 1994 for the Dagstuhl seminar on reactive systems, or the real-time-based ASM modeling and verification in [253, 34, 35] of the Railroad Crossing Problem to which Heitmeyer and Mandrioli's book [277] is dedicated (see Sect. 5.2).

9.4.2 Industrial Pilot Projects and Further Applications

ASMs at Siemens/Munich. For his sabbatical year 1995/96 Börger decided to find out whether the above-described applications of ASMs to requirements capture and to design and analysis of controller software scale to the needs of industrial design environments. This developed into a fruitful cooperation during the years 1996–1999 with Päppinghaus at Siemens in Munich, largely focussed on design methods for railway-related software [96]. It led to a rather successful application of ASMs in a middle-sized industrial software development project (FALKO, May 1998–March 1999, reported in [121], see `Falko` (⇝ CD)). The salient methodological outcome of this cooperation was the creation of the first *prototypical ASM based industrial development environment*, which supports a seamless flow from the definition of an ASM ground model to compilable (in the specific case C++) code, including high-level testing and code maintenance.

Obviously, to make this project succeed, appropriate ASM tools had to be created and used extensively. In the Spring of 1995 Del Castillo had started his PhD work, located at the university of Paderborn, to build a tool environment for the specification and simulation of ASMs. The FALKO ground model was formulated in the ASM-SL language that Del Castillo meanwhile had defined for the ASM Workbench [170], so that in the FALKO design phase early versions of this machine could be used for extensive testing of the high-level FALKO models, prior to coding. At the end of the design phase, as part of his PhD project, which started in the summer of 1998 at Siemens Corporate Research in Munich, Schmid developed a compiler from ASM-SL to C++ [391] – it generated the program which since March 1999 is in daily, failure-free use by the Vienna Subway Operator for the validation of subway operational services. For documentation and maintenance purposes Schmid developed a literate programming tool allowing one to keep a single collection of consistent HTML documents from which the ASM-SL code can be extracted as input to the ASM Workbench or to the compiler, but also in pretty-printed form for the human reader.

In the third part of his PhD work [392, Chap. 2], Schmid successfully applied this tool-supported structured ASM modeling and refinement technique also in a large ASIC design and verification project at Siemens München. This includes the definition of a notion of ASM components which was used for the behavioral specification of digital hardware circuits, and of the development of a compiler from ASM components to VHDL.

CO-Monitoring system. Another early industrial application of tool-supported ASMs, namely for an automated fire detection system adopted by three major German coal mines, is reported in [148]. Kappel's Prolog-based interpreter for basic ASMs [301] was extended for this project to support the parallel execution of independent modules, representing distributed processes which are synchronized via stream based communication. The extension comes with a visualization mechanism for run data.

DFG projects Deduktion/Verifix. In 1994 Börger suggested to the German Research Council project "Deduktion" to apply mechanical proof verification for proving properties of ASM models of real-life programs [393]. In particular, some of the refinement steps in the above-mentioned WAM correctness proof have been mechanically verified using Isabelle [369], whereas using KIV the entire proof has been elaborated for a mechanical check, using not only all the 12 refinement steps from [132], but adding one more intermediate model to make the proof feasible for the machine [388, 386]. In [387] a scheme is extracted from that work for proving the correctness of ASM refinements using generalized forward simulation (see Sect. 3.2.3). This use of ASMs for proving the correctness of compilation schemes has been further developed in a part of the Verifix project [229] of the German Research Council, where instead of schemes for compilation into virtual machine code the correctness of concrete compilers compiling into real-life machine languages is investigated. We mention only a few examples from the subpart of the Verifix project where ASMs are used, which appeared in [211, 439, 188, 212, 274, 275, 231, 273]. A ground model ASM for the DEC-Alpha processor family has been extracted from the manufacturer's handbook; compiler back-ends have been built based on realistic intermediate languages to prove their correctness, using generic PVS theories developed in [187] to define refinement relations between ASMs; and ASMs have been used to describe compilers which verify the correctness of the code they generate, etc. For the specification of source languages Montages (see below) has also been used, adopting however attribute grammars to formulate static semantics features. In [335] appropriate ASM models are defined to prove the correctness for the static link technique.

Montages at ETH Zürich. A related research effort has been undertaken at ETH Zürich, triggered by Gurevich's ASM lectures delivered there in the Spring of 1995. It was driven by the Montages project [311, 19], geared to support, by an appropriate combination of graphical and textual elements, the simultaneous specification of the static and dynamic semantics of programming languages, exploiting the syntax-driven modularity which is typical for sequential languages where instructions are executed one after the other and one per step. The method is illustrated by a complete definition of the syntax, static and dynamic semantics of Oberon in [310] and of C in [285]. In [18] a development tool (Gem-Mex) for creating Montages is presented which has been applied to provide an executable semantics for Mosses' Action Notation [16]. A successful application of Montages to the design and specification of a domain-specific language for a Swiss bank is reported in [312].

SDL-2000 standard. Another remarkable industrial exploitation of ASMs comes with the abstract operational ASM definition of the new industrial standard for the design language SDL, widely used for over 20 years to develop distributed systems. In November 2000, the international standardization body ITU-T for telecommunications accepted the ASM model as the official definition of the 2000 standard of the language. The project of modeling the intricate static and dynamic semantics of the distributed real-time features of SDL as executable ASMs, in the context of rich data and hierarchical control structures together with advanced object-oriented and exception-handling features, was started in [225], proposed to the SDL Forum in [220], and led through three years of intensive work by a body of experts [221, 222, 195, 367] to the complete standard definition [292]. It covers the static and the dynamic part of the semantics, currently an SDL-to-AsmL compiler, and further tool support of the ASM model for the standard are in development [194].

ASM analysis of Java and the JVM. The project to apply ASMs to a systematic investigation of Java and its implementation on the Java Virtual Machine was born from a debate, in March 1997 in Dagstuhl, on *How to use Abstract State Machines in Software Engineering* [83]. The modeling experiments during the first two years [138, 137, 139, 140, 141] were geared to establish through this concrete real-life case study the respective merits of functional, axiomatic and operational abstract-state-based specification features. They were followed by two more years of mathematical and experimental analysis, streamlining and structuring the ASM models of Java and of the Java Virtual Machine (see Figs. 2.10, 2.11, 2.12), and adding correctness and completeness proofs for a standard compiler of Java programs to JVM code and for the security critical bytecode verifier component of the JVM [406]. The technique of structuring the ASM models into language layers and machine components [89] is based upon the composition and submachine concepts which were developed in [134] (see the concept of turbo ASMs in Sect. 4.1). It led to a natural refinement of the high-level Java/JVM models to AsmGofer executable models [390] which can be used for code-testing purposes. In a recent evaluation of about 40 Java/JVM research projects worldwide it is stated that "the Jbook (i.e. [406]) ... gives the most comprehensive and consistent formal account of the combination of Java and the JVM, to date"[272, Sect. 6.2]. In [23] the new Java memory models are modeled and analyzed using ASMs.

Architecture projects. Mention has already been made above of the industrial extensions of the ASM models for VHDL [111, 112] to analog VHDL and Verilog [383, 384, 380, 382, 381], to PHDL [349], to SystemC [345] and to SpecC [344], as well as of Schmid's compiler from ASM components to VHDL [392, Chap. 2]. The ASM modeling method for instruction set architectures developed in the APE100 project [102] has been enhanced in [174, 173] to instrument models to collect data for evaluating design alternatives.

In [217, 407] some steps were taken to mechanically verify the pipelining correctness proof using the KIV system and PVS, but unfortunately without covering the complete hierarchy of the four models in [119], so that an omission of a hazard case in the last refinement step remained undetected until Hinrichsen found it during his work on generating pipelined systems from sequential processor specifications [279] (see Sect. 3.3). The design and verification method of [119] has been applied in [286] to an advanced commercial RISC processor with a simpler pipelining scheme. It is reused in [412] to illustrate an approach to automatically transform register transfer descriptions of microprocessors into XASM-executable ASMs [15], thus allowing one to generate a simulator for a processor architecture from its netlist description or from a graphical description of its data-path. In the same spirit, in [413] an ASM model was developed for a VLIW digital signal processor of the Texas

Instruments TMS3200 C6200 family. These papers are part of an architecture and compiler co-generation project, led by Teich at the University of Paderborn, where ASMs and their execution in XASM [15] were systematically applied to the hierarchical modeling of application specific instruction set processors [411].

Further ASM applications. Since 1994 numerous advanced applications of ASMs appeared for protocol verification and in other fields of computer science, namely formal grammars, databases and electronic commerce, software architecture, finite model theory, complexity theory. In [297, 341, 342] async ASMs are used to model various formal and natural language grammars (see the definition of EXECUTE(G) on p. 115, Exercise 3.2.3 and `Backtracking` (⇝ CD)). In [260] the database undo-redo recovery algorithm is modeled at several levels of abstraction, showing the ground model to be correct and proving the correctness of each of the four refinement steps, leading to a model incorporating cache and log management (see Sect. 3.2). An ASM-based definition of the concept of database transactions is given in [368]. In [38] ASMs are used to describe the semantics of a domain-specific language, tailored to program the control for event-driven database applications. In [200] an ASM-based prototype system is described for specifying electronic commerce applications. The contribution of ASMs here is to provide a rigorous and transparent way for describing the state changes involved in electronic commerce negotiations, concerning the traded products, the negotiators, their orders, the laws accepted as basis for the particular negotiation, etc. This paper and its companion paper [1] triggered the recent study of decision problems for restricted classes of relational ASM-transducers [404]. In [300] ASMs are used to specify a name-management model. In [22] ASMs are used to define the semantics of patterns and for correctness proofs for workarounds. In [208, 28, 237, 216, 209] ASMs are exploited for testing issues, as suggested in [86, p. 36] and [87, p. 6] on the basis of the positive experience with running ASM test suites in one of the first industrial ASM projects [121] (see the beginning of Sect. 9.4.2 and the discussion of the ASM Workbench in Sect. 9.4.3). In [410] the proposal of an ASM-based approach to software architecture design is made, allowing one to specify software systems by appropriately connecting components which are characterized abstractly in terms of the services they export or import (see Sect. 3.1.2 and `ComponentModel` (⇝ CD)). In [395, 328] interacting ASMs are defined. In [154] the task and data parallelism of the programming language P3L is analyzed on the basis of an ASM model. In [352] the theme of modeling PVM [107, 108] is taken up again using ASMs to propose a definition for the semantics of grid systems.

The early ASM specifications and verifications of protocols have been continued in [45], where the Kerberos Authentication System is modeled by a hierarchy of proven-t- be-correct stepwise refined ASMs, disclosing the minimum assumptions to guarantee the correctness of the system as well as its security weaknesses. In [46] the Needham–Schroeder protocol is analyzed to illustrate a general ASM framework for a practical analysis of cryptographic protocols. For a recent AsmL-executable model of abstract encryption see [377]. Stroetmann defined a ground model ASM for the constrained shortest path problem and proved it to be correct from a small number of natural axioms [409]. He then refined the ground model in a proven-to-be-correct way down to (but excluding) the level of an efficient proprietary Siemens implementation of the algorithm in C++ (see Sect. 3.2). During his research stay in Pisa in 1997/98, Durand investigated the cache coherence protocol in the Stanford FLASH multiprocessor system. In [189] a high-level ASM specification and a correctness proof are provided which detected some incoherent and incomplete features in the given protocol description. This model has been used in [425] as a case study for a real-life application of model checking to ASMs, using the SMV model checker. In [193] a two-level ASM specification of a distributed termination detection algorithm is given together with an equivalence proof between the two machines. In [47] async ASMs are used to model a new routing layer protocol for mobile ad hoc networks (see Sect. 6.1.8). In [254] two ASMs for a ring-buffer algorithm proposed by Lamport are used to illustrate that the notion of equivalence depends on the level of abstraction at which the algorithm is viewed (see Exercise 6.1.10).

Foundational progress. It is characteristic for ASM workshops and collections of ASM papers [361, 79, 85, 84, 226, 255, 110, 109, 55, 106] to contain both theoretical and practical contributions. In fact in addition to the theoretical papers mentioned already above, there have been important contributions of ASMs also in complexity and finite model theory, although the observation in [191] is still true, namely that the theoretical potential of ASMs has been recognized and explored to a less extent than their practical applications. In [56] ASMs are used as computation models to reflect the practical experience that for real-life computations, constant factors matter. Based upon this model, linear-time hierarchy theorems for random access machines and ASMs are proven. It is shown that there exists a sequential ASM U and a constant c such that, under honest time counting, U simulates every other sequential ASM in lock-step with log factor c.

In [233] finite model theory is extended to metafinite models, covering the mixture of finite and potentially infinite features as they appear typically in practical applications in the states of an ASM. Reference [57] is an investigation into the notion of the reserve set of an (untyped) ASM, exploring the ideas of adding structure within the reserve and the non-determinism of importing new elements (see Sect. 2.4.4).

In [62, 59, 63] a polynomial time version of ASMs is defined and investigated which captures the portion of the complexity class PTIME, where parallel algorithms (with arbitrary finite structures as inputs) are not allowed arbitrary choice. In [64] this choiceless polynomial-time variant of ASMs is explored as a query language for relational databases. In [235, 402] a restriction is defined to capture log-space computable functions on structures. The

ASM choice construct *(choose $x : F(x)$)* motivated also [58], where extensions of first-order logic with the choice construct are studied.

References [401, 403, 354] deal with decision problems for restricted classes of ASMs. In [44] ASMs are used to model knowledge discovery and belief revision. In [234] quantum algorithms are axiomatized in the spirit of [61] and the ASM thesis is proved for them from these axioms.

Recent industrial ASM applications. Since the Fall of 1998 when Gurevich joined Microsoft Research, various applications of ASMs within Microsoft have been reported. The first one [264] is an ASM specification of the Windows Card Runtime Environment with a verification of certain safety properties. During 1999/2000, a command-line debugger of a stack-based run-time environment has been reverse engineered from the given 30k lines of C++ code, using three successive abstraction steps: one to extract the ground model which defines the same core functionality as the debugger, one to reflect the compile time structure of the underlying architecture (of modules of classes containing functions containing code) together with a restricted view of the run-time structure, and eventually a control state ASM focussing on the interaction between the command issuing user and the reacting run-time environment (see Sect. 3.1.2 and `Debugger` (⇝ CD)). In [224] an async real-time constrained ASM has been developed to specify the Universal Plug and Play (UPnP) architecture for peer-to-peer network connectivity of intelligent devices (see the COMMUNICATOR on p. 108). A refined AsmL executable model is derived which can be used to inspect ASM runs for conformance testing.

In [428] an ongoing industrial project in Australia is mentioned where ASMs are used to specify and model check a railway interlocking system. In [150] the development of a high-level abstract model of the core functionality of an entity in a mobile telephony network is reported which served as the basis for a C++ product implementation.

9.4.3 Tool Integration

Already at the time of the very first industrial application of ASMs in the context of the ISO standardization of Prolog, the issue of how to turn the abstract specifications into executable ones for high-level simulations prior to coding was recognized as crucial. Since then it has been resolved in various ways, as we are going to survey in this section. The progress made over the years in using ASMs in industrial projects for building models of software systems at various abstraction levels made it clear, however, that it is not enough to build simulators. They must also be linked to standard verification, testing, refinement, versioning and maintenance methods and tools as used in current development environments.

ASM interpreters. The first interpreter for sequential ASMs was developed in 1989/90, in Kappel's Diplom thesis at the University of Dortmund [301]. At the same time, at Quintus a special interpreter for Prolog

ASMs was built [143]. Both interpreters served to execute the ASM models proposed in 1990 and accepted in 1995 for the ISO standardization [101]. In 1995 an elegant 9-line Prolog interpreter for sequential ASMs appeared [36]. The same year in Oslo [180] a functional ASM interpreter was implemented. As Huggins reports (see [87, footnote 3]), at the University of Michigan an interpreter for sequential ASMs was developed (in C) by Harrison and Huggins from 1991–1994. Huggins also implemented an automated partial evaluator for sequential ASMs [252] which was extended in [178]. From 1994–1996 the Michigan interpreter was upgraded by Mani to async ASMs, but no larger application has been reported.

In 1995, Glässer, Del Castillo and Durdanovic proposed an abstract ASM machine (at the time called EAM for abstract evolving algebra machine). It is defined by stepwise refinement as a platform for implementing ASM tools and led to the design of a virtual machine architecture as a basis for a sequential implementation of the EAM [171]. This was the start for two PhD projects, both located at the University of Paderborn and with the goal of developing a practically useful tool support for ASMs. The first result to come out was Del Castillo's ASM Workbench [169, 170], which has been used in a methodologically interesting way in the FALKO project at Siemens [121] (see `Falko` (⇝ CD)). The high-level ASMs of the to-be-developed railway process software were tested for some scenarios provided by the customers by running the scenarios on the Workbench, where, upon calling an abstract function for some argument, the requested value is taken from its instance in the particular scenario (read from a file which describes the given use case). Reference [190] continues the ideas developed in [171] by building an ASM Virtual Architecture as the basis for a comprehensive ASM tool environment which comes up to industrial efficiency standards.

In [366] the MAX tool supporting the generation of language-specific software from formal specifications is presented which uses functional algebraic attribution techniques for the static semantics and ASMs for the dynamic semantics. The development tool Gem-Mex presented in [18] for creating Montages has been extended by Anlauff to the system XASM [15] for producing efficiently executable and easily reusable ASMs and comes with an XASM-compiler, a run-time system and a graphical debugging and animation interface. It contains a mechanism for structuring ASMs based on components which can be compiled separately and thus be put into a library for later reuse. Technically this is achieved by enhancing a static module concept, where submachines can be used as ASM rules or as external functions, by access/update constructs which provide information on permissions to read/write locations of submachines. An application of this component concept is presented in [20]. (For a different set of modularity concepts, geared to define and refine the I/O-behavior of programs according to their abstract syntax, see [230].) XASM offers an interface to C allowing (a) C-functions to be used as static or monitored ASM-functions, and (b) ASMs to be called from within C-programs. In [309] a denotational semantics is proposed for Xasm. The applications of Xasm for Montages and Teich's architecture and compiler co-generation project are described above.

The development of AsmGofer [390] was driven by the goal of enabling the executability of the models for Java and the JVM defined in [406], although it represents a general interpreter for a large class of structured ASMs (e.g. see its use in the Light Control case study [125] or the use of its variation AsmHugs in [26]). It comes with GUI support, debugging facilities and support for a literate programming technique. The structuring and composition concepts implemented in AsmGofer have been defined in [134] (see Sect. 4.1) and have been used to decompose the Java/JVM models into language layered and functional components [89] . The definition of these concepts was driven by the double concern (a) to distill some forms of standard programming constructs which are really needed in large applications, and (b) to integrate them as natural standard refinements into the ASM world with simultaneous multiple updates of a global state. As a consequence these concepts are special cases of the more general algebraic concepts investigated in [330].

The MOSES tool suite in [293] is tailored for the specification and prototypical implementation of visual notations for discrete-event systems. It comes with a graph editor (from visual notation), a simulator with animator, a debugger and some management tools.

Recently a new specification language AsmL running on the .NET platform has been made accessible at Microsoft Research [201]. In [27, 28, 30] some applications of AsmL within Microsoft are reported. An important new feature of AsmL is the way it exploits the abstraction potential of ASMs to offer component-based and object-oriented structuring principles. The naturalness with which object-oriented features can be described by ASMs had been observed and used already in [232, 53, 348, 347, 366, 310, 295, 294] and has led to Zamulin's systematic investigation of objects and generic types for ASMs [434, 433, 432, 435, 436, 437, 438]. A related effort was made in [99, 98, 296] and in Cavarra's PhD thesis [152] to exploit ASMs for a rigorous support of object-oriented UML techniques and concepts (see Sect. 6.5.1), triggered in the summer of 1999 by an evaluation of the high-level design characteristics of UML and of ASMs in an industrial software development environment [100]. This has inspired also the work in [355], where the ASM Workbench [170] is used to define an executable semantics for UML which covers real-time aspects, and the work in [356, 153].

Linking ASMs to verification tools. The successful link of ASMs to theorem-proving systems like Isabelle, KIV and PVS, and to model checking [426] has been mentioned above. An interesting variation of model checking, applied to the railroad crossing ASM of [253], appears in [34, 35]. See [213] for interfacing ASMs with the MDG tool. Recently [408] KIV has been used also for checking the programs in Sun's Java manual against their ASM specification in [138]. A description of the tableau proof method in terms of ASMs at various levels of refinement leading from the textbook level to an implementation appears in [136].

Numerous logics have been developed to formalize ASMs and to support mechanical reasoning for them. See [238, 363, 365, 394, 414, 35]. In [370] the co-induction proof scheme is justified for ASMs, characterizing them as a class of Di-algebras and proposing "the *Di-algebra* thesis which improves the Turing thesis and which corresponds directly to the evolving algebra thesis: *Real world computable algorithms coincide with algorithms specifiable by completely constraint Di-algebra specifications*" [370, Sect. 3]. The logic in [405] unifies some of these logics. It is based on an atomic predicate for function updates and on a definedness predicate for the termination of the evaluation of structured sequential ASM rules. For a more detailed exposition see Section 8.1.

9.5 Conclusion and Outlook

In this chapter we have described how the *mathematical notion* of Abstract State Machines, discovered by a reflection upon how to sharpen Turing's thesis, through a collaborative effort led to an *engineering method* which supports the trustworthy design and analysis of complex real-life systems. It is characteristic for the ASM method and for its short history that it spans from an accurate theoretical foundation to practical applications in industrial development environments.

In fact on the theoretical side, the concept of ASMs provides the foundation for a logic-based taxonomy of discrete systems into classes of "sequential" and asynchronous systems [90]. The epistemological appropriateness of this classification of models of computation by logic-driven criteria can be justified by the ASM thesis explained in Sect. 7.2 and is supported by the arguments which prove from a few postulates its sequential and its parallel synchronous version. The universality of the ASM model of computation is pragmatically confirmed by

- the *naturalness* with which other *models of computation* can be defined as ASM instances, directly, without any extraneous encoding (but typically not vice versa), as shown in Sect. 7.1,
- the *flexibility* with which ASMs could be adapted to the diversity *of modeling* problems and techniques in different application *domains* and at different design *levels*, as illustrated in Chapters 3–6 of this book,
- the degree by which ASMs support a truly *generic form of programming*, yielding programs – in fact semantically well-defined pseudo-code – which can be instantiated by a variety of refinements of their abstract control and data structures and can be ported between languages and platforms in a semantically transparent way, as discussed in Sect. 8.3.

The retrospective shows that the driving force for the development of the ASM method came from the practical computer *technology* side, that is to say from the use of ASMs in cutting edge, mostly industrial, *applications* and case studies where the interesting problems to be solved showed up.[12] The simple character of the underlying notion of Virtual Machine, stripped down to its essentials – see the definition of ASMs in Sect. 2.2 and its mathematical underpinning in Sect. 2.4 – is the key to the industrial practicability of the method and to its wide-ranged usability. It makes the ASM method rather unique that within a single precise conceptual framework it supports the major activities which occur during the typical software life cycle, namely:

- **requirements capture** by constructing satisfactory *ground models*,
- **detailed design** by *stepwise refinement* of models to executable code,
- **validation** of models by their simulation, which is possible due to the notion of *run* coming with ASMs,
- **verification** of model properties by proof techniques or model checking,
- **documentation** for inspection, reuse and maintenance, by providing through the intermediate models explicit descriptions of the software structure and of the major design decisions.

We have tried in this book to introduce the reader to each of these possible uses of ASMs. In doing this we had to stick to small or medium-size examples and case studies we have chosen from the literature, given the proprietary character of the larger industrial applications mentioned above in this chapter. A real-life public-domain case study which involves all the design and analysis capabilities offered by the ASM method is contained in the Jbook [406], which however will attract only those who are interested in the specific theme of modeling and investigating the principles of the static and dynamic semantics of a modern programming language and of its implementation.

To conclude we want to draw the attention of the reader also to a pragmatic way in which the ASM method brings theory and practice fruitfully together. Namely the method not only turned out to be useful for the daily work of the software practicianer, but it appears to be attractive also for the more theoretically minded persons. This stems from the fact that the method offers a concrete practical way, not obscured by the futile formalization features that Christian Morgenstern warns us against (see p. V), to turn the construction and investigation of real-life sw/hw systems into an intellectually rewarding engineering task – a noble task of applied, experimentally backed-up mathematics, or of scientifically well-founded engineer-

[12] Also the name change to *Abstract State Machines* was triggered by the concern to better render the *practical* relevance of the notion and in fact was pushed by our colleagues in industry [191]. The new name was found by Päppinghaus after two extensive community-wide electronic discussions and replaced the more theoretical sounding name *evolving algebras* which itself already was a replacement of the original *dynamic structures* and later *dynamic algebras*.

ing of computing devices if one prefers to look at it the other way round.[13] It is interesting to observe that though coming from an engineering instead of a logico-mathematical perspective, Abrial's B method, which is similar in various respects and has been rather successful for a variety of safety-relevant large-scale industrial applications, [14] also shares this insight that

> the task of programming (in the small as well as in the large) can be accomplished *by returning to mathematics* [5, p. xi].

Thus we are back at Leonardo da Vinci's observation on p. V which opened this book.

[13] See the following remark by an observer of ASMs, N. Shankar (e-mail of February 12, 2002 to Börger): "The ASM model imposes a logical structure on transition systems that has inspired very capable mathematicians to construct remarkably elegant proofs. It is one of the few formal notations that appeals to both mathematicians and engineers."

[14] See the rich documentation of the MATISSE project (Methodologies and Technologies for Industrial Strength Systems Engineering, IST-1999-11435), in particular the interesting three handbooks for the project mangager, the program manager and the practitioner, at `http://www.matisse.qinetiq.com/`).

References

1. S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proc. 17th ACM Sympos. Principles of Database Systems (PODS 1998)*, pp. 179–187. ACM Press, 1998.

   Relational transducers mapping sequences of input relations to sequences of output relations are proposed for high-level declarative specifications of business models. See [404] for a related class of ASM-transducers. 360, 399, 425

2. U. Abraham. *Models for Concurrency.* Gordon and Breach, 1999. 266, 351

3. U. Abraham. Bakery algorithms. Manuscript of 41 pages from University of Beer Sheva, Nov 19, 2001. 266, 351

4. S. Abramsky. Semantics of interaction. In A. Pitts and P. Dybjer (eds.), *Semantics and Logics of Computation*, pp. 1–31. Cambridge University Press, Cambridge, 1997. 300

5. J.-R. Abrial. *The B-Book.* Cambridge University Press, Cambridge, 1996. 4, 22, 54, 62, 157, 295, 297, 367

6. J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias (ed.), *Proc. 1st Conf. on the B Method*, pp. 169–190. IRIN Institut de recherche en informatique de Nantes, 1996. 4, 296

7. J.-R. Abrial, E. Börger, and H. Langmaack. *Methods for Semantics and Specification*, Vol. 117. Dagstuhl Seminar No. 9523, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, May 1995. 229, 354, 369

8. J.-R. Abrial, E. Börger, and H. Langmaack (eds.). *Formal Methods for Industrial Applications. Specifying and Programming the Steam Boiler Control*, Lecture Notes in Computer Science, Vol. 1165. Springer-Verlag, 1996.

   Contains the problem description for the steam boiler control competition [7] and 22 proposed solutions obtained using the major known formal methods, with text and complete documentation on the accompanying CD. 229, 354

9. J.-R. Abrial, E. Börger, and H. Langmaack. The steam boiler case study: Competition of formal program specification and development methods. In J.-R. Abrial, E. Börger, and H. Langmaack (eds.), *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, Lecture Notes in Computer Science, Vol. 1165, pp. 1–12. Springer-Verlag, 1996.

   Motivation of the steam-boiler control competition [7] and short characterization of the 22 problem solutions appearing in the book. 188, 229, 354

10. J.-R. Abrial and L. Mussat. Specification and design of a transmission protocol by successive refinements using B. In M. Broy and B. Schieder (eds.), *Mathematical Methods in Program Development.* Springer-Verlag, 1996. 241, 296

11. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert (ed.), *B'98: Recent Advances in the Development and Use of the B Method*, Lecture Notes in Computer Science, Vol. 1393, pp. 82–128. Springer-Verlag, 1998. 134, 296

12. M. Allemand, C. Attiogbé, and H. Habrias (eds.). *Comparing Systems Specification Techniques.*, ISBN 2-906082-29-5. IRIN Nantes, March 1998. 88, 89, 157

13. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 287

14. R. Alur and T. Henzinger. Real-time system = discrete system + clock variables. *Software Tools for Technology Transfer*, 1:86–109, 1997. 200

15. M. Anlauff. XASM – an extensible, component-based Abstract State Machines language. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 69–90. Springer-Verlag, 2000.

    The XASM (Extensible ASM) language for producing executable ASMs is presented. A development environment for XASM systems is described. (XASM was formerly known as Aslan.) Also appears in TIK-Report No. 87, pp. 1–21, ETH Zürich, March 2000. See [17]. 342, 359, 360, 363, 370, 375, 414, 427

16. M. Anlauff, S. Chakraborty, P. Kutter, A. Pierantonio, and L. Thiele. Generating an Action Notation environment from Montages descriptions. *Software Tools and Technology Transfer*, 3:431–455, 2001.

    Montages [19] are used to provide executable semantics for Action Notation. A preliminary version was published by M. Anlauff, P. Kutter, A. Pierantonio and L. Thiele under the title "Generating an Action Notation Environment from Montages Descriptions" in Proc. 2nd Int. Workshop on Action Semantics (AS'99), ed. by P. Mosses and D. Watt, University of Aarhus, Department of Computer Science, BRICS Notes Series NS-99-3, March 1999, pp. 1–42. 356, 358

17. M. Anlauff and P. Kutter. Xasm Open Source. Web pages at `http://www.xasm.org/`, 2001.

    The extensible ASM project [15] is Open Source. 173, 342, 370

18. M. Anlauff, P. Kutter, and A. Pierantonio. Montages/Gem-Mex: a meta visual programming generator. TIK-Report 35, ETH Zürich, Switzerland, 1998.

    An introduction to Montages [311] and Gem-Mex, the development tool for creating Montages (using a graphical editor) and generating language interpreters from them, supporting also debugging and animation features. A description of their use and some small examples can be found in *Formal Aspects of and Development Environments for Montages* by the same authors, published in M. Sellink (ed.): 2nd Int. Workshop on the Theory and Practice of Algebraic Specifications, Springer Workshops in Computing 1997. Another description is in *Tool Support for Language Design and Prototyping with Montages* published by the same authors in Proc. of Compiler Construction (CC'99), Springer Lecture Notes in Computer Science, Vol. 1575, pp. 296–300, Springer-Verlag 1999. Another illustration is in [20]. 358, 363, 371, 427

19. M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In D. Bjoerner and M. Broy (eds.), *Proc. Perspectives of System*

Informatics (PSI'99), Lecture Notes in Computer Science, Vol. 1755, pp. 40–53. Springer-Verlag, 1999.

A refined definition of Montages, based on the notion of finite state machines, triggered by the use of Montages for defining the static semantics of Java in [421] which showed some shortcomings of the original formulation in [311]. 358, 370, 371, 414, 428

20. M. Anlauff, P. Kutter, A. Pierantonio, and A. Sünbül. Using domain-specific languages for the realization of component composition. In T. Maibaum (ed.), *Fundamental Approaches to Software Engineering (FASE 2000)*, Lecture Notes in Computer Science, Vol. 1783, pp. 112–126, 2000.

    An illustration of how to apply Montages [311, 19] and of its tool environment Gem-Mex [18] for the implementation of component interaction. 363, 370

21. L. Araujo. Correctness proof of a distributed implementation of Prolog by means of Abstract State Machines. *J. Universal Computer Science*, 3(5):416–422, 1997.

    Building upon [132], a specification and a proof of correctness for the Prolog Distributed Processor (PDP), a WAM extension for parallel execution of Prolog on distributed memory, are provided. A preliminary version appeared in 1996 under the title *Correctness Proof of a Parallel Implementation of Prolog by Means of Evolving Algebras* as Technical Report DIA 21-96 of Dpto. Informática y Automática, Universidad Complutense de Madrid. 347, 380, 389

22. U. Assmann, A. Heberle, W. Löwe, A. Ludwig, and R. Neumann. Language concepts and design patterns. Manuscript, 1999.

    ASMs are used to define the semantics of patterns and for correctness proofs for workarounds. 360

23. V. Awhad and C. Wallace. A unified formal specification and analysis of the new Java memory models. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 166–185. Springer-Verlag, 2003.

    See [421]. 359, 428

24. R. J. R. Back. On correct refinement of programs. *J. Computer and System Sciences*, 23(1):49–68, 1979. 22, 134

25. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. 22, 134

26. M. Barnett, E. Börger, Y. Gurevich, W. Schulte, and M. Veanes. Using Abstract State Machines at Microsoft: A case study. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 367–380. Springer-Verlag, 2000.

    A description of a reverse-engineering case study, modeling a command-line debugger of a stack-based run-time environment at three levels of abstraction. For a powerpoint slide show see `Debugger` (⇝ CD). 25, 103, 113, 364

27. M. Barnett, C. Campbell, W. Schulte, and M. Veanes. Specification, simulation and testing of COM components using Abstract State Machines. In R. Moreno-Díaz and A. Quesada-Arencibia (eds.), *Formal Methods and Tools for Computer Science (Local Proceedings of Eurocast 2001)*, pp. 266–270, Canary Islands, Spain, February 2001. Universidad de Las Palmas de Gran Canaria.

A description of the use of AsmL to specify, simulate, and test the interfaces of Microsoft COM components. 341, 364

28. M. Barnett, L. Nachmanson, and W. Schulte. Conformance checking of components against their non-deterministic specifications. Technical Report MSR-TR-2001-56, Microsoft Research, Redmond, Washington, June 2001.

    A method for testing a Microsoft COM (Component Object Model) component against a (possibly non-deterministic) ASM specification is presented. See [31]. 86, 341, 360, 364, 372

29. M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In G. T. Leavens, M. Sitaraman, and D. Giannakopoulou (eds.), *Workshop on Specification and Verification of Component-Based Systems*, pp. 7–13. Technical Report TR 01-09a, Iowa State University, 2001.

    A predecessor of [31]. 193, 372

30. M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, 2002.

    See [31]. 341, 364, 372

31. M. Barnett and W. Schulte. Contracts, components and their runtime verification on the .NET platform. *J. Systems and Software*, Special Issue on Component-Based Software Engineering, 2002, to appear.

    Continuing [28, 29, 30] AsmL is proposed to implement behavioral interface specifications, including component interaction, on the .NET platform. 18, 106, 341, 372

32. A. Bartoloni et al. A hardware implementation of the APE100 architecture. *Int. J. Mod. Phys.*, C(4):969, 1993.

    The architecture which has been chosen by Börger for Del Castillo's Tesi di Laurea to try out ASMs for modeling real-world architectures, in the context of a reengineering project for this machine which had been launched by a group of physicists in Pisa and Rome; see [97],[102]. 349, 372

33. A. Bartoloni et al. The software of the APE100 architecture. *Int. J. Mod. Phys.*, C(4):955, 1993.

    See comment to [32]. 349

34. D. Beauquier and A. Slissenko. The railroad crossing problem: Towards semantics of timed algorithms and their model-checking in high-level languages. In M. Bidoit and M. Dauchet (eds.), *TAPSOFT'97: Theory and Practice of Software Development, 7th Int. Joint Conf. CAAP/FASE*, Lecture Notes in Computer Science, Vol. 1214, pp. 201–212. Springer-Verlag, 1997.

    A semantics of ASMs with continuous time using infinitesimals is defined, their runs are decribed in some first order logic. The framework is used to discuss the verification of the railroad crossing problem, based upon its ASM specification in [253]. An early version appeared in 1996 as Technical Report 96-10 of Dept. of Informatics, Université Paris 12. For a continuation see [35]. 203, 356, 364, 372, 373, 407

35. D. Beauquier and A. Slissenko. A first-order logic for specification of timed algorithms: Basic properties and a decidable class. *Annals of Pure and Applied Logic*, 113(1–3):13–52, 2001.

    A continuation of [34]. The authors define (a) a class of algorithms (a modified version of ASMs) with explicit continuous time, and (b) a First-Order Timed Logic which suffices to write requirements specifications close to natural language, enhancing the logic in [34]. The timed logic description of the semantics of that class of ASMs can be viewed as a basic set of inductive invariants for proving the properties of timed ASMs. The authors consider a decidable class of verification problems and outline a compact verification proof of the Generalized Railroad Crossing Problem [253]. A first version of this work appeared under the title *On Semantics of Algorithms with Continuous Time* in October 1997 as TR 97-15 of Dept. of Informatics, Université Paris 12. A survey of that TR and of [34] appears under the title *Verification of Timed Algorithms: Gurevich Abstract State Machines versus First-Order Timed Logic* in Y. Gurevich and P. Kutter and M. Odersky and L. Thiele (eds.): Abstract State Machines – ASM 2000, *Int. Workshop on Abstract State Machines*, Monte Verita, Switzerland, Local Proceedings, March 2000, ETH Zürich, TIK-Report No. 87, pp. 22–39. Continuation in TR-00-23 of June 2000, Université Paris-12, by the same authors and with the title *A First Order Logic for Specification of Timed Algorithms: Basic Properties and a Deciadable Class*. For a set of more efficient inductive invariants for ASMs and a short formal proof of the Generalized Railroad Crossing Problem along the same lines, see "A Predicate Logic Framework for Mechanical Verification of Real-Time Gurevich Abstract State Machines: A Case Study with PVS", by the same authors together with T. Colard, Technical Report TR–00–25, Université Paris 12, Department of Informatics, 2000, available at `http://www.univ-paris12.fr/lacl/`. 203, 356, 364, 365, 372, 407

36. B. Beckert and J. Posegga. leanEA: A lean evolving algebra compiler. In H. K. Büning (ed.), *Proc. the Annual Conf. of the European Association for Computer Science Logic (CSL'95)*, Lecture Notes in Computer Science, Vol. 1092, pp. 64–85. Springer-Verlag, 1996.

    A 9-line Prolog interpreter for sequential ASMs, including discussion of extensions for layered ASMs. A preliminary version appeared in April 1995 under the title *leanEA: A poor man's evolving algebra compiler* as internal report 25/95 of Fakultät für Informatik, Universität Karlsruhe. 340, 363

37. P. Behm, P. Benoit, A. Faivre, and J. M. Meynadier. Meteor: A successful application of B in a large project. In *FM'99*, Lecture Notes in Computer Science, Vol. 1708, pp. 348–387. Springer-Verlag, 1999. 27, 313

38. H. Behrends. *Beschreibung ereignisgesteuerter Aktivitäten in datenbankgestützten Informationssystemen*. PhD thesis, University of Oldenburg, Germany, 1995.

    Uses ASMs to define the semantics of a language which is tailored to program the control of event-driven database applications. The specification proceeds by stepwise refinement of *event processing*, *rule selection* among the event triggered rules, and *action execution* following the priorities of the selected rules. Thesis supervised by Appelrath. Issued as TR 3/95 of CS Departement of the University of Oldenburg, October 1995, 278 pages. 360

39. C. Beierle. Formal design of an abstract machine for constraint logic programming. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 377–382, Elsevier, Amsterdam, 1994.

    Develops a general implementation scheme for CLP(X) over an unspecified constraint domain X, namely by designing a generic extension WAM(X) of the Warren Abstract Machine and a corresponding generic compilation scheme of CLP(X) programs to WAM(X) code. The scheme is based on the specification and correctness proof for compilation of Prolog programs in [132] and on joint work with Börger; see [40, 42, 41]. 298, 347, 419

40. C. Beierle and E. Börger. Correctness proof for the WAM with types. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter (eds.), *Computer Science Logic*, Lecture Notes in Computer Science, Vol. 626, pp. 15–34. Springer-Verlag, 1992.

    The specification and correctness proof for compiling Prolog to WAM [132] is extended in modular fashion to the type-constraint logic programming language Protos-L which extends Prolog with polymorphic order-sorted (dynamic) types. In this paper, the notion of types and dynamic type constraints are kept abstract (as constraints) in order to permit applications to different constraint formalisms like Prolog III or CLP(R). The theorem is proved that for every appropriate type-constraint logic programming system L, every compiler to the WAM extension with an abstract notion of types which satisfies the specified conditions, is correct. Reference [41] extends the specification and the correctness proof to the full Protos Abstract Machine by refining the abstract type constraints to the polymorphic order-sorted types of PROTOS-L. Also issued as IBM Germany Science Center Research Report IWBS 205, 1991. Revised and final version published in [42]. 298, 346, 373, 374, 390

41. C. Beierle and E. Börger. Refinement of a typed WAM extension by polymorphic order-sorted types. *Formal Aspects of Computing*, 8(5):539–564, 1996.

    Continuation of [42] which is extended to the full Protos Abstract Machine by refining the abstract type constraints to the polymorphic order-sorted types of PROTOS-L. Preliminary version published under the title *A WAM Extension for Type-Constraint Logic Programming: Specification and Correctness Proof* as Research Report IWBS 200, IBM Germany Science Center, Heidelberg, December 1991. 23, 156, 298, 346, 347, 373, 374, 378, 381, 389

42. C. Beierle and E. Börger. Specification and correctness proof of a WAM extension with abstract type constraints. *Formal Aspects of Computing*, 8(4):428–462, 1996.

    Revised version of [40]. 23, 156, 298, 346, 347, 373, 374, 378, 381, 389

43. C. Beierle, E. Börger, I. Durdanović, U. Glässer, and E. Riccobene. Refining abstract machine specifications of the steam boiler control to well documented executable code. In J.-R. Abrial, E. Börger, and H. Langmaack (eds.), *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, Lecture Notes in Computer Science, No. 1165, pp. 62–78. Springer-Verlag, 1996.

    The steam-boiler control specification problem is used to illustrate how ASMs applied to the specification and the verification of complex systems can be exploited for a reliable and well-documented development of executable, but formally inspectable and systematically modifiable code. A hierarchy of stepwise refined abstract machine models is developed, the ground version of which can be checked for whether it faithfully reflects the informally given problem. The sequence of machine models yields various abstract views of the system, making the various design decisions transparent, and leads to a C++ program. This program has been demonstrated during the Dagstuhl Seminar on Methods for Semantics and Specification, in June 1995, to control the FZI Steam-Boiler simulator satisfactorily. The proofs of properties of the ASM models provide insight into the structure of the system which supports easily maintainable extensions and modifications of both the abstract specification and the implementation. For a continuation of this use of ASMs for reliable software development see [120, 125]. 188, 229, 340, 354

44. C. Beierle and G. Kern-Isberner. Modeling knowledge discovery and belief revision by Abstract State Machines. In E. Börger, A. Gargantini, and

E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 186–203. Springer-Verlag, 2003. 362

45. G. Bella and E. Riccobene. Formal analysis of the Kerberos authentication system. *J. Universal Computer Science*, 3(12):1337–1381, 1997.

    A formal model of the whole system is reached through stepwise refinements of ASMs, and is used as a basis both to discover the minimum assumptions to guarantee the correctness of the system and to analyse its security weaknesses. Each refined model comes together with a correctness refinement theorem. 360

46. G. Bella and E. Riccobene. A realistic environment for crypto-protocol analyses by ASMs. In U. Glässer and P. Schmitt (eds.), *Proc. 5th Int. Workshop on Abstract State Machines*, pp. 127–138. Magdeburg University, 1998.

    ASMs are used to give a model of a general, realistic environment in which cryptographic protocols can be faithfully analyzed. The Needham–Schroeder protocol is investigated as an example. 360, 403

47. A. Benczur, U. Glässer, and T. Lukovszki. Formal description of a distributed location service for mobile ad hoc networks. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 204–217. Springer-Verlag, 2003.

    Models a new routing layer protocol of mobile ad hoc networks by an async ASM, coming with sublayers for the location service and for the position based routing between known locations. 210, 223, 225, 361

48. D. M. Berry. The importance of ignorance in requirements engineering. *J. Systems and Software*, 28(2):179–184, 1995. 18, 157

49. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992. 311

50. J. Billington. Protocol specification using p-graphs, a technique based on coloured Petri nets. In W. Reisig and G. Rozenberg (eds.), *Lectures on Petri Nets II: Applications*, Lecture Notes in Computer Science, Vol. 1492. Springer-Verlag, 1998. 52

51. A. Binemann-Zdanowicz. Towards information systems modeling on the basis of ASM semantics. Technical Report 12/01, Brandenburg University of Technology at Cottbus, Germany, December 2001.

    Proposes an ASM based approach to modeling information services, using a special-purpose language *SiteLang*. Specifications which are written in *SiteLang* are compiled to input for XASM [15]. 342

52. S. Bistarelli and E. Riccobene. An operational model for the SCLP language. ILPS Workshop on Tools and Environments for CLP held in Port Jefferson USA, 1997.

    Refinement and parallelization of the ASM model for Prolog to a semi-ring based constraint system, replacing the Call and Select rules of [71] by a Reduction rule which activates a child process simultaneously for each alternative of the current process. For the proceedings see `http://www.clip.dia.fi.upm.es/Tools_Environ/proceedings.html`. 348

53. B. Blakley. *A Smalltalk Evolving Algebra and its Uses.* PhD thesis, University of Michigan, Ann Arbor, Michigan, 1992.

A reduced version of Smalltalk is formalized by sequential ASMs. A Hoare-style proof system is defined for reasoning about storage allocation and deallocation in ASMs. Missing constructs concern processes, inheritance, memory allocation and deallocation. Thesis supervised by Gurevich. 347, 364

54. A. Blass. Abstract State Machines and pure mathematics. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 9–21. Springer-Verlag, 2000.

    A discussion of connections, similarities, and differences between concepts and issues arising in the study of ASMs and those of set theory and logic. 28, 348

55. A. Blass, E. Börger, and Y. Gurevich. *Abstract State Machines*. Dagstuhl Seminar No. 02101, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, March 2002. 361

56. A. Blass and Y. Gurevich. The linear time hierarchy theorems for Abstract State Machines. *J. Universal Computer Science*, 3(4):247–278, 1997.

    Contrary to polynomial time, linear time depends on the computation model. In 1992, N. Jones designed specific computation models where the linear-speed-up theorem fails and linear-time computable functions form a proper hierarchy. In this paper linear-time hierarchy theorems for random access machines and ASMs are proven. In particular it is shown that there exists a lock-step universal sequential ASM U, i.e. with a constant c such that, under honest time counting, U simulates every other sequential ASM in lock-step with log factor c. The result has been anounced under the title *Evolving Algebras and Linear Time Hierarchy* in B. Pehrson and I. Simon (eds.), IFIP 13th World Computer Congress, Vol. I: Technology/Foundations, Elsevier, Amsterdam, 1994, 383–390. 310, 361, 380, 407, 419

57. A. Blass and Y. Gurevich. Background, reserve, and Gandy machines. In P. Clote and H. Schwichtenberg (eds.), *Computer Science Logic (Proceedings of CSL 2000)*, Lecture Notes in Computer Science, Vol. 1862, pp. 1–17. Springer-Verlag, 2000.

    An investigation into the notion of the reserve set of an ASM, exploring the ideas of adding structure within the reserve (such as the hereditarily finite sets of [62]) and the non-determinism of importing new elements. 36, 361

58. A. Blass and Y. Gurevich. The logic of choice. *J. Symbolic Logic*, 65(3):1264–1310, 2000.

    Motivated by the choice construct of ASMs, extensions of first-order logic with the choice construct *(choose* $x : F(x)$*)* are studied. Some results about Hilbert's ϵ operator are proven. The main part of the paper concerns the case where all choices are independent. Previously appeared as Technical Report CSE-TR-369-98, EECS Dept., University of Michigan, 1998. 362

59. A. Blass and Y. Gurevich. New zero-one law and strong extension axioms. *Bull. EATCS*, 72:103–122, 2000.

    A formulation of Shelah's proof of a zero-one law for the choiceless polynomial time variant of ASMs [62]. 361

60. A. Blass and Y. Gurevich. Algorithms vs. machines. *Bull. EATCS*, 74:96–118, 2002.

    In reaction to [340] the mergesort algorithm is described in terms of distributed ASMs. See the description of recursive algorithms by turbo ASMs in [95]. 171

61. A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Trans. Computational Logic*, 2002, to appear.

 The axiomatization of sequential algorithms as the basis for a derivation of the sequential ASM thesis from the proposed axioms [249] is extended to parallel synchronous algorithms. A preliminary version appeared in November 2001 as Microsoft Research Technical Report TR-2001-117. See the adaptation to quantum algorithms in [234]. 35, 284, 310, 311, 344, 362, 405, 406

62. A. Blass, Y. Gurevich, and S. Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100:141–187, 1999.

 The question "Is there a computation model whose machines do not distinguish between isomorphic structures and compute exactly polynomial time properties?" became a central question of finite model theory. The negative answer was conjectured in [244]. A related question is what portion of PTIME can be naturally captured by a computation model (when inputs are arbitrary finite structures). A PTIME version of ASMs is used to capture the portion of PTIME where algorithms are not allowed arbitrary choice but parallelism is allowed and, in some cases, implements choice. Earlier versions appeared as Technical Report CSE-TR-338-97, EECS Department, University of Michigan, 1997, and Technical Report MSR-TR-99-08, Microsoft Research, February 1999. See [235]. 361, 376, 377, 405, 425

63. A. Blass, Y. Gurevich, and S. Shelah. On polynomial time computation over unordered structures. *J. Symbolic Logic*, 67(3):1093–1125, 2001.

 A consideration of several algorithmic problems near the border of the known, logically defined complexity classes contained in polynomial time, including the choiceless polynomial time defined in [62]. 361

64. A. Blass, Y. Gurevich, and J. Van den Bussche. Abstract State Machines and computationally complete query languages. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 22–33. Springer-Verlag, 2000.

 The use of the choiceless polynomial-time variant of ASMs [62] as a query language for relational databases is explored. Also appears in TIK-Report No. 87, pp. 40–65, ETH Zürich, March 2000, and as Microsoft Research Technical Report MSR-TR-99-95. Republished in *Information and Computation* 174(1):20–36, 2002. 361

65. R. Bloomfield, D. Craigen, F. Koob, M. Ullman, and S. Wittmann. Formal methods diffusion: Past lessons and future prospects. In *Proc. SAFECOMP 2000*, Lecture Notes in Computer Science, Vol. 1943, pp. 211–226. Springer-Verlag, 2000.

 The full technical report is available at `http://www.bsi.bund.de/aufgaben/projekte/fmethode/sonstige/fms_v1.0.pdf`. 299, 313

66. C. Böhm and G. Jacopini. Flow diagrams, Turing Machines, and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966. 163

67. T. Bolognesi and E. Börger. Abstract State Processes. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 22–32. Springer-Verlag, 2003.

 Process-algebraic structuring techniques and concurrency patterns are combined with the state-based abstraction mechanism and synchronous parallelism of ASMs. Extended ASM programs are defined which are structured and evolve like process-algebraic behaviour expressions operating on evolving states. This supersedes the preceding definition given in the extended abstract presented to the Dagstuhl Seminar on ASMs in March 2002. 185

68. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. and ISDN Syst.*, 14(1):25–59, 1987. 52, 135

69. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999. 4

70. E. Börger. *Computability, Complexity, Logic (English translation of "Berechenbarkeit, Komplexität, Logik")*, Studies in Logic and the Foundations of Mathematics, Vol. 128. North-Holland, 1989. 28, 54, 285, 292, 300

71. E. Börger. A logical operational semantics for full Prolog. Part I: Selection core and control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld (eds.), *CSL'89. 3rd Workshop on Computer Science Logic*, Lecture Notes in Computer Science, Vol. 440, pp. 36–64. Springer-Verlag, 1990.

    See Comments to [74]. 9, 38, 86, 156, 340, 345, 347, 348, 375, 378, 382, 383, 388, 389, 390, 413

72. E. Börger. A logical operational semantics of full Prolog. Part II: Built-in predicates for database manipulation. In B. Rovan (ed.), *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, Vol. 452, pp. 1–14. Springer-Verlag, 1990.

    See Comments to [74]. 9, 86, 156, 340, 345, 347, 378, 382, 383, 388, 389, 390, 413

73. E. Börger. Dynamische Algebren und Semantik von Prolog. In E. Börger (ed.), *Berechenbarkeit, Komplexität, Logik*, pp. 476–499. Vieweg, 3rd edn., 1992.

    The first textbook definition of ASMs, elaborating notes of a series of lectures on the Semantics of Programming Languages delivered to a summer school organized by Börger in Cortona in 1989. The definition is illustrated with machines operating on standard data structures and by the tree based version [75] of the core Prolog ASM in [71]. 348, 378, 381, 388, 389, 406

74. E. Börger. A logical operational semantics for full Prolog. Part III: Built-in predicates for files, terms, arithmetic and input-output. In Y. N. Moschovakis (ed.), *Logic From Computer Science*, Berkeley Mathematical Sciences Research Institute Publications, Vol. 21, pp. 17–50. Springer-Verlag, 1992.

    This paper, along with [71] and [72] are the original 3 papers which provide a complete ASM formalization of Prolog with all features discussed in the international Prolog standardization working group (WG17 of ISO/IEC JTCI SC22); see [101]. The specification proposed as the ground model for Prolog is developed by stepwise refinement, describing orthogonal language features by modular rule sets. An improved (tree instead of stack based) version is found in [75, 73, 131]. These three papers were also published in 1990 as IBM Germany Science Center Research Reports 111, 115 and 117 respectively. The refinement technique, used in combination with corresponding methods of proof, is further developed in [132, 133, 114, 42, 41, 104, 119, 136, 120, 138, 406, 387]. For a systematic exposition and survey see [94]. Together with the technique of building ground models, ASM refinements became a constituent of the ASM method. 340, 345, 378, 379, 382, 383, 389, 390

75. E. Börger. A natural formalization of full Prolog. *Newsletter of the Association for Logic Programming*, 5(1):8–9, 1992.

The paper explains the abstract tree structure and the four ASM transition rules which govern the user-defined core of Prolog. See [74]. 346, 348, 378, 381, 387, 388, 389, 391, 418

76. E. Börger. Logic programming: The Evolving Algebra approach. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 391–395, Elsevier, Amsterdam, 1994.

    Surveys the work which has been done from 1988–1994 on specifications of logic programming systems by ASMs. 9, 86, 156, 343, 346, 419

77. E. Börger. Review of E. W. Dijkstra and C. S. Scholten *Predicate Calculus and Program Semantics* (Springer-Verlag 1989). *Science of Computer Programming*, 23:1–11, 1994.

    Discusses the weakness of identifying the notion of proof with "formal proofs" and furthermore with "formal proofs in a strict format". Critically evaluates the authors' restricted view on the role of formal methods for program design and verification concerns. An abridged version appeared in *J. Symbolic Logic* 59:673–678, 1994. 356

78. E. Börger. Annotated bibliography on Evolving Algebras. In E. Börger (ed.), *Specification and Validation Methods*, pp. 37–51. Oxford University Press, 1995.

    An annotated bibliography of papers (as of 1994) which deal with or use ASMs. For an updated version see [116]. 379, 381, 385

79. E. Börger. *Specification and Validation Methods.* Oxford University Press, 1995.

    The ASM related papers appearing in this volume are [248, 78, 133, 420, 114, 283, 256]. 348, 349, 351, 361

80. E. Börger. Why use Evolving Algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiederman (eds.), *Proc. SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, Lecture Notes in Computer Science, Vol. 1012, pp. 236–271. Springer-Verlag, 1995.

    A presentation of the salient features of ASMs, as part of a discussion and survey of the use of ASMs in the design and analysis of hardware and software systems. The leading example is detailed and improved in [119]. 86, 343, 345, 346, 352, 356

81. E. Börger. Evolving Algebras and Parnas tables. In H. Ehrig, F. von Henke, J. Meseguer, and M. Wirsing (eds.), *Specification and Semantics.* Dagstuhl Seminar No. 9626, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, July 1996.

    Extended abstract showing that Parnas' approach to the use of function tables for precise program documentation can be generalized in a natural way by using ASMs for well-documented program development. 381

82. E. Börger. Remarks on the history and some perspectives of Abstract State Machines in software engineering. In W. Aspray, R. Keil-Slawik, and D. L. Parnas (eds.), *The History of Software Engineering*, pp. 12–17. Dagstuhl Seminar No. 9635, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, August 1996.

    Survey of the development of the ASM method as of 1996. For an update in 2000 see [87]. 380, 381

83. E. Börger. How to use Abstract State Machines in software engineering. In S. Jähnichen, J. Loeckx, D. Smith, and M. Wirsing (eds.), *Logic for Systems Engineering*, Vol. 171, pp. 5–7. Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, 3–7 March 1997.

The talk which triggered the first two years of work on the Java/JVM ASM project, as a comparative field test of purely declarative (functional or axiomatic) methods and their enhancement within an integrated abstract state-based operational (ASM) framework [138, 137, 139, 140, 141]. See preface to [406]. 359

84. E. Börger. JUCS Special ASM Issue. Part II. In E. Börger (ed.), *J. Universal Computer Science*, Vol. 3(5). Springer-Verlag, 1997.

Introduction to the second part of the special ASM issue of the J. Universal Computer Science. This May issue contains [311, 310, 439, 21, 120, 332, 424]. 361

85. E. Börger. Ten Years of Gurevich's Abstract State Machines. In E. Börger (ed.), *J. Universal Computer Science*, Vol. 3(4). Springer-Verlag, 1997.

Introduction to the first special ASM issue of the *J. Universal Computer Science*. This April issue contains [261, 56, 177, 409, 260, 313, 388]. 361, 381

86. E. Börger. High-level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann (eds.), *Current Trends in Applied Formal Methods (FM-Trends 98)*, Lecture Notes in Computer Science, Vol. 1641, pp. 1–43. Springer-Verlag, 1999.

A general introduction to and survey of the ASM method, including the definition of the ASM concept and an illustration of the main characteristics of the method, a comparison with other well-known system design and analysis approaches, and experimental evidence for the ASM thesis. 9, 86, 287, 311, 343, 345, 352, 353, 360, 381, 426

87. E. Börger. Abstract State Machines at the cusp of the millenium. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 1–8. Springer-Verlag, 2000.

A brief survey of the history of the development of the ASM method and the current challenges in the field (continuation of [82]). 18, 86, 356, 360, 363, 379, 381

88. E. Börger. *Hardware Design and Validation Methods*. Springer-Verlag, 2000.

The ASM related paper appearing in this volume is [140]. 349

89. E. Börger. Design for reuse via structuring techniques for ASMs. In R. Moreno-Díaz, B. Buchberger, and J.-L. Freire (eds.), *Computer Aided Systems Theory–EUROCAST 2001*, Lecture Notes in Computer Science, Vol. 2178, pp. 20–35. Springer-Verlag, 2001.

The composition and structuring concepts for sequential ASMs defined in [134] are used to illustrate a modular high-level definition of the architecture of the Java Virtual Machine, unfolding its language layering and its functional components for loader, verifier, and interpreter. Extracted from [406]. 359, 364, 390

90. E. Börger. Discrete systems modeling. In R. A. Meyers (ed.), *Encyclopedia of Physical Science and Technology*, Vol. 4, pp. 535–546. Academic Press, San Diego, 2001.

A classification of discrete systems and of methods for their mathematical verification and experimental validation, using ASMs as the framework for the taxonomy. 365

91. E. Börger. Computation and specification models. A comparative study. In P. D. Mosses (ed.), *Proc. 4th Int. Workshop on Action Semantics*, BRICS Series, Vol. NS-02-8, pp. 107–130. Department of Computer Science at University of Aarhus, December 2002.

    Continuing the work in [81, 86, 142] representative computation models in the literature are characterized as naturally arising special classes of ASMs. Classical automata (Moore-Mealy, Co-Design FSM, Timed FSM, PushDown, Turing, Scott, Eilenberg, Minsky, Wegner, Alternating TM), grammar formalisms, tree computation machines, structured and functional programs are covered, as well as system design models like UNITY, COLD, B, SCR (Parnas tables), Petri nets, Neural Nets. Also logic based, functional-denotational and process-algebraic systems including CSP, Z, VDM are discussed. A draft has been presented under the title *Definitional Suggestions for Computation Theory* to the Dagstuhl Seminar "Theory and Application of Abstract State Machines", March 3–8, 2002. The final version appears in [93]. 311, 381

92. E. Börger. The origins and the development of the ASM method for high-level system design and analysis. *J. Universal Computer Science*, 8(1):2–74, 2002.

    A historical and bibliographical survey of the ASM related literature from 1984–2002. Elaboration of [78, 82, 85, 116, 87]. 343, 344, 406

93. E. Börger. Abstract State Machines: A unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic*, 2003, to appear.

    An elaboration of [91]. 311, 381

94. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 14, 2003, to appear.

    An exposition of the ASM refinement method, including a survey of its development in [75, 73, 131, 132, 133, 114, 42, 41, 104, 119, 136, 120, 138, 406, 387]. 378

95. E. Börger and T. Bolognesi. Remarks on turbo ASMs for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003 – Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 218–228. Springer-Verlag, 2003.

    The notation for value-returning turbo ASMs defined in [134] is extended to simultaneous calls of multiple submachines and used to answer the question raised in [340] of how to naturally model widely used forms of recursion by abstract machines. The notation allows one to seamlessly integrate functional description and programming techniques into ASMs. 185, 376

96. E. Börger, H. Busch, J. Cuellar, P. Päppinghaus, E. Tiden, and I. Wildgruber. Konzept einer hierarchischen Erweiterung von EURIS. Siemens ZFE T SE 1 Internal Report BBCPTW91-1 (pp. 1–43), Summer 1996.

    ASMs are proposed for extending the EURIS method for the tool-supported design of railway-related software. 356

97. E. Börger, G. D. Castillo, P. Glavan, and D. Rosenzweig. Towards a mathematical specification of the APE100 architecture: the APESE model. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 396–401, Elsevier, Amsterdam, 1994.

Defines an ASM model of the high-level programmer's view of the APE100 parallel architecture. This model is refined in [102] to an ASM processor model. 103, 349, 372, 419

98. E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus (ed.), *Algebraic Methodology and Software Technology, 8th Int. Conf., AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000 Proceedings*, Lecture Notes in Computer Science, Vol. 1816, pp. 293–308. Springer-Verlag, 2000.

    ASMs are used to disambiguate the semantics for activity diagrams in UML, defining a special subclass of ASMs appropriate to modeling such diagrams. As illustration a one-page UML activity diagram definition is given for the ASM model of Occam which appeared in [105]. For a continuation of this work to make semantic features of UML precise see [99]. 47, 88, 164, 276, 280, 282, 364, 382, 393

99. E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 223–241. Springer-Verlag, 2000.

    The work in [98] providing a rigorous semantics for basic UML features is extended by an ASM definition of the dynamic semantics of UML state machines. These machines integrate statecharts with the UML object model. A rational reconstruction is given for the event-driven run to completion scheme of UML (including the sequential entry/exit actions, the concurrent internal activities, and the event-deferring mechanism) and for the concepts of action and durative action. The models make the *semantic variation points* of UML explicit, as well as various ambiguities and omissions in the official UML documents. For an executable version of these models see [152], where various conflict situations are also described which may arise through the concurrent behavior of active objects. This argument has been reconsidered by the same authors in *Solving Conflicts in UML State Machines Concurrent States* presented to the Workshop on Concurrency Issues in UML – UML 2001, Toronto/Canada, and in *A precise semantics of UML State Machines: Making Semantic Variation Points and Ambiguities Explicit* in Proc. Int. Workshop on Semantic Foundations of Engineering Design Languages (SFEDL'02) in conjunction with the 5th European Joint Conferences on Theory and Practice of Software (ETAPS'02). See the continuation in [153]. 47, 88, 164, 169, 276, 280, 282, 364, 382, 393, 412

100. E. Börger, M. Cesaroni, M. Falqui, and T. L. Murgi. Caso di Studio: Mail From Form System. Internal Report FST-2-1-RE-02, Fabbrica Servizi Telematici FST (Gruppo Atlantis), Uta (Cagliari), 1999.

    Feasability study of using ASMs for software analysis and design in an industrial object-oriented software development environment. Two company internal case studies are developed. In view of a possible integration, the use of the ASM method for building ground models and refining them to code is compared to the use of UML based tools, in particular Rational Rose. 364

101. E. Börger and K. Dässler. Prolog: DIN papers for discussion. ISO/IEC JTCI SC22 WG17 Prolog Standardization Document 58, National Physical Laboratory, Middlesex, England, 1990.

    A version of [71, 72, 74] proposed to the International Prolog Standardization Committee as a complete formal semantics of Prolog. A streamlined version is in [131], representing the definition of the dynamic core of Prolog which has been accepted as the ISO standard [291]. 87, 340, 345, 363, 378, 389, 391

102. E. Börger and G. Del Castillo. A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study). In B. Werner (ed.), *Proc. 1st IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS'95)*, pp. 145–148, November 1995.

Presents an ASM based technique by which a behavioural description of a processor is obtained as the result of the composition of its (formally specified) basic architectural components. The technique is illustrated by the example of a subset of the zCPU processor (used as the control unit of the APE100 parallel architecture). A more complete version, containing the full formal description of the zCPU components, of their composition and of the whole zCPU processor, appeared in Y. Gurevich and E. Börger (eds.), *Evolving Algebras – Mini-Course, BRICS Technical Report (BRICS-NS-95-4)*, 195–222, University of Aarhus, Denmark, July 1995. This work is based upon G. Del Castillo's Tesi di Laurea "Descrizione Matematica dell'Architettura Parallela APE100", Università di Pisa, academic year 1993/94. An extended abstract "An Evolving Algebra model for the APE100 parallel architecture" was presented to the 3d Annual Meeting of the Working Group 0.1.6 "Logik in der Informatik" of the German Association for Computer Science, University of Karlsruhe CS TR 23/95, pp. 48–51. 103, 105, 112, 155, 157, 349, 359, 372, 382, 383, 395, 407

103. E. Börger and B. Demoen. A framework to specify database update views for Prolog. In M. J. Maluszynski (ed.), *PLILP'91. Third Int. Sympos. on Programming Languages Implementation and Logic Programming.*, Lecture Notes in Computer Science, Vol. 528, pp. 147–158. Springer-Verlag, 1991.

Provides a precise definition of the major Prolog database update views (immediate, logical, minimal, maximal), within a framework closely related to [71, 72, 74]. A preliminary version of this was published as *The View on Database Updates in Standard Prolog: A Proposal and a Rationale* in ISO/ETC JTCI SC22 WG17 Prolog Standardization Report no. 74, February 1991, pp. 3–10. 345, 388

104. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.

The final draft version has been issued in BRICS Technical Report (BRICS-NS-95-4); see [250]. It sharpens the refinement method of [132] to cope also with parallelism and non-determinism for an imperative programming language. The paper provides a mathematical definition of the Transputer instruction-set architecture for executing Occam together with a correctness proof for a general compilation schema of Occam programs into Transputer code.

Starting from the Occam model developed in [105], constituted by an abstract processor running a high- and a low-priority queue of Occam processes (which formalizes the semantics of Occam at the abstraction level of atomic Occam instructions), increasingly more refined levels of Transputer semantics are developed, proving correctness (and when possible also completeness) for each refinement step.

Along the way proof assumptions are collected, thus obtaining a set of natural conditions for compiler correctness, so that the proof is applicable to a large class of compilers. The formalization of the Transputer instruction-set architecture has been the starting point for applications of the ASM refinement method to the modeling of other architectures (see [102, 119]). 42, 88, 112, 156, 281, 298, 350, 378, 381, 384, 389, 405, 407, 408

105. E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and compiler correctness. Part I: Simple mathematical interpreters. In U. Montanari and E. R. Olderog (eds.), *Proc. PROCOMET'94 (IFIP Working Conf. on Programming Concepts, Methods and Calculi)*, pp. 489–508. North-Holland, 1994.

Improving upon the parse tree determined ASM in [257], a truly concurrent ASM model of Occam is defined as the basis for a proven-to-be-correct, smooth transition to the Transputer instruction-set architecture. This model is stepwise refined, in a proven-to-be-correct way, providing: (a) an asynchronous implementation of synchronous channel communication, (b) its optimization for internal channels, (c) the sequential implementation of processors using priority and time-slicing. See [104] for the extension of this work to cover the compilation to Transputer code. 88, 134, 280, 282, 351, 382, 383, 408

106. E. Börger, A. Gargantini, and E. Riccobene (eds.). *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589. Springer-Verlag, 2003.

This is the Proc. 10th Int. ASM Workshop (Taormina March 2003). 361

107. E. Börger and U. Glässer. A formal specification of the PVM architecture. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 402–409, Elsevier, Amsterdam, 1994.

After Börger's lectures on ASMs at the University of Paderborn in the early summer of 1993, Glässer suggested providing an ASM model for the Parallel Virtual machine (PVM [214], the Oak Ridge National Laboratory software system that serves as a general-purpose environment for heterogeneous distributed computing). The model in this paper defines PVM at the C-interface, at the level of abstraction which is tailored to the programmer's understanding. Cf. the survey *An abstract model of the parallel virtual machine (PVM)* presented at *7th Int. Conf. on Parallel and Distributed Computing Systems* (PDCS'94), Las Vegas/Nevada, 5.-9.10.1994. See [108] for an elaboration of this paper. 38, 274, 350, 360, 401, 419

108. E. Börger and U. Glässer. Modeling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In Y. Gurevich and E. Börger (eds.), *Evolving Algebras – Mini-Course, BRICS Technical Report BRICS-NS-95-4*, pp. 128–153. University of Aarhus, Denmark, July 1995.

This is a tutorial introduction to the ASM approach to design and verification of complex computing systems. The salient features of the method are explained by showing how one can develop from scratch an easily understandable and transparent ASM model for PVM [214], the widespread virtual architecture for heterogeneous distributed computing. 274, 350, 360, 384, 401, 407

109. E. Börger and U. Glässer. Abstract State Machines 2001: New developments and applications. In E. Börger and U. Glässer (eds.), *J. Universal Computer Science*, Vol. 7(11), pp. 914–917. Springer-Verlag, 2001.

Introduction to the third special ASM issue of JUCS, with papers selected from those submitted after the Int. ASM'2001 Workshop held in Las Palmas. This issue contains [262, 387, 405, 142, 194, 208, 391]. 361, 385

110. E. Börger and U. Glässer. Abstract State Machines Workshop 2001. In R. Moreno-Díaz and A. Quesada-Arencibia (eds.), *Formal Methods and Tools for Computer Science*, pp. 212–304. IUCTC Universidad de Las Palmas de Gran Canaria, 2001.

Abstracts of talks presented to the Int. ASM'2001 Workshop held in Las Palmas de Gran Canaria from February 13–19, 2001, as part of Eurocast 2001. See [109]. 361, 405

111. E. Börger, U. Glässer, and W. Müller. The semantics of behavioral VHDL'93 descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pp. 500–505, Los Alamitos, California, 1994. IEEE Computer Society Press.

Provides a transparent but precise ASM definition of the signal behavior and time model of full *elaborated* VHDL'93. This includes guarded signals, delta and time delays, the two main propagation delay modes *transport* and *inertial*, and the three process suspensions (wait on/until/for). Shared variables, postponed processes and rejection pulses are covered. The work is extended in [112]. 20, 43, 44, 87, 350, 359, 385, 411, 417, 418, 422

112. E. Börger, U. Glässer, and W. Müller. Formal definition of an abstract VHDL'93 simulator by ea-machines. In C. Delgado Kloos and P. T. Breuer (eds.), *Formal Semantics for VHDL*, pp. 107–139. Kluwer Academic Publishers, 1995.

Extends the work in [111] by including the treatment of variable assignments and of value propagation by ports. References [383, 380] extend the VHDL model to analog VHDL and to Verilog. 38, 39, 87, 350, 359, 385, 411, 417, 418, 422

113. E. Börger and R. Gotzhein. The light control case study. *J. Universal Computer Science*, 6(7):580–585, 2000.

The introductory pp. 580–585 present the requirements engineering case study, discussed during a Dagstuhl Seminar on Requirements Engineering [115], and a synopsis of the six solutions published in the journal issue. For the solution which uses ASMs see the comment to [125]. 229, 231, 355, 385

114. E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger (ed.), *Specification and Validation Methods*, pp. 231–243. Oxford University Press, 1995.

One ASM A1 is constructed to reflect faithfully the algorithm. Then a more abstract ASM A2 is constructed. It is checked that A2 is safe and fair, and that A1 correctly implements A2. The proofs work for atomic as well as for, mutatis mutandis, durative actions. See also [157, 258]. 156, 260, 282, 349, 351, 378, 379, 381, 393, 407, 408, 415

115. E. Börger, B. Hörger, D. L. Parnas, and D. Rombach. *Requirements Capture, Documentation, and Validation*, Vol. 241. Dagstuhl Seminar No. 99241, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, June 1999.

The Light Control Case Study was proposed to the participants of the seminar to discuss methods for solving requirements engineering problems. See [113] for a detailed exposition of some of the proposed solutions, including [125]. 229, 355, 356, 385, 388

116. E. Börger and J. Huggins. Abstract State Machines 1988–1998: Commented ASM bibliography. *Bull. EATCS*, 64:105–127, 1998.

The 1997 version of the annotated bibliography of papers which deal with or use ASMs. An update of [78]. 379, 381

117. E. Börger, P. Joannou, and D. L. Parnas. *Practical Methods for Code Documentation and Inspection*, Vol. 178. Dagstuhl Seminar No. 9720, Schloss

Dagstuhl, Int. Conf. and Research Center for Computer Science, May 1997. 188, 355, 387

118. E. Börger, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A model for mathematical analysis of functional logic programs and their implementations. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 410–415, Elsevier, Amsterdam, 1994.

    Defines an ASM model for the innermost version of the functional logic programming language BABEL, extending the Prolog model of [131] by rules which describe the reduction of expressions to normal form. The model is stepwise refined towards a mathematical specification of the implementation of Babel by a graph-narrowing machine. The refinements are proved to be correct. A full version containing optimizations and proofs appeared under the title *Towards a Mathematical Specification of a Narrowing Machine* as research report DIA 94/5, Dpto. Informática y Automática, Universidad Complutense, Madrid 1994. 114, 115, 346, 419

119. E. Börger and S. Mazzanti. A practical method for rigorously controllable hardware design. In J. P. Bowen, M. B. Hinchey, and D. Till (eds.), *ZUM'97: The Z Formal Specification Notation*, Lecture Notes in Computer Science, Vol. 1212, pp. 151–187. Springer-Verlag, 1997.

    A technique for specifying and verifying the control of pipelined microprocessors is described, illustrated through ASM models for Hennessy and Patterson's RISC architecture DLX. A serial DLX model is stepwise refined to the parallel DLX with five-stage pipeline which is proved to be correct. Each refinement deals with a different pipelining problem (structural hazards, data hazards, control hazards) and the methods for its solution. This makes the approach applicable to design-driven verification as well as to the verification-driven design of RISC machines. A preliminary version appeared under the title *A correctness proof for pipelining in RISC architectures* as DIMACS (Rutgers University, Princeton University, ATT Bell Laboratories, Bellcore) research report TR 96-22, pp. 1–60, Brunswick, New Jersey, July 1996. The specification was worked out in 1994/95 by S. Mazzanti for her Tesi di Laurea *Algebre Dinamiche per il DLX*, Università di Pisa, 1995, supervised by Börger. For a machine verification using KIV and PVS of the refinement of the serial to the parallel model see [217, 407]. An omission in the proof for the second refinement step has been pointed out by Holger Hinrichsen, see [279] and Sect. 3.2 of this book for a correction. The specification and proof method has ben applied in [286] to the commercial ARM2 RISC Microprocessor with a simpler three-stage pipeline and enhanced in [412] to automatically transform register transfer descriptions of microprocessors into executable ASMs. 137, 152, 156, 298, 349, 359, 378, 379, 381, 383, 401, 410, 411, 426

120. E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *J. Universal Computer Science*, 3(5):603–665, 1997.

    Presents a structured software engineering method which allows the software engineer to control efficiently the *modular development* and the *maintenance* of well documented, formally inspectable and easily modifiable code out of rigorous ASM *models for requirement specifications*. Shows that the code properties of interest (like correctness, safety, liveness and performance conditions) can be proved at high levels of abstraction by traditional and reusable mathematical arguments which – where needed – can be computer verified. Shows also that the proposed method is appropriate for dealing in a rigorous but

transparent manner with hardware–software co-design aspects of system development.
The approach is illustrated by developing a C++ program for the production-cell case study. The program has been validated by extensive experimentation with the FZI production cell simulator in Karlsruhe and has been submitted for inspection to the Dagstuhl Seminar on "Practical Methods for Code Documentation and Inspection" [117]. Source code (the ultimate refinement) for the case study appears in [332]; model checking results for the ASM models appear in [424] and in [362], where an error was detected in a refinement step for the deposit belt, due to an erroneous assumption of symmetry between unloading actions for feed belt, press and deposit belt. For a PVS verification of the case see [207]. An abstract appeared under the title "The Evolving Algebra Approach to Modular Development of Well Documented Software for Complex Systems. A Case Study: The Production Cell Control Program" in the Proc. DIMACS Workshop on Controllers for Manufacturing and Automation: Specification, Synthesis, and Verification Issues–CONMASSYV, May 1996, DIMACS. The work was part of Mearelli's Tesi di Laurea *Sviluppo Sistematico di un Programma di Controllo per un Impianto di Produzione Robotizzato*, Pisa 1994/95, supervised by Börger. 156, 188, 190, 193, 195, 355, 374, 378, 380, 381, 400, 401, 415, 416, 419, 423, 428

121. E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 361–366. Springer-Verlag, 2000.

    A report on the successful use of ASMs at Siemens AG (from May 1998 to March 1999) to redesign and implement the railway process model component of FALKO, a railway timetable validation and construction program. Extensive testing was done for the ASM model prior to its coding, using the ASM Workbench [170]. For a powerpoint slide show see `Falko` (⇝ CD). 24, 38, 86, 88, 103, 104, 340, 341, 357, 360, 363, 394, 423

122. E. Börger and E. Riccobene. A mathematical model of concurrent Prolog. Research Report CSTR-92-15, Dept. of Computer Science, University of Bristol, Bristol, England, 1992.

    An ASM formalization of Ehud Shapiro's Concurrent Prolog. Adaptation of the model defined for PARLOG in [123]. 282, 348, 403, 421

123. E. Börger and E. Riccobene. A formal specification of Parlog. In M. Droste and Y. Gurevich (eds.), *Semantics of Programming Languages and Model Theory*, pp. 1–42. Gordon and Breach, 1993.

    An ASM formalization of Parlog, a well-known parallel version of Prolog. This formalization separates explicitly the two kinds of parallelism occurring in Parlog: AND–parallelism and OR–parallelism. It uses an implementation-independent, abstract notion of terms and substitutions and is obtained by combining the concurrent features of the Occam model of [257] with the logic programming model of [75]. Also published as Technical Report TR 1/93 from Dipartmento di Informatica, Università di Pisa, 1993. An improved and extended version of the following two papers by the same authors: *Logical Operational Semantics of Parlog. Part I: And-Parallelism* in H. Boley and M. M. Richter (eds.): Processing Declarative Knowledge (Lecture Notes in Artificial Intelligence, Vol. 567, pp. 191–198, Springer-Verlag, 1991). *Logical Operational Semantics of Parlog. Part II: Or-Parallelism* in A. Voronkov (ed.): Logic Programming (Lecture Notes in Artificial Intelligence, Vol. 592, pp. 27–34, Springer-Verlag, 1992). For an extension to Pandora see [374]. 282, 348, 387, 413, 421

124. E. Börger and E. Riccobene. Logic + control revisited: An abstract interpreter for Gödel programs. In G. Levi (ed.), *Advances in Logic Programming Theory*, pp. 231–154. Oxford University Press, 1994.

Develops a simple ASM interpreter for Gödel programs. This interpreter abstracts from the deterministic and sequential execution strategies of Prolog [132] and thus provides a precise interface between logic and control components for execution of Gödel programs. The construction is given in abstract terms which cover the general logic programming paradigm and allow for concurrency. 346, 421

125. E. Börger, E. Riccobene, and J. Schmid. Capturing requirements by Abstract State Machines: The light control case study. *J. Universal Computer Science*, 6(7):597–620, 2000.

ASMs are applied to the Light Control Case Study discussed during a Dagstuhl Seminar on Requirements Engineering [115]. A ground model is defined which captures the informal requirements as far as possible and documents their ambiguity and incompleteness. The ground model is then refined into a form directly executable by AsmGofer [390]. 230, 341, 356, 364, 374, 385, 423, 424

126. E. Börger and D. Rosenzweig. An analysis of prolog database views and their uniform implementation. ISO/IEC JTCI SC22 WG17 Prolog Standardization Document 80, National Physical Laboratory, Teddington, Middlesex, England, 1991.

A mathematical analysis of the Prolog database views defined in [103]. The analysis is derived by stepwise refinement of the stack model for Prolog from [132]. It leads to the proposal of a uniform implementation of the different views which discloses the tradeoffs between semantic clarity and efficiency of database update view implementations. Also issued as Research Report CSE-TR-89-91 by the EECS Dept., University of Michigan, Ann Arbor. 345

127. E. Börger and D. Rosenzweig. A formal specification of Prolog by tree algebras. In V. Čeric, V. Dobrić, V. Lužar, and R. Paul (eds.), *Information Technology Interfaces*, pp. 513–518. University Computing Center, Zagreb, Zagreb, 1991.

Prompted by discussion in the international Prolog standardization committee (ISO/IEC JTC1 SC22 WG17), this paper suggests replacing the stack-based model of [71] and the stack implementation of the tree-based model of [72] by a pure tree model for Prolog. See also [75, 73], which is the basis for [131], where a mistake in the treatment of the *catch* built-in predicate is corrected. 389, 413

128. E. Börger and D. Rosenzweig. From Prolog algebras towards WAM – a mathematical study of implementation. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld (eds.), *CSL'90, 4th Workshop on Computer Science Logic*, Lecture Notes in Computer Science, Vol. 533, pp. 31–66. Springer-Verlag, 1991.

Refines Börger's Prolog model [72] by elaborating the conjunctive component – as reflected by compilation of clause structure into WAM code – and the disjunctive component – as reflected by compilation of predicate structure into WAM code. The correctness proofs for these refinements include last call optimization, determinacy detection and virtual copying of dynamic code. Extended in [129] and improved in [132]. 347, 389

129. E. Börger and D. Rosenzweig. WAM algebras – a mathematical study of implementation, Part 2. In A. Voronkov (ed.), *Logic Programming*, Lecture Notes in Artificial Intelligence, Vol. 592, pp. 35–54. Springer-Verlag, 1992.

Refines the Prolog model of [128] by elaborating the WAM code for representation and unification of terms. The correctness proof for this refinement includes environment trimming, Warren's variable classification and switching instructions. Improved in [132]. Also issued as Technical Report CSE-TR-88-91 from EECS Dept, University of Michigan, Ann Arbor, Michigan, 1991. 156, 347, 388, 389

130. E. Börger and D. Rosenzweig. The mathematics of set predicates in Prolog. In G. Gottlob, A. Leitsch, and D. Mundici (eds.), *Computational Logic and Proof Theory*, Lecture Notes in Computer Science, Vol. 713, pp. 1–13. Springer-Verlag, 1993.

Provides a logical (proof–theoretical) specification of the solution-collecting predicates *findall, bagof* of Prolog. This abstract ASM-based definition allows a logico–mathematical analysis, rationale and criticism of various proposals made for implementations of these predicates (in particular of *setof*) in current Prolog systems. Foundational companion to Sect. 5, on solution collecting predicates, in [131]. Also issued as *Prolog. Copenhagen papers 2*, ISO/IEC JTC1 SC22 WG17 Standardization report no. 105, National Physical Laboratory, Middlesex, 1993, pp. 33–42. 345

131. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.

An abstract ASM specification of the semantics of Prolog, rigorously defining the international ISO 1995 Prolog standard by stepwise refinement. Revised and final version of [71, 72, 101, 127, 75, 73]. An abstract of this was issued as *Full Prolog in a Nutshell* in *Logic Programming* (Proc. 10th Int. Conf. on Logic Programming) (D. S. Warren, Ed.), MIT Press 1993. A preliminary version appeared under the title *A Simple Mathematical Model for Full Prolog* as research report TR-33/92, Dipartimento di Informatica, Università di Pisa, 1992. 20, 87, 114, 115, 157, 347, 378, 381, 382, 386, 388, 389, 413

132. E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In C. Beierle and L. Plümer (eds.), *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, Vol. 11, Chap. 2, pp. 20–90. North-Holland, 1995.

The successive-refinement method introduced for ASMs in [71, 72, 74] is applied to provide a hierarchy of models as a mathematical basis for constructing provably correct compilers from Prolog to WAM. Various refinement steps take care of different distinctive features ("orthogonal components") of WAM, making the specification as well as the correctness proof modular and extendible; examples of such extensions are found in [41, 42, 133, 21, 313]. An extension of this work to an imperative language with parallelism and non-determinism has been provided in [104] and is further developed in [137, 141, 406]. See [369, 388] for machine-checked versions of the correctness proofs for the refinement steps. Preliminary versions appeared in [128, 129] and as Research Report TR-14/92, Dipartimento di Informatica, Università di Pisa, 1992. 23, 38, 112, 134, 156, 298, 347, 350, 357, 371, 373, 374, 378, 381, 383, 388, 389, 390, 415, 420, 423, 424

133. E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP($\mathcal{R}$) programs. In E. Börger (ed.), *Specification and Validation Methods*, pp. 97–130. Oxford University Press, 1995.

Extends the specification and correctness proof, for compiling Prolog programs to the WAM [132], to CLP($\mathcal{R}$) and the constraint logical arithmetical machine (CLAM) developed at IBM Yorktown Heights. For full proofs, see R. Salamone, "Una Specifica Astratta e Modulare della CLAM (An Abstract and Modular Specification of the CLAM)", Tesi di Laurea, supervised by Börger at Università di Pisa, Italy, academic year 1992/93, pp. 113. 23, 114, 115, 298, 347, 349, 378, 379, 381, 389

134. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg (eds.), *Computer Science Logic (Proceedings of CSL 2000)*, Lecture Notes in Computer Science, Vol. 1862, pp. 41–60. Springer-Verlag, 2000.

    Structuring concepts for sequential composition and iteration, parameterization, and encapsulation in ASMs are defined. The concept of recursive submachines has been developed for its use in [406] to provide a modular definition of the statics and the dynamics of Java and of the JVM architecture [89] which can be naturally refined to an executable model, namely written in AsmGofer [390]. 86, 185, 341, 342, 359, 364, 380, 381, 409, 416, 423, 425

135. E. Börger and P. Schmitt. A formal operational semantics for languages of type Prolog III. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld (eds.), *CSL'90, 4th Workshop on Computer Science Logic*, Lecture Notes in Computer Science, Vol. 533, pp. 67–79. Springer-Verlag, 1991.

    An ASM formalization of Alain Colmerauer's constraint logic programming language Prolog III, obtained from the Prolog model in [71, 72, 74] through extending unifications by constraint systems. This extension was the starting point for the extension of [132] in [40]. A preliminary version of this was issued as IBM Germany IWBS Report 144, 1990. 346

136. E. Börger and P. Schmitt. A description of the tableau method using Abstract State Machines. *J. Logic and Computation*, 7(5):661–683, 1997.

    Starting from the textbook formulation of the tableau calculus, an operational description of the tableau method is given in terms of ASMs at various levels of refinement ending after four stages at a specification that is close to the leanT^AP implementation of the tableau calculus in Prolog. Proofs of correctness and completeness of the refinement steps are given. 365, 378, 381

137. E. Börger and W. Schulte. Defining the Java Virtual Machine as platform for provably correct Java compilation. In L. Brim, J. Gruska, and J. Zlatuska (eds.), *Mathematical Foundations of Computer Science 1998, 23rd Int. Sympos., MFCS'98, Brno, Czech Republic*, Lecture Notes in Computer Science, Vol. 1450, pp. 17–35. Springer-Verlag, August 1998.

    A definition of the Java Virtual Machine, along with a provably correct compilation scheme for Java programs to the JVM, based on the ASM semantics for Java presented in [138]. Streamlined, corrected and completed in [406]. The full version appears as Technical Report, Universität Ulm, Fakultät für Informatik, Ulm, Germany, 1998. 359, 380, 389, 391, 409, 428

138. E. Börger and W. Schulte. Programmer friendly modular definition of the semantics of Java. In J. Alves-Foss (ed.), *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science, Vol. 1523. Springer-Verlag, 1998.

    Provides a system- and machine-independent definition of the semantics of the full programming language Java as seen by the Java programmer. The definition is modular, coming as a series of refined ASMs, dealing in succession with Java's imperative core, its object-oriented features, exceptions and threads. Streamlined, corrected and completed in [406]. An extended abstract has been presented by Börger to the IFIP WG 2.2 (University of Graz, 22–26 September, 1997) and by Schulte under the title *Modular Dynamic Semantics of Java* to the Workshop on Programming Languages (Ahrensdorp, FEHMARN Island, September 25, 1997), see University of Kiel, Dept. of CS Research Report Series, TR *Arbeitstagung Programmiersprachen* 1997. An independently developed Java model using ASMs and Montages was published later as a technical report in [421]. For an ASM model of Java which is geared to the analysis of the concurrency features see [259]. 156, 359, 365, 378, 380, 381, 390, 391, 409, 428

139. E. Börger and W. Schulte. Initialization problems for Java. *Software – Concepts and Tools*, 19(4):175–178, 2000.

 Using the models in [138, 137] and reporting results of experiments with current implementations of the JVM it is shown that the treatment of initialization of classes and interfaces in Java and in the Java Virtual Machine do not match, afflicting the portability of Java programs. It is shown that concurrent initialization may deadlock and that various Java compilers violate the initialization semantics through standard optimization techniques. 359, 380, 409, 428

140. E. Börger and W. Schulte. Modular design for the Java VM architecture. In E. Börger (ed.), *Architecture Design and Validation Methods*, pp. 297–357. Springer-Verlag, 2000.

 Provides a modular definition of the Java VM architecture, at different layers of abstraction. The layers partly reflect the layers made explicit in the specification of the Java language in [138]. The ASM model for JVM defined here and the ASM model for Java defined in [138] provide a rigorous framework for a machine independent mathematical analysis of the language and of its implementation, including compilation correctness conditions, safety and optimization issues. Streamlined, corrected and completed in [406]. 359, 380, 409, 428

141. E. Börger and W. Schulte. A practical method for specification and analysis of exception handling: A Java/JVM case study. *IEEE Trans. Software Eng.*, 26(10):872–887, October 2000.

 ASM models for exception handling in Java and the Java Virtual Machine (JVM) are given, along with a compilation scheme for Java to JVM code. It is proven that corresponding runs of the Java and JVM throw the same exceptions with equivalent effect. A different proof is offered in [406]. 112, 359, 380, 389, 409, 428

142. E. Börger and D. Sona. A neural abstract machine. *J. Universal Computer Science*, 7(11):1007–1024, 2001.

 A parameterized Neural Abstract Machine is defined whose instantiations cover the major neural networks in the literature. The refinement for feedforward networks with back-propagation training is shown. 199, 298, 381, 384

143. D. Bowen. Implementation at Quintus of Börger's Prolog ASM. Personal Communication to Börger at Quintus in Palo Alto on November 5 and e-mail of November 11, 1990.

 The four ASM rules which constitute the core for user-defined predicates in Börger's Prolog model [101, 75] have been implemented, making use of the code available at Quintus to compute the abstract functions which appear in that model, in particular the function *unify*, and the function *procdef* which for a given goal (literal) and a given program yields the ordered set of alternatives the program offers for resolving the goal. 340, 347, 363

144. M. Broy, S. Merz, and K. Spies. The RPC memory case study: A synopsis. In M. Broy, S. Merz, and K. Spies (eds.), *Formal Systems Specification – The RPC-Memory Specification Case Study*, Lecture Notes in Computer Science, Vol. 1169. Springer-Verlag, August 1996.

    For an ASM solution of the case study see [284]. 356, 411

145. A. Brüggemann, L. Priese, D. Rödding, and R. Schätz. Modular decomposition of automata. In E. Börger, G. Hasenjäger, and D. Rödding (eds.), *Logic and Machines: Decision Problems and Complexity*, Lecture Notes in Computer Science, Vol. 171, pp. 198–236. Springer-Verlag, 1984. 164

146. B. Buchberger and B. Roider. Input/output codings and transition functions in effective systems. *Int. J. General Systems*, 4:201–209, 1978. 54

147. J. R. Burch. Techniques for verifying superscalar microprocessors. In *Proc. 33rd Annual Conf. on Design Automation Conference*, pp. 552–557, Las Vegas, Nevada, 3–7 June 1996. ACM Press. 155

148. W. Burgard, A. B. Cremers, D. Fox, M. Heidelbach, A. M. Kappel, and S. Lüttringhaus-Kappel. Knowledge-enhanced CO-monitoring in coal mines. In *Proc. Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA-AIE)*, pp. 511–521, Fukuoka, Japan, 4–7 June 1996.

    Extends the ASM interpreter of [301] by modules, which can be executed in parallel, so that distributed processes can be represented which are synchronized via stream communication. Also a graphical visualization is added, which is needed for industrial applications of the system in a time- and security-critical coal mining application reported in the paper. Available at `http://www.informatik.uni-bonn.de/~angelica/publications.html` . 357, 413

149. C. Campbell and Y. Gurevich. Table ASMs. In R. Moreno-Díaz and A. Quesada-Arencibia (eds.), *Formal Methods and Tools for Computer Science (Local Proceedings of Eurocast 2001)*, pp. 286–290, Canary Islands, Spain, February 2001. Universidad de Las Palmas de Gran Canaria.

    A special table notation for a class of basic ASMs is presented. 29

150. G. D. Castillo and P. Päppinghaus. Designing software for internet telephony: experiences in an industrial development process. In A. Blass, E. Börger, and Y. Gurevich (eds.), *Theory and Applications of Abstract State Machines*, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, 2002.

    The development of a high-level abstract model of the core functionality of an entity in a mobile telephony network is reported. The model served as the basis for a C++ product implementation. 340, 362

151. S. Cater and J. Huggins. An ASM dynamic semantics for standard ML. Technical Report CPSC-1999-2, Kettering University, Flint, Michigan, October 1999.

    ASMs are used to provide dynamic semantics for the functional programming language Standard ML. An extended abstract appears in Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, eds., *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 203–222, Springer-Verlag, 2000, and in TIK-Report No. 87, pp. 68–99, ETH Zürich, March 2000.

152. A. Cavarra. *Applying Abstract State Machines to Formalize and Integrate the UML Lightweight Method*. PhD thesis, University of Catania, Sicily, Italy, 2000.

The thesis, which was supervised by Börger and Riccobene, studies the use of ASMs to rigorously support semi-formal specification techniques as they are used in industrial practice, with a focus on UML notations and concepts. In addition to the work, which has been published in [98, 99], a simulator for UML state machines has been developed using AsmGofer [390]. 341, 356, 364, 382, 393, 423

153. A. Cavarra, E. Riccobene, and P. Scandurra. Integrating UML static and dynamic views and formalizing the interaction mechanism of UML state machines. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 229–243. Springer-Verlag, 2003. 47, 88, 364, 382

154. A. Cavarra, E. Riccobene, and A. Zavanella. A formal model for the parallel semantics of P3L. In J. Carroll, E. Damiani, H. Haddad, and D. Oppenheim (eds.), *Proc. 2000 ACM Sympos. Applied Computing*, Lecture Notes in Computer Science, Vol. 2, pp. 804–812. ACM Press, 2000.

Provides an ASM formalization of the semantics of P3L, a programming language with task and data parallelism. The model describes (a) how the compiler defines a network of processes starting from a given program, and (b) the computation of the running processes. Some rewrite rules for trimming the compiler for better program performance are proved to be correct. 360

155. Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison Wesley, 2000.

See also `http://java.sun.com/products/javacard/havacard21.html`. 88

156. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999. 299

157. J. Cohen and A. Slissenko. On verification of refinements of asynchronous timed distributed algorithms. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 34–49. Springer-Verlag, 2000.

A study of the role of timing constraints for proving the correctness of refinements of distributed asynchronous algorithms with continuous time, specified as distributed ASMs. The ASM investigation of Lamport's Bakery Algorithm in [114] is used as a case study. Also appears in TIK-Report No. 87, pp. 100–114, ETH Zürich, March 2000. 352, 385

158. K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, EECS Department, University of Michigan, 2000.

Using the ideas developed in [98, 99, 152], ASMs are used to give semantics for UML state machines, as a basis for constructing an automated tool for verifying the properties of UML state machines. An extended abstract appears as "A Semantic Model for the State Machine in the Unified Modeling Language" in G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa (eds.), "Dynamic Behaviour in UML Models: Semantic Questions", Workshop Proceedings, UML 2000 Workshop, Ludwig-Maximilians-Universität München, Institut für Informatik, Bericht 0006, October 2000, pp. 25–31.

159. D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Trans. Software Eng.*, 21(2):90–98, 1995. 299

160. A. B. Cremers, U. Griefahn, and R. Hinze. *Deduktive Datenbanken*. Vieweg, 1994. 272

161. A. B. Cremers and T. N. Hibbard. Formal modeling of virtual machines. *IEEE Trans. Software Eng.*, SE-4(5):426–436, 1978. 353

162. F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 6:175–187, 1991. 240, 252

163. F. DaCruz. *Kermit: A File Transfer Protocol*. Digital Press, 1987.

     See also the Kermit Web site `http://www.columbia.edu/kermit`. 240, 352, 410

164. O. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, 1972. 9, 353

165. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001. 22, 104

166. M. Davis. *The Universal Computer: The Road from Leibniz to Turing*. W.W. Norton, New York, 2000. 166

167. W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, Cambridge, 1998. 22, 134, 156

168. M. Dehof and S. Tahar. Implementierung des DLX-RISC-Prozessors in einer Standardzellen-Entwurfsumgebung. Technical Report SBF 358-C2-9/94, Institue of Computer Design and Fault Tolerance, University of Karlsruhe, Germany, 1994. 137

169. G. Del Castillo. Towards comprehensive tool support for Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann (eds.), *Applied Formal Methods – FM-Trends 98*, Lecture Notes in Computer Science, Vol. 1641, pp. 311–325. Springer-Verlag, 1999.

     A description of the ASM Workbench, an integrated environment for various ASM tools; see [170]. Another description appears under the title *The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines* in [226, pp. 139–154]. 363, 403

170. G. Del Castillo. *The ASM Workbench. A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models*. PhD thesis, Universität Paderborn, Germany, 2001.

     Published in: HNI-Verlagsschriftenreihe Vol. 83, 217 pages. The main contribution of the thesis, supervised by Börger and Glässer, is the definition of the ASM-based specification language ASM-SL and a tool architecture – the ASM Workbench – based on ASM-SL. The tool environment includes basic functionalities such as parsing, abstract syntax trees, type checking, pretty printing, etc., and in particular a transformation of ASMs into FSMs which can be model checked using SMV; see [175]. In the thesis a case study from the domain of automated manufacturing is treated, namely the distributed control for a material flow system. The ASM Workbench has been extensively used for testing purposes in the FALKO project at Siemens [121]. It has been used in [355] to provide an executable semantics for UML. 340, 341, 357, 363, 364, 387, 394, 419, 423, 429

171. G. Del Castillo, I. Durdanović, and U. Glässer. An evolving algebra abstract machine. In H. K. Büning (ed.), *Proc. Ann. Conf. of the European Association for Computer Science Logic (CSL'95)*, Lecture Notes in Computer Science, Vol. 1092, pp. 191–214. Springer-Verlag, 1996.

    Introduces the concept of an abstract machine (EAM) as a platform for the systematic development of ASM tools and gives a formal definition of the EAM ground model in terms of a universal ASM. The definition proceeds by stepwise refinement and leads to the design of a simple virtual machine architecture as a basis for a sequential implementation of the EAM. A preliminary version appeared under the title *Specification and Design of the EAM (EAM – Evolving Algebra Abstract Machine)* as Technical Report TR-RSFB-96-003, Paderborn University, 1996. 355, 363, 397

172. G. Del Castillo and U. Glässer. Computer-aided analysis and validation of heterogeneous system specifications. In F. Pichler, R. Moreno-Díaz, and P. Kopacek (eds.), *Computer Aided Systems Theory: Proc. 7th Int. Workshop on Computer Aided Systems Theory (EUROCAST'99)*, Lecture Notes in Computer Science, Vol. 1798, pp. 55–79. Springer-Verlag, 2000.

    ASMs are proposed as a method for combining heterogeneous specifications. As a case study, Petri-net and SDL specifications of a material flow system are combined via ASMs and validated using SMV [175]. 356, 395

173. G. Del Castillo and W. Hardt. Fast dynamic analysis of complex hardware/software systems based on Abstract State Machine models. In *Proc. 6th Int. Workshop on Hardware/Software Codesign (CODES/CASHE'98) (March 15–18, Seattle, Washington)*, pp. 77–81, 1998.

    Provides experimental results for [174]. 359, 395

174. G. Del Castillo and W. Hardt. Towards a unified analysis methodology of HW/SW systems based on Abstract State Machines: Modeling of instruction sets. In *Proc. GI/ITG/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, Paderborn, Germany, March 1998.

    Extending the processor description technique from [102], ASMs are used for high-level analysis of hardware/software systems. The authors show how to model instruction sets using ASMs and to instrument such models to collect data for evaluating design alternatives. Experimental results appear in [173]. 359, 395

175. G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach (eds.), *Proc. 6th Int. Conf. TACAS 2000*, Lecture Notes in Computer Science, Vol. 1785, pp. 331–346. Springer-Verlag, 2000.

    Extending [424], the authors introduce an interface from the ASM Workbench to the SMV model checking tool, based on an ASM-to-SMV transformation. Previously appeared as Universität-GH Paderborn Technical Report TR-RI-99-209. For an extension see [425, 426]. For an experiment with this interface see [172]. 341, 394, 395, 428, 429

176. J. Derrick and E. Boiten. *Refinement in Z and Object-Z.* Formal Approaches to Computing and Information Technology. Springer-Verlag, 2001. 21, 22, 134, 156, 185

177. S. Dexter, P. Doyle, and Y. Gurevich. Gurevich Abstract State Machines and Schönhage Storage Modification Machines. *J. Universal Computer Science*, 3(4):279–303, 1997.

A demonstration that, in a strong sense, Schönhage's storage modification machines are equivalent to unary basic ASMs without external functions. The unary restriction can be removed if the storage modification machines are equipped with a pairing function in an appropriate way. 301, 344, 380, 424

178. V. Di Iorio, R. Bigonha, and M. Maia. A self-applicable partial evaluator for ASM. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines – ASM 2000, Int. Workshop on Abstract State Machines, Monte Verità, Switzerland, Local Proceedings*, TIK-Report, No. 87, pp. 115–130. ETH Zürich, March 2000.

    A partial evaluator for ASMs is described which is self-applicable. The use of such a tool for compiler generation and techniques for describing language semantics suitable for partial evaluation are discussed. Implementation details are in *An ASM Implementation of a Self-Applicable Partial Evaluator* by V. Di Iorio and R. Bigonha, Technical Report LLP-004-2000 of Programming Languages Laboratory, DCC, Universidade Federal de Minas Gerais, 2000. Extends the work of [252]. 363, 407

179. S. Diehl. Transformations of evolving algebras. In *Proc. LIRA'97 (VIII Int. Conf. on Logic and Computer Science)*, pp. 43–50, Novi Sad, Yugoslavia, September 1997.

    Constant propagation is introduced as a transformation on ASMs. ASMs are extended by macro definitions, folding and unfolding transformations for macros, a simple transformation to flatten transition rules and a pass separation transformation for ASMs are defined. For all transformations the operational equivalence of the resulting ASMs with the original ASMs is proven. In the case of pass separation, it is shown that the results of the computations in the original and the transformed ASMs are equal. Pass separation is applied to a simple interpreter. A preliminary version appeared in 1995 as Technical Report 02/95 of Universität des Saarlandes.

180. D. Diesen. *Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages.* D Sc Thesis, Dept. of Informatics, University of Oslo, Norway, March 1995.

    A description of a functional interpreter for ASMs, with applications for functional programming languages, along with a proposed extension to the language of ASMs. 363

181. B. DiFranco. *Specification of ISO SQL using Montages.* Master's thesis, Università di l'Aquila, Italy, 1997.

    Tesi di Laurea, in Italian. 414

182. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 117, 121

183. E. W. Dijkstra. Structure of the T.H.E. multiprogrammming system. *Commun. ACM*, 11:341–346, 1968. 8, 297, 353

184. E. W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (eds.), *Structured Programming*, pp. 1–82. Academic Press, 1972. 21, 24

185. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976. 4

186. S. Distefano. *Architettura X86 e sistema dei processi di MINIX dalla specifica ASM al codice eseguibile.* Master's thesis, University of Catania, Sicily, Italy, 1998/99.

The multiprogramming operating system MINIX together with the architecture X86 are modeled as ASMs and refined to an implementation in Java and in C. 39, 282, 340

187. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.

 A technique for formally representing ASMs using the automated verification system PVS is described, along with generic PVS theories which define refinement relations between ASMs. An application to *Representing the Alpha Processor Family using PVS* by the same author appears as Verifix/Uni Ulm/4.1, University of Ulm, Germany, November 1995. 300, 358

188. A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt (eds.), *Proc. 5th Int. Workshop on Abstract State Machines*, pp. 50–67. Magdeburg University, 1998.

 Using techniques from [439], an approach is described for mechanically proving the correctness of back-end rewrite system (BURS) specifications where source and target languages are described by ASMs. The approach can be used in conjunction with BURS-based back-end compiler generators. PVS proof strategies are defined for the automatic verification of BURS rules. Similar aspects are treated by A. Dold and T. Gaul and W. Zimmermann in *Mechanized Verification of Compiler Back-Ends* in Proc. Int. Workshop on Software Tools for Technology Transfer (STTT'98), Aalborg, Denmark, July 12–13, 1998. 300, 358, 403

189. A. Durand. Modeling cache coherence protocol – a case study with FLASH. In U. Glässer and P. Schmitt (eds.), *Proc. 5th Int. Workshop on Abstract State Machines*, pp. 111–126. Magdeburg University, 1998.

 During his research stay in Pisa in 1997/98, upon Börger's suggestion Durand investigated the cache coherence protocol in the Stanford FLASH multiprocessor system, for which he provides a high-level specification and correctness proofs related to data consistency. For a model checking verification of the model using SMV see [425]. 361, 403, 428

190. I. Durdanović. From operational specifications to real architectures. Draft of PhD Thesis (NEC Research Institute Princeton), March 2, 2000.

 This PhD project, supervised by Börger, continues the ideas presented in [171]. An ASM Virtual Architecture is defined as the basis for a comprehensive ASM tool environment. The developed base system contains an ASM parser and a compiler into ASM/VA code which is a form of high-level C++ programs whose actual refinement into C++ is supported by programs in a C++ library. 363

191. E.A. Community. Name change to replace EA by something better. Electronic Discussion at ea@ira.uka.de, September 6 to October 11 (1996).

 The discussion was proposed with the following motivation: “Algebra” makes the theoreticians think that the approach belongs to the algebraic specification and verification research area – and their dissatisfaction and misjudgement comes from our violating so many (I would say almost all) of their cherished concepts and beliefs. “Algebra” makes the practitioners think that we want them to use algebraic notation and equations or laws – and this is enough for them not even to look further at what we really do. “Evolving” is either not understood at all or in the best of all cases interpreted as implying that the signature should be allowed to change – this comes from the analogy with biological systems where the concept is used that way. (Börger on

Sept. 6)

In a lively discussion, two dozen names were proposed, resulting in Päppinghaus' proposal of (Gurevich's) *Abstract State Machines* to become generally accepted. Here are the concluding messages of October 10/11 which resume this decision.

From: Erik.Tiden@zfe.siemens.de
To: eboerger@prosun.first.gmd.de
Subject: Name of the beast.

Dear Prof. Börger, I write in English, so that you can quote me to your community if you wish. The name "Gurevich Machines" is impossible in industry, because it only evokes associations of useless (in industrial practice) theoretical concepts. The name"Abstract State Machines" on the other hand, is fine. That's also what we will keep on calling them here at Siemens central research. Thus, if you stick to "Gurevich Machines" you will end up with two names. Now, if you regard ASMs as a theoretical exercise, investigation, whatever, into the foundations of CS or some such worthy cause, then you can call them whatever you like of course. If you want to make ASMs into something which is useful in practice, calling them GM is simply foolish. Best regards, Erik Tiden.

Answer of October 11.
From: Egon Börger eboerger@prosun.first.gmd.de
To: Erik.Tiden@zfe.siemens.de
Subject: Abstract State Machines (Gurevich Machines).

Dear Dr. Tiden, thanks for your valuable comment on the EA name problem which I am going to answer in English so that the whole community can follow this. I do not know whether you did follow the entire discussion; I had started it pushed by the need to find a name which helps those of us who aim at practical (in particular industrial) applications of the specification, verification and code development method which has been built around Gurevich's notion of evolving algebras. I am glad that through the discussion we have found such a name, namely Gurevich ('s Abstract State) Machines. By the way, the first step to this solution, namely the proposal to call the beasts Abstract State Machines, came from one of your collaborators, Dr. Paeppinghaus, to whom I am grateful for his suggestion. Adding the inventor's name to Abstract State Machines is in accordance with usual practice and provided the chance to conclude the search not with two really different names (EA and ASM) but with ONE name which is generally accepted by the community. Gurevich Machines or Abstract State Machines are not two different names but only shorthands for Gurevich's Abstract State Machines. Here is another variation, appearing in the title for one of my forthcoming lectures: Abstract State Machines (Gurevich Machines). An interesting feature which makes Gurevich's ASMs unique is that they have both practical AND theoretical relevance (although surprisingly enough the theoretical potential of the notion of Gurevich Machines has been recognized and explored even less than its practical relevance). Therefore it IS valuable to have a unique name which takes into account the sometimes diverging interests. I hope this is a satisfactory answer to your message. With best wishes, Egon Börger. 361, 366

192. S. Eilenberg. *Automata, Machines and Languages*, Vol. A. Academic Press, 1974. 290

193. R. Eschbach. A termination detection algorithm: Specification and verification. In J. Wing, J. C. P. Woodcock, and J. Davies (eds.), *Proc.*

FM'99, Vol. II, Lecture Notes in Computer Science, Vol. 1709, pp. 1720–1737. Springer-Verlag, 1999.

A two-level specification of a distributed termination detection algorithm is given using ASMs. The lower-level specification of the algorithm is proved equivalent to the upper-level specification. 361

194. R. Eschbach, U. Gässer, R. Gotzhein, M. v. Löwis, and A. Prinz. Formal definition of SDL-2000 – compiling and running SDL specifications as ASM models. *J. Universal Computer Science*, 7(11):1025–1050, 2001.

Contains the most recent and detailed survey of the SDL-2000 formal semantics definition [292] that was accepted in 2000 by ITU-T, the international standardization body for telecommunication. The focus of this survey is on the dynamic semantics, where ASMs have been applied as the underlying framework. In particular, the SDL Abstract Machine (SAM) model including real time, the definition of SAM programs and their execution by the SDL Virtual Machine (SVM) (SDL-to-ASM compiler and further tool support) are presented. 87, 341, 359, 384, 402, 412

195. R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz. On the formal semantics of SDL-2000: A compilation approach based on an abstract SDL machine. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 242–265. Springer-Verlag, 2000.

An overview of the semantics of SDL-2000, whose complete and final definition which uses ASMs appears in [292]. A simplified language SPL is defined and described using ASMs to point out some of the unique features of the semantics of SDL-2000. Also in TIK-Report No. 87, pp. 131–151, ETH Zürich, March 2000. 358, 403

196. H. Eveking. Machine assisted verification. In E. Börger (ed.), *Architecture Design and Validation Methods*, pp. 191–242. Springer-Verlag, 2000. 299

197. L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design.* Cambridge University Press, Cambridge, 1992. 180, 294, 405

198. L. M. G. Feijs, H. B. M. Jonkers, C. P. J. Koymans, and G. R. Renardel de Lavalette. Formal definition of the design language COLD-K. Technical Report No. 234/87, Philips Research Laboratories, 1987.

Also appeared as ESPRIT Document No. METEOR/t7/PRLE/7, 1987. Final update in August 1989. 405

199. J. Fitzgerald and P. G. Larsen. *Modeling Systems. Practical Tool and Techniques in Software Development.* Cambridge University Press, Cambridge, 1998. 156, 295

200. B. Fordham, S. Abiteboul, and Y. Yesha. Evolving databases: An application to electronic commerce. In *Proc. Int. Database Engineering and Applications Sympos. (IDEAS)*, pp. 191–200, Montreal, August 1997.

The paper describes an ASM-based prototype system, in the spirit of active databases, for specifying electronic commerce applications. An extensible database model called "evolving databases" (EDB) is defined based upon ASMs. It is applied to capture in a rigorously transparent way the state changes involved in electronic commerce negotiations, concerning the traded products, the negotiators, their orders and the laws accepted as the basis for the particular negotiation. See [1]. 360

201. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at `http://research.microsoft.com/foundations/AsmL/`, 2001. 30, 185, 341, 364

202. G. Franceschinis and M. Ribaud. Efficient performance analysis techniques for stochastic well-formed nets and stochastic-process algebras. In W. Reisig and G. Rozenberg (eds.), *Lectures on Petri Nets II: Applications*, Vol. 1492, pp. 386–437. Springer-Verlag, 1998. 241

203. N. G. Fruja and R. F. Stärk. The hidden computation steps of turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 244–262. Springer-Verlag, 2003. 86, 178, 185

204. N. E. Fuchs. Specifications are (preferably) executable. *Software Eng. J.*, 7(5):323–334, September 1992.

    Reprinted in: J. P. Bowen, M. G. Hinchey, *High-Integrity System Specification and Design*, pp. 583–608. Springer-Verlag, London, 1999. 19

205. N. E. Fuchs, U. Schwertel, and R. Schwitter. Attempto controlled English – not just another logic specification. In P. Flener (ed.), *Logic-Based Program Synthesis and Transformation, 8th Int. Workshop LOPSTR'98*, Lecture Notes in Computer Science, Vol. 1559, pp. 1–20. Springer-Verlag, 1998. 19

206. M. Gaieb. *Géneration de spécifications Centaur á partir de specifications Montages*. Master's thesis, Université de Nice – Sophia Antipolis, France, June 1997.

    This works investigate the possibilities of mapping the operational ASM semantics of the static analysis phase of Montages [311] into the declarative Natural Semantics framework. A formalization for the list arrows of Montages is found – a feature that has not been fully formalized in [311]. In addition, the Gem-Mex Montages tool is interfaced to the Centaur system (which executes Natural Semantics specificaions), and the tool suport of Centaur is exploited in order to generate structural editors for languages defined with Montages.

207. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 303–322. Springer-Verlag, 2000.

    A framework for automatic translation from ASM to PVS is presented. Following a suggestion by Börger, the ASM specification of the Production Cell problem [120] is used as a case study. Also appears in TIK-Report No. 87, pp. 152–173, ETH Zürich, March 2000. 188, 300, 355, 387, 425

208. A. Gargantini and E. Riccobene. ASM-based testing: Coverage criteria and automatic test sequence generation. *J. Universal Computer Science*, 7(11):1051–1068, 2001.

    ASMs are used for testing purposes, defining adequacy criteria measuring the coverage achieved by a test suite, and determining whether sufficient testing has been performed. An algorithm is defined to generate from ASMs test sequences with desired coverage, exploiting the counter example generation of SMV. See the continuation in [209]. 86, 360, 384, 400

209. A. Gargantini and E. Riccobene. Using Spin to generate tests from ASM specifications. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 263–277. Springer-Verlag, 2003.

    Continuation of [208]. 360, 400

210. M. C. Gaudel. *Génération et Preuve de Compilateurs Basées sur une Sémantique Formelle des Langages de Programmation.* Thèse, L'Institut National Polytechnique de Lorraine, France, 1980.

The work to which mostly the idea is attributed of using Tarski structures as the most general notion of states. But see [359]. 8, 353, 419

211. T. Gaul. An Abstract State Machine specification of the DEC-Alpha processor family. Verifix Working Paper Verifix/UKA/4, University of Karlsruhe, Germany, 1995.

An ASM for the DEC-Alpha processor family is derived directly from the original manufacturer's handbook. The specification omits certain less-used instructions and VAX compatibility parts. 358, 401

212. T. Gaul, A. Heberle, and W. Zimmermann. An ASM specification of the operational semantics of MIS. Verifix Working Paper Verifix/UKA/3, University of Karlsruhe, Germany, 1998.

An ASM specification of MIS, an intermediate programming language used in the Verifix project for provably correct compilation to the DEC-Alpha microprocessor [211]. 358

213. A. Gawanmeh, S. Tahar, and K. Winter. Interfacing ASMs with the MDG tool. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 278–292. Springer-Verlag, 2003. 364

214. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

A high-level model of the system described in this book has been developed in [107, 108]. 273, 350, 384

215. S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, 11(1):21–28, January 1994. 299

216. F. Giannuzzi. *Studi di un metodo per la derivazione dei casi di test da specifiche ASM.* Tesi di laurea, Universitá di Pisa, Italy, July 2001.

Studies the derivation of test cases from ASM specifications, illustrating an application of the cause-effect-graph method for the Production Cell ASM [120]. Supervised by Bertolino and Börger. 360

217. M. Giese, D. Kempe, and A. Schönegge. KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining Architektur. Interner Bericht 16/97, Universität Karlsruhe, Germany, 1997.

The Karlsruhe Interactive Verifier (KIV system) is used for the formal verification of the parallelization of the ASM specification for the serial DLX architecture, the first step of the refinement to its parallel version with five-stage pipelining in [119]. Two additions to the KIV system are described which were designed in the course of this case study. 137, 142, 146, 148, 156, 359, 386

218. U. Glässer. Systems level specification and modeling of reactive systems: Concepts, methods, and tools. In F. Pichler, R. Moreno-Díaz, and R. Albrecht (eds.), *Computer Aided Systems Theory–EUROCAST'95:* Proc. 5th Int. Workshop on Computer Aided Systems Theory (Innsbruck, Austria, May 1995*)*, Lecture Notes in Computer Science, Vol. 1030, pp. 375–385. Springer-Verlag, 1996.

The paper investigates the derivation of formal requirements and design specifications at systems level as part of a comprehensive design concept for complex reactive systems. In this context the meaning of correctness with respect to the embedding of mathematical models into the physical world is discussed.

219. U. Glässer. Combining Abstract State Machines with predicate transition nets. In F. Pichler and R. Moreno-Díaz (eds.), *Computer Aided Systems Theory – EUROCAST'97 (Proc. 6th Int. Workshop on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, Feb. 1997)*, Lecture Notes in Computer Science, Vol. 1333, pp. 108–122. Springer-Verlag, 1997.

The work investigates the formal relation between ASMs and Pr/TPredicate Transition (Pr/T-) Nets with the aim of integrating both approaches into a common framework for modeling concurrent and reactive system behavior, where Pr/T-nets are considered as a graphical interface for distributed ASMs. For the class of *strict Pr/T-nets* (which constitutes the basic form of Pr/T-nets) a transformation to distributed ASMs is given. 356

220. U. Glässer. ASM semantics of SDL: Concepts, methods, tools. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz (eds.), *Proc. 1st Workshop of the SDL Forum Society on SDL and MSC*, Informatik-Berichte, Vol. 104 (ISSN 0863-095), pp. 271–280. Humboldt-Universität Berlin, 1998.

Proposal to the SDL Forum to use ASMs for a definition of the semantics of SDL which is abstract but through its operational character is apt to be transformed to an executable model. Detailed in [225]. 358

221. U. Glässer. *Analysis and Validation of Formal Requirement Specifications in Model-Based Engineering of Concurrent Systems.* Habilitationsschrift, University of Paderborn, Germany, 1999.

Contains a systematic treatment of the work started in [225] providing ASM models for the dynamic semantics of SDL. Completed in [292]; see [194] for a survey. 358, 403

222. U. Glässer, R. Gotzhein, and A. Prinz. Towards a new formal SDL semantics based on Abstract State Machines. In G. v. Bochmann, R. Dssouli, and Y. Lahav (eds.), *SDL'99 – The Next Millenium, Proc. 9th SDL Forum*, pp. 171–190. Elsevier Science B.V., 1999.

Based upon the idea proposed in [225], ASMs are applied to formally define the behavior model of a sample SDL-2000 specification. See also "SDL Formal Semantics Definition" by the same authors, published as University of Paderborn Report No. TR SFBR-99-065, June 1999. See the completion of the work in [292] and the survey [194]. 358, 403, 419

223. U. Glässer, Y. Gurevich, and M. Veanes. An abstract communication model. Technical Report MSR-TR-2002-55, Microsoft Research, Redmond, Washington, May 2002.

From [224] an abstract communication model is extracted. 105, 108, 402

224. U. Glässer, Y. Gurevich, and M. Veanes. High-level executable specification of the universal plug and play architecture. In *Proc. 35th Hawaii Int. Conf. on System Sciences – 2002*, pp. 1–10. IEEE Computer Society Press, 2002.

An AsmL specification of the Universal Plug and Play (UPnP) architecture for peer-to-peer network connectivity of intelligent devices. A more detailed version appeared in June 2001 as Microsoft Research Technical Report MSR-TR-2001-59 under the title "Universal Plug and Play Models". See [223]. 88, 298, 341, 362, 402

225. U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *J. Universal Computer Science*, 3(12):1382–1414, 1997.

A formal semantic model of Basic SDL-92 – according to the *ITU-T Recommendation Z.100* – is defined in terms of an abstract SDL machine based on the concept of a *multi-agent real-time ASM*. The resulting interpretation model is not only mathematically precise but also reflects the common understanding of SDL in a direct and intuitive manner; it provides a *concise* and *understandable* representation of the complete dynamic semantics of Basic SDL-92. Moreover, the model can easily be *extended* and *modified*. The article considers the behavior of channels, processes and timers with respect to signal transfer operations and timer operations. Continuation of this work and merging it with work by Gotzhein and by Prinz [222, 221, 367] led to the ITU-T standard definition of SDL-2000 [292, 195]. 358, 402

226. U. Glässer and P. Schmitt (eds.). *Proc. 5th Int. Workshop on Abstract State Machines*, Germany, 1998. GI Jahrestagung 1998, Otto-von-Guericke-Universität Magedeburg.

Extended abstracts of the talks presented to the workshop which was organized as part of the 28th Annual Conf. of the German Computer Science Society (GI Jahrestagung). See [435, 395, 328, 188, 274, 414, 189, 46, 169]. 361, 394, 427

227. P. Glavan and D. Rosenzweig. Communicating evolving algebras. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter (eds.), *Computer Science Logic*, Lecture Notes in Computer Science, Vol. 702, pp. 182–215. Springer-Verlag, 1993.

A theory of concurrent computation within the framework of ASMs is developed, generalizing [257, 122]. As an illustration models are given for the Chemical Abstract Machine and the π-calculus. See [248] for a more general definition of the notion of distributed ASM runs. 282, 348

228. P. Glavan and D. Rosenzweig. Evolving algebra model of programming language semantics. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 416–422, Elsevier, Amsterdam, 1994.

Defines an ASM interpretation of many-step SOS, denotational semantics and Hoare logic for the language of while-programs and states correctness and completeness theorems, based on a simple flowchart model of the language. 419

229. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. In P. Fritzson (ed.), *Int. Conf. on Compiler Construction, Proc. Poster Session of CC'96*, Linköping, Sweden, 1996. IDA Technical Report LiTH-IDA-R-96-12.

In this project a method is developed to establish, modulo hardware correctness, the correctness of reliable initial compilers (not only compiler specifications) for an appropriate high-level system programming language. The approach is based upon multiple-phase compilation (with closely related intermediate languages) and a diagonal bootstrapping technique. The following three major steps are performed. (1) Verification of a specification of the compilation function with respect ot the semantics of the source and target language and a correctness definition. Here ASMs are used to rigorously define source and target language semantics and the correctness property. PVS is used for proof support. (2) Verification of a compiler implementation in a high-level language, using generators to generate the front end and parts of the back end. Small (proven to be correctly implemented) checker routines are used to verify by syntactical a posteriori code inspection that the input and output of the generators have the needed properties (program checking). (3) Verification of a compiler implementation in binary. An initial bootstrap compiler is used which is proved (once) to be correctly implemented in binary. In the project this is a compiler from COMLISP to Transputer code, whose semantics are defined by SOS methods. No further binary code verification is necessary. For the program checker and other system software it suffices to implement them correctly in the high-level source language of the initial compiler (using standard program transformation or verification techniques). 300, 358

230. G. Goos, A. Heberle, W. Löwe, and W. Zimmermann. On modular definitions and implementations of programming languages. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines – ASM 2000, Int. Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings*, TIK-Report, No. 87, pp. 174–208. ETH Zürich, March 2000.

A formal composition and refinement (correct implementation) mechanism for state-transition systems is presented which exploits the abstract syntax of programs. Applications are made to language semantic definitions using ASMs. Montages [311] is characterized as a set of parameterized ASMs. 363

231. G. Goos and W. Zimmermann. Verifying compilers and ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 177–202. Springer-Verlag, 2000.

ASMs are used to describe verifying compilers: compilers which verify the correctness of their generated code. 112, 358

232. G. Gottlob, G. Kappel, and M. Schrefl. Semantics of object-oriented data models – the evolving algebra approach. In J. W. Schmidt and A. A. Stogny (eds.), *Next Generation Information Technology*, Lecture Notes in Computer Science, Vol. 504, pp. 144–160. Springer-Verlag, 1991.

Uses ASMs to define, in the context of a graphical object-oriented data model design language, the operational semantics of object creation, of overriding and dynamic binding, and of inheritance at the type level (type specialization) and at the instance level (object specialization). Issued also as technical report MooD-TR 90/02, Technische Universität Wien, December 20, 1990. See [397]. 347, 364, 424

233. E. Grädel and Y. Gurevich. Metafinite model theory. *Information and Computation*, 140(1):26–81, 1998.

Computer systems, *e.g.* databases, are not necessarily finite because they may involve, for example, arithmetic. Motivated by such computer science challenges and by ASM applications, metafinite structures, as they typically appear in ASM states, are defined and finite model theory is extended to metafinite models. An early version has been presented under the title *Towards a Model Theory of Metafinite Structures* to the Logic Colloquium 1994; see the abstract in the *J. Symbolic Logic*. An intermediate version appeared in *Logic and Computational Complexity, Selected Papers*, Lecture Notes in Computer Science, Vol. 960, pp. 313–366, Springer-Verlag, 1995. 361

234. E. Grädel and A. Nowack. Quantum computing and Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State*

Machines 2003–Advances in Theory and Applications, Lecture Notes in Computer Science, Vol. 2589, pp. 309–323. Springer-Verlag, 2003.

Derives the ASM thesis for quantum algorithms from postulates which are inspired by the axiomatization of parallel algorithms in [61]. 362, 377

235. E. Grädel and M. Spielmann. Logspace reducibility via Abstract State Machines. In J. Wing, J. Woodcock, and J. Davies (eds.), *Proc. FM'99, Vol. II*, Lecture Notes in Computer Science, Vol. 1709, pp. 1738–1757. Springer-Verlag, 1999.

    ASMs are used to investigate logspace reducibility among structures, capturing the choiceless fragment of logspace. A continuation of [62]. See also [402]. 361, 377, 425

236. I. Graham. *The Transputer Handbook*. Prentice-Hall, 1990.

    Together with [289, 290], this book served as the basis for the ASM model developed for the Transputer in [104]. 350, 411

237. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from Abstract State Machines. In *Software Engineering Notes*, Vol. 27.4, pp. 112–122. ISSTA 02, 2002.

    Proposes a scheme for grouping ASM states into finitely many equivalence classes ("hyperstates") on which an FSM is induced to reflect the ASM transitions for testing purposes. A preliminary version was presented to the Int. ASM'01 Workshop in February 2001 [110]. It appeared in October 2001 under the title "Conformance Testing with Abstract State Machines" as MSR-TR-2001-97 and in May 2002 revised under the same number with the new title. 86, 122, 123, 360

238. R. Groenboom and G. R. Renardel de Lavalette. A formalization of evolving algebras. In S. Fischer and M. Trautwein (eds.), *Proc. Accolade 95*, pp. 17–28. Dutch Research School in Logic, ISBN 90-74795-31-5, 1995.

    The authors present the syntax and semantics for a Formal Language for Evolving Algebra (FLEA) covering sequential ASMs. This language is then extended to a multi-modal language FLEA$'$, and it is sketched how one can transfer the axioms of the logic MLCM to FLEA$'$. MLCM is a Modal Logic of Creation and Modification, a dynamic logic which is incorporated in Jonker's Common Object-Oriented Language for Design, COLD [197, 198]. See [405]. 321, 322, 337, 365, 425

239. M. Grosse-Rhode. A formal specification framework for evolving algebras. Unpublished manuscript. Technical University of Berlin, 1996.

    Applies some algebraic-categorical composition schemes to ASMs, illustrated on an alternating-bit protocol specification. 352

240. Y. Gurevich. Reconsidering Turing's Thesis: Toward more realistic semantics of programs. Technical Report CRL-TR-36-84, EECS Department, University of Michigan, September 1984.

    An attempt to reconsider Turing's Thesis, taking into account that resources are bounded. The earliest known paper in which the ideas behind ASMs began to take form. See the continuation in [241]. 343, 344, 405

241. Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, 6(4):317, August 1985.

    Following [240], for the first time the ASM Thesis is stated, but no definition for ASMs is given yet. See the continuation in [244, 242]. 301, 344, 405, 406

242. Y. Gurevich. Algorithms in the world of bounded resource. In R. Herken (ed.), *The Universal Turing Machine – A Half-Century Story*, pp. 407–416. Oxford University Press, 1988.

Early complexity theoretical motivation for the introduction of ASMs is discussed. 344, 405

243. Y. Gurevich. Kolmogorov machines and related issues. *Bull. EATCS*, 35:71–82, 1988.

The Kolmogorov–Uspenskii thesis is stated that every computation, performing only one restricted local action at a time, can be viewed as the computation of an appropriate Komogorov–Uspenskii machine. 344

244. Y. Gurevich. Logic and the challenge of computer science. In E. Börger (ed.), *Current Trends in Theoretical Computer Science*, pp. 1–57. Computer Science Press, 1988.

Part 2 contains the first small examples for ASMs, drawn from Gurevich's lectures in Semantics of Programming Languages delivered in Pisa in May 1986 (not "in the Spring of 1987" as stated erroneously, e.g. in [92]. 345, 377, 405

245. Y. Gurevich. Evolving algebras. A tutorial introduction. *Bull. EATCS*, 43:264–284, 1991.

The ASM thesis is stated. A slightly revised version was reprinted under the title "Evolving Algebras: An attempt to discover semantics" in G. Rozenberg and A. Salomaa Eds, *Current Trends in Theoretical Computer Science*, World Scientific, 1993, pp. 266–292. A german textbook version of the definition appeared in [73]. For a more elaborate and complete definition see [248]. 32, 35, 86, 345, 347, 406, 407

246. Y. Gurevich. Evolving Algebras. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 423–427, Elsevier, Amsterdam, 1994.

The opening talk at the first ASM workshop. Sections: Introduction, The ASM Thesis, Remarks, Future Work. 407, 419

247. Y. Gurevich. Logic activities in Europe. *ACM SIGACT News*, 25(2):11–24, 1994.

A critical analysis of European logic activities in computer science. Subsection 4.6 *Mathematics and Pedantics* discusses the separation of different levels of verification in the context of modeling with ASMs. 356

248. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger (ed.), *Specification and Validation Methods*, pp. 9–36. Oxford University Press, 1995.

The notion of sequential ASMs defined in [245] is extended to cover distributed computations. A later update *May 1997 Draft of the ASM Guide* appeared as Technical Report CSE-TR-336-97, EECS Dept., University of Michigan. 8, 32, 35, 64, 77, 86, 282, 343, 348, 349, 379, 403, 406, 407, 414

249. Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Trans. Computational Logic*, 1(1):77–111, July 2000.

The notion of "sequential algorithm" is axiomatized to derive from three basic axioms the sequential version of the "ASM thesis" which was proposed in [241, 245]. An early version appeared under different titles as Microsoft Research Technical Reports MSR-TR-99-09 and MSR-TR-99-65, and in Bull. EATCS 67 (February 1999), 93–124. See [61] for an extension to the notion of "synchronous parallel algorithms". 306, 311, 344, 348, 377

250. Y. Gurevich and E. Börger. Evolving algebras – mini course. BRICS Technical Report BRICS-NS-95-4, ISSN 0909-3206, University of Aarhus, Denmark, July 1995.

Contains reprints of the papers [56, 245, 246, 248, 252, 254, 251, 114, 102, 104, 108] which were used as material for a course on ASMs delivered by the two authors at BRICS, Aarhus, in the summer of 1995. 383

251. Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter (eds.), *Computer Science Logic*, Lecture Notes in Computer Science, Vol. 702, pp. 274–309. Springer-Verlag, 1993.

The method of successive refinements is used to give a succinct dynamic semantics of the C programming language. For a correction of minor errors and omissions see the ERRATA in Lecture Notes in Computer Science, Vol. 832, pp. 334–336, Springer-Verlag, 1994. An early version appeared under the title *The Evolving Algebra Semantics of C: Preliminary Version* as Technical Report CSE-TR-141-92, EECS Department, University of Michigan, Ann Arbor, 1992. This work is included in the PhD thesis *Evolving Algebras: Tools for Specification, Verification, and Program Transformation* of the second author, pp. IX+91, supervised by Gurevich at the University of Michigan, Ann Arbor, 1995. For an extension to C++ see [420]. For an addition of the statics of C to the model see [285]. 347, 407, 411, 428

252. Y. Gurevich and J. Huggins. Evolving algebras and partial evaluation. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 587–592, Elsevier, Amsterdam, 1994.

The paper describes an automated partial evaluator for sequential ASMs implemented at the University of Michigan. It takes an ASM and a portion of its input and produces a specialized ASM using the provided input to execute rules when possible and generating new rules otherwise. A full version appears as J. Huggins, "An Offline Partial Evaluator for Evolving Algebras", Technical Report CSE-TR-229-95, EECS Department, University of Michigan, Ann Arbor, 1995. This work is included in the PhD thesis *Evolving Algebras: Tools for Specification, Verification, and Program Transformation* of the second author, pp. IX+91, University of Michigan, Ann Arbor, 1995. For an extension of this work see [178]. 363, 396, 407

253. Y. Gurevich and J. Huggins. The railroad crossing problem: An experiment with instantaneous actions and immediate reactions. In *Proc. CSL'95 (Computer Science Logic)*, Lecture Notes in Computer Science, Vol. 1092, pp. 266–290. Springer-Verlag, 1996.

An ASM solution for the railroad crossing problem in [277]. The paper experiments with agents that perform instantaneous actions in continuous time at the moment they are enabled. A preliminary version appeared under the title *The Railroad Crossing Problem: An Evolving Algebra Solution* as research report LITP 95/63 of Centre National de la Recherche Scientifique, Paris, and under the title *The Generalized Railroad Crossing Problem: An Evolving Algebra Based Solution* as research report CSE-TR-230-95 of EECS Department, University of Michigan, Ann Arbor, MI. For a further investigation see [34, 35]. 201, 203, 205, 356, 364, 372, 373, 410

254. Y. Gurevich and J. Huggins. Equivalence is in the eye of the beholder. *Theoretical Computer Science*, 179(1–2):353–380, 1997.

A response to a paper of Leslie Lamport, "Processes are in the Eye of the Beholder" which is published in the same volume. It is discussed how the same two algorithms may and may not be considered equivalent. In addition, a direct proof is given of an appropriate equivalence of two particular algorithms considered by Lamport. A preliminary version appeared as research report CSE-TR-240-95, EECS Dept., University of Michigan, Ann Arbor, Michigan 1995. 228, 361, 407

255. Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.). *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912. Springer-Verlag, 2000.

Proc. Int. Workhop ASM2000 held at Monte Verità, Switzerland, March 2000. 361

256. Y. Gurevich and R. Mani. Group membership protocol: Specification and verification. In E. Börger (ed.), *Specification and Validation Methods*, pp. 295–328. Oxford University Press, 1995.

A processor-group membership protocol involving timing constraints is formally specified and verified using distributed ASMs. 240, 252, 259, 349, 352, 379

257. Y. Gurevich and L. S. Moss. Algebraic operational semantics and Occam. In E. Börger, H. Kleine Büning, and M. M. Richter (eds.), *CSL'89, 3rd Workshop on Computer Science Logic*, Lecture Notes in Computer Science, Vol. 440, pp. 176–192. Springer-Verlag, 1990.

The first application of ASMs to distributed parallel computing with the challenge of true concurrency. For an improved (no longer parse tree determined, but truly concurrent) ASM model for Occam and its refinement to a Transputer implementation see [105, 104]. 282, 348, 384, 387, 403, 413

258. Y. Gurevich and D. Rosenzweig. Partially ordered runs: A case study. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 131–150. Springer-Verlag, 2000.

The ASM investigation [114] of Lamport's Bakery Algorithm is sharpened in terms of partially ordered runs, abstracting from the mapping of moves to linear realtime. Some properties are proved which are useful for reasoning about partially ordered runs. The paper also appeared as a technical report in TIK-Report No. 87, ETH Zürich, March 2000, and in MSR-TR-99-98. 271, 351, 385

259. Y. Gurevich, W. Schulte, and C. Wallace. Investigating Java concurrency using Abstract State Machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 151–176. Springer-Verlag, 2000.

An ASM specification and verification of Java's model of concurrency, including threads and synchronization. Also in TIK-Report No. 87, pp. 227–271, ETH Zürich, March 2000, and in University of Delaware Department of Computer & Information Sciences TR 2000-04. 391

260. Y. Gurevich, N. Soparkar, and C. Wallace. Formalizing database recovery. *J. Universal Computer Science*, 3(4):320–340, 1997.

A database recovery algorithm (the undo-redo algorithm) is modeled at several levels of abstraction, with verification of the correctness of the high-level model and of each of the four refinement steps. An updated version of the Technical Reports CSE-TR-249-95 and CSE-TR-327-97 of EECS Department, University of Michigan, Ann Arbor, and of the paper *Formalizing Recovery in Transaction-Oriented Database Systems* of C. Wallace and

Y. Gurevich and N. Soparkar, published in S. Chaudhuri and A. Deshpande and R. Krishnamurthy (eds.): Proc. 7th Int. Conf. on Management of Data, Tata McGraw-Hill, New Delhi, India, 1995, pp. 166–185. 117, 133, 360, 380

261. Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *J. Universal Computer Science*, 3(4):233–246, 1997.

 A definition of recursive ASMs in terms of distributed ASMs is suggested. A preliminary version appeared as Technical Report CSE-TR-322-96, EECS Department, University of Michigan, Ann Arbor, 1996. For a definition of recursive ASMs in terms of sequential ASMs see [134]. 171, 380

262. Y. Gurevich and N. Tillmann. Partial updates: Exploration. *J. Universal Computer Science*, 7(11):918–952, 2001.

 A solution is proposed for the problem of cumulative updates for counters, steps and maps. See the continuation in [263]. 53, 348, 384, 409

263. Y. Gurevich and N. Tillmann. Partial updates exploration II. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 57–86. Springer-Verlag, 2003.

 Continuation of [262]. 53, 409

264. Y. Gurevich and C. Wallace. Specification and verification of the Windows Card runtime environment using Abstract State Machines. Technical Report MSR-TR-99-07, Microsoft Research, Redmond, Washington, February 1999.

 An ASM specification of the Windows Card Runtime Environment and a verification of certain safety properties. 362

265. J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12), 1978. 8, 353, 419

266. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993. 38, 159, 161, 187, 273, 282, 311

267. J. A. Hall. Taking Z seriously. In *ZUM'97*, Lecture Notes in Computer Science, Vol. 1212, pp. 89–91. Springer-Verlag, 1997. 296, 299

268. D. Harel. Dynamic logic. In D. M. Gabbay and F. Guenthner (eds.), *Handbook of Philosophical Logic*, Vol. II, pp. 497–604. Reidel, Dordrecht, 1983. 314, 323

269. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000. 314

270. D. Harel and R. Marelly. Capturing and executing behavioral requirements: the play-in/play-out approach. Technical Report MCS01-15, Weizmann Institute of Science, Israel, 2001. 22, 104

271. D. Harel and M. Politi. *Modeling reactive systems with statecharts*. McGraw-Hill, 1998. 4

272. P. Hartel and L. Moreau. Formalizing the safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, 2001.

 A review of the literature on formal approaches of Java and its implementation with focus on safety issues and their impact on smart cards. Sect. 6.2 evaluates the ASM based work in this area [138, 137, 139, 140, 141, 406, 421]. 359, 425

273. A. Heberle. *Korrekte Transformationsphase – der Kern korrekter Übersetzer*. PhD thesis, Universität Karlsruhe, Germany, 2000.

 The essential results of the thesis (which is written in German) are published in [274, 275]. 112, 358, 410

274. A. Heberle and W. Löwe. On ASM-based specification of programming language semantics and reusable correct compilations. In U. Glässer and P. Schmitt (eds.), *Proc. 5th Int. Workshop on Abstract State Machines*, pp. 68–90. Magdeburg University, 1998.

General equivalence-preserving transformations on ASM specifications of programming languages are defined, to be used for the definition of provably correct compilation schemes. An extensible language AL is introduced for specifying dynamic language semantics in a way which facilitates the reuse of verified transformations. Some of the results are from [273]. 358, 403, 409

275. A. Heberle, W. Löwe, and M. Trapp. Safe reuse of source to intermediate language compilations. In R. Chillarege (ed.), *Proc. 9th. Int. Symp. on Software Reliability Engineering*, Fast Abstract and Industrial Tracts, Paderborn, Germany, 4–7 November 1998.

See `http://www.chillarege.com/issre/fastabstracts/98417.html`. Contains some results of [273]. 358, 409

276. C. Heitmeyer. Using SCR methods to capture, document, and verify computer system requirements. In E. Börger, B. Hörger, D. L. Parnas, and D. Rombach (eds.), *Requirements Capture, Documentation, and Validation*. Dagstuhl Seminar No. 99241, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, 1999. 294

277. C. Heitmeyer and D. Mandrioli. *Formal Methods for Real-Time Computing*, Trends in Software, Vol. 5. John Wiley, 1996.

Extensive study of the Railroad Crossing Problem, proposed as a case study for real-time computing and solved using various popular specification and verification methods. For an ASM solution see [253]. 187, 198, 201, 356, 407

278. J. L. Hennessy and D. A. Patterson. *Architecture: A Quantitative Approach*. Morgan Kaufman, 2nd edn., 1996. 137

279. H. Hinrichsen. Formally correct construction of a pipelined DLX architecture. Technical Report TR 98-5-1, Darmstad University of Technology, Dept. of Electrical and Computer Engineering, Germany, 1998.

In an e-mail to Börger on February 11, 1998, Hinrichsen points out that for a correct handling of the instruction sequence 1. LOAD R1 A, 2. LOAD R2 B, 3. ADD R3 R1 R2, the ADD instruction must be stalled for one clock cycle. This corrects an omission of a hazard case in the last refinement step of [119]. 137, 359, 386

280. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 583, 1969. 299

281. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 22, 272

282. M. Holcombe and F. Ipate. *Correct Systems. Building a Business Process Solution*. Springer-Verlag, 1998. 291

283. J. Huggins. Kermit: Specification and verification. In E. Börger (ed.), *Specification and Validation Methods*, pp. 247–293. Oxford University Press, 1995.

The Kermit file-transfer protocol [163] is specified and verified using ASMs at several layers of abstraction. This work is part of the author's PhD thesis *Evolving Algebras: Tools for Specification, Verification, and Program Transformation*, pp. IX+91, University of Michigan, Ann Arbor, 1995. 240, 241, 349, 352, 379

284. J. Huggins. Broy-Lamport Specification Problem: A Gurevich Abstract State Machine Solution. Technical Report CSE-TR-320-96, EECS Dept., University of Michigan, 1996.

Upon Börger's suggestion, Huggins developed an ASM solution to the specification problem proposed by Broy and Lamport, in conjunction with the Dagstuhl Seminar on Reactive Systems, held in Dagstuhl, Germany, 26–30 September, 1994. A preliminary version appeared as Technical Report CSE-TR-223-94, EECS Department, University of Michigan, Ann Arbor, 1994. Other solutions of this problem were published in [144]. 356, 392

285. J. Huggins and W. Shen. The static and dynamic semantics of C. Technical Report CPSC-2000-4, Kettering University, Computer Science Program, Flint, Michigan, 2000.

The ASM for C in [251] is extended to provide both static and dynamic semantics for C, using Montages [311]. An extended abstract appears in Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, eds., Abstract State Machines – ASM 2000, *Int. Workshop on Abstract State Machines*, Monte Verita, Switzerland, Local Proceedings, TIK-Report No. 87, pp. 272–283, ETH Zürich, March 2000. A previous version appears as Kettering University Computer Science Program Technical Report CPSC-1999-1. 88, 358, 407

286. J. Huggins and D. Van Campenhout. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Trans. Des. Autom. of Electron. Syst.*, 3(4):563–580, 1998.

An extended abstract describing a layered ASM specification of the advanced RISC machine processor ARM2, one of the early commercial RISC microprocessors. The method developed in [119] is applied for the layered specification and the correctness proof for the ARM2's pipelining techniques. In [412] this ASM model of the ARM is used to illustrate an approach to automatically transform register transfer descriptions of microprocessors into executable ASMs. A full version of the paper appears as University of Michigan EECS Department Technical Report CSE-TR-371-98. An earlier version appears in *Proc. IEEE Int. High Level Design Validation and Test Workshop (HLDTV'97)*, November 1997. 157, 359, 386, 427

287. M. Ibanez and H. Rempp. European user survey analysis. Technical Report TR 95104, European Software Institute, Bilbao, Spain, January 30 1996. 16

288. IEEE Std 1076-1993. *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, USA, 1993.

The standard description of the hardware design language VHDL'93 which has been formalized by an ASM ground model in [111, 112]. 350

289. INMOS. *Transputer Instruction Set – A Compiler Writer's Guide*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

INMOS Document 72 TRN 119 05. See the comment to [236]. 350, 405

290. INMOS. *Transputer Implementation of Occam – Communication Process Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

See comment to [236]. 350, 405

291. ISO. Prolog-Part 1: General core. ISO Standard Information Technology–Programming Languages ISO/IEC 13211-1, ISO/ICE, January 1995. 87, 345, 382

292. ITU-T. SDL formal semantics definition. ITU-T Recommendation Z.100 Annex F, International Telecommunication Union, November 2000.

This document contains the complete, internationally standardized formal semantics definition of SDL-2000, a design language for the development of distibuted real-time systems in general and telecommunication systems in particular. SDL is industrially applied in the telecommunications industry, for instance, to the development of UMTS protocols and Intelligent Networks. The dynamic semantics of SDL-2000 is defined using ASMs as the underlying mathematical framework. For further information see `http://rn.informatik.uni-kl.de/projects/sdl`. For a survey see [194]. 20, 87, 209, 358, 399, 402, 403

293. J. W. Janneck. *Syntax and Semantics of Graphs*. PhD thesis, ETH Zürich, Switzerland, 2000.

Published in: Berichte aus der Informatik, TIK Series Vol. 38, Shaker Verlag Aachen (ISBN 3-8265-7688-8), pp. XI+177. The classical networks of stream processing finite state machines (with their notion of network components with input and output ports to communicate among each other) are enriched by ASM state transformations of individual components. The resulting machines are applied to give a uniform rigorous semantics to common visual notations for discrete event systems, together with a prototypical implementation. Illustration by Petri nets. 287, 364

294. J. W. Janneck and P. Kutter. Mapping automata: Simple Abstract State Machines. TIK-Report 49, ETH Zürich, Switzerland, June 1998.

Mapping automata are defined as ASMs where the state is formed by a single binary function (interpreted as mapping which assigns to every object in the base set U a unary function over objects), and the rules are built from updates of that binary function in the usual way. Using the standard coding of arbitrary structures into the structure of one binary function, the resulting correspondence between mapping automata and ASMs is shown to preserve the desired computational equivalence. Also appears in Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, eds., Abstract State Machines – ASM 2000, *Int. Workshop on Abstract State Machines*, Monte Verita, Switzerland, Local Proceedings, TIK-Report No. 87, pp. 310–325, ETH Zürich, March 2000. An implementation in Java is reported in *Object-Based Mapping Automata (Reference Manual)* by J. W. Janneck, TIK-Report No. 50, ETH Zürich, June 1998. Mapping automata are used in [296] for a description of the semantics of UML statecharts. 364, 412

295. J. W. Janneck and P. Kutter. Object-based Abstract State Machines. TIK-Report 47, ETH Zürich, Switzerland, 1998.

Proposes to view ASMs as classes attached to objects which communicate only by message passing. Illustration by a class definition for Petri net places and transitions. 364

296. Y. Jin, R. Esser, and J. W. Janneck. Describing the syntax and semantics of UML statecharts in a heterogeneous modeling environment. In *Proc. DIAGRAMS 2002*, pp. 320–334, 2002.

Based upon the syntactical description of UML statecharts by attributed graphs coming with well-formedness conditions, the mapping automata of [294] are used to describe the semantics of UML statecharts. Compared with the ASM model of UML statecharts in [99], the focus here is on a discussion of transition conflicts. 364, 412

297. D. E. Johnson and L. S. Moss. Grammar formalisms viewed as Evolving Algebras. *Linguistics and Philosophy*, 17:537–560, 1994.

Distributed ASMs are used to model formalisms for natural language syntax. The authors start by defining an ASM model of context-free derivations which abstracts from the parse tree descriptions used in [257, 123] and from the dynamic tree generation appearing in [127, 131]. Then the basic model of context-free rules is extended to characterize in a uniform and natural way different context-sensitive languages in terms of ASMs. See [341, 342]. 114, 360, 417

298. J. Jürjens. *Principles for secure system development.* PhD thesis, Wolfson College Oxford, England, 2001. 291

299. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Using ASM specifications for compiler testing. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, p. 415. Springer-Verlag, 2003.

     Also presented under the title "Using ASM specification for automatic test suite generation for mpC parallel programming language compiler" to AS2000 (4th international workshop on Action Semantics and related frameworks), Copenhagen, July 2002. 342

300. A. Kaplan and J. Wileden. Formalization and application of a unifying model for name management. In *Proc. 3rd ACM SIGSOFT Sympos. on the Foundations of Software Engineering*, Software Engineering Notes, Vol. 20(4), pp. 161–172, October 1995.

     Presents a unifying model for name management, using ASMs as the specification language for the model. A preliminary version appeared in July 1995 as CMPSCI Technical Report 95-60 of Computer Science Department, University of Massachusetts, Amherst. 360

301. A. M. Kappel. *Implementation of Dynamic Algebras with an Application to Prolog.* Diplom thesis, Computer Science Dept., Universität Dortmund, Germany, 1990.

     This Diplom thesis was triggered by Börger's lectures on ASM models for Prolog [71, 72], delivered in June 1989 to A. B. C. Cremers' and H. Ganzinger's "Diplomanden-und Doktorandenseminar" at the University of Dortmund. Kappel defines a language for the specification of sequential ASMs and designs an abstract target machine (namely a Prolog program) for executing a class of sequential ASMs, including those of the ASM models for Prolog in [71, 72]. A prototype of the compiler has been implemented in Prolog, all the examples have been tested for Quintus Prolog on a SPARC station 1+ and for LPA Prolog on an IBM PC AT. A short version of the paper appeared in [302]; a parallel extension of the interpreter appears in [148]. 340, 347, 357, 362, 392, 413

302. A. M. Kappel. Executable specifications based on dynamic algebras. In A. Voronkov (ed.), *Logic Programming and Automated Reasoning*, Lecture Notes in Artificial Intelligence, Vol. 698, pp. 229–240. Springer-Verlag, 1993.

     Short version of [301]. 347, 413

303. C. Kern and M. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. of Electron. Syst.*, 4:123–193, 1999. 299

304. C. M. R. Kintala, Kong-Yee Pun, and D. Wotschke. Concise representations of regular languages by degree and probabilistic finite automata. *Math. Systems Theory*, 26(4):379–395, 1993. 41

305. E. Kohlbrenner, D. Morris, and B. Morris. The history of virtual machines. Web pages at http://cne.gmu.edu/itcore/virtualmachine/history.htm. 8, 297, 353

306. A. N. Kolmogorov and V. A. Uspenskii. On the definition of an algorithm. *AMS Translations, 2nd Series*, 29:217–245, 1993. 344

307. T. Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, 1999. 299

308. P. Kutter. An ASM macro language for sets. TIK-Report 34, ETH Zürich, Switzerland, January 1998.

A small set of simple, generic macros that allow one to manipulate and parametrize sets in ASMs, without changing the semantics given in [248].

309. P. Kutter. *Montages – Engineering of Computer Languages*. PhD thesis, ETH Zürich, Switzerland, 2002.

Contains a denotational semantics of XASM [15] (announced on June 5, 2002 as TIK Report No. 136 under the title “The formal definition of Anlauff’s eXtensible Abstract State Machines”), which is an ASM semantics of Montages, an example language illustrating the description of language features found in sequential Java (see [421]). 364, 428

310. P. Kutter and A. Pierantonio. The formal specification of Oberon. *J. Universal Computer Science*, 3(5):443–503, 1997.

A presentation of the syntax, static semantics and dynamic semantics of Oberon, using ASMs and Montages [311]. The dynamic semantics previously appeared as P. Kutter, “Dynamic Semantics of the Oberon Programming Language”, TIK-Report No. 25, ETH Zürich, Feburary 1997. 88, 358, 364, 380

311. P. Kutter and A. Pierantonio. Montages: Specifications of realistic programming languages. *J. Universal Computer Science*, 3(5):416–442, 1997.

The authors introduce Montages, a version of ASMs specifically tailored for specifying the static and dynamic semantics of programming languages. Montages combine graphical and textual elements to yield specifications similar in structure, length and complexity to those in common language manuals, but with a formal semantics. A preliminary version appeared in July 1996 under the title *Montages: Unified Static and Dynamic Semantics of Programming Languages* as Technical Report 118 of Universita de L’Aquila. At that same university also the first application of Montages appeared in a Tesi di Laurea [181]. See [19] for an extension of Montages with a finite-state machine model. 358, 370, 371, 380, 400, 404, 411, 414, 428

312. P. Kutter, D. Schweizer, and L. Thiele. Integrating domain specific language design in the software life cycle. In *Proc. Int. Workshop on Current Trends in Applied Formal Methods*, Lecture Notes in Computer Science, Vol. 1641, pp. 196–212. Springer-Verlag, 1998.

A report on an industrial case study, applying ASMs and Montages [311] to the design, specification and implementation of a driver specification language needed in the context of a complex data warehouse problem at Union Bank of Switzerland. 88, 358

313. K. Kwon. A structured presentation of a closure-based compilation method for a scoping notion in logic programming. *J. Universal Computer Science*, 3(4):341–376, 1997.

An extension to logic programming which permits scoping of procedure definitions is described at a high level of abstraction (using ASMs) and refined (in a provably-correct manner) to a lower level, building upon the method developed in [132]. The PhD thesis upon which this paper is based was submitted to Duke University on December 12, 1994, under the title "Towards a Verified Abstract Machine for a Logic Programmming Language with a Notion of Scope", No. CS 1994-36, pp. 189. 347, 380, 389

314. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

Definition of the bakery algorithm to solve the mutual exclusion problem; see also [315]. An ASM analysis of this algorithm appears in [114]. 260, 351, 415

315. L. Lamport. On interprocess communication. Part I: Basic formalism. Part II: Algorithms. *Distributed Computing*, 1:77–101, 1986.

See [314]. 260, 271, 351, 415

316. B. W. Lampson. Principles of computer systems. MIT Lecture Notes 6.826 and `http://research.microsoft.com/Lampson`, Spring 1999. 4, 116, 160, 296

317. P. J. Landin. A λ-calculus approach. In L. Fox (ed.), *Advances in Programming and Non-Numerical Computation*, pp. 97–141. Pergamon Press London, 1966.

See also the paper *A formal description of Algol 60* by the same author in T. B. Steel (ed.). *Formal Language Description Languages for Computer Programming*, North-Holland, Amsterdam, 1966. 301

318. H. Langmaack. The ProCoS approach to correct systems. *Real-Time Systems*, 13:253–275, 1997. 350

319. L. Lavagno, A. Sangiovanni-Vincentelli, and E. M. Sentovitch. Models of computation for system design. In E. Börger (ed.), *Architecture Design and Validation Methods*, pp. 243–295. Springer-Verlag, 2000. 287

320. N. G. Leveson. Completeness in formal specification language design for process-control systems. In *Formal Methods in Software Practice*, pp. 75–87. ACM Press, 2000. 20, 287

321. N. G. Leveson and J. D. Reese. SpecTRM: A toolset to support the safeware methodology. In *Proc. 16th Int. System Safety Conf.*, pp. 256–262, 1998. 287

322. N. G. Leveson, J. D. Reese, S. Koga, L. D. Pinnel, and S. D. Sandys. Analyzing requirements specifications for mode confusion errors. In *Proc. Workshop on Human Error and System Development*, Glasgow, Scotland, 20–22 March 1997. 8

323. T. Lindner. Task description. In C. Lewerentz and T. Lindner (eds.), *Formal Development of Reactive Systems. Case Study "Production Cell"*, Lecture Notes in Computer Science, Vol. 891, pp. 9–21. Springer-Verlag, 1995.

Description of the Production Cell case study which has been derived from a metal-processing plant in Karlsruhe. The book contains solutions of the problem which use various formal methods. The book inspired work on an ASM solution of the problem; see [120]. 187, 188, 189, 190, 193, 197, 355

324. A. Lisitsa and G. Osipov. Evolving algebras and labelled deductive systems for the semantic network based reasoning. In *Proc. Workshop on Applied Semiotics, ECAI'96*, pp. 5–12, August 1996.

ASMs are used to present the high-level semantics for MIR, an AI semantic network system. Another formalization of MIR is given in terms of labeled deduction systems, and the two formalizations are compared.

325. B. H. Lisko and S. N. Zilles. Specification techniques for data abstraction. *IEEE Trans. Software Eng.*, SE-1, March 1975. 8, 353, 419

326. A. Lötzbeyer. Simulation of a steam boiler. In J.-R. Abrial, E. Börger, and H. Langmaack (eds.), *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, Lecture Notes in Computer Science, Vol. 1165, pp. 493–499. Springer-Verlag, 1996. 355

327. M. Maia and R. Bigonha. An ASM-Based Approach for Mobile Systems. Technical Report LLP-12/99, Programming Language Laboratory, Computer Science Department, Universidade Federal de Minas Gerais, Brasil, 1999.

     Using the Interacting ASM techniques introduced in [328], the authors describe the use of ASMs to specify the semantics of active mobile objects. Mobility is expressed by dynamic changes in the communication topology. An earlier version appears as Technical Report LP 002/99 (same institution). 416

328. M. Maia, V. Iorio, and R. Bigonha. Interacting Abstract State Machines. In U. Glässer and P. Schmitt (eds.), *Proc. 5th Int. Workshop on Abstract State Machines*, pp. 37–49. Magdeburg University, 1998.

     An extended abstract describing an extension to ASMs supporting the interaction of independent ASM agents by means of message passing. The full version appears as M. Maia and R. Bigonha, *Formal Semantics for Interactive Abstract State Machine Language*, Technical Report RT 005/98, Universidade Federal de Minas Gerais, Brazil, 1998. Continued in [327]. 352, 360, 403, 416

329. K. Mani Chandy and J. Misra. *Parallel Program Design. A Foundation.* Addison Wesley, 1988. 293

330. W. May. Specifying complex and structured systems with evolving algebras. In *TAPSOFT'97: Theory and Practice of Software Development, 7th Int. Joint Conf. CAAP/FASE*, Lecture Notes in Computer Science, Vol. 1214, pp. 535–549. Springer-Verlag, 1997.

     An approach is presented for specifying structured systems with ASMs by means of aggregation and composition. An earlier version appeared under the title "Modeling Rule-Based and Structured Systems with Evolving Algebras" as Technical Report, Freiburg, 1996. For some of the structuring concepts defined here, simpler definitions are given in [134] which are geared to their natural integration into the basic parallelism of multiple simultaneous machine actions of ASMs. 364

331. P. J. McCann and G.-C. Roman. Programming abstractions for mobile computing. *IEEE Trans. Software Eng.*, 24(2):97–110, 1998. 293

332. L. Mearelli. Refining an ASM specification of the production cell to C++ code. *J. Universal Computer Science*, 3(5):666–688, 1997.

     Source code for the ASM specification of the Production Cell described in [120]. For the generation of this code see [391]. 188, 193, 340, 355, 380, 387

333. B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992. 17, 341

334. J. M.Morris. A theoretical basis for stepwise refinement. *Science of Computer Programming*, 9(3), 1987. 22, 134

335. M. Mohnen. A compiler correctness proof for the static link technique by means of evolving algebras. *Fundamenta Informatica*, 29(3):257–303, 1997.

     The static link technique is a common method used by stack-based implementations of imperative programming languages. The author uses ASMs to prove the correctness of this well-known technique in a non-trivial subset of Pascal. 358

336. E. F. Moore. The shortest path through a maze. In *Proc. Int. Sympos. on Theory of Switching*, The Annals of the Computation Laboratory of Harvard University, Vol. 30.II. Harvard University Press, 1959. 122

337. C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 1990,^{2}1994. 22, 134

338. J. Morris. *Algebraic Operational Semantics and Modula-2.* PhD thesis, University of Michigan, Ann Arbor, Michigan, 1988.

Thesis supervised by Gurevich. The earliest ASM formalization of a programming language. The semantical description is parse-tree directed, but flat. An extended abstract appeared as Y. Gurevich and J. Morris, “Algebraic Operational Semantics and Modula-2”, in E. Börger, H. Kleine Büning and M. M. Richter, eds., *CSL'87, 1st Workshop on Computer Science Logic*, Lecture Notes in Computer Science, Vol. 329, pp. 81–101, Springer-Verlag, 1988. 345, 347

339. J. Morris and G. Pottinger. Ada-Ariel semantics. Odyssey Research Associates, unpublished manuscript, July 1990.

340. Y. N. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid (eds.), *Mathematics Unlimited – 2001 and beyond*, pp. 919–936. Springer-Verlag, 2001. 171, 376, 381

341. L. S. Moss and D. E. Johnson. Dynamic interpretations of constraint-based grammar formalisms. *J. Logic, Language, and Information*, 4(1):61–79, 1995.

Extends the work of [297] to grammar formalisms based on Kasper–Rounds logics. See [342]. 360, 413

342. L. S. Moss and D. E. Johnson. Evolving algebras and mathematical models of language. In L. Polos and M. Masuch (eds.), *Applied Logic: How, What, and Why*, Synthese Library, Vol. 626, pp. 143–175. Kluwer Academic Publishers, 1995.

Extends the work of [297] to several other grammar formalisms. 360, 413, 417

343. P. D. Mosses. *Action Semantics.* Cambridge University Press, Cambridge, 1992.

344. W. Mueller, R. Dömer, and A. Gerstlauer. The formal execution semantics of SpecC. Technical Report TR ICS 01-59, Center for Embedded Computer Systems at the University of California at Irvine, 2001.

Adapting the async ASM model of VHDL in [111, 112] and the work in [345], an async ASM model for the semantics of SpecC is developed which covers the execution of SpecC behaviors and their interaction with the simulation kernel. This includes wait, waitfor, par, pipe, and try statements. 45, 350, 359

345. W. Mueller, J. Ruf, D. W. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of SystemC. In *Proc. Design Automation and Test in Europe (DATE 2001)*, pp. 64–70, IEEE CS Press, March 2001.

Adapting the distributed ASM model of VHDL in [111, 112], a distributed ASM model for the semantics of SystemC is developed which covers method, thread, clocked thread behavior, and their interaction with the simulation kernel. Watching statements, signal assignment and wait statements are formalized for version V1.0 of SystemC. An extended version will appear in [346]. 350, 359, 417, 418

346. W. Mueller, J. Ruf, and W. Rosenstiel. An ASM-based semantics of systemC simulation. In W. Mueller, J. Ruf, and W. Rosenstiel (eds.), *SystemC - Methodologies and Applications*. Kluwer Academic Publishers, 2003.

See [345]. 417

347. B. Müller. *Eine objektorientierte Prolog-Erweiterung zur Entwicklung wissensbasierter Systeme*. PhD thesis, University of Oldenburg, Germany, 1994.

Thesis supervised by Appelrath and Börger. Defines an object-oriented extension of Prolog to be applied for the development of knowledge based systems. The semantics is defined (in Chap. 5) as an extension of Börger's Prolog model [75]. 346, 364, 418

348. B. Müller. A semantics for hybrid object-oriented Prolog systems. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 428–433, Elsevier, Amsterdam, 1994.

On Börger's suggestion this work extends the rules given in [75] for the user-defined core of Prolog to define the semantics of a hybrid object-oriented Prolog system. The definition covers the central object-oriented features of object creation and deletion, data encapsulation, inheritance, messages, polymorphism and dynamic binding. See [347]. 346, 364, 419

349. W. Müller. *Executable Graphics for VHDL-Based Systems Design*. PhD thesis, University of Paderborn, Germany, 1996.

Uses ASMs to define the behavioral semantics of PHDL, a pictorial extension of VHDL'93. The ASMs for VHDL defined in [111, 112] are reused. 350, 359

350. M. Müller-Olm. *Modular Compiler Verification. A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, Lecture Notes in Computer Science, Vol. 1283. Springer-Verlag, 1997.

The author's PhD thesis. The considered language is a sublanguage of Occam with real-time features. See also the PROCOS II Esprit Basic Research 7071 Report MMO 12/3 (1996), University of Kiel: Structuring Code Generator Correctness Proofs by Stepwise Abstracting the Machine Language's Semantics. 350

351. Z. Németh. Definition of a parallel execution model with Abstract State Machines. *Acta Cybernetica*, 15(3):417–455, 2002.

Two ASMs are defined and related by a refinement correctness proof, as preparation for designing and verifying a distributed parallel Prolog execution model. 347

352. Z. Németh and V. Sunderam. A formal framework for defining grid systems. In *Proc. Int. Sympos. on Cluster Computing and the Grid (CCGrid2002)*, pp. 202–211, Berlin, 21–24 May 2002. IEEE Computer Society Press.

ASMs are used to define a model for grid systems. 360

353. M. Nicolosi-Asmundo and E. Riccobene. Consistent integration for sequential Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 324–340. Springer-Verlag, 2003.

Two operations to compose basic ASMs are defined and illustrated by two case studies. 188

354. A. Nowack. Deciding the verification problem for Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 341–355. Springer-Verlag, 2003. 362

355. I. Ober. More meaningful UML models. In *Proc. TOOLS*, pp. 146–157, Sydney, Australia, 20–23 November 2000. IEEE Computer Society Press.

ASMs are used to define an executable semantics for UML which covers real-time aspects. The work is inspired by the ASM model for SDL in [222] and uses the ASM Workbench [170]. 341, 364, 394

356. I. Ober. An ASM semantics for UML derived from the meta-model and incorporating actions. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 356–371. Springer-Verlag, 2003. 364

357. M. Odersky. Programming with variable functions. In *ICFP'98, Proc. 3rd ACM SIGPLAN Int. Conf. on Functional Programming*, ACM SIGPLAN Notices, Vol. 34 (1), pp. 105–116, January 1999.

The use of "variable functions" (functions which can be updated at specified points in their domains) is proposed as a method for deriving efficient imperative programs from functional programs. The notion of a variable function is drawn from the dynamic functions of ASMs.

358. C. Pahl. Towards an action refinement calculus for Abstract State Machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines – ASM 2000, Int. Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings*, TIK-Report, No. 87, pp. 326–340. Swiss Federal Institute of Technology (ETH) Zurich, March 2000.

A refinement calculus for (a reformulation of) ASMs is presented.

359. C. Pair. Types Abstraits et Sémantique Algébrique des Langages de Programmation. Technical Report TR 80-R-011, Centre de Recherche en Informatique de Nancy, France, 1980.

Often refered to as the first publication where the idea is formulated that the most general notion of state of computing systems is Tarski's notion of structures. See [210]. The similarity between data types and algebras is however already observed in [325, 265]. 8, 353, 401

360. D. L. Parnas and J. Madey. Functional documents for computer systems. *Sci. of Comput. Program.*, 25:41–62, 1995. 198, 294

361. B. Pehrson and I. Simon. I: Technology/foundations. In *IFIP 13th World Computer Congress 94*, Elsevier, Amsterdam, 1994.

Stream C (Evolving Algebras) (pp. 377–441), organized by Gurevich, contains short versions of the talks presented to the first international ASM workshop; see [39, 56, 76, 97, 107, 118, 228, 246, 348, 365, 376]. 348, 361

362. C. N. Plonka. *Model Checking for the Design with Abstract State Machines.* Diplom thesis, CS Department of University of Ulm, Germany, January 2000.

A feasibility study, carried out upon Börger's suggestion at Siemens Research, of model checking ASMs for two industrial case studies: the Production Cell [120] and a statistical multiplexing unit. An error was detected in [120] concerning a refinement step for the deposit belt, due to an erroneous (easily repaired) symmetry assumption made during the specification for the unloading actions of feedbelt, press and deposit belt. Due to additional scheduling assumptions, made for the model checking of the Production Cell ASM in [424] to guarantee maximal performance of the model, the mistake had remained undiscovered there. 188, 195, 355, 387, 428

363. A. Poetzsch-Heffter. Interprocedural data flow analysis based on temporal specifications. Technical Report 93-1397, Cornell University, Ithaca, New York, 1993.

Investigates the specification of data flow problems by temporal logic formulas and proves fixpoint analyses correct. Temporal formulas are interpreted with respect to programming language semantics given in the framework of ASMs. 365

364. A. Poetzsch-Heffter. Comparing action semantics and evolving algebra based specifications with respect to applications. In *Proc. 1st Int. Workshop on Action Semantics*, pp. 43–47, 1994.

Action semantics is compared to ASM based language specifications. In particular, different aspects relevant to language documentation and programming tool development are discussed.

365. A. Poetzsch-Heffter. Deriving partial correctness logics from evolving algebras. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 434–439, Elsevier, Amsterdam, 1994.

A proposal for deriving partial correctness logics from simple ASM models of programming languages. A basic axiom (schema) is derived from an ASM and is used to obtain more convenient logics. See [405]. 323, 365, 419, 425

366. A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34:737–772, 1997.

A tool supporting the generation of language-specific software from specifications is presented, enabling in particular the generation and refinement of interpreters based on formal language specifications. Static semantics is defined by an attribution technique (e.g. for the specification of flow graphs). The dynamic semantics is defined by ASMs. As an example, an object-oriented programming language with parallelism is specified. Part of this work has appeared as Report TR 93-1396, Cornell University and in 1994 as *Developing Efficient Interpreters based on Formal Language Specifications* in P. Fritzson (ed.): Compiler Construction, Lecture Notes in Computer Science, Vol. 786, pp. 233–247, Springer-Verlag, 1994. 363, 364

367. A. Prinz. *Formal Semantics for SDL. Definition and Implementation.* Habilitationsschrift, Humboldt University of Berlin, Germany, 2000.

Contains a complete definition and implementation of the static and dynamic semantics of a characteristic sublanguage of SDL. 358, 403

368. A. Prinz and B. Thalheim. Operational semantics of transactions. In X. Zhou and K.-D. Schewe (eds.), *Proc. 14th Australian Database Conf. (ADC2003)*, Australian Computer Science Commun., Vol. 25(2), pp. 169–179. Australian Computer Society, 2003.

Defines an ASM model for database transactions which is instantiated for the in-private and the in-place setting and used to explain the constraint enforcement used in SQL'99. 178, 360

369. C. Pusch. Verification of compiler correctness for the WAM. In J. von Wright, J. Grundy, and J. Harrison (eds.), *Theorem Proving in Higher Order Logics (TPHOLs'96)*, Lecture Notes in Computer Science, Vol. 1125, pp. 347–362. Springer-Verlag, 1996.

See comment to [132]. 300, 357, 389, 424

370. H. Reichel. Unifying ADT and evolving algebra specifications. *Bull. EATCS*, 59:112–126, 1996.

Di-algebras, a notion unifying algebras and co-algebras, are used to combine algebraic specifications of abstract data types with ASMs. A characterization of ASMs as terminally constraint Di-algebras is introduced to justify the co-induction proof principle for ASMs. Also a Di-algebra thesis is stated as the algebraic counterpart of the ASM thesis. 365

371. W. Reisig. Petri nets in software engineering. In W. Brauer, W. Reisig, and G. Rozenberg (eds.), *Petri Nets: Applications and Relationships to other Models of Concurrency*, Lecture Notes in Computer Science, Vol. 255, pp. 63–96. Springer-Verlag, 1987. 62

372. W. Reisig. *Elements of Distributed Algorithms.* Springer-Verlag, 1998. 210, 216, 241, 271, 297

373. G. R. Renardel de Lavalette. A logic of modification and creation. In C. Condoravdi and G. R. Renardel de Lavalette (eds.), *Logical Perspectives on Language and Information.* CSLI publications, Stanford, CA, 2001. 328

374. E. Riccobene. A formal computational model for PANDORA. Technical Report CSTR-92-16 and ACRC-92-15, University of Bristol, Department of Computer Science, 1992.

    The ASM model for Parlog developed in [123] is extended by the don't-know non-determinism of Pandora. 282, 348, 388, 421

375. E. Riccobene. *Modelli Matematici per Linguaggi Logici.* PhD thesis, University of Catania, Sicily, Italy, Academic year 1991/92.

    Systematic treatment of ASM models for Gödel [124], Parlog [123], Pandora [374], Concurrent Prolog [122], GHC. Thesis supervised by Börger. 282, 348

376. D. Rosenzweig. Distributed computations: Evolving algebra approach. In B. Pehrson and I. Simon (eds.), *IFIP 13th World Computer Congress*, Vol. I: Technology/Foundations, pp. 440–441, Elsevier, Amsterdam, 1994.

    Remarks on some ASM models of concurrent and parallel computation. 419

377. D. Rosenzweig, D. Runje, and N. Slani. Privacy, abstract encryption and protocols: an ASM model–part I. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 372–390. Springer-Verlag, 2003.

    Provides an AsmL executable model of abstract encryption. 361

378. H. Rust. Hybrid Abstract State Machines: Using the hyperreals for describing continuous changes in a discrete notation. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings*, TIK-Report, No. 87, pp. 341–356. ETH Zürich, March 2000.

    A hybrid version of ASMs, incorporating the hyperreals for continuously changing quantities, is described.

379. H. Rust. *A non-standard approach to operational semantics for timed systems.* Habilitation thesis, BTU Cottbus, Germany, 2002.

    Time moments are defined as multiples of some arbitrary, but fixed infinitesimal, allowing us to model real-time system behavior with infinitesimal exactness and refinements of actions. ASMs are used to model hybrid systems, adding interleaving to the synchronous parallel execution model. 31, 180, 198

380. H. Sasaki. A formal semantics for Verilog-VHDL simulation interoperability by Abstract State Machines. In *Proc. IEEE Conf. DATE'99 (Design, Automation and Test in Europe), ICM Munich, Germany*, pp. 353–357, 9–12 March 1999.

Based upon the VHDL models developed in [111, 112], a formal semantics for Verilog-HDL and VHDL focusing on the simulation model (with signal scheduling and time control) is defined. The semantics presented is faithful to the language reference manual and is proposed as a first step towards semantic interoperability analysis on multi-semantic domains such as Verilog-AMS and VHDL-AMS. Extended in [381]. 350, 359, 385, 422

381. H. Sasaki. A formal semantics on net delay in Verilog-HDL. In *Proc. Asia Pacific Conf. on Chip Design Languages (APCHDL'99)*, pp. 100–106, Fukuoka, Japan, 6–8 October 1999.

An extension of [380] giving semantics for net delays in Verilog-HDL using ASMs. 350, 359, 422

382. H. Sasaki. A new dynamic equation scheduling to extend VHDL-AMS. In *Proc. Asia Pacific Conf. on Chip Design Languages (APCHDL'99)*, pp. 47–52, Fukuoka, Japan, 6–8 October 1999.

An extension to VHDL-AMS for dynamic equation scheduling is proposed. The semantics of the extension is given in terms of the ASM model for VHDL-AMS presented in [383]. 350, 359, 422

383. H. Sasaki, K. Mizushima, and T. Sasaki. Semantic validation of VHDL-AMS by an Abstract State Machine. In *Proc. BMAS'97 (IEEE/VIUF Int. Workshop on Behavioral Modeling and Simulation)*, pp. 61–68, Arlington, VA, 20–21 October 1997.

An extension of the ASM model defined for VHDL in [111, 112] to provide a rigorous definition of VHDL-AMS, following the IEEE Language Reference Manual for the analog extension of VHDL. For an extension see [384]. See also [381, 382, 380]. 350, 359, 385, 422

384. T. Sasaki, H. Sasaki, and K. Mizushima. Semantic analysis of VHDL-ASM by attribute grammar. In *Proc. FDL'98 (Forum on Design Languages), Lausanne, Switzerland*, pp. 123–131, 6–10 September 1998.

An extension of [383] to provide a formal semantics of the VHDL Analog Mixed Signal extension by means of attribute grammars. The formulation treats both the static and the dynamic aspects of semantics and permits one to show the equality of process behavior. 350, 359, 422

385. J. Sauer. *Wissensbasiertes Lösen von Ablaufsplanungsproblemen durch explizite Heuristiken*. PhD thesis, Universität Oldenburg, Germany, 1993.

Published in: Dissertationen zur Künstlichen Intelligenz, Vol. 37, Infix-Verlag, Dr. Ekkehardt Hundt, St. Augustin, 1993. Uses ASMs to define the semantics for the HERA language (and its implementation in Prolog), a special-purpose programming language for the representation and manipulation of scheduling knowledge on the basis of heuristics, tailored to program efficient and reusable scheduling algorithms for production planning and control. See also J. Sauer, "Evolving Algebras for the Description of a Meta-Scheduling System", in H. Kleine Büning, ed., *Workshop der GI-Fachgruppe Logik in der Informatik*, Technical Report TR-RI-94-146, Universität Paderborn, 1994. 88, 346

386. G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Germany, 1999.

ASMs are embedded into dynamic logic. Two refinement notions are extracted from typical ASM refinements and formalized in dynamic logic. A general modularisation theorem is proved for schemes to prove the correctness of refinements. An improved version of this theorem appears in [387]. The KIV system is enhanced to apply those proof techniques for a KIV verification of the WAM correctness proof in [132]. An English version of the thesis is available at Schellhorn's web site. 300, 357, 423, 424

387. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.

    See [386]. 112, 133, 134, 156, 357, 378, 381, 384, 423, 424

388. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.

    The Karlsruhe Interactive Verifier (KIV system) is applied to mechanically verify the proof of correctness of the Prolog to WAM transformation described in [132]. The starting point was the Diplom Thesis *Von Prolog zur WAM. Verifikation der Prozedurübersetzung mit KIV* of W. Ahrendt, Universität Karlsruhe (Germany) 1995. See comment to [132] and [389, 386]. 134, 300, 357, 380, 389, 423, 424

389. G. Schellhorn and W. Ahrendt. The WAM case study: Verifying compiler correctness for Prolog with KIV. In W. Bibel and P. Schmitt (eds.), *Automated Deduction – A Basis for Applications*, Vol. III: Applications, pp. 165–194. Kluwer Academic Publishers, 1998.

    Continuation of [388]. 134, 300, 423

390. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at `http://www.tydo.de/AsmGofer`.

    The web site for the machine to execute, equipped with graphical user interface, ASMs which are enhanced with the structuring and composition concepts defined in [134]. AsmGofer executes the Light Control ASM `http://www.tydo.de/AsmGofer/light` defined in [125] and all the ASMs defined in [406]. In [152] AsmGofer has been used to build a simulator for UML state diagrams. 173, 185, 230, 240, 341, 359, 364, 388, 390, 393

391. J. Schmid. Compiling Abstract State Machines to C++. *J. Universal Computer Science*, 7(11):1069–1088, 2001.

    Introduces a scheme for compiling ASMs from the syntax of the ASM Workbench [170] to C++, coding algebraic types, pattern matching, functional expressions, dynamic functions and simultaneous updates in such a way that efficient C++ code is obtained without losing the structure of the original ASM specification. The compiler has been successfully applied in the FALKO project at Siemens [121]. In an early application C++ code was generated from a translation of the Production Cell ASM in [120] to the ASM Workbench format ASM-SL [170]. An HTML version is available at `http://www.tydo.de/ProductionCell/`. 193, 341, 355, 357, 384, 416, 423

392. J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, University of Ulm, Germany, 2002.

    Thesis supervised by Börger and located at Siemens Corporate Research in München from August 1998 to July 2000. The thesis enriches ASMs by structuring and composition concepts [134] and their implementation in the AsmGofer system, developed for executing ASMs in an environment with a graphical user interface. The concepts have been successfully applied in a middle-sized software development project at Siemens [121, 391], in the Light

Control Case Study [125], in an industrial ASIC design and verification project (including a compiler from ASM to VHDL), and for the modeling and implementation of Java and the JVM in [406]. Electronic version available at `http://www.tydo.org/files/papers/dissJS.pdf`. 341, 357, 359

393. P. Schmitt. Proving WAM compiler correctness. Technical Report 33/94, Universität Karlsruhe, Fakultät für Informatik, Germany, 1994.

Feasibility analysis of Börger's proposal to the DFG project "Deduktion" to mechanize the Prolog-to-WAM compiler correctness proof in [132]. See [388, 387, 386, 369]. 357

394. A. Schönegge. Extending dynamic logic for reasoning about evolving algebras. Technical Report 49/95, Universität Karlsruhe, Fakultät für Informatik, Germany, 1995.

EDL, an extension of dynamic logic, is presented, which permits one to directly represent statements about ASMs. Such a logic lays the foundation for extending KIV (Karlsruhe Interactive Verifier) to reason about ASMs directly. See [405]. 322, 365, 425

395. W. Schönfeld. Interacting Abstract State Machines. In U. Glässer and P. Schmitt (eds.), *Proc. 5th Int. Workshop on Abstract State Machines*, pp. 22–36. Magdeburg University, 1998.

An extension to ASMs which permits one to specify forced synchronization of agent moves (à la Petri nets) is proposed and explored on some examples. 360, 403

396. A. Schönhage. Storage modification machines. *SIAM J. Comp.*, 9:490–508, 1980.

Shown in [177] to be equivalent to a class of unary sequential ASMs. 344

397. M. Schrefl and G. Kappel. Cooperation contracts. In T. J. Teorey (ed.), *Proc. 10th Int. Conf. on the Entity Relationship Approach (ER'91)*, pp. 285–307, San Mateo, California, 23–25 October 1991. Entity Relationship Institute.

The authors introduce the concept of *cooperative* message handling where multiple objects can establish cooperation contracts governing their answers to jointly received messages. An ASM rule is defined (Fig. 9, p. 304) to formalize the run-time search of the most specific cooperation contract which implements a cooperative message. See [232]. 347, 404

398. D. Scott. Definitional suggestions for automata theory. *J. Computer and System Sciences*, 1:187–212, 1967. 290

399. M. Shaw and D. Garlan. Formulations and formalisms in software architecture. In J. van Leeuwen (ed.), *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science, Vol. 1000. Springer-Verlag, 1995. 5

400. I. Soloviev. *Exploration and Experimental Implementation of Recursive Patterns and Functions Embedding Into Prolog Language Syntactical Environment.* PhD thesis, St. Petersburg University, Russia, 1995.

In Russian. A functional extension of Prolog with a specialized unification algorithm is proposed. ASMs are used to define the operational semantics of the language.

401. M. Spielmann. Automatic verification of Abstract State Machines. In N. Halbwachs and D. A. Peled (eds.), *Proc. 11th Int. Conf. on Computer-Aided Verification (CAV '99)*, Lecture Notes in Computer Science, Vol. 1633, pp. 431–442. Springer-Verlag, 1999.

A class of restricted ASM programs is introduced, along with a PSPACE algorithm for verifying the correctness of certain CTL*-like temporal-logic properties of such programs. The limits on the verifiability of generalizations of this class are discussed. 362, 425

402. M. Spielmann. *Abstract State Machines: Verification Problems and Complexity*. PhD thesis, University of Aachen, Germany, 2000.

Investigation of the complexity of decision problems for certain classes of ASMs. Most of the results appear in [401, 404, 403]. The second part of the thesis relates to the work in [62]. A restricted ASM model to capture log-space computable functions on structures is defined; see also[235]. 361, 405

403. M. Spielmann. Model checking Abstract State Machines and beyond. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 323–340. Springer-Verlag, 2000.

Decision problems for ASMs are investigated, i.e. problems to decide, for an ASM M of a given class and for a property P of a given form, whether M satisfies P. For particular classes of machines and of property describing formulae, the computational complexity of such problems is studied for the following two cases: (a) given M, P and input I, decide whether P holds during all M-computations over I (called model-checking problem); (b) given M and P, decide whether for every input I, P holds during all M-computations over I (called the verification problem). Appeared also as TIK-Report No. 87, pp. 357–375, ETH Zürich, March 2000. 362, 425

404. M. Spielmann. Verification of relational transducers for electronic commerce. In *Proc. 19th ACM Sympos. Principles of Database Systems (PODS 2000)*, pp. 92–103, Dallas, Texas, 2000. ACM Press.

An investigation into decision problems for certain transaction protocols specifying the interaction of multiple parties, each equipped with an active database. Inspired by the relational transducers in [1], ASM-transducers are defined and shown to have various solvable decision problems. 360, 369, 425

405. R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. Universal Computer Science*, 7(11):981–1006, 2001.

A new logic for sequential, non-distributed ASMs is presented which is based on an atomic predicate for function updates and on a definedness predicate for the termination of the evaluation of ASM rules. The logic allows for sequential and hierarchical recursive submachine composition as defined in [134]. It is proven complete for hierarchical non-recursive ASMs. This logic provides a unifying view of the logics for ASMs developed in [238, 394, 365, 207]. A preliminary version appeared in L. Fribourg (ed.): Computer Science Logic (CSL 2001), Lecture Notes in Computer Science, Vol. 2142, pp. 217–231, Springer-Verlag, 2001. 185, 313, 328, 365, 384, 405, 420, 424

406. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

A high-level description, together with a mathematical and an experimental analysis (verification and validation), of Java and of the Java Virtual Machine (JVM), including a standard compiler of Java programs to JVM code and the security-critical bytecode verifier component of the JVM. Includes an executable ASM specification written for AsmGofer. For an evaluation see [272, Sect. 6.2]; for more information see `http://www.inf.ethz.ch/~jbook/`. 2, 10, 23, 42, 48, 49, 60, 87, 88, 109, 112, 156, 163, 169, 193, 298, 324, 325, 341, 359, 364, 366, 378, 380, 381, 389, 390, 391, 409, 423, 424, 428

407. M. M. Stegmüller. *Formale Verifikation des DLX RISC-Prozessors: Eine Fallstudie basierend auf abstrakten Zustandsmaschinen.* Diplom thesis, University of Ulm, Germany, 1998.

PVS is used for the formal verification of the parallelization of the ASM specification for the serial DLX architecture, the first step of the refinement to its parallel version with five-stage pipelining in [119]. 137, 148, 156, 359, 386

408. K. Stenzel. Verification of JavaCard programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, Germany, 2001.

Available at `http://www.Informatik.Uni-Augsburg.DE/swt/fmg/papers/`. The report is about the formal verification of JavaCard or sequential Java programs (i.e. without synchronized statements). A calculus in dynamic logic is defined and implemented in KIV. KIV parses the original JavaCard or Java program, resolves names and types in the same manner as a normal Java compiler, and produces an annotated abstract syntax tree that is the input for the verification. All sequential Java statements are supported, including exceptions, breaks, static initialization, objects, dynamic method lookup and arrays. The abstract syntax of Java programs, the proof rules, and the underlying algebraic specifications for the object store and the primitive data types, and a formal semantic is described in detail. An example proof and a list of validation programs conclude the report. For information on preliminary work on formalizing ASM models for Java in KIV see `http://www.informatik.uni-augsburg.de/swt/fmg/applications/`. 364

409. K. Stroetmann. The constrained shortest path problem: A case study in using ASMs. *J. Universal Computer Science*, 3(4):304–319, 1997.

Upon Börger's suggestion, an abstract, non-deterministic form of the constrained shortest path problem is defined as an ASM and proven correct, and then refined to the level of implementation. 117, 122, 361, 380

410. A. Sünbül. *Architectural Design of Evolutionary Software Systems in Continuous Software Engineering.* PhD thesis, TU Berlin, Germany, 2001.

Taking up a suggestion in [86, Sect. 4] this dissertation develops a language for specifying software systems by linking components via connectors. Components are abstractly characterized by the services they import and export which are defined by high-level specifications (possibly depending on given views) and have to satisfy certain constraints on well-formedness and on the ordering of usage (called use structure). For connectors, which connect services required in one component to services offered by other components, a refinement concept is defined. ASM rules are provided to check the consistency of software architectures developed in that language, namely checking componentwise (a) for each imported service its correct connection to a corresponding exported service (with respect to signature and specification), (b) for each exported service that the imported services it uses satisfy the constraints of the used components, and (c) that the (optional) refinement is correct with respect to the system constituents (types, views, components, connectors). The proposed machine has been made executable in XASM. Extended abstracts of some of the ideas in the thesis have been published by M. Anlauff and A. Sünbül as *Software Architecture Based Composition of Components* (Gesellschaft für Informatik, Sicherheit und Zuverlässigkeit softwarebasierter Systeme, May 1999), *Component Based Software Engineering for Telecommunication Software* (Proc. SCI/ISAS Conf., Orlando, Florida 1999), *Domain-Specific Languages in Software Architecture* (Proc. Integrated Design and Process Technology IDPT99, June 1999). 105, 108, 360

411. J. Teich. Project Buildabong at University of Paderborn. http://www-date.upb.de/RESEARCH/BUILDABONG/buildabong.html, 2001.

The project, led by Teich at the University of Paderborn, uses ASMs to provide behavioral and structural descriptions of application-specific instruction set processors, from which (using XASM [15] and Gem-Mex [18]) bit-true and cycle-accurate simulators and debuggers are derived. See the paper "Design Space Characterization for Architecture/Compiler Co-Exploration" by D. Fischer, J. Teich, R. Weper, U. Kastens, M. Thies in: Proc. ACM Conf. CASES'01, 16–17 November, 2001, Atlanta, Georgia, USA. 137, 157, 360, 427

412. J. Teich, P. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 266–286. Springer-Verlag, 2000.

A method for transforming register transfer descriptions of microprocessors into executable ASM specifications is described and illustrated using the ASM model developed in [286] for the ARM2 RISC processor. The description exploits the natural correspondence between the simultaneous execution of all guarded update rules of an ASM and a single-clock hardware step executing a set of Leuper's guarded register transfer patterns. XASM [15] is used together with the Gem-Mex tool [18] which generates a graphical simulator for the given architecture. See also [413]. Also appears in TIK-Report No. 87, pp. 376–397, ETH Zürich, March 2000. 157, 359, 386, 411, 427

413. J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joint architecture/compiler design environment for ASIPs. In *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, pp. 26–33, San Jose, CA, USA, November 2000. ACM Press.

An ASM model is developed for a VLIW digital signal processor of the Texas Instruments TMS320 C6200 family to illustrate the Buildabong method [411]. See also [412]. 359, 427

414. H. Tonino. *A Theory of Many-sorted Evolving Algebras.* PhD thesis, Delft University of Technology, Netherlands, 1997.

Based on a two-valued many-sorted logic of partial functions (with a complete and sound Fitch-style axiomatization), a structural operational and a Hoare-style axiomatic semantics is given for many-sorted non-distributed deterministic ASMs. The SOS semantics is defined in two levels, one for the sequential and one for the parallel ASM constructs. Two (sound but not complete) Hoare-style descriptions are given, one for partial and one for total correctness. A first part appeared under the title "A Formalization of Many-sorted Evolving Algebras" as Report TR 93-115 at TU Delft. An extended abstract appeared under the title *A Sound and Complete SOS-Semantics for Non-Distributed Deterministic Abstract State Machines* in [226, pp. 91–110]. 365, 403

415. H. Tonino and J. Visser. Stepwise refinement of an Abstract State Machine for WHNF-reduction of λ-terms. Technical Report 96–154, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Netherlands, 1996.

A series of ASMs for finding the weak head normal form (WHNF) of an arbitrary term of the λ-calculus is presented.

416. K. J. Turner (ed.). *Using Formal Description Techniques. An Introduction to Estelle, LOTOS and SDL.* John Wiley, New York, 1993. 49, 50, 52

417. M. Vale. The evolving algebra semantics of COBOL. Part I: Programs and control. Technical Report CSE-TR-162-93, EECS Dept., University of Michigan, 1993.

     An ASM for the control constructs of COBOL. A description of a plan for a series of ASMs for all of COBOL is sketched (but not carried out). Missing constructs concern source text manipulations, report writer, communication, debug, and segmentation modules. 347

418. J. Visser. *Evolving Algebras*. Master's thesis, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft, Netherlands, 1996.

     The monad programming method is used to write a compiler/run-analyzer for ASMs in Gofer. Static functions can be supplied to the ASMs by means of Gofer functions.

419. F. von Henke. Putting software technology to work. In K. Duncan and K. Krueger (eds.), *IFIP 13th World Computer Congress 1994*, pp. 345–350. Elsevier, 1994. 299

420. C. Wallace. The semantics of the C++ programming language. In E. Börger (ed.), *Specification and Validation Methods*, pp. 131–164. Oxford University Press, 1995.

     The description in [251] of the semantics of C is extended to C++. 88, 347, 349, 379, 407

421. C. Wallace. The semantics of the Java programming language: Preliminary version. Technical Report CSE-TR-355-97, EECS Dept., University of Michigan, December 1997.

     A specification of the static and dynamic semantics of Java, using ASMs and Montages. This work showed the shortcomings of the original formulation of Montages [311] and led to its state machine based reformulation in [19]. See [309] and the independent earlier Java modeling work [138] which was continued in [137, 139, 140, 141] and [406]. See also [23]. 371, 391, 409, 414

422. C. Wallace, G. Tremblay, and J. N. Amaral. An Abstract State Machine specification and verification of the location consistency memory model and cache protocol. *J. Universal Computer Science*, 7(11):1089–1113, 2001.

423. P. Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40:80–91, 1997. 291

424. K. Winter. Model checking for Abstract State Machines. *J. Universal Computer Science*, 3(5):689–701, 1997.

     Inspired by Börger's lectures on ASMs in Freiburg in the Fall of 1994, Winter develops a framework for using a model checker to verify ASM specifications. It is applied to the production cell control model described in [120]. See [362] for an interesting problem with refining abstractions for model checking purposes. For an extension see [175, 425, 426, 427]. 188, 300, 355, 380, 387, 395, 419, 428, 429

425. K. Winter. Towards a methodology for model checking ASM: Lessons learned from the flash case study. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, Vol. 1912, pp. 341–360. Springer-Verlag, 2000.

     A general discussion of applying model checking to ASMs. Following a suggestion by Börger, the ASM specification of the FLASH cache coherence protocol [189] is checked using SMV as a case study. An extension of [175, 424]. Also appears in TIK-Report No. 87, pp. 398–425, ETH Zürich, March 2000. 300, 361, 395, 397, 428, 429

426. K. Winter. *Model Checking Abstract State Machines*. PhD thesis, Technical University of Berlin, Germany, 2001.

     Based upon [424, 175, 425, 427], a transformation of ASMs to FSMs and abstraction mechanisms in the context of model checking large ASMs are investigated and implemented. The underlying tools are the ASM Workbench [170], SMV and Multiway Decision Graphs (for the latter see also [427]). 338, 364, 395, 428

427. K. Winter. Model checking with abstract types. In S. D. Stoller and W. Visser (eds.), *Workshop on Software Model Checking*, Electronic Notes in Theoretical Computer Science, Vol. 55 (3), Paris, France, July 23 2001. Elsevier Science B.V.

     Investigates an interface from ASMs to Multiway Decision Graphs. See also the Report TR 01-16, Software Verification Research Center, The University of Queensland, November 2001. 428, 429

428. K. Winter. Automated checking of control tables. E-mail to E. Börger, December 24, 2001.

     Case study for automated checking of Control Tables, used by the Software Verification Research Centre at the University of Queensland, Australia, to specify railway interlocking systems. The control tables are formalized as ASMs and then transformed by the algorithm described in [175] to become input for the SMV model checker. 300, 362

429. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), 1971. 9, 21, 24

430. J. Woodcock and M. Loomes. *Software Engineering Mathematics*. Pitman, 1988. 100

431. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996. 156, 295

432. A. Zamulin. Algebraic specification of dynamic objects. In *Proc. LMO'97 (Acte du Colloque Langage et Modeles a Objets)*, pp. 111–127, Paris, 22–24 October 1997. Edition Hermes.

     A model for describing the behavior of dynamic objects is presented, using a state-transition system with the same semantics as (though not explicitly identified as) ASMs. 364

433. A. Zamulin. Specification of an Oberon compiler by means of a typed Gurevich machine. Technical Report 589.3945009.00007-01, Institute of Informatics Systems of the Siberian Division of the Russian Academy of Sciences, Novosibirsk, Russia, 1997.

     A Typed Gurevich Machine [434] is used to define a compiler for Oberon to an algebraically-specified abstract target machine. 364, 430

434. A. Zamulin. Typed Gurevich machines revisited. *Joint CS & IIS Bulletin, Computer Science*, 7:95–122, 1997.

     An approach to combining type-structured algebraic specifications and ASMs is proposed. A preliminary version appeared in 1996 as preprint 36 of the Institute of Informatics Systems, Novosibirsk. 364, 429

435. A. Zamulin. Object-oriented Abstract State Machines. In U. Glässer and P. Schmitt (eds.), *Proc. 5th Int. Workshop on Abstract State Machines*, pp. 1–21. Otto-von-Guericke-Universität Magedeburg, 1998.

     Proposes an extension of ASMs to include objects. 364, 403, 430

436. A. Zamulin. Specification of dynamic systems by typed Gurevich machines. In Z. Bubnicki and A. Grzech (eds.), *Proc. 13th Int. Conf. on System Science*, pp. 160–167, Wroclaw, Poland, 15–18 September 1998.

A combination of many-sorted algebraic specifications for states and ASM-rules for transitions is proposed as an approach for dynamic system specification. The approach is used in [433] to specify an Oberon compiler. 364, 430

437. A. Zamulin. Generic facilities in object-oriented ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines – ASM 2000, Int. Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings*, TIK-Report, No. 87, pp. 426–446. ETH Zürich, March 2000.

The object-oriented ASM framework introduced in [435] is extended to allow the definition of generic object types, type categories, functions, and procedures. Examples from the C++ Standard Template Library (STL) are provided. Previously appeared in Preprint 60, Institute of Informatics Systems, Siberian Division of the Russian Academy of Sciences, Novosibirsk, 1999. 364

438. A. Zamulin. Specifications in-the-large by typed ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines – ASM 2000, Int. Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings*, TIK-Report, No. 87, pp. 447–461. ETH Zürich, March 2000.

A discussion of combining typed ASMs (as proposed in [436]) to produce larger ASMs. 364

439. W. Zimmerman and T. Gaul. On the construction of correct compiler back-ends: An ASM approach. *J. Universal Computer Science*, 3(5):504–567, 1997.

The authors use ASMs to construct provably correct compiler back-ends based on realistic intermediate languages (and check the correctness of their proofs using PVS). 112, 300, 358, 380, 397

440. W. Zimmermann and A. Dold. A framework for modeling the semantics of expression evaluation with Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene (eds.), *Abstract State Machines 2003–Advances in Theory and Applications*, Lecture Notes in Computer Science, Vol. 2589, pp. 391–406. Springer-Verlag, 2003.

An ASM framework is defined for a uniform characterization of expression evaluation tactics used in programming languages like Fortran, ADA95, C, C++, Java, and C#. 300

List of Problems

List of Figures

List of Tables

Index