Roberto Bruni
Juergen Dingel (Eds.)

LNCS 6722

Formal Techniques for Distributed Systems

Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011
and 30th IFIP WG 6.1 International Conference, FORTE 2011
Reykjavik, Iceland, June 2011, Proceedings

ifip

Lecture Notes in Computer Science 6722

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Roberto Bruni   Juergen Dingel (Eds.)

Formal Techniques for Distributed Systems

Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011
and 30th IFIP WG 6.1 International Conference, FORTE 2011
Reykjavik, Iceland, June 6-9, 2011
Proceedings

Springer

Volume Editors

Roberto Bruni
Università di Pisa
Dipartimento di Informatica
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
E-mail: bruni@di.unipi.it

Juergen Dingel
Queen's University
School of Computing
723 Goodwin Hall, Kingston, ON, K7L 3N6, Canada
E-mail: dingel@cs.queensu.ca

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-21460-8 e-ISBN 978-3-642-21461-5
DOI 10.1007/978-3-642-21461-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011928308

CR Subject Classification (1998): D.2, D.2.4, I.2.2, D.3, F.3, F.4, I.2.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© IFIP International Federation for Information Processing 2011
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

In 2011 the 6th International Federated Conferences on Distributed Computing Techniques (DisCoTec) took place in Reykjavik, Iceland, during June 6-9. It was hosted and organized by Reykjavik University. The DisCoTec series of federated conferences, one of the major events sponsored by the International Federation for Information processing (IFIP), included three conferences: Coordination, DAIS, and FMOODS/FORTE.

DisCoTec conferences jointly cover the complete spectrum of distributed computing subjects ranging from theoretical foundations to formal specification techniques to practical considerations. The 13th International Conference on Coordination Models and Languages (Coordination) focused on the design and implementation of models that allow compositional construction of large-scale concurrent and distributed systems, including both practical and foundational models, run-time systems, and related verification and analysis techniques. The 11th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS) elicited contributions on architectures, models, technologies and platforms for large-scale and complex distributed applications and services that are related to the latest trends in bridging the physical/virtual worlds based on flexible and versatile service architectures and platforms. The 13th Formal Methods for Open Object-Based Distributed Systems and 31st Formal Techniques for Networked and Distributed Systems (FMOODS/FORTE) together emphasized distributed computing models and formal specification, testing and verification methods.

Each of the three days of the federated event began with a plenary speaker nominated by one of the conferences. On the first day, Giuseppe Castagna (CNRS, Paris 7 University, France) gave a keynote titled "On Global Types and Multi-Party Sessions." On the second day, Paulo Verissimo (University of Lisbon FCUL, Portugal) gave a keynote talk on "Resisting Intrusions Means More than Byzantine Fault Tolerance." On the final and third day, Pascal Costanza (ExaScience Lab, Intel, Belgium) presented a talk that discussed "Extreme Coordination—Challenges and Opportunities from Exascale Computing."

In addition, there was a poster session, and a session of invited talks from representatives of Icelandic industries including Ossur, CCP Games, Marorka, and GreenQloud.

There were five satellite events:

1. The 4th DisCoTec workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS)
2. The Second International Workshop on Interactions Between Computer Science and Biology (CS2BIO) with keynote lectures by Jasmin Fisher (Microsoft Research - Cambridge, UK) and Gordon Plotkin (Laboratory for Foundations of Computer Science - University of Edinburgh, UK)

3. The 4th Workshop on Interaction and Concurrency Experience (ICE) with keynote lectures by Prakash Panangaden (McGill University, Canada), Rocco de Nicola (University of Florence, Italy), and Simon Gay (University of Glasgow, UK)
4. The First Workshop on Process Algebra and Coordination (PACO) with keynote lectures by Jos Baeten (Eindhoven University of Technology, The Netherlands), Dave Clarke (Katholieke Universiteit Leuven, Belgium), Rocco De Nicola (University of Florence, Italy), and Gianluigi Zavattaro (University of Bologna, Italy)
5. The 7th International Workshop on Automated Specification and Verification of Web Systems (WWV) with a keynote lecture by Elie Najm (Telecom Paris, France)

I believe that this rich program offered each participant an interesting and stimulating event. I would like to thank the Program Committee Chairs of each conference and workshop for their effort. Moreover, organizing DisCoTec 2011 was only possible thanks to the dedicated work of the Publicity Chair Gwen Salaun (Grenoble INP - INRIA, France), the Workshop Chairs Marcello Bonsangue (University of Leiden, The Netherlands) and Immo Grabe (CWI, The Netherlands), the Poster Chair Martin Steffen (University of Oslo, Norway), the Industry Track Chairs Björn Jónsson (Reykjavik University, Iceland), and Oddur Kjartansson (Reykjavik University, Iceland), and the members of the Organizing Committee from Reykjavik University: Árni Hermann Reynisson, Steinar Hugi Sigurðarson, Georgiana Caltais Goriac, Eugen-Ioan Goriac and Ute Schiffel. To conclude I want to thank the International Federation for Information Processing (IFIP), Reykjavik University, and CCP Games Iceland for their sponsorship.

June 2011 Marjan Sirjani

Preface

This volume contains the proceedings of the FMOODS/FORTE 2011 conference, a joint conference combining the 13th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS) and the 31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE).

FMOODS/FORTE was hosted together with the 13th International Conference on Coordination Models and Languages (COORDINATION) and the 11th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS) by the federated conference event DisCoTec 2011, devoted to distributed computing techniques and sponsored by the International Federation for Information Processing (IFIP).

FMOODS/FORTE provides a forum for fundamental research on the theory and applications of distributed systems. Of particular interest are techniques and tools that advance the state of the art in the development of concurrent and distributed systems and that are drawn from a wide variety of areas including model-based design, component and object technology, type systems, formal specification and verification and formal approaches to testing. The conference encourages contributions that combine theory and practice in application areas of telecommunication services, Internet, embedded and real-time systems, networking and communication security and reliability, sensor networks, service-oriented architecture, and Web services.

The keynote speaker of FMOODS/FORTE 2011 was Giuseppe Castagna of CNRS, University Paris Diderot, who is known for his foundational work on semantic subtyping, contracts for Web services, and efficient and effective languages for the processing of XML documents. Castagna contributed a paper on global types for multi-party sessions to these proceedings.

The FMOODS/FORTE 2011 program consisted of the above invited paper and 21 regular papers which were selected by the Program Committee (PC) out of 65 submissions received from 29 different countries. Each submitted paper was evaluated on the basis of at least three detailed reviews from 28 PC members and 76 external reviewers. Additional expert reviews were solicited if the reviews of a paper had diversified or controversial assessments or if the reviewers indicated low confidence. The final decision of acceptance was preceeded by a 9-day online discussion of the PC members. The selected papers constituted a strong program of stimulating, timely, and diverse research. Papers presented techniques from formal verification (using model checking, theorem proving, and rewriting), formal modeling and specification (using process algebras and calculi, type systems, and refinement), run-time monitoring, and testing to address challenges in many different application areas including dynamic and ad hoc networks, mobile

and adaptive computation, reactive and timed systems, business processes, and distributed and concurrent systems and algorithms.

We are deeply indebted to the PC members and external reviewers for their hard and conscientious work in preparing 198 reviews. We thank Marjan Sirjani, the DisCoTec General Chair, for his support, and the FMOODS/FORTE Steering Committee for their guidance. Our gratitude goes to the authors for their support of the conference by submitting their high-quality research works. We thank the providers of the EasyChair conference tool that was a great help in organizing the submission and reviewing process.

April 2011

Roberto Bruni
Juergen Dingel

Organization

Program Committee

Saddek Bensalem	VERIMAG, France
Dirk Beyer	University of Passau, Germany
Gregor Bochmann	University of Ottawa, Canada
Roberto Bruni	Università di Pisa, Italy
Nancy Day	David R. Cheriton School of Computer Sceince, University of Waterloo, Canada
John Derrick	University of Sheffield, UK
Juergen Dingel	Queen's University, Kingston, Canada
Khaled El-Fakih	American University of Sharjah, UAE
Holger Giese	Hasso Plattner Institute, Germany
John Hatcliff	Kansas State University, USA
Valerie Issarny	INRIA, France
Claude Jard	ENS Cachan Bretagne, France
Einar Broch Johnsen	University of Oslo, Norway
Ferhat Khendek	Concordia University, Canada
Jay Ligatti	University of South Florida, USA
Luigi Logrippo	Université du Québec en Outaouais, Canada
Niels Lohmann	Universität Rostock, Germany
Fabio Massacci	University of Trento, Italy
Uwe Nestmann	Technische Universität Berlin, Germany
Peter Olveczky	University of Oslo, Norway
Alexandre Petrenko	CRIM, Canada
Frank Piessens	K.U. Leuven, Belgium
Andre Platzer	Carnegie Mellon University, USA
António Ravara	Universidade Nova de Lisboa, Portugal
Kenneth Turner	University of Stirling, UK
Keiichi Yasumoto	Nara Institute of Science and Technology, Japan
Nobuko Yoshida	Imperial College London, UK
Elena Zucca	DISI - University of Genoa, Italy

Steering Committee

Gilles Barthe	IMDEA Software, Spain
Gregor Bochmann	University of Ottawa, Canada
Frank S. de Boer	Centrum voor Wiskunde en Informatica, The Netherlands
John Derrick	University of Sheffield, UK
Khaled El-Fakih	American University of Sharjah, UAE

Roberto Gorrieri	University of Bologna, Italy
John Hatcliff	Kansas State University, USA
David Lee	The Ohio State University, USA
Antonia Lopes	University of Lisbon, Portugal
Elie Najm	Telecom ParisTech, France
Arnd Poetzsch-Heffter	University of Kaiserslautern, Germany
Antonio Ravara	Technical University of Lisbon, Portugal
Carolyn Talcott	SRI International, USA
Ken Turner	University of Stirling, UK
Keiichi Yasumoto	Nara Institute of Science and Technology, Japan
Elena Zucca	University of Genoa, Italy

Additional Reviewers

Abraham, Erika
Amtoft, Torben
Bae, Kyungmin
Becker, Basil
Ben Hafaiedh, Khaled
Bentea, Lucian
Borgström, Johannes
Bouchy, Florent
Carbone, Marco
Chatzikokolakis, Konstantinos
Costa Seco, João
Delzanno, Giorgio
Denilou, Pierre-Malo
Desmet, Lieven
Devriese, Dominique
Dury, Arnaud
Fahrenberg, Uli
Fararooy, Ashgan
Fokkink, Wan
Gabrysiak, Gregor
Gamboni, Maxime
Gori, Roberta
Haller, Philipp
Hildebrandt, Stephan
Hüttel, Hans
Israr, Toqeer
Jaghoori, Mohammad Mahdi
Jean, Quilboeuf
Jones, Simon
Kaschner, Kathrin
Kazemeyni, Fatemeh
Keremoglu, Erkan
Kolberg, Mario
Krause, Christian
Krichen, Moez
Lagorio, Giovanni
Legay, Axel
Lehmann, Andreas
Loos, Sarah
Magill, Evan
Martins, Francisco
Merro, Massimo
Mousavi, Mohammadreza
Mühlberg, Jan Tobias
Nakata, Akio
Neumann, Stefan
Nikiforakis, Nick
Okano, Kozo
Peters, Kirstin
Philippou, Anna
Phillips, Andrew
Pous, Damien
Quo, Larry
Reggio, Gianna
Ricca, Filippo
Rosenthal, Malte
Santos, Tiago
Sasse, Ralf
Schaefer, Andreas
Schlatte, Rudolf

Shankland, Carron
Simao, Adenilso
Smans, Jan
Stolz, Volker
Tapia Tarifa, Silvia Lizeth
Taylor, Ramsay
Tiezzi, Francesco
Tran, Thi Mai Thuong
Vanoverberghe, Dries
Vieira, Hugo Torres
Vogel, Thomas
Von Essen, Christian
Wendler, Philipp
Wimmel, Harro
Yamaguchi, Hirozumi
Zavattaro, Gianluigi

Table of Contents

On Global Types and Multi-party Sessions 1
Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani

Linear-Time and May-Testing in a Probabilistic Reactive Setting 29
Lucia Acciai, Michele Boreale, and Rocco De Nicola

A Model-Checking Tool for Families of Services 44
Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi

Partial Order Methods for Statistical Model Checking and Simulation 59
Jonathan Bogdoll, Luis María Ferrer Fioriti, Arnd Hartmanns, and Holger Hermanns

Counterexample Generation for Markov Chains Using SMT-Based Bounded Model Checking 75
Bettina Braitling, Ralf Wimmer, Bernd Becker, Nils Jansen, and Erika Ábrahám

Adaptable Processes (Extended Abstract) 90
Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro

A Framework for Verifying Data-Centric Protocols 106
Yuxin Deng, Stéphane Grumbach, and Jean-François Monin

Relational Concurrent Refinement: Timed Refinement 121
John Derrick and Eerke Boiten

Galois Connections for Flow Algebras 138
Piotr Filipiuk, Michał Terepeta, Hanne Riis Nielson, and Flemming Nielson

An Accurate Type System for Information Flow in Presence of Arrays 153
Séverine Fratani and Jean-Marc Talbot

Analysis of Deadlocks in Object Groups 168
Elena Giachino and Cosimo Laneve

Monitoring Distributed Systems Using Knowledge 183
Susanne Graf, Doron Peled, and Sophie Quinton

Global State Estimates for Distributed Systems 198
Gabriel Kalyon, Tristan Le Gall, Hervé Marchand, and Thierry Massart

A Process Calculus for Dynamic Networks 213
Dimitrios Kouzapas and Anna Philippou

On Asynchronous Session Semantics 228
Dimitrios Kouzapas, Nobuko Yoshida, and Kohei Honda

Towards Verification of the Pastry Protocol Using TLA$^+$ 244
Tianxiang Lu, Stephan Merz, and Christoph Weidenbach

Dynamic Soundness in Resource-Constrained Workflow Nets 259
María Martos-Salgado and Fernando Rosa-Velardo

SimGrid MC: Verification Support for a Multi-API Simulation Platform 274
Stephan Merz, Martin Quinson, and Cristian Rosa

Ownership Types for the Join Calculus 289
Marco Patrignani, Dave Clarke, and Davide Sangiorgi

Contracts for Multi-instance UML Activities 304
Vidar Slåtten and Peter Herrmann

Annotation Inference for Separation Logic Based Verifiers 319
Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans

Analyzing BGP Instances in Maude 334
Anduo Wang, Carolyn Talcott, Limin Jia, Boon Thau Loo, and Andre Scedrov

Author Index 349

On Global Types and Multi-party Sessions

Giuseppe Castagna[1], Mariangiola Dezani-Ciancaglini[2], and Luca Padovani[2]

[1] CNRS, Université Paris Diderot – Paris 7
[2] Dipartimento d'Informatica, Università degli Studi di Torino

Abstract. We present a new, streamlined language of global types equipped with a trace-based semantics and whose features and restrictions are semantically justified. The multi-party sessions obtained projecting our global types enjoy a liveness property in addition to the traditional progress and are shown to be sound and complete with respect to the set of traces of the originating global type. Our notion of completeness is less demanding than the classical ones, allowing a multi-party session to leave out redundant traces from an underspecified global type.

1 Introduction

Relating the global specification of a system of communicating entities with an implementation (or description) of the single entities is a great classic in many different areas of computer science. The recent development of *session-based computing* has renewed the interest in this problem. In this work we attack it from the behavioral type and process algebra perspectives and briefly compare the approaches used in other areas.

A (multi-party) session is a place of interaction for a restricted number of participants that communicate messages. The interaction may involve the exchange of arbitrary sequences of messages of possibly different types. Sessions are restricted to a (usually fixed) number of participants, which makes them suitable as a structuring construct for systems of communicating entities. In this work we define a language to describe the interactions that may take place among the participants implementing a given session. In particular, we aim at a definition based on few "essential" assumptions that should not depend on the way each single participant is implemented. To give an example, a bargaining protocol that includes two participants, "seller" and "buyer", can be informally described as follows:

> *Seller sends buyer a price* <u>*and*</u> *a description of the product;* <u>*then*</u> *buyer sends seller acceptance* <u>*or*</u> *it quits the conversation.*

If we abstract from the value of the price and the content of the description sent by the seller, this simple protocol describes just two possible executions, according to whether the buyer accepts or quits. If we consider that the price and the description are in distinct messages then the possible executions become four, according to which communication happens first. While the protocol above describes a finite set of possible interactions, it can be easily modified to accommodate infinitely many possible executions, as well as additional conversations: for instance the protocol may allow "buyer" to answer "seller" with a counteroffer, or it may interleave this bargaining with an independent bargaining with a second seller.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 1–28, 2011.
© IFIP International Federation for Information Processing 2011

All essential features of protocols are in the example above, which connects some basic communication actions by the flow control points we underlined in the text. More generally, a protocol is a possibly infinite set of finite sequences of interactions between a fixed set of participants. We argue that the set of sequences that characterizes a protocol—and thus the protocol itself—can be described by a language with one form of atomic actions and three composition operators.

Atomic actions. The only atomic action is the interaction, which consists of one (or more) sender(s) (*eg*, “seller sends”), the content of the communication (*eg*, “a price”, “a description”, “acceptance”), and one (or more) receiver(s) (*eg*, “buyer”).

Compound actions. Actions and, more generally, protocols can be composed in three different ways. First, two protocols can be composed sequentially (*eg*, “Seller sends buyer a price...; ***then*** buyer sends...”) thus imposing a precise order between the actions of the composed protocols. Alternatively, two protocols can be composed unconstrainedly, without specifying any order (*eg*, “Seller sends a price ***and*** (sends) a description”) thus specifying that any order between the actions of the composed protocols is acceptable. Finally, protocols can be composed in alternative (*eg*, “buyer sends acceptance ***or*** it quits”), thus offering a choice between two or more protocols only one of which may be chosen.

More formally, we use $\mathtt{p} \xrightarrow{a} \mathtt{q}$ to state that participant $\mathtt{p}$ sends participant $\mathtt{q}$ a message whose content is described by a, and we use “;”, “$\wedge$”, and “$\vee$” to denote sequential, unconstrained, and alternative composition, respectively. Our initial example can thus be rewritten as follows:

$$(\mathtt{seller} \xrightarrow{descr} \mathtt{buyer} \wedge \mathtt{seller} \xrightarrow{price} \mathtt{buyer});\\ (\mathtt{buyer} \xrightarrow{accept} \mathtt{seller} \vee \mathtt{buyer} \xrightarrow{quit} \mathtt{seller}) \tag{1}$$

The first two actions are composed unconstrainedly, and they are to be followed by one (and only one) action of the alternative before ending. Interactions of unlimited length can be defined by resorting to a Kleene star notation. For example to extend the previous protocol so that the buyer may send a counter-offer and wait for a new price, it suffices to add a Kleene-starred line:

$$(\mathtt{seller} \xrightarrow{descr} \mathtt{buyer} \wedge \mathtt{seller} \xrightarrow{price} \mathtt{buyer});\\ (\mathtt{buyer} \xrightarrow{offer} \mathtt{seller}; \mathtt{seller} \xrightarrow{price} \mathtt{buyer})*;\\ (\mathtt{buyer} \xrightarrow{accept} \mathtt{seller} \vee \mathtt{buyer} \xrightarrow{quit} \mathtt{seller}) \tag{2}$$

The description above states that, after having received (in no particular order) the price and the description from the seller, the buyer can initiate a loop of zero or more interactions and then decide whether to accept or quit.

Whenever there is an alternative there must be a participant that decides which path to take. In both examples it is `buyer` that makes the choice by deciding whether to send *accept* or *quit*. The presence of a participant that decides holds true in loops too, since it is again `buyer` that decides whether to enter or repeat the iteration (by sending *offer*) or to exit it (by sending *accept* or *quit*). We will later show that absence of such decision-makers gives protocols impossible to implement. This last point critically

depends on the main hypothesis we assume about the systems we are going to the describe, that is the absence of *covert channels*. One the one hand, we try to develop a protocol description language that is as generic as possible; on the other hand, we limit the power of the system and require all communications between different participants to be explicitly stated. In doing so we bar out protocols whose implementation essentially relies on the presence of secret/invisible communications between participants: a protocol description must contain all and only the interactions used to implement it.

Protocol specifications such as the ones presented above are usually called *global types* to emphasize the fact that they describe the acceptable behaviours of a system from a global point of view. In an actual implementation of the system, though, each participant autonomously implements a different part of the protocol. To understand whether an implementation satisfies a specification, one has to consider the set of all possible sequences of synchronizations performed by the implementation and check whether this set satisfies five basic properties:

1. *Sequentiality:* if the specification states that two interactions must occur in a given order (by separating them by a ";"), then this order must be respected by all possible executions. So an implementation in which `buyer` may send *accept* before receiving *price* violates the specification.
2. *Alternativeness:* if the specification states that two interactions are alternative, then every execution must exhibit one and only one of these two actions. So an implementation in which `buyer` emits both *accept* and *quit* (or none of them) in the same execution violates the specification.
3. *Shuffling:* if the specification composes two sequences of interactions in an unconstrained way, then all executions must exhibit some shuffling (in the sense used in combinatorics and algebra) of these sequences. So an implementation in which `seller` emits *price* without emitting *descr* violates the specification.
4. *Fitness:* if the implementation exhibits a sequence of interactions, then this sequence is expected by (*ie*, it fits) the specification. So any implementation in which `seller` sends `buyer` any message other than *price* and *descr* violates the specification.
5. *Exhaustivity:* if some sequence of interactions is described by the specification, then there must exist at least an execution of the implementation that exhibits these actions (possibly in a different order). So an implementation in which no execution of `buyer` emits *accept* violates the specification.

Checking whether an implemented system satisfies a specification by comparing the actual and the expected sequences of interactions is non-trivial, for systems are usually infinite-state. Therefore, on the lines of [9,16], we proceed the other way round: we extract from a global type the local specification (usually dubbed *session type* [20,15]) of each participant in the system and we type-check the implementation of each participant against the corresponding session type. If the projection operation is done properly and the global specification satisfies some well-formedness conditions, then we are guaranteed that the implementation satifies the specification. As an example, the global type (1) can be projected to the following behaviors for `buyer` and `seller`:

$$\begin{aligned} \mathtt{seller} &\mapsto \mathtt{buyer}!\mathit{descr}.\mathtt{buyer}!\mathit{price}.(\mathtt{buyer}?\mathit{accept} + \mathtt{buyer}?\mathit{quit}) \\ \mathtt{buyer} &\mapsto \mathtt{seller}?\mathit{descr}.\mathtt{seller}?\mathit{price}.(\mathtt{seller}!\mathit{accept} \oplus \mathtt{seller}!\mathit{quit}) \end{aligned}$$

or to

$$\begin{array}{r}\texttt{seller} \mapsto \texttt{buyer}!\mathit{price}.\texttt{buyer}!\mathit{descr}.(\texttt{buyer}?\mathit{accept} + \texttt{buyer}?\mathit{quit})\\ \texttt{buyer} \mapsto \texttt{seller}?\mathit{price}.\texttt{seller}?\mathit{descr}.(\texttt{seller}!\mathit{accept} \oplus \texttt{seller}!\mathit{quit})\end{array}$$

where $\texttt{p}!a$ denotes the output of a message a to participant $\texttt{p}$, $\texttt{p}?a$ the input of a message a from participant $\texttt{p}$, $\texttt{p}?a.T + \texttt{q}?b.S$ the (external) choice to continue as T or S according to whether a is received from $\texttt{p}$ or b is received from $\texttt{q}$ and, finally, $\texttt{p}!a.T \oplus \texttt{q}!b.S$ denotes the (internal) choice between sending a to $\texttt{p}$ and continue as T or sending S to $\texttt{q}$ and continue as T. We will call *session environments* the mappings from participants to their session types. It is easy to see that any two processes implementing `buyer` and `seller` will satisfy the global type (1) if *and only if* their visible behaviors matches one of the two session environments above (these session environments thus represent some sort of minimal typings of processes implementing `buyer` and `seller`). In particular, both the above session environments are fitting and exhaustive with respect to the specification since they precisely describe what the single participants are expected and bound to do.

We conclude this introduction by observing that there are global types that are intrinsically flawed, in the sense that they do not admit any implementation (without covert channels) satisfying them. We classify flawed global types in three categories, according to the seriousness of their flaws.

[*No sequentiality*] The mildest flaws are those in which the global type specifies some sequentiality constraint between independent interactions, such as in $(\texttt{p} \xrightarrow{a} \texttt{q};\texttt{r} \xrightarrow{b} \texttt{s})$, since it is impossible to implement $\texttt{r}$ so that it sends b only after that $\texttt{q}$ has received a (unless this reception is notified on a covert channel, of course). Therefore, it is possible to find exhaustive (but not fitting) implementations that include some unexepected sequences which differ from the expected ones only by a permutation of interactions done by independent participants. The specification at issue can be easily patched by replacing some ";" by "$\wedge$".

[*No knowledge for choice*] A more severe kind of flaw occurs when the global type requires some participant to behave in different ways in accordance with some choice it is unaware of. For instance, in the global type

$$(\texttt{p} \xrightarrow{a} \texttt{q};\texttt{q} \xrightarrow{a} \texttt{r};\texttt{r} \xrightarrow{a} \texttt{p}) \quad \vee \quad (\texttt{p} \xrightarrow{b} \texttt{q};\texttt{q} \xrightarrow{a} \texttt{r};\texttt{r} \xrightarrow{b} \texttt{p})$$

participant $\texttt{p}$ chooses the branch to execute, but after having received a from $\texttt{q}$ participant $\texttt{r}$ has no way to know whether it has to send a or b. Also in this case it is possible to find exhaustive (but not fitting) implementations of the global type where the participant $\texttt{r}$ chooses to send a or b independently of what $\texttt{p}$ decided to do.

[*No knowledge, no choice*] In the worst case it is not possible to find an exhaustive implementation of the global type, for it specifies some combination of incompatible behaviors, such as performing and input or an output in mutual exclusion. This typically is the case of the absence of a decision-maker in the alternatives such as in

$$\texttt{p} \xrightarrow{a} \texttt{q} \vee \texttt{q} \xrightarrow{b} \texttt{p}$$

where each participant is required to choose between sending or receiving. There seems to be no obvious way to patch these global types without reconsidering also the intended semantics.

Contributions and outline. A first contribution of this work is to introduce a streamlined language of global specifications—that we dub *global types* (Section 2)—and to relate it with *session environments* (Section 3), that is, with sets of independent, sequential, asynchronous *session types* to be type-checked against implementations. Global types are just regular expressions augmented with a shuffling operator and their semantics is defined in terms of finite sequences of interactions. The second contribution, which is a consequence of the chosen semantics of global types, is to ensure that every implementation of a global type preserves the possibility to reach a state where *every* participant has successfully terminated.

In Section 4 we study the relationship between global types and sessions. We do so by defining a projection operation that extracts from a global type *all* the (sets of) possible session types of its participants. This projection is useful not only to check the implementability of a global description (and, incidentally, to formally define the notions of errors informally described so far) but, above all, to relate in a compositional and modular way a global type with the sets of distributed processes that implement it. We also identify a class of well-formed global types whose projections need no covert channels. Interestingly, we are able to effectively characterize well-formed global types solely in terms of their semantics.

In Section 5 we present a projection algorithm for global types. The effective generation of all possible projections is impossible. The reason is that the projectability of a global type may rely on some global knowledge that is no longer available when working at the level of single session types: while in a global approach we can, say, add to some participant new synchronization offerts that, thanks to our global knowledge, we know will never be used, this cannot be done when working at the level of single participant. Therefore in order to work at the projected level we will use stronger assumptions that ensure a sound implementation in all possible contexts.

In Section 6 we show some limitations deriving from the use of the Kleene star operator in our language of global types, and we present one possible way to circumvent them. We conclude with an extended discussion on related work (Section 7) and a few final considerations (Section 8).

Proofs, more examples and an extended survey of related work were omitted and can be found in the long version available on the authors' home pages.

2 Global Types

In this section we define syntax and semantics of global types. We assume a set $\mathscr{A}$ of *message types*, ranged over by a, b, $\dots$, and a set Π of *roles*, ranged over by p, q, $\dots$, which we use to uniquely identify the participants of a session; we let $\pi, \dots$ range over non-empty, finite sets of roles.

Global types, ranged over by $\mathscr{G}$, are the terms generated by the grammar in Table 1. Their syntax was already explained in Section 1 except for two novelties. First, we include a skip atom which denotes the unit of sequential composition (it plays the same role as the empty word in regular expressions). This is useful, for instance, to express

Table 1. Syntax of global types

$\mathscr{G}$ ::=		**Global Type**		
	skip	(skip)	$\mid \pi \xrightarrow{a} \mathtt{p}$	(interaction)
$\mid$	$\mathscr{G};\mathscr{G}$	(sequence)	$\mid \mathscr{G} \wedge \mathscr{G}$	(both)
$\mid$	$\mathscr{G} \vee \mathscr{G}$	(either)	$\mid \mathscr{G}^*$	(star)

optional interactions. Thus, if in our example we want the buyer to do at most one counteroffer instead of several ones, we just replace the starred line in (2) by

$$(\mathtt{buyer} \xrightarrow{\mathit{offer}} \mathtt{seller}; \mathtt{seller} \xrightarrow{\mathit{price}} \mathtt{buyer}) \vee \mathsf{skip}$$

which, using syntactic sugar of regular expressions, might be rendered as

$$(\mathtt{buyer} \xrightarrow{\mathit{offer}} \mathtt{seller}; \mathtt{seller} \xrightarrow{\mathit{price}} \mathtt{buyer})?$$

Second, we generalize interactions by allowing a finite set of roles on the l.h.s. of interactions. Therefore, $\pi \xrightarrow{a} \mathtt{p}$ denotes the fact that (the participant identified by) $\mathtt{p}$ waits for an a message from all of the participants whose tags are in π. We will write $\mathtt{p} \xrightarrow{a} \mathtt{q}$ as a shorthand for $\{\mathtt{p}\} \xrightarrow{a} \mathtt{q}$.

To be as general as possible, one could also consider interactions of the form $\pi \xrightarrow{a} \pi'$, which could be used to specify broadcast communications between participants. It turns out that this further generalization is superfluous in our setting since the interaction $\pi \xrightarrow{a} \{\mathtt{p}_i\}_{i \in I}$ can be encoded as $\bigwedge_{i \in I}(\pi \xrightarrow{a} \mathtt{p}_i)$. The encoding is made possible by the fact that communication is asynchronous and output actions are not blocking (see Section 3), therefore the order in which the participants in π send a to the $\mathtt{p}_i$'s is irrelevant. Vice versa, we will see that allowing sets of multiple senders enriches the expressiveness of the calculus, because $\pi \xrightarrow{a} \mathtt{p}$ can be used to *join* different activities involving the participants in $\pi \cup \{\mathtt{p}\}$, while $\bigwedge_{i \in I}(\mathtt{p} \xrightarrow{a} \mathtt{q}_i)$ represents *fork* of parallel activities. For example, we can represent two buyers waiting for both the price from a seller and the mortgage from a bank before deciding the purchase:

$$\begin{array}{l}(\mathtt{seller} \xrightarrow{\mathit{price}} \mathtt{buyer1} \wedge \mathtt{bank} \xrightarrow{\mathit{mortgage}} \mathtt{buyer2}); \\ (\{\mathtt{buyer1},\mathtt{buyer2}\} \xrightarrow{\mathit{accept}} \mathtt{seller} \wedge \{\mathtt{buyer1},\mathtt{buyer2}\} \xrightarrow{\mathit{accept}} \mathtt{bank})\end{array} \tag{3}$$

In general we will assume $\mathtt{p} \notin \pi$ for every interaction $\pi \xrightarrow{a} \mathtt{p}$ occurring in a global type, that is, we forbid participants to send messages to themselves. For the sake of readability we adopt the following precedence of global type operators $\longrightarrow \; ^* \; ; \; \wedge \; \vee$.

Global types denote languages of legal interactions that can occur in a multi-party session. These languages are defined over the alphabet of interactions

$$\Sigma = \{\pi \xrightarrow{a} \mathtt{p} \mid \pi \subset_{\mathit{fin}} \Pi, \mathtt{p} \in \Pi, \mathtt{p} \notin \pi, a \in \mathscr{A}\}$$

and we use α as short for $\pi \xrightarrow{a} \mathtt{p}$ when possible; we use φ, ψ, ... to range over strings in Σ^* and ε to denote the empty string, as usual. To improve readability we will sometimes use ";" to denote string concatenation.

In order to express the language of a global type having the shape $\mathscr{G}_1 \wedge \mathscr{G}_2$ we need a standard shuffling operator over languages, which can be defined as follows:

Definition 2.1 (shuffling). *The* shuffle *of L_1 and L_2, denoted by $L_1 \sqcup\!\sqcup L_2$, is the language defined by: $L_1 \sqcup\!\sqcup L_2 \stackrel{def}{=} \{\varphi_1\psi_1 \cdots \varphi_n\psi_n \mid \varphi_1 \cdots \varphi_n \in L_1 \wedge \psi_1 \cdots \psi_n \in L_2\}$.*

Observe that, in $L_1 \sqcup\!\sqcup L_2$, the order of interactions coming from one language is preserved, but these interactions can be interspersed with other interactions coming from the other language.

Definition 2.2 (traces of global types). *The set of* traces *of a global type is inductively defined by the following equations:*

$$\begin{array}{lll} \mathrm{tr}(\mathsf{skip}) = \{\varepsilon\} & \mathrm{tr}(\mathscr{G}_1;\mathscr{G}_2) = \mathrm{tr}(\mathscr{G}_1)\mathrm{tr}(\mathscr{G}_2) & \mathrm{tr}(\mathscr{G}_1 \vee \mathscr{G}_2) = \mathrm{tr}(\mathscr{G}_1) \cup \mathrm{tr}(\mathscr{G}_2) \\ \mathrm{tr}(\pi \xrightarrow{a} \mathtt{p}) = \{\pi \xrightarrow{a} \mathtt{p}\} & \mathrm{tr}(\mathscr{G}^*) = (\mathrm{tr}(\mathscr{G}))^\star & \mathrm{tr}(\mathscr{G}_1 \wedge \mathscr{G}_2) = \mathrm{tr}(\mathscr{G}_1) \sqcup\!\sqcup \mathrm{tr}(\mathscr{G}_2) \end{array}$$

where juxtaposition denotes concatenation and $(\cdot)^\star$ is the usual Kleene closure of regular languages.

Before we move on, it is worth noting that $\mathrm{tr}(\mathscr{G})$ is a regular language (recall that regular languages are closed under shuffling). Since a regular language is made of *finite strings*, we are implicitly making the assumption that a global type specifies interactions of finite length. This means that we are considering interactions of arbitraty length, but such that the termination of all the involved participants is always within reach. This is a subtle, yet radical change from other multi-party session theories, where infinite interactions are considered legal.

3 Multi-party Sessions

We devote this section to the formal definition of the behavior of the participants of a multiparty session.

3.1 Session Types

We need an infinite set of recursion variables ranged over by $X, \ldots$. Pre-session types, ranged over by $T, S, \ldots$, are the terms generated by the grammar in Table 2 such that all recursion variables are guarded by at least one input or output prefix. We consider pre-session types modulo associativity, commutativity, and idempotence of internal and external choices, fold/unfold of recursions and the equalities

$$\pi!a.T \oplus \pi!a.S = \pi!a.(T \oplus S) \qquad\qquad \pi?a.T + \pi?a.S = \pi?a.(T + S)$$

Pre-session types are behavioral descriptions of the participants of a multiparty session. Informally, end describes a successfully terminated party that no longer participates to a session. The pre-session type $\mathtt{p}!a.T$ describes a participant that sends an a message to participant $\mathtt{p}$ and afterwards behaves according to T; the pre-session type $\pi?a.T$ describes a participant that waits for an a message from all the participants in π

Table 2. Syntax of pre-session types

T ::=	**Pre-Session Type**		
end	(termination)	$\mid X$	(variable)
$\mid$ $\mathtt{p}!a.T$	(output)	$\mid \pi?a.T$	(input)
$\mid$ $T \oplus T$	(internal choice)	$\mid T + T$	(external choice)
$\mid$ $\mathtt{rec}\, X.T$	(recursion)		

and, upon arrival of the message, behaves according to T; we will usually abbreviate $\{\mathtt{p}\}?a.T$ with $\mathtt{p}?a.T$. Behaviors can be combined by means of behavioral choices $\oplus$ and $+$: $T \oplus S$ describes a participant that internally decides whether to behave according to T or S; $T+S$ describes a participant that offers to the other participants two possible behaviors, T and S. The choice as to which behavior is taken depends on the messages sent by the other participant. In the following, we sometimes use n-ary versions of internal and external choices and write, for example, $\bigoplus_{i=1}^{n} \mathtt{p}_i!a_i.T_i$ for $\mathtt{p}_1!a_1.T_1 \oplus \cdots \oplus \mathtt{p}_n!a_n.T_n$ and $\sum_{i=1}^{n} \pi_i?a_i.T_i$ for $\pi_1?a_1.T_1 + \cdots + \pi_n?a_n.T_n$. As usual, terms X and $\mathtt{rec}\, X.T$ are used for describing recursive behaviors. As an example, $\mathtt{rec}\, X.(\mathtt{p}!a.X \oplus \mathtt{p}!b.\mathsf{end})$ describes a participant that sends an arbitrary number of a messages to $\mathtt{p}$ and terminates by sending a b message; dually, $\mathtt{rec}\, X.(\mathtt{p}?a.X + \mathtt{p}?b.\mathsf{end})$ describes a participant that is capable of receiving an arbitrary number of a messages from $\mathtt{p}$ and terminates as soon a b message is received.

Session types are the pre-session types where internal choices are used to combine outputs, external choices are used to combine inputs, and the continuation after every prefix is uniquely determined by the prefix. Formally:

Definition 3.1 (session types). *A pre-session type T is a* session type *if either:*

- $T = \mathsf{end}$, *or*
- $T = \bigoplus_{i\in I} \mathtt{p}_i!a_i.T_i$ *and* $\forall i, j \in I$ *we have that* $\mathtt{p}_i!a_i = \mathtt{p}_j!a_j$ *implies* $i = j$ *and each* T_i *is a session type, or*
- $T = \sum_{i\in I} \pi_i?a_i.T_i$ *and* $\forall i, j \in I$ *we have that* $\pi_i \subseteq \pi_j$ *and* $a_i = a_j$ *imply* $i = j$ *and each* T_i *is a session type.*

3.2 Session Environments

A session environment is defined as the set of the session types of its participants, where each participant is uniquely identified by a role. Formally:

Definition 3.2 (session environment). *A* session environment *(briefly,* session*) is a finite map* $\{\mathtt{p}_i : T_i\}_{i\in I}$.

In what follows we use Δ to range over sessions and we write $\Delta \uplus \Delta'$ to denote the union of sessions, when their domains are disjoint.

To describe the operational semantics of a session we model an asynchronous form of communication where the messages sent by the participants of the session are stored within a *buffer* associated with the session. Each message has the form $\mathtt{p} \xrightarrow{a} \mathtt{q}$ describing the sender $\mathtt{p}$, the receiver $\mathtt{q}$, and the type a of message. Buffers, ranged over by $\mathbb{B}$,

..., are finite sequences $\mathtt{p}_1 \xrightarrow{a_1} \mathtt{q}_1 :: \cdots :: \mathtt{p}_n \xrightarrow{a_n} \mathtt{q}_n$ of messages considered modulo the least congruence $\simeq$ over buffers such that:

$$\mathtt{p} \xrightarrow{a} \mathtt{q} :: \mathtt{p}' \xrightarrow{b} \mathtt{q}' \simeq \mathtt{p}' \xrightarrow{b} \mathtt{q}' :: \mathtt{p} \xrightarrow{a} \mathtt{q} \qquad \text{for } \mathtt{p} \neq \mathtt{p}' \text{ or } \mathtt{q} \neq \mathtt{q}'$$

that is, we care about the order of messages in the buffer only when these have both the same sender and the same receiver. In practice, this corresponds to a form of communication where each pair of participants of a multiparty session is connected by a distinct FIFO buffer.

There are two possible reductions of a session:

$$\mathbb{B} \,⨾\, \{\mathtt{p} : \bigoplus_{i\in I} \mathtt{p}_i!a_i.T_i\} \uplus \Delta \quad \longrightarrow \quad (\mathtt{p} \xrightarrow{a_k} \mathtt{p}_k)::\mathbb{B} \,⨾\, \{\mathtt{p} : T_k\} \uplus \Delta \qquad (k\in I)$$

$$\mathbb{B}::(\mathtt{p}_i \xrightarrow{a} \mathtt{p})_{i\in I} \,⨾\, \{\mathtt{p} : \sum_{j\in J} \pi_j?a_j.T_j\} \uplus \Delta \xrightarrow{\pi_k \xrightarrow{a} \mathtt{p}} \mathbb{B} \,⨾\, \{\mathtt{p} : T_k\} \uplus \Delta \qquad \begin{pmatrix} k\in J & a_k = a \\ \pi_k = \{\mathtt{p}_i | i\in I\} \end{pmatrix}$$

The first rule describes the effect of an output operation performed by participant p, which stores the message $\mathtt{p} \xrightarrow{a_k} \mathtt{p}_k$ in the buffer and leaves participant p with a residual session type T_k corresponding to the message that has been sent. The second rule describes the effect of an input operation performed by participant p. If the buffer contains enough messages of type a coming from all the participants in π_k, those messages are removed from the buffer and the receiver continues as described in T_k. In this rule we decorate the reduction relation with $\pi_k \xrightarrow{a} \mathtt{p}$ that describes the occurred interaction (as we have already remarked, we take the point of view that an interaction is completed when messages are received). This decoration will allow us to relate the behavior of an implemented session with the traces of a global type (see Definition 2.2). We adopt some conventional notation: we write $\Longrightarrow$ for the reflexive, transitive closure of $\longrightarrow$; we write $\stackrel{\alpha}{\Longrightarrow}$ for the composition $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ and $\stackrel{\alpha_1\cdots\alpha_n}{\Longrightarrow}$ for the composition $\stackrel{\alpha_1}{\Longrightarrow} \cdots \stackrel{\alpha_n}{\Longrightarrow}$.

We can now formally characterize the "correct sessions" as those in which, no matter how they reduce, it is always possible to reach a state where all of the participants are successfully terminated and the buffer has been emptied.

Definition 3.3 (live session). *We say that* Δ *is a* live session *if* $\varepsilon \,⨾\, \Delta \stackrel{\varphi}{\Longrightarrow} \mathbb{B} \,⨾\, \Delta'$ *implies* $\mathbb{B} \,⨾\, \Delta' \stackrel{\psi}{\Longrightarrow} \varepsilon \,⨾\, \{\mathtt{p}_i : \mathsf{end}\}_{i\in I}$ *for some* ψ.

We adopt the term "live session" to emphasize the fact that Definition 3.3 states a *liveness property*: every finite computation $\varepsilon, \Delta \stackrel{\varphi}{\Longrightarrow} \mathbb{B} \,⨾\, \Delta'$ can always be extended to a successful computation $\varepsilon \,⨾\, \Delta \stackrel{\varphi}{\Longrightarrow} \mathbb{B} \,⨾\, \Delta' \stackrel{\psi}{\Longrightarrow} \varepsilon \,⨾\, \{\mathtt{p}_i : \mathsf{end}\}_{i\in I}$. This is stronger than the progress property enforced by other multiparty session type theories, where it is only required that a session must never get stuck (but it is possible that some participants starve for messages that are never sent). As an example, the session

$$\Delta_1 = \{\mathtt{p} : \mathtt{rec}\ X.(\mathtt{q}!a.X \oplus \mathtt{q}!b.\mathsf{end})\ ,\ \mathtt{q} : \mathtt{rec}\ Y.(\mathtt{p}?a.Y + \mathtt{p}?b.\mathsf{end})\}$$

is alive because, no matter how many a messages p sends, q can receive all of them and, if p chooses to send a b message, the interaction terminates successfully for both p and q. This example also shows that, despite the fact that session types describe finite-state

processes, the session Δ_1 is not finite-state, in the sense that the set of configurations $\{(\mathbb{B} \,\S\, \Delta') \mid \exists\varphi, \mathbb{B}, \Delta' : \varepsilon \,\S\, \Delta_1 \stackrel{\varphi}{\Longrightarrow} \mathbb{B} \,\S\, \Delta'\}$ is infinite. This happens because there is no bound on the size of the buffer and an arbitrary number of a messages sent by $\mathtt{p}$ can accumulate in $\mathbb{B}$ before $\mathtt{q}$ receives them. As a consequence, the fact that a session is alive cannot be established in general by means of a brute force algorithm that checks every reachable configuration. By contrast, the session

$$\Delta_2 = \{\mathtt{p} : \mathtt{rec}\, X.\mathtt{q}!a.X \ , \ \mathtt{q} : \mathtt{rec}\, Y.\mathtt{p}?a.Y\}$$

which is normally regarded correct in other session type theories, is not alive because there is no way for $\mathtt{p}$ and $\mathtt{q}$ to reach a successfully terminated state. The point is that hitherto correctness of session was associated to progress (*ie*, the system is never stuck). This is a weak notion of correctness since, for instance the session $\Delta_2 \uplus \{\mathtt{r} : \mathtt{p}?c.\mathsf{end}\}$ satisfies progress even though role $\mathtt{r}$ starves waiting for its input. While in this example starvation is clear since no c message is ever sent, determining starvation is in general more difficult, as for

$$\Delta_3 = \{\mathtt{p} : \mathtt{rec}\, X.\mathtt{q}!a.\mathtt{q}!b.X \ , \ \mathtt{q} : \mathtt{rec}\, Y.(\mathtt{p}?a.\mathtt{p}?b.Y + \mathtt{p}?b.\mathtt{r}!c.\mathsf{end}) \ , \ \mathtt{r} : \mathtt{q}?c.\mathsf{end}\}$$

which satisfies progress, where every input corresponds to a compatible output, and viceversa, but which is not alive.

We can now define the traces of a session as the set of sequences of interactions that can occur in every possible reduction. It is convenient to define the traces of an incorrect (*ie*, non-live) session as the empty set (observe that $\mathrm{tr}(\mathscr{G}) \neq \emptyset$ for every $\mathscr{G}$).

Definition 3.4 (session traces)

$$\mathrm{tr}(\Delta) \stackrel{def}{=} \begin{cases} \{\varphi \mid \varepsilon \,\S\, \Delta \stackrel{\varphi}{\Longrightarrow} \varepsilon \,\S\, \{\mathtt{p}_i : \mathsf{end}\}_{i \in I}\} & \textit{if } \Delta \textit{ is a live session} \\ \emptyset & \textit{otherwise} \end{cases}$$

It is easy to verify that $\mathrm{tr}(\Delta_1) = \mathrm{tr}((\mathtt{p} \stackrel{a}{\longrightarrow} \mathtt{q})^*; \mathtt{p} \stackrel{b}{\longrightarrow} \mathtt{q})$ while $\mathrm{tr}(\Delta_2) = \mathrm{tr}(\Delta_3) = \emptyset$ since neither Δ_2 nor Δ_3 is a live session.

4 Semantic Projection

In this section we show how to project a global type to the session types of its participants —*ie*, to a session— in such a way that the projection is correct with respect to the global type. Before we move on, we must be more precise about what we mean by correctness of a session Δ with respect to a global type $\mathscr{G}$. In our setting, correctness refers to some relationship between the traces of Δ and those of $\mathscr{G}$. In general, however, we cannot require that $\mathscr{G}$ and Δ have exactly the same traces: when projecting $\mathscr{G}_1 \wedge \mathscr{G}_2$ we might need to impose a particular order in which the interactions specified by $\mathscr{G}_1$ and $\mathscr{G}_2$ must occur (shuffling condition). At the same time, asking only $\mathrm{tr}(\Delta) \subseteq \mathrm{tr}(\mathscr{G})$ would lead us to immediately loose the exhaustivity property, since for instance $\{\mathtt{p} : \mathtt{q}!a.\mathsf{end} \ , \ \mathtt{q} : \mathtt{p}?a.\mathsf{end}\}$ would implement $\mathtt{p} \stackrel{a}{\longrightarrow} \mathtt{q} \vee \mathtt{p} \stackrel{b}{\longrightarrow} \mathtt{q}$ even though

Table 3. Rules for semantic projection

(SP-SKIP)
$$\Delta \vdash \mathsf{skip} \rhd \Delta$$

(SP-ACTION)
$$\{\mathtt{p}_i : T_i\}_{i\in I} \uplus \{\mathtt{p} : T\} \uplus \Delta \vdash \{\mathtt{p}_i\}_{i\in I} \xrightarrow{a} \mathtt{p} \rhd \{\mathtt{p}_i : \mathtt{p}!a.T_i\}_{i\in I} \uplus \{\mathtt{p} : \{\mathtt{p}_i\}_{i\in I}?a.T\} \uplus \Delta$$

(SP-SEQUENCE)
$$\frac{\Delta \vdash \mathscr{G}_2 \rhd \Delta' \qquad \Delta' \vdash \mathscr{G}_1 \rhd \Delta''}{\Delta \vdash \mathscr{G}_1;\mathscr{G}_2 \rhd \Delta''}$$

(SP-ALTERNATIVE)
$$\frac{\Delta \vdash \mathscr{G}_1 \rhd \{\mathtt{p} : T_1\} \uplus \Delta' \qquad \Delta \vdash \mathscr{G}_2 \rhd \{\mathtt{p} : T_2\} \uplus \Delta'}{\Delta \vdash \mathscr{G}_1 \vee \mathscr{G}_2 \rhd \{\mathtt{p} : T_1 \oplus T_2\} \uplus \Delta'}$$

(SP-ITERATION)
$$\frac{\{\mathtt{p} : T_1 \oplus T_2\} \uplus \Delta \vdash \mathscr{G} \rhd \{\mathtt{p} : T_1\} \uplus \Delta}{\{\mathtt{p} : T_2\} \uplus \Delta \vdash \mathscr{G}^* \rhd \{\mathtt{p} : T_1 \oplus T_2\} \uplus \Delta}$$

(SP-SUBSUMPTION)
$$\frac{\Delta \vdash \mathscr{G}' \rhd \Delta' \qquad \mathscr{G}' \leqslant \mathscr{G} \qquad \Delta'' \leqslant \Delta'}{\Delta \vdash \mathscr{G} \rhd \Delta''}$$

the implementation systematically exhibits only one ($\mathtt{p} \xrightarrow{a} \mathtt{q}$) of the specified alternative behaviors. In the end, we say that Δ is a correct implementation of $\mathscr{G}$ if: first, every trace of Δ is a trace of $\mathscr{G}$ (*soundness*); second, every trace of $\mathscr{G}$ is the permutation of a trace of Δ (*completeness*). Formally:

$$\operatorname{tr}(\Delta) \subseteq \operatorname{tr}(\mathscr{G}) \subseteq \operatorname{tr}(\Delta)^\circ$$

where L° is the closure of L under arbitrary permutations of the strings in L:

$$L^\circ \stackrel{\text{def}}{=} \{\alpha_1 \cdots \alpha_n \mid \text{there exists a permutation } \sigma \text{ such that } \alpha_{\sigma(1)} \cdots \alpha_{\sigma(n)} \in L\}$$

Since these relations between languages (of traces) play a crucial role, it is convenient to define a suitable pre-order relation:

Definition 4.1 (implementation pre-order). *We let $L_1 \leqslant L_2$ if $L_1 \subseteq L_2 \subseteq L_1^\circ$ and extend it to global types and sessions in the natural way, by considering the corresponding sets of traces. Therefore, we write $\Delta \leqslant \mathscr{G}$ if $\operatorname{tr}(\Delta) \leqslant \operatorname{tr}(\mathscr{G})$.*

It is easy to see that soundness and completeness respectively formalize the notions of fitness and exhaustivity that we have outlined in the introduction. For what concerns the remaining three properties listed in the introduction (*ie*, sequentiality, alternativeness, and shuffling), they are entailed by the formalization of the semantics of a global type in terms of its traces (Definition 2.2). In particular, we have that soundness implies sequentiality and alternativeness, while completeness implies shuffling. Therefore, in the formal treatment that follows we will focus on soundness and completeness as to the only characterizing properties connecting sessions and global types. The relation $\Delta \leqslant \mathscr{G}$ summarizes the fact that Δ is both sound and complete with respect to $\mathscr{G}$, namely that Δ is a correct implementation of the specification $\mathscr{G}$.

Table 3 presents our rules to build the projections of global types. Projecting a global type basically means compiling it to an "equivalent" set of session types. Since the source language (global types) is equipped with sequential composition while the

target language (session types) is not, it is convenient to parameterize projection on a continuation, *ie*, we consider judgments of the shape:

$$\Delta \vdash \mathscr{G} \triangleright \Delta'$$

meaning that if Δ is the projection of some $\mathscr{G}'$, then Δ' is the projection of $\mathscr{G};\mathscr{G}'$. This immediately gives us the rule (SP-SEQUENCE). We say that Δ' is a *projection* of $\mathscr{G}$ with *continuation* Δ.

The projection of an *interaction* $\pi \xrightarrow{a} \mathtt{p}$ adds $\mathtt{p}!a$ in front of the session type of all the roles in π, and $\pi?a$ in front of the session type of $\mathtt{p}$ (rule (SP-ACTION)). For example we have:

$$\{\mathtt{p} : \mathsf{end}, \mathtt{q} : \mathsf{end}\} \vdash \mathtt{p} \xrightarrow{a} \mathtt{q} \triangleright \{\mathtt{p} : \mathtt{q}!a.\mathsf{end},\ \ \mathtt{q} : \mathtt{p}?a.\mathsf{end}\}$$

An *alternative* $\mathscr{G}_1 \vee \mathscr{G}_2$ (rule (SP-ALTERNATIVE)) can be projected only if there is a participant $\mathtt{p}$ that actively chooses among different behaviors by sending different messages, while all the other participants must exhibit the same behavior in both branches. The subsumption rule (SP-SUBSUMPTION) can be used to fulfil this requirement in many cases. For example we have $\Delta_0 \vdash \mathtt{p} \xrightarrow{a} \mathtt{q} \triangleright \{\mathtt{p} : \mathtt{q}!a.\mathsf{end}, \mathtt{q} : \mathtt{p}?a.\mathsf{end}\}$ and $\Delta_0 \vdash \mathtt{p} \xrightarrow{b} \mathtt{q} \triangleright \{\mathtt{p} : \mathtt{q}!b.\mathsf{end}, \mathtt{q} : \mathtt{p}?b.\mathsf{end}\}$, where $\Delta_0 = \{\mathtt{p} : \mathsf{end}, \mathtt{q} : \mathsf{end}\}$. In order to project $\mathtt{p} \xrightarrow{a} \mathtt{q} \vee \mathtt{p} \xrightarrow{b} \mathtt{q}$ with continuation Δ_0 we derive first by subsumption $\Delta_0 \vdash \mathtt{p} \xrightarrow{a} \mathtt{q} \triangleright \{\mathtt{p} : \mathtt{q}!a.\mathsf{end}\ ,\ \mathtt{q} : T\}$ and $\Delta_0 \vdash \mathtt{p} \xrightarrow{b} \mathtt{q} \triangleright \{\mathtt{p} : \mathtt{q}!b.\mathsf{end}\ ,\ \mathtt{q} : T\}$ where $T = \mathtt{p}?a.\mathsf{end} + \mathtt{p}?b.\mathsf{end}$. Then we obtain

$$\Delta_0 \vdash \mathtt{p} \xrightarrow{a} \mathtt{q} \vee \mathtt{p} \xrightarrow{b} \mathtt{q} \triangleright \{\mathtt{p} : \mathtt{q}!a.\mathsf{end} \oplus \mathtt{q}!b.\mathsf{end}\ ,\ \ \mathtt{q} : T\}$$

Notice that rule (SP-ALTERNATIVE) imposes that in alternative branches there must be one *and only one* participant that takes the decision. For instance, the global type

$$\{\mathtt{p}, \mathtt{q}\} \xrightarrow{a} \mathtt{r} \ \vee\ \{\mathtt{p}, \mathtt{q}\} \xrightarrow{b} \mathtt{r}$$

cannot be projected since we would need a covert channel for $\mathtt{p}$ to agree with $\mathtt{q}$ about whether to send to $\mathtt{r}$ the message a or b.

To project a *starred* global type we also require that one participant $\mathtt{p}$ chooses between repeating the loop or exiting by sending messages, while the session types of all other participants are unchanged. If T_1 and T_2 are the session types describing the behavior of $\mathtt{p}$ when it has respectively decided to perform one more iteration or to terminate the iteration, then $T_1 \oplus T_2$ describes the behavior of $\mathtt{p}$ before it takes the decision. The projection rule requires that one execution of $\mathscr{G}$ followed by the choice between T_1 and T_2 projects in a session with type T_1 for $\mathtt{p}$. This is possible only if T_1 is a recursive type, as expected, and it is the premise of rule (SP-ITERATION). For example if $T_1 = \mathtt{q}!a.\mathtt{rec}\, X.(\mathtt{q}!a.X \oplus \mathtt{q}!b.\mathsf{end})$, $T_2 = \mathtt{q}!b.\mathsf{end}$, and $S = \mathtt{rec}\, Y.(\mathtt{p}?a.Y + \mathtt{p}?b.\mathsf{end})$ we can derive $\{\mathtt{p} : T_1 \oplus T_2, \mathtt{q} : S\} \vdash \mathtt{p} \xrightarrow{a} \mathtt{q} \triangleright \{\mathtt{p} : T_1, \mathtt{q} : S\}$ and then

$$\{\mathtt{p} : T_2, \mathtt{q} : S\} \vdash (\mathtt{p} \xrightarrow{a} \mathtt{q})^* \triangleright \{\mathtt{p} : T_1 \oplus T_2,\ \ \mathtt{q} : S\}$$

Notably there is no rule for "$\wedge$", the *both* constructor. We deal with this constructor by observing that all interleavings of the actions in $\mathscr{G}_1$ and $\mathscr{G}_2$ give global types $\mathscr{G}$ such that $\mathscr{G} \leqslant \mathscr{G}_1 \wedge \mathscr{G}_2$, and therefore we can use the subsumption rule to eliminate every occurrence of $\wedge$. For example, to project the global type $\mathtt{p} \xrightarrow{a} \mathtt{q} \wedge \mathtt{r} \xrightarrow{b} \mathtt{s}$ we can use $\mathtt{p} \xrightarrow{a} \mathtt{q};\mathtt{r} \xrightarrow{b} \mathtt{s}$: since the two actions that compose both global types have disjoint participants, then the projections of these global types (as well as that of $\mathtt{r} \xrightarrow{b} \mathtt{s};\mathtt{p} \xrightarrow{a} \mathtt{q}$) will have exactly the same set of traces.

Other interesting examples of subsumptions useful for projecting are

$$\mathtt{r} \xrightarrow{b} \mathtt{p};\mathtt{p} \xrightarrow{a} \mathtt{q} \leqslant (\mathtt{p} \xrightarrow{a} \mathtt{q};\mathtt{r} \xrightarrow{b} \mathtt{p}) \vee (\mathtt{r} \xrightarrow{b} \mathtt{p};\mathtt{p} \xrightarrow{a} \mathtt{q}) \tag{4}$$

$$\mathtt{r} \xrightarrow{b} \mathtt{p};(\mathtt{p} \xrightarrow{a} \mathtt{q} \vee \mathtt{p} \xrightarrow{b} \mathtt{q}) \leqslant (\mathtt{r} \xrightarrow{b} \mathtt{p};\mathtt{p} \xrightarrow{a} \mathtt{q}) \vee (\mathtt{r} \xrightarrow{b} \mathtt{p};\mathtt{p} \xrightarrow{b} \mathtt{q}) \tag{5}$$

In (4) the $\leqslant$-larger global type describes the shuffling of two interactions, therefore we can project one particular ordering still preserving completeness. In (5) we take advantage of the flat nature of traces to push the $\vee$ construct where the choice is actually being made.

We are interested in projections without continuations, that is, in judgments of the shape $\{\mathtt{p}_i : \mathsf{end} \mid \mathtt{p}_i \in \mathscr{G}\} \vdash \mathscr{G} \rhd \Delta$ (where $\mathtt{p} \in \mathscr{G}$ means that $\mathtt{p}$ occurs in $\mathscr{G}$) which we shortly will write as

$$\vdash \mathscr{G} \rhd \Delta$$

The mere existence of a projection does not mean that the projection behaves as specified in the global type. For example, we have

$$\vdash \mathtt{p} \xrightarrow{a} \mathtt{q};\mathtt{r} \xrightarrow{a} \mathtt{s} \rhd \{\mathtt{p} : \mathtt{q}!a.\mathsf{end},\ \ \mathtt{q} : \mathtt{p}?a.\mathsf{end},\ \ \mathtt{r} : \mathtt{s}!a.\mathsf{end},\ \ \mathtt{s} : \mathtt{r}?a.\mathsf{end}\}$$

but the projection admits also the trace $\mathtt{r} \xrightarrow{a} \mathtt{s};\mathtt{p} \xrightarrow{a} \mathtt{q}$ which is not allowed by the global type. Clearly the problem resides in the global type, which tries to impose a temporal ordering between interactions involving disjoint participants. What we want, in accordance with the traces of a global type $\mathscr{G}_1;\mathscr{G}_2$, is that no interaction in $\mathscr{G}_2$ can be completed before all the interactions in $\mathscr{G}_1$ are completed. More in details:

- an action $\pi \xrightarrow{a} \mathtt{p}$ is completed when the participant $\mathtt{p}$ has received the message a from all the participants in π;
- if $\varphi;\pi \xrightarrow{a} \mathtt{p};\pi' \xrightarrow{b} \mathtt{p}';\psi$ is a trace of a global type, then either the action $\pi' \xrightarrow{b} \mathtt{p}'$ cannot be completed before the action $\pi \xrightarrow{a} \mathtt{p}$ is completed, or they can be executed in any order. The first case requires $\mathtt{p}$ to be either $\mathtt{p}'$ or a member of π', in the second case the set of traces must also contain the trace $\varphi;\pi' \xrightarrow{b} \mathtt{p}';\pi \xrightarrow{a} \mathtt{p};\psi$.

This leads us to the following definition of well-formed global type.

Definition 4.2 (well-formed global type). *We say that a set of traces L is* well formed *if $\varphi;\pi \xrightarrow{a} \mathtt{p};\pi' \xrightarrow{b} \mathtt{p}';\psi \in L$ implies either $\mathtt{p} \in \pi' \cup \{\mathtt{p}'\}$ or $\varphi;\pi' \xrightarrow{b} \mathtt{p}';\pi \xrightarrow{a} \mathtt{p};\psi \in L$. We say that a global type $\mathscr{G}$ is* well formed *if so is* $\mathrm{tr}(\mathscr{G})$.

It is easy to decide well-formedness of an arbitrary global type $\mathscr{G}$ by building in a standard way the automaton that recognises the language of traces generated by $\mathscr{G}$.

Projectability and well-formedness must be kept separate because it is sometimes necessary to project ill-formed global types. For example, the ill-formed global type $\mathtt{p} \xrightarrow{a} \mathtt{q}; \mathtt{r} \xrightarrow{a} \mathtt{s}$ above is useful to project $\mathtt{p} \xrightarrow{a} \mathtt{q} \wedge \mathtt{r} \xrightarrow{a} \mathtt{s}$ which is well formed.

Clearly, if a global type is projectable (*ie*, $\vdash \mathscr{G} \rhd \Delta$ is derivable) then well-formedness of $\mathscr{G}$ is a necessary condition for the soundness and completeness of its projection (*ie*, for $\Delta \leqslant \mathscr{G}$). It turns out that well-formedness is also a sufficient condition for having soundness and completeness of projections, as stated in the following theorem.

Theorem 4.1. *If $\mathscr{G}$ is well formed and $\vdash \mathscr{G} \rhd \Delta$, then $\Delta \leqslant \mathscr{G}$.*

In summary, if a well-formed global type $\mathscr{G}$ is projectable, then its projection is a *live* projection (it cannot be empty since $\mathrm{tr}(\mathscr{G}) \subseteq \mathrm{tr}(\Delta)^{\circ}$) which is sound and complete wrt $\mathscr{G}$ and, therefore, satisfies the sequentiality, alternativeness, and shuffling properties outlined in the introduction.

We conclude this section by formally characterizing the three kinds of problematic global types we have described earlier. We start from the least severe problem and move towards the more serious ones. Let $L^{\#}$ denote the smallest well-formed set such that $L \subseteq L^{\#}$.

No sequentiality. Assuming that there is no Δ that is both sound and complete for $\mathscr{G}$, it might be the case that we can find a session whose traces are complete for $\mathscr{G}$ and sound for the global type $\mathscr{G}'$ obtained from $\mathscr{G}$ by turning some « ; »'s into « $\wedge$ »'s. This means that the original global type $\mathscr{G}$ is ill formed, namely that it specifies some sequentiality constraints that are impossible to implement. For instance, $\{\mathtt{p} : \mathtt{q}!a.\mathsf{end},\ \mathtt{q} : \mathtt{p}?a.\mathsf{end},\ \mathtt{r} : \mathtt{s}!b.\mathsf{end},\ \mathtt{s} : \mathtt{r}?b.\mathsf{end}\}$ is a complete but not sound session for the ill-formed global type $\mathtt{p} \xrightarrow{a} \mathtt{q}; \mathtt{r} \xrightarrow{b} \mathtt{s}$ (while it is a sound and complete session for $\mathtt{p} \xrightarrow{a} \mathtt{q} \wedge \mathtt{r} \xrightarrow{b} \mathtt{s}$). We characterize the global types $\mathscr{G}$ that present this error as:

$$\nexists \Delta : \Delta \leqslant \mathscr{G} \text{ and } \exists \Delta : \mathrm{tr}(\mathscr{G}) \subseteq \mathrm{tr}(\Delta) \subseteq \mathrm{tr}(\mathscr{G})^{\#}.$$

No knowledge for choice. In this case every session Δ that is complete for $\mathscr{G}$ invariably exhibits some interactions that are not allowed by $\mathscr{G}$ despite the fact that $\mathscr{G}$ is well formed. This happens when the global type specifies alternative behaviors, but some participants do not have enough information to behave consistently. For example, the global type

$$(\mathtt{p} \xrightarrow{a} \mathtt{q}; \mathtt{q} \xrightarrow{a} \mathtt{r}; \mathtt{r} \xrightarrow{a} \mathtt{p}) \vee (\mathtt{p} \xrightarrow{b} \mathtt{q}; \mathtt{q} \xrightarrow{a} \mathtt{r}; \mathtt{r} \xrightarrow{b} \mathtt{p})$$

mandates that $\mathtt{r}$ should send either a or b in accordance with the message that $\mathtt{p}$ sends to $\mathtt{q}$. Unfortunately, $\mathtt{r}$ has no information as to which message $\mathtt{q}$ has received, because $\mathtt{q}$ notifies $\mathtt{r}$ with an a message in both branches. A complete implementation of this global type is

$$\begin{array}{l}\{\mathtt{p} : \mathtt{q}!a.(\mathtt{r}?a.\mathsf{end} + \mathtt{r}?b.\mathsf{end}) \oplus \mathtt{q}!b.(\mathtt{r}?a.\mathsf{end} + \mathtt{r}?b.\mathsf{end}),\\ \ \mathtt{q} : \mathtt{p}?a.\mathtt{r}!a.\mathsf{end} + \mathtt{p}?b.\mathtt{r}!a.\mathsf{end}, \mathtt{r} : \mathtt{q}?a.(\mathtt{q}!a.\mathsf{end} \oplus \mathtt{q}!b.\mathsf{end})\}\end{array}$$

which also produces the traces $\mathtt{p} \xrightarrow{a} \mathtt{q}; \mathtt{q} \xrightarrow{a} \mathtt{r}; \mathtt{r} \xrightarrow{b} \mathtt{p}$ and $\mathtt{p} \xrightarrow{b} \mathtt{q}; \mathtt{q} \xrightarrow{a} \mathtt{r}; \mathtt{r} \xrightarrow{a} \mathtt{p}$. We characterize this error as:

$$\nexists \Delta : \mathrm{tr}(\mathscr{G}) \subseteq \mathrm{tr}(\Delta) \subseteq \mathrm{tr}(\mathscr{G})^{\#} \text{ and } \exists \Delta : \mathrm{tr}(\mathscr{G}) \subseteq \mathrm{tr}(\Delta).$$

No knowledge, no choice. In this case we cannot find a complete session Δ for $\mathscr{G}$. This typically means that $\mathscr{G}$ specifies some combination of incompatible behaviors. For example, the global type $\mathtt{p} \xrightarrow{a} \mathtt{q} \vee \mathtt{q} \xrightarrow{a} \mathtt{p}$ implies an agreement between p and q for establishing who is entitled to send the *a* message. In a distributed environment, however, there can be no agreement without a previous message exchange. Therefore, we can either have a sound but not complete session that implements just one of the two branches (for example, $\{\mathtt{p} : \mathtt{q}!a.\mathsf{end}, \mathtt{q} : \mathtt{p}?a.\mathsf{end}\}$) or a session like $\{\mathtt{p} : \mathtt{q}!a.\mathtt{q}?a.\mathsf{end}, \mathtt{q} : \mathtt{p}?a.\mathtt{p}!a.\mathsf{end}\}$ where both p and q send their message but which is neither sound nor complete. We characterize this error as:

$$\nexists\Delta : \mathrm{tr}(\mathscr{G}) \subseteq \mathrm{tr}(\Delta)\,.$$

5 Algorithmic Projection

We now attack the problem of *computing* the projection of a global type. We are looking for an algorithm that "implements" the projection rules of Section 4, that is, that given a session continuation Δ and a global type $\mathscr{G}$, produces a projection Δ' such that $\Delta \vdash \mathscr{G} : \Delta'$. In other terms this algorithm must be sound with respect to the semantic projection (completeness, that is, returning a projection for every global type that is semantically projectable, seems out of reach, yet).

The deduction system in Table 3 is not algorithmic because of two rules: the rule (SP-ITERATION) that does not satisfy the subformula property since the context Δ used in the premises is the result of the conclusion; the rule (SP-SUBSUMPTION) since it is neither syntax-directed (it is defined for a generic $\mathscr{G}$) nor does it satisfy the subformula property (the $\mathscr{G}'$ and Δ'' in the premises are not uniquely determined).[1] The latter rule can be expressed as the composition of the two rules

$$\text{(SP-SUBSUMPTIONG)}\ \frac{\Delta \vdash \mathscr{G}' \triangleright \Delta' \qquad \mathscr{G}' \leqslant \mathscr{G}}{\Delta \vdash \mathscr{G} \triangleright \Delta'} \qquad\qquad \text{(SP-SUBSUMPTIONS)}\ \frac{\Delta \vdash \mathscr{G} \triangleright \Delta' \qquad \Delta'' \leqslant \Delta'}{\Delta \vdash \mathscr{G} \triangleright \Delta''}$$

Splitting (SP-SUBSUMPTION) into (SP-SUBSUMPTIONG) and (SP-SUBSUMPTIONS) is useful to explain the following problems we have to tackle to define an algorithm:

1. How to eliminate (SP-SUBSUMPTIONS), the subsumption rule for sessions.
2. How to define an algorithmic version of (SP-ITERATION), the rule for Kleene star.
3. How to eliminate (SP-SUBSUMPTIONG), the subsumption rule for global types.

We address each problem in order and discuss the related rule in the next sections.

5.1 Session Subsumption

Rule (SP-SUBSUMPTIONS) is needed to project alternative branches and iterations (a loop is an unbound repetition of alternatives, each one starting with the choice of

[1] The rule (SP-ALTERNATIVE) is algorithmic: in fact there is a finite number of participants in the two sessions of the premises and at most one of them can have different session types starting with outputs.

whether to enter the loop or to skip it): each participant different from the one that actively chooses must behave according to the same session type in both branches. More precisely, to project $\mathscr{G}_1 \vee \mathscr{G}_2$ the rule (SP-ALTERNATIVE) requires to deduce for $\mathscr{G}_1$ and $\mathscr{G}_2$ the same projection: if different projections are deduced, then they must be previously subsumed to a common lower bound. The algorithmic projection of an alternative (see the corresponding rule in Table 4) allows premises with two different sessions, but then *merges* them. Of course not every pair of projections is mergeable. Intuitively, two projections are mergeable if so are the behaviors of each participant. This requires participants to respect a precise behavior: as long as a participant cannot determine in which branch (*ie*, projection) it is, then it must do the same actions in all branches (*ie*, projections). For example, to project $\mathscr{G} = (\mathtt{p} \xrightarrow{a} \mathtt{q};\mathtt{r} \xrightarrow{c} \mathtt{q};\dots) \vee (\mathtt{p} \xrightarrow{b} \mathtt{q};\mathtt{r} \xrightarrow{c} \mathtt{q};\dots)$ we project each branch separately obtaining $\Delta_1 = \{\mathtt{p} : \mathtt{q}!a\dots, \mathtt{q} : \mathtt{p}?a.\mathtt{r}?c\dots, \mathtt{r} : \mathtt{q}!c\dots\}$ and $\Delta_2 = \{\mathtt{p} : \mathtt{q}!b\dots, \mathtt{q} : \mathtt{p}?b.\mathtt{r}?c\dots, \mathtt{r} : \mathtt{q}!c\dots\}$. Since $\mathtt{p}$ performs the choice, in the projection of $\mathscr{G}$ we obtain $\mathtt{p} : \mathtt{q}!a\dots \oplus \mathtt{q}!b\dots$ and we must merge $\{\mathtt{q} : \mathtt{p}?a.\mathtt{r}?c\dots, \mathtt{r} : \mathtt{q}!c\dots\}$ with $\{\mathtt{q} : \mathtt{p}?b.\mathtt{r}?c\dots, \mathtt{r} : \mathtt{q}!c\dots\}$. Regarding $\mathtt{q}$, observe that it is the receiver of the message from $\mathtt{p}$, therefore it becomes aware of the choice and can behave differently right after the first input operation. Merging its behaviors yields $\mathtt{q} : \mathtt{p}?a.\mathtt{r}?c\dots + \mathtt{p}?b.\mathtt{r}?c\dots$. Regarding $\mathtt{r}$, it has no information as to which choice has been made by $\mathtt{p}$, therefore it must have the same behavior in both branches, as is the case. Since merging is idempotent, we obtain $\mathtt{r} : \mathtt{q}!c\dots$. In summary, *mergeability* of two branches of an "$\vee$" corresponds to the "awareness" of the choice made when branching (see the discussion in Section 4 about the "No knowledge for choice" error), and it is possible when, roughly, each participant performs the same internal choices and disjoint external choices in the two sessions.

Special care must be taken when merging external choices to avoid unexpected interactions that may invalidate the correctness of the projection. To illustrate the problem consider the session types $T = \mathtt{p}?a.\mathtt{q}?b.\mathsf{end}$ and $S = \mathtt{q}?b.\mathsf{end}$ describing the behavior of a participant $\mathtt{r}$. If we let $\mathtt{r}$ behave according to the merge of T and S, which intuitively is the external choice $\mathtt{p}?a.\mathtt{q}?b.\mathsf{end} + \mathtt{q}?b.\mathsf{end}$, it may be possible that the message b from $\mathtt{q}$ is read *before* the message a from $\mathtt{p}$ arrives. Therefore, $\mathtt{r}$ may mistakenly think that it should no longer participate to the session, while there is still a message targeted to $\mathtt{r}$ that will never be read. Therefore, T and S are *incompatible* and it is not possible to merge them safely. On the contrary, $\mathtt{p}?a.\mathtt{p}?b.\mathsf{end}$ and $\mathtt{p}?b.\mathsf{end}$ are compatible and can be merged to $\mathtt{p}?a.\mathtt{p}?b.\mathsf{end} + \mathtt{p}?b.\mathsf{end}$. In this case, since the order of messages coming from the same sender is preserved, it is not possible for $\mathtt{r}$ to read the b message coming from $\mathtt{p}$ before the a message, assuming that $\mathtt{p}$ sent both. More formally:

Definition 5.1 (compatibility). *We say that an input* $\mathtt{p}?a$ *is* compatible *with a session type* T *if either* (i) $\mathtt{p}?a$ *does not occur in* T, *or* (ii) $T = \bigoplus_{i\in I} \mathtt{p}_i!a_i.T_i$ *and* $\mathtt{p}?a$ *is compatible with* T_i *for all* $i \in I$, *or* (iii) $T = \sum_{i\in I} \pi_i?a_i.T_i$ *and for all* $i \in I$ *either* $\mathtt{p} \in \pi_i$ *and* $a \neq a_i$ *or* $\mathtt{p} \notin \pi_i$ *and* $\mathtt{p}?a$ *is compatible with* T_i.

We say that an input $\pi?a$ *is* compatible *with a session type* T *if* $\mathtt{p}?a$ *is compatible with* T *for some* $\mathtt{p} \in \pi$.

Finally, $T = \sum_{i\in I} \pi_i?a_i.T_i + \sum_{j\in J} \pi_j?a_j.T_j$ *and* $S = \sum_{i\in I} \pi_i?a_i.S_i + \sum_{h\in H} \pi_h?a_h.S_h$ *are* compatible *if* $\pi_j?a_j$ *is compatible with* S *for all* $j \in J$ *and* $\pi_h?a_h$ *is compatible with* T *for all* $h \in H$.

Table 4. Rules for algorithmic projection

(AP-SKIP)

$$\Delta \vdash_{\mathsf{a}} \mathsf{skip} \rhd \Delta$$

(AP-ACTION)

$$\{\mathtt{p}_i : T_i\}_{i\in I} \uplus \{\mathtt{p} : T\} \uplus \Delta \vdash_{\mathsf{a}} \{\mathtt{p}_i\}_{i\in I} \xrightarrow{a} \mathtt{p} \rhd \{\mathtt{p}_i : \mathtt{p}!a.T_i\}_{i\in I} \uplus \{\mathtt{p} : \{\mathtt{p}_i\}_{i\in I}?a.T\} \uplus \Delta$$

(AP-SEQUENCE)

$$\frac{\Delta \vdash_{\mathsf{a}} \mathscr{G}_2 \rhd \Delta' \qquad \Delta' \vdash_{\mathsf{a}} \mathscr{G}_1 \rhd \Delta''}{\Delta \vdash_{\mathsf{a}} \mathscr{G}_1 ; \mathscr{G}_2 \rhd \Delta''}$$

(AP-ALTERNATIVE)

$$\frac{\Delta \vdash_{\mathsf{a}} \mathscr{G}_1 \rhd \{\mathtt{p} : T_1\} \uplus \Delta_1 \qquad \Delta \vdash_{\mathsf{a}} \mathscr{G}_2 \rhd \{\mathtt{p} : T_2\} \uplus \Delta_2}{\Delta \vdash_{\mathsf{a}} \mathscr{G}_1 \vee \mathscr{G}_2 \rhd \{\mathtt{p} : T_1 \oplus T_2\} \uplus (\Delta_1 \bowtie \Delta_2)}$$

(AP-ITERATION)

$$\frac{\{\mathtt{p} : X\} \uplus \{\mathtt{p}_i : X_i\}_{i\in I} \vdash_{\mathsf{a}} \mathscr{G} \rhd \{\mathtt{p} : S\} \uplus \{\mathtt{p}_i : S_i\}_{i\in I}}{\{\mathtt{p} : T\} \uplus \{\mathtt{p}_i : T_i\}_{i\in I} \uplus \Delta \vdash_{\mathsf{a}} \mathscr{G}^* \rhd \{\mathtt{p} : \mathtt{rec}\, X.(T \oplus S)\} \uplus \{\mathtt{p}_i : \mathtt{rec}\, X_i.(T_i \bowtie S_i)\}_{i\in I} \uplus \Delta}$$

The merge operator just connects sessions with the *same* output guards by internal choices and with *compatible* input guards by external choices:

Definition 5.2 (merge). *The* merge *of T and S, written $T \bowtie S$, is defined coinductively and by cases on the structure of T and S thus:*

- *if $T = S = \mathsf{end}$, then $T \bowtie S = \mathsf{end}$;*
- *if $T = \bigoplus_{i\in I} \mathtt{p}_i!a_i.T_i$ and $S = \bigoplus_{i\in I} \mathtt{p}_i!a_i.S_i$, then $T \bowtie S = \bigoplus_{i\in I} \mathtt{p}_i!a_i.(T_i \bowtie S_i)$;*
- *if $T = \sum_{i\in I} \pi_i?a_i.T_i + \sum_{j\in J} \pi_j?a_j.T_j$ and $S = \sum_{i\in I} \pi_i?a_i.S_i + \sum_{h\in H} \pi_h?a_h.S_h$ are compatible, then $T \bowtie S = \sum_{i\in I} \pi_i?a_i.(T_i \bowtie S_i) + \sum_{j\in J} \pi_j?a_j.T_j + \sum_{h\in H} \pi_h?a_h.S_h$.*

We extend merging to sessions so that $\Delta \bowtie \Delta' = \{\mathtt{p} : T \bowtie S \mid \mathtt{p} : T \in \Delta \;\&\; \mathtt{p} : S \in \Delta'\}$.

Rules (AP-ALTERNATIVE) and (AP-ITERATION) of Table 4 are the algorithmic versions of (SP-ALTERNATIVE) and (SP-ITERATION), but instead of relying on subsumption they use the merge operator to compute common behaviors.

The merge operation is a sound but incomplete approximation of session subsumption insofar as the merge of two sessions can be undefined even though the two sessions completed with the participant that makes the decision have a common lower bound according to $\leqslant$. This implies that there are global types which can be semantically but not algorithmically projected. Take for example $\mathscr{G}_1 \vee \mathscr{G}_2$ where $\mathscr{G}_1 = \mathtt{p} \xrightarrow{a} \mathtt{r}; \mathtt{r} \xrightarrow{a} \mathtt{p}; \mathtt{p} \xrightarrow{a} \mathtt{q}; \mathtt{q} \xrightarrow{b} \mathtt{r}$ and $\mathscr{G}_2 = \mathtt{p} \xrightarrow{b} \mathtt{q}; \mathtt{q} \xrightarrow{b} \mathtt{r}$. The behavior of $\mathtt{r}$ in $\mathscr{G}_1$ and $\mathscr{G}_2$ respectively is $T = \mathtt{p}?a.\mathtt{p}!a.\mathtt{q}?b.\mathsf{end}$ and $S = \mathtt{q}?b$. Then we see that $\mathscr{G}_1 \vee \mathscr{G}_2$ is semantically projectable, for instance by inferring the behavior $T + S$ for $\mathtt{r}$. However, T and S are incompatible and $\mathscr{G}_1 \vee \mathscr{G}_2$ is not algorithmically projectable. The point is that the $\leqslant$ relation on projections has a comprehensive perspective of the *whole* session and "realizes" that, if $\mathtt{p}$ initially chooses to send a, then $\mathtt{r}$ will not receive a b message coming from $\mathtt{q}$ until $\mathtt{r}$ has sent a to $\mathtt{p}$. The merge operator, on the other hand, is defined locally on pairs of session types and ignores that the a message that $\mathtt{r}$ sends to $\mathtt{p}$ is used to enforce the arrival of the b message from $\mathtt{q}$ to $\mathtt{r}$ only afterwards. For this reason it conservatively declares T and S incompatible, making $\mathscr{G}_1 \vee \mathscr{G}_2$ impossible to project algorithmically.

5.2 Projection of Kleene Star

Since an iteration $\mathscr{G}^*$ is intuitively equivalent to $\mathsf{skip} \vee \mathscr{G};\mathscr{G}^*$ it comes as no surprise that the algorithmic rule (AP-ITERATION) uses the merge operator. The use of recursion variables for continuations is also natural: in the premise we project $\mathscr{G}$ taking recursion variables as session types in the continuation; the conclusion projects $\mathscr{G}^*$ as the choice between exiting and entering the loop. There is, however, a subtle point in this rule that may go unnoticed: although in the premises of (AP-ITERATION) the only actions and roles taken into account are those occurring in $\mathscr{G}$, in its conclusion the projection of $\mathscr{G}^*$ may require a continuation that includes actions and roles that precede $\mathscr{G}^*$. The point can be illustrated by the global type

$$(\mathtt{p} \xrightarrow{a} \mathtt{q}; (\mathtt{p} \xrightarrow{b} \mathtt{q})^*)^*; \mathtt{p} \xrightarrow{c} \mathtt{q}$$

where $\mathtt{p}$ initially decides whether to enter the outermost iteration (by sending a) or not (by sending c). If it enters the iteration, then it eventually decides whether to also enter the innermost iteration (by sending b), whether to repeat the outermost one (by sending a), or to exit both (by sending c). Therefore, when we project $(\mathtt{p} \xrightarrow{b} \mathtt{q})^*$, we must do it in a context in which both $\mathtt{p} \xrightarrow{c} \mathtt{q}$ and $\mathtt{p} \xrightarrow{a} \mathtt{q}$ are possible, that is a continuation of the form $\{\mathtt{p} : \mathtt{q}!a \ldots \oplus \mathtt{q}!c.\mathsf{end}\}$ even though no a is sent by an action (syntactically) following $(\mathtt{p} \xrightarrow{b} \mathtt{q})^*$. For the same reason, the projection of $(\mathtt{p} \xrightarrow{b} \mathtt{q})^*$ in $(\mathtt{p} \xrightarrow{a} \mathtt{q}; \mathtt{p} \xrightarrow{a} \mathtt{r}; (\mathtt{p} \xrightarrow{b} \mathtt{q})^*)^*; \mathtt{p} \xrightarrow{c} \mathtt{q}; \mathtt{q} \xrightarrow{c} \mathtt{r}$ will need a recursive session type for $\mathtt{r}$ in the continuation.

5.3 Global Type Subsumption

Elimination of global type subsumption is the most difficult problem when defining the projection algorithm. While in the case of sessions the definition of the merge operator gives us a sound—though not complete—tool that replaces session subsumption in very specific places, we do not have such a tool for global type containment. This is unfortunate since global type subsumption is necessary to project several usage patterns (see for example the inequations (4) and (5)), but most importantly it is the only way to eliminate $\wedge$-types (neither the semantic nor the algorithmic deduction systems have projection rules for $\wedge$). The minimal facility that a projection algorithm should provide is to feed the algorithmic rules with all the variants of a global type obtained by replacing occurrences of $\mathscr{G}_1 \wedge \mathscr{G}_2$ by either $\mathscr{G}_1;\mathscr{G}_2$ or $\mathscr{G}_2;\mathscr{G}_1$. Unfortunately, this is not enough to cover all the occurrences in which rule (SP-SUBSUMPTIONG) is necessary. Indeed, while $\mathscr{G}_1;\mathscr{G}_2$ and $\mathscr{G}_2;\mathscr{G}_1$ are in many cases projectable (for instance, when $\mathscr{G}_1$ and $\mathscr{G}_2$ have distinct roles and are both projectable), there exist $\mathscr{G}_1$ and $\mathscr{G}_2$ such that $\mathscr{G}_1 \wedge \mathscr{G}_2$ is projectable only by considering a clever interleaving of the actions occurring in them. Consider for instance $\mathscr{G}_1 = (\mathtt{p} \xrightarrow{a} \mathtt{q}; \mathtt{q} \xrightarrow{c} \mathtt{s}; \mathtt{s} \xrightarrow{e} \mathtt{q}) \vee (\mathtt{p} \xrightarrow{b} \mathtt{r}; \mathtt{r} \xrightarrow{d} \mathtt{s}; \mathtt{s} \xrightarrow{f} \mathtt{r})$ and $\mathscr{G}_2 = \mathtt{r} \xrightarrow{g} \mathtt{s}; \mathtt{s} \xrightarrow{h} \mathtt{r}; \mathtt{s} \xrightarrow{i} \mathtt{q}$. The projection of $\mathscr{G}_1 \wedge \mathscr{G}_2$ from the environment $\{\mathtt{q} : \mathtt{p}!a.\mathsf{end}, \mathtt{r} : \mathtt{p}!b.\mathsf{end}\}$ can be obtained only from the interleaving $\mathtt{r} \xrightarrow{g} \mathtt{s}; \mathscr{G}_1; \mathtt{s} \xrightarrow{h} \mathtt{r}; \mathtt{s} \xrightarrow{i} \mathtt{q}$. The reason is that $\mathtt{q}$ and $\mathtt{r}$ receive messages only in one of the two branches

of the $\vee$, so we need to compute the $\bigwedge\!\!\!\bigwedge$ of their types in these branches with their types in the continuations. The example shows that to project $\mathscr{G}_1 \wedge \mathscr{G}_2$ it may be necessary to arbitrarily decompose one or both of $\mathscr{G}_1$ and $\mathscr{G}_2$ to find the particular interleaving of actions that can be projected. As long as $\mathscr{G}_1$ and $\mathscr{G}_2$ are finite (no non-trivial iteration occurs in them), we can use a brute force approach and try to project all the elements in their shuffle, since there are only finitely many of them. In general —*ie*, in presence of iteration— this is not an effective solution. However, we conjecture that even in the presence of infinitely many traces one may always resort to the finite case by considering only zero, one, and two unfoldings of starred global types. To give a rough idea of the intuition supporting this conjecture consider the global type $\mathscr{G}^* \wedge \mathscr{G}'$: its projectability requires the projectability of $\mathscr{G}'$ (since $\mathscr{G}$ can be iterated zero times), of $\mathscr{G} \wedge \mathscr{G}'$ (since $\mathscr{G}$ can occur only once) and of $\mathscr{G};\mathscr{G}$ (since the number of occurences of $\mathscr{G}$ is unbounded). It is enough to require also that either $\mathscr{G};(\mathscr{G} \wedge \mathscr{G}')$ or $(\mathscr{G} \wedge \mathscr{G}');\mathscr{G}$ can be projected, since then the projectability of either $\mathscr{G}^n;(\mathscr{G} \wedge \mathscr{G}')$ or $(\mathscr{G} \wedge \mathscr{G}');\mathscr{G}^n$ for an arbitrary n follows (see the appendix in the extended version).

So we can —or, conjecture we can— get rid of all occurences of $\wedge$ operators automatically, without loosing in projectability. However, examples (4) and (5) in Section 4 show that rule (SP-SUBSUMPTIONG) is useful to project also global types in which the $\wedge$-constructor does not occur. A fully automated approach may consider (4) and (5) as right-to-left rewriting rules that, in conjunction with some other rules, form a rewriting system generating a set of global types to be fed to the algorithm of Table 4. The choice of such rewriting rules must rely on a more thorough study to formally characterize the sensible classes of approximations to be used in the algorithms. An alternative approach is to consider a global type $\mathscr{G}$ as somewhat underspecified, in that it may allow for a large number of *different* implementations (exhibiting *different* sets of traces) that are sound and complete. Therefore, rule (SP-SUBSUMPTIONG) may be interpreted as a human-assisted refinement process where the designer of a system proposes one particular implementation $\mathscr{G} \leqslant \mathscr{G}'$ of a system described by $\mathscr{G}'$. In this respect it is interesting to observe that checking whether $L_1 \leqslant L_2$ when L_1 and L_2 are regular is decidable, since this is a direct consequence of the decidability of the Parikh equivalence on regular languages [18].

5.4 Properties of the Algorithmic Rules

Every deduction of the algorithmic system given in Table 4, possibly preceeded by the elimination of $\wedge$ and other possible sources of failures by applying the rewritings/heuristics outlined in the previous subsection, induces a similar deduction using the rules for semantic projection (Table 3).

Theorem 5.1. *If* $\vdash_{\mathsf{a}} \mathscr{G} \rhd \Delta$*, then* $\vdash \mathscr{G} \rhd \Delta$*.*

As a corollary of Theorems 4.1 and 5.1, we immediately obtain that the projection Δ of $\mathscr{G}$ obtained through the algorithm is sound and complete with respect to $\mathscr{G}$.

6 *k*-Exit Iterations

The syntax of global types (Table 1) includes that of regular expressions and therefore is expressive enough for describing any protocol that follows a regular pattern. Nonetheless, the simple Kleene star prevents us from projecting some useful protocols. To illustrate the point, suppose we want to describe an interaction where two participants $\mathtt{p}$ and $\mathtt{q}$ alternate in a negotiation in which each of them may decide to bail out. On $\mathtt{p}$'s turn, $\mathtt{p}$ sends either a *bailout* message or a *handover* message to $\mathtt{q}$; if a *bailout* message is sent, the negotiation ends, otherwise it continues with $\mathtt{q}$ that behaves in a symmetric way. The global type

$$(\mathtt{p} \xrightarrow{handover} \mathtt{q}; \mathtt{q} \xrightarrow{handover} \mathtt{p})^*; (\mathtt{p} \xrightarrow{bailout} \mathtt{q} \vee \mathtt{p} \xrightarrow{handover} \mathtt{q}; \mathtt{q} \xrightarrow{bailout} \mathtt{p})$$

describes this protocol as an arbitrarily long negotiation that may end in two possible ways, according to the participant that chooses to bail out. This global type cannot be projected because of the two occurrences of the interaction $\mathtt{p} \xrightarrow{handover} \mathtt{q}$, which make it ambiguous whether $\mathtt{p}$ actually chooses to bail out or to continue the negotiation. In general, our projection rules (SP-ITERATION) and (AP-ITERATION) make the assumption that an iteration can be exited in one way only, while in this case there are two possibilities according to which role bails out. This lack of expressiveness of the simple Kleene star used in a nondeterministic setting [17] led researchers to seek for alternative iteration constructs. One proposal is the *k-exit iteration* [2], which is a generalization of the binary Kleene star and has the form

$$(\mathscr{G}_1, \dots, \mathscr{G}_k)^{\,k*}\,(\mathscr{G}'_1, \dots, \mathscr{G}'_k)$$

indicating a loop consisting of k subsequent phases $\mathscr{G}_1, \dots, \mathscr{G}_k$. The loop can be exited just before each phase through the corresponding $\mathscr{G}'_i$. Formally, the traces of the k-exit iteration can be expressed thus:

$$\mathrm{tr}((\mathscr{G}_1, \dots, \mathscr{G}_k)^{\,k*}\,(\mathscr{G}'_1, \dots, \mathscr{G}'_k)) \stackrel{\text{def}}{=} \mathrm{tr}((\mathscr{G}_1; \dots; \mathscr{G}_k)^*; (\mathscr{G}'_1 \vee \mathscr{G}_1; \mathscr{G}'_2 \vee \dots \vee \mathscr{G}_1; \dots; \mathscr{G}_{k-1}; \mathscr{G}'_k))$$

and, for example, the negotiation above can be represented as the global type

$$(\mathtt{p} \xrightarrow{handover} \mathtt{q}, \mathtt{q} \xrightarrow{handover} \mathtt{p})^{\,2*}\,(\mathtt{p} \xrightarrow{bailout} \mathtt{q}, \mathtt{q} \xrightarrow{bailout} \mathtt{p}) \tag{6}$$

while the unary Kleene star $\mathscr{G}^*$ can be encoded as $(\mathscr{G})^{\,1*}\,(\mathsf{skip})$.

In our setting, the advantage of the k-exit iteration over the Kleene star is that it syntactically identifies the k points in which a decision is made by a participant of a multi-party session and, in this way, it enables more sophisticated projection rules such as that in Table 5. Albeit intimidating, rule (SP-k-EXIT ITERATION) is just a generalization of rule (SP-ITERATION). For each phase i a (distinct) participant $\mathtt{p}_i$ is identified: the participant may decide to exit the loop behaving as S_i or to continue the iteration behaving as T_i. While projecting each phase $\mathscr{G}_i$, the participant $\mathtt{p}_{(i \bmod k)+1}$ that will decide at the next turn is given the continuation $T_{(i \bmod k)+1} \oplus S_{(i \bmod k)+1}$, while the others must behave according to some R_i that is the same for every phase in which

Table 5. Semantic projection of k-exit iteration

(SP-k-EXIT ITERATION)

$$\frac{\begin{array}{c}\Delta \vdash \mathscr{G}'_i \rhd \{\mathtt{p}_i : S_i\} \uplus \{\mathtt{p}_j : R_j\}_{j=1,\dots,i-1,i+1,\dots,k} \uplus \Delta' \;{}^{(i\in\{1,\dots,k\})} \\ \{\mathtt{p}_2 : T_2 \oplus S_2\} \uplus \{\mathtt{p}_i : R_i\}_{i=1,3,\dots,k} \uplus \Delta' \vdash \mathscr{G}_1 \rhd \{\mathtt{p}_1 : T_1\} \uplus \{\mathtt{p}_i : R_i\}_{i=2,\dots,k} \uplus \Delta' \\ \{\mathtt{p}_3 : T_3 \oplus S_3\} \uplus \{\mathtt{p}_i : R_i\}_{i=1,2,4,\dots,k} \uplus \Delta' \vdash \mathscr{G}_2 \rhd \{\mathtt{p}_2 : T_2\} \uplus \{\mathtt{p}_i : R_i\}_{i=1,3,\dots,k} \uplus \Delta' \\ \vdots \\ \{\mathtt{p}_1 : T_1 \oplus S_1\} \uplus \{\mathtt{p}_i : R_i\}_{i=2,\dots,k} \uplus \Delta' \vdash \mathscr{G}_k \rhd \{\mathtt{p}_k : T_k\} \uplus \{\mathtt{p}_i : R_i\}_{i=1,\dots,k-1} \uplus \Delta'\end{array}}{\Delta \vdash (\mathscr{G}_1,\dots,\mathscr{G}_k)^{k*}\,(\mathscr{G}'_1,\dots,\mathscr{G}'_k) \rhd \{\mathtt{p}_1 : T_1 \oplus S_1\} \uplus \{\mathtt{p}_i : R_i\}_{i=2,\dots,k} \uplus \Delta'}$$

they play no active role. Once again, rule (SP-SUBSUMPTION) is required in order to synthesize these behaviors. For example, the global type (6) is projected to

$$\begin{array}{l}\{\mathtt{p} : \mathtt{rec}\, X.(\mathtt{q}!\mathit{handover}.(\mathtt{q}?\mathit{handover}.X + \mathtt{q}?\mathit{bailout}.\mathsf{end}) \oplus \mathtt{q}!\mathit{bailout}.\mathsf{end}), \\ \;\mathtt{q} : \mathtt{rec}\, Y.(\mathtt{p}?\mathit{handover}.(\mathtt{p}!\mathit{handover}.Y \oplus \mathtt{p}!\mathit{bailout}.\mathsf{end}) + \mathtt{p}?\mathit{bailout}.\mathsf{end})\}\end{array}$$

as one expects.

7 Related Work

The formalization and analysis of the relation between a global description of a distributed system and a more machine-oriented description of a set of components that implements it, is a problem that has been studied in several contexts and by different communities. In this context, important properties that are consider are the *verification* that an implementation satisfies the specification, the *implementability* of the specification by automatically producing an implementation from it, and the study of different properties on the specification that can then be transposed to every (possibly automatically produced) implementation satisfying it. In this work we concentrated on the implementability problem, and we tackled it from the "Web service coordination" angle developed by the community that focuses on behavioral types and process algebrae. We are just the latest to attack this problem. So many other communities have been considering it before us that even a sketchy survey has no chance to be exhaustive.

In what follows we compare the "behavioral types/process algebra" approach we adopted, with two alternative approaches studied by important communities with a large amount of different and important contributions, namely the "automata" and "cryptographic protocols" approaches. In the full version of this article the reader will find a deeper survey of these two approaches along with a more complete comparison. In a nutshell, the "automata/model checking" community has probably done the most extensive research on the problem. The paradigmatic global descriptions language they usually refer to are *Message Sequence Charts* (MSC, ITU Z.120 standard) enriched with branching and iteration (which are then called *Message Sequence Graphs* or, as in the Z.120 standard, *High-Level Message Sequence Charts*) and they are usually projected into *Communicating Finite State Machines* (CFM) which form the theoretical core of the *Specification and Description Language* (SDL ITU Z.100 standard). This community has investigated the expressive power of the two formalisms and their properties,

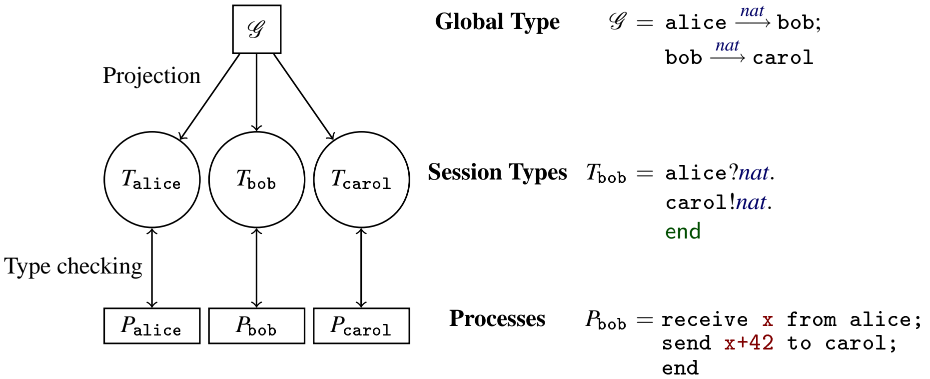


Fig. 1. Global types and multiparty sessions in a nutshell

studied different notions of implementability (but not the notion we studied here which, as far as we know, is original to our work), and several variants of these formalisms especially to deal with the decidability or tractability of the verification of properties, in particular model-checking. The community that works on the formal verification of cryptographic protocols uses MSC as global descriptions, as well, though they are of different nature from the previous ones. In particular, for cryptographic protocols much less emphasis is put on control (branching and iteration have a secondary role) and expressivity, while much more effort is devoted to the description of the messages (these include cryptographic primitives, at least), of properties of the participants, and of local treatment of the messages. The global descriptions are then projected into local descriptions that are more detailed than in the automata approach since they precisely track single values and the manipulations thereof. The verification of properties is finer grained and covers execution scenari that fall outside the global description since the roles described in the global type can be concurrently played by different participants, messages can be intercepted, read, destroyed, and forged and, generally, the communication topology may be changed. Furthermore different executions of the protocol may be not independent as attackers can store and detour information in one execution to use it in a later execution.

Our work springs from the research done to formally describe and verify compositions of Web services. This research has mainly centered on using process algebras to describe and verify visible local behavior of services and just recently (all the references date of the last five years) has started to consider global *choreographic* descriptions of multiple services and the problem of their projection. This yielded the three layered structure depicted in Figure 1 where a global type describing the choreography is projected into a set of session types that are then used to type-check the processes that implement it (as well as guide their implementation). The study thus focuses on defining the relation between the different layers. Implementability is the relation between the first and second layer. Here the important properties are that projection produces systems that are sound and complete with respect to the global description (in the sense stated by Theorem 4.1) and deadlock free (*eg*, we bar out specifications as

$\mathtt{p} \xrightarrow{a} \mathtt{q} \vee \mathtt{p} \xrightarrow{a} \mathtt{r}$ when it has no continuation, since whatever the choice either q or r will be stuck). Typeability is the relation between the second and third layer. Here the important properties are subject reduction (well-typed processes reduce only to well-typed processes) and progress (which in this context implies deadlock freedom).

Although in this work we disregarded the lower layer of processes, it is nevertheless an essential component of this research. In particular, it explains the nature of the messages that characterize this approach, which are *types*. One of the principal aims of this research, thus, is to find the right level of abstraction that must be expressed by types and session types. Consider again Figure 1. The process layer clearly shows the relation between the message received by bob and the one it sends to carol, but this relation (actually, any relation) is abstracted out both in the session and the global type layers. The level of abstraction is greater than that of cryptographic protocols since values are not tracked by global descriptions. Although tracking of values could be partially recovered by resorting to singleton types, there is a particular class of values that deserves special care and whose handling is one of the main future challenges of this research, that is, *channels*. The goal is to include higher order types in global specifications thus enabling the transmission of session channels and therefore the reification of dynamic reconfiguration of session topology. We thus aim at defining reconfiguration in the specification itself, as opposed to the case of cryptographic protocols where the reconfiguration of the communication topology is considered at meta-level for verification purposes. As a matter of fact, this feature has already been studied in the literature. For instance, the extension of WS-CDL [1] with channel passing is studied in [11] (as the automata approach has the MSC as their reference standard, so the Web service-oriented research refers to the WS-CDL standard whose implementability has been studied in [19]); the paper that first introduced global descriptions for session types [9] explicitly mentions channels in messages that can be sent to other participants to open new sessions on them. In our opinion the existing works on session types are deeply syntactic in nature and suffer from the fact that their global types are defined in function of the languages used to define processes and session types. The consequence is that the design choices done in defining session types are amplified in the passage to global types yielding a somewhat unnatural syntax for global types and restrictive conditions devoid of semantic characterizations. Here we preferred to take a step back and to start by defining global descriptions whose restrictions are semantically justified. So we favored a less rich language with few semantically justified features and leave the addition of more advanced features for a later time.

Coming back to the comparison of the three approaches, the Web service-oriented approach shares several features in common with the other two. As for the automata approach we (in the sense of the Web service-oriented research) focus on the expressiveness of the control, the possibility of branching and iterate, and the effective implementability into deadlock-free local descriptions. However the tendency for Web services is to impose syntactic restrictions from the beginning rather than study the general case and then devise appropriate restrictions with the sought properties (in this respects our work and those of Bravetti, Zavattaro and Lanese [6,7,5] are few exceptions in the panorama of the Web service approach). Commonalities with the cryptographic protocol approach are more technical. In particular we share the dynamism of the

communication topology (with the caveat about whether this dynamism is performed at the linguistic or meta-linguistic level) and the robustness with respect to reconfiguration (the projected session types should ensure that well-typed process will be deadlock free even in the presence of multiple interleaved sessions and session delegation, though few works actually enforce this property [3,13]). As for cryptographic protocols, this dynamism is also accounted at level of participants since recent work in session types studies global descriptions of roles that can then be implemented by several different agents [12]. Finally, there are some characteristics specific to our approach such as the exploration of new linguistic features (for instance in this work we introduced actions with multi-senders and encoded multi-receivers) and a pervasive use of compositional deduction systems that we inherit from type theory. We conclude this section with a more in-depth description of the main references in this specific area so as to give a more detailed comparison with our work.

Multiparty session types. Global types were introduced in [9] for dyadic sessions and in [16] for multi-party sessions. Channels are present in the global types of both [9] and [16] while the first also allows messages to have a complex structure. Their presence, however, requires the definition of syntactic restrictions that ensure projectability: channels need to be "well-threaded" (to avoid that the use of different channels disrupts the sequentiality constraints of the specification) and message structures must be used "coherently" in different threads (to assure that a fixed server offers the same services to different clients). We did not include such features in our treatment since we wanted to study the problems of sequentiality (which yielded Definition 4.2 of well-formed global type) and of coherence (which is embodied by the subsession relation whose algorithmic counterpart is the $\wedge\!\!\wedge$ merge operator) in the simplest setting without further complexity induced by extra features. As a consequence of this choice, our merge between session types is a generalization of the merge in [21,12] since we allow inputs from different senders (this is the reason why our compatibility is more demanding than the corresponding notion in [21]).

Another feature we disregarded in the present work is *delegation*. This was introduced in [16] for multi-party sessions and is directly inherited from that of dyadic sessions [15]. A participant can delegate another agent to play his role in a session. This delegation is transparent for all the remaining participant of the session. Delegation is implemented by exchanging channels, *ie*, by allowing higher-order channels. In this way the topology of communications may dynamically evolve.

Our crusade for simplification did not restrict itself to exclude features that seemed inessential or too syntax dependent, but it also used simpler forms of existing constructs. In particular an important design choice was to use Kleene star instead of more expressive recursive global types used in [15,9,16,12]. Replacing the star for the recursion gives us a fair implementation of the projected specification almost for free. Fairness seems to us an important —though neglected by current literature— requirement for multi-party sessions. Without it a session in which a part continuously interacts leaving a second one to starve is perfectly acceptable. This is what happens in all the papers referred in this subsection. Without Kleene star, fairness would be more difficult to enforce. Clearly recursion is more expressive than iteration, even though we can partially bridge this gap using k-exit iterations (Section 6).

Finally, although we aimed at simplifying as much as possible, we still imposed few restrictions that seemed unavoidable. Foremost, the sequentiality condition of Section 4, that is, that any two actions that are bound by a semicolon must always appear in the same order in all traces of (sound and complete) implementations. Surprisingly, in all current literature of multi-party session types we are aware of, just one work [9] enforces the sequential semantics of "$;$". In [9] the sequentiality condition, called *connectedness* is introduced (albeit in a simplified setting since—as in [15,16]— instead of the "$;$" the authors consider the simpler case of prefixed actions) and identified as one of three basic principles for global descriptions under which a sound and complete implementation can be defined. All others (even later) works admit to project, say, $\mathtt{q} \xrightarrow{a} \mathtt{p}; \mathtt{r} \xrightarrow{a} \mathtt{p}$ in implementations in which $\mathtt{p}$ reads from $\mathtt{r}$ before having read from $\mathtt{q}$. While the technical interest of relaxing the sequentiality constraint in the interpretation of the "$;$" operator is clear —it greatly simplifies projectability— we really cannot see any semantically plausible reason to do it.

Of course all this effort of simplification is worth only if it brings clear advantages. First and foremost, our simpler setting allows us to give a semantic justification of the formalism and of the restrictions and the operators we introduced in it. For these reasons many restrictions that are present in other formalisms are pointless in our framework. For instance, two global types whose actions can be interleaved in an arbitrary way (*ie*, composed by $\wedge$ in our calculus) can share common participants in our global types, while in the work of [9] and [16] (which use the parallel operator for $\wedge$) this is forbidden. So these works fail to project (actually, they reject) protocols as simple as the first line of the example given in the specification (1) in the introduction. Likewise we can have different receivers in a choice like, for example, the case in which two associated buyers wait for a price from a given seller:

$$\mathtt{seller} \xrightarrow{price} \mathtt{buyer1}; \mathtt{buyer1} \xrightarrow{price} \mathtt{buyer2} \vee \mathtt{seller} \xrightarrow{price} \mathtt{buyer2}; \mathtt{buyer2} \xrightarrow{price} \mathtt{buyer1}$$

while such a situation is forbidden in [9,16].

Another situation possible in our setting but forbidden in [9,16,12] is to have different sets of participants for alternatives, such as in the following case where a buyer is notified about a price by the broker or directly by the seller, but in both cases gives an answer to the broker:

$$(\mathtt{seller} \xrightarrow{agency} \mathtt{broker}; \mathtt{broker} \xrightarrow{price} \mathtt{buyer} \vee \mathtt{seller} \xrightarrow{price} \mathtt{buyer}); \mathtt{buyer} \xrightarrow{answer} \mathtt{broker}$$

A similar situation may arise when choosing between repeating or exiting a loop:

$$\begin{array}{l}\mathtt{seller} \xrightarrow{agency} \mathtt{broker}; (\mathtt{broker} \xrightarrow{offer} \mathtt{buyer}; \mathtt{buyer} \xrightarrow{counteroffer} \mathtt{broker})^*;\\ (\mathtt{broker} \xrightarrow{result} \mathtt{seller} \wedge \mathtt{broker} \xrightarrow{result} \mathtt{buyer})\end{array}$$

which is again forbidden in [9,16,12].

The fact of focusing on a core calculus did not stop us from experimenting. On the contrary, having core definitions for global and session types allowed us to explore new linguistic and communication primitives. In particular, an original contribution of our work is the addition of actions with multiple senders and encoded multiple receivers (as explained at the beginning of Section 2). This allows us to express both joins and forks of interactions as shown by the specification (3) given in Section 2.

Choreographies. Global types can be seen as choreographies [1] describing the interaction of some distributed processes connected through a private multiparty session. Therefore, there is a close relationship between our work and [6,7,5], which concern the projection of choreographies into the contracts of their participants. The choreography language in these works essentially coincides with our language of global types and, just like in our case, a choreography is correct if it preserves the possibility to reach a state where all of the involved Web services have successfully terminated. There are some relevant differences though, starting from choreographic interactions that invariably involve exactly one sender and one receiver, while in the present work we allow for multiple senders and we show that this strictly improves the expressiveness of the formalism, which is thus capable of specifying the join of independent activities. Other differences concern the communication model and the projection procedure. In particular, the communication model is synchronous in [6] and based on FIFO buffers associated with each participant of a choreography in [7]. In our model (Section 3) we have a single buffer and we add the possibility for a receiver to specify the participant from which a message is expected. In [6,7,5] the projection procedure is basically an homomorphism from choreographies to the behavior of their participants, which is described by a contract language equipped with parallel composition, while our session types are purely sequential. [6,7] give no conditions to establish which choreographies produce correct projects. In contrast, [5] defines three *connectedness conditions* that guarantee correctness of the projection. The interesting aspect is that these conditions are solely stated on properties of the set of traces of the choreography, while we need the combination of projectability (Table 3) and well-formedness (Definition 4.2). However, the connectedness conditions in [5] impose stricter constraints on alternative choreographies by requiring that the roles in both branches be the same. This forbids the definition of the two global types described just above that involve the `broker` participant. In addition, they need a causal dependency between actions involving the same operation which immediately prevents the projection of recursive behaviors (the choreography language in [5] lacks iteration and thus can only express finite interactions).

Finally, in discussing MSG in the long version of this work we argue that requiring the specification and its projection produce the same set of traces (called *standard implementation* in [14]) seemed overly constraining and advocated a more flexible solution such as the definitions of soundness and completeness introduced here. However it is interesting that Bravetti and Zavattaro in [5] take the opposite viewpoint, and make this relation even more constraining by requiring the relation between a choreography and its projection to be a strong bisimulation.

Other calculi. In this brief overview we focused on works that study the relation between global specifications and local machine-oriented implementations. However in the literature there is an important effort to devise new description paradigms for either global descriptions or local descriptions. In the latter category we want to cite [15,4], while [10] seems a natural candidate in which to project an eventual higher order extension of our global types. For what concerns global descriptions, the Conversation Calculus [8] stands out for the originality of its approach.

8 Conclusion

We think that the design-by-contract approach advocated in [9,16] and expanded in later works is a very reasonable way to implement distributed systems that are correct by construction. In this work we have presented a theory of global types in an attempt of better understanding their properties and their relationship with multi-party session types. We summarize the results of our investigations in the remaining few lines. First of all, we have defined a proper algebra of global types whose operators have a clear meaning. In particular, we distinguish between sequential composition, which models a strictly sequential execution of interactions, and unconstrained composition, which allows the designer to underspecify the order of possibly dependent interactions. The semantics of gobal types is expressed in terms of regular languages. Aside from providing an accessible intuition on the behavior of the system being specified, the most significant consequence is to induce a *fair* theory of multi-party session types where correct sessions preserve the ability to reach a state in which all the participants have successfully terminated. This property is stronger than the usual progress property within the same session that is guaranteed in other works. We claim that eventual termination is both desirable in practice and also technically convenient, because it allows us to easily express the fact that every participant of a session makes progress (this is non-trivial, especially in an asynchronous setting). We have defined two projection methods from global to session types, a semantic and an algorithmic one. The former allows us to reason about *which* are the global types that can be projected, the latter about *how* these types are projected. This allowed us to define three classes of flawed global types and to suggest if and how they can be amended. Most notably, we have characterized the absence of sequentiality solely in terms of the traces of global types, while we have not been able to provide similar trace-based characterizations for the other flaws. Finally, we have defined a notion of completeness relating a global type and its implementation which is original to the best of our knowledge. In other theories we are aware of, this property is either completely neglected or it is stricter, by requiring the equivalence between the traces of the global type and those of the corresponding implementation.

Acknowledgments. We are indebted to several people from the LIAFA lab: Ahmed Bouajjani introduced us to Parikh's equivalence, Olivier Carton explained us subtle aspects of the shuffle operator, Mihaela Sighireanu pointed us several references to global specification formalisms, while Wiesław Zielonka helped us with references on trace semantics. Anca Muscholl helped us on surveying MSC and Martín Abadi and Roberto Amadio with the literature on security protocols (see the long version). Finally, Roberto Bruni gave us serveral useful suggestions to improve the final version of this work. This work was partially supported by the ANR Codex project, by the MIUR Project IPODS, by a visiting researcher grant of the "Fondation Sciences Mathématiques de Paris", and by a visiting professor position of the Université Paris Diderot.

References

1. Web services choreography description language version 1.0. W3C Candidate Recommendation (2005), `http://www.w3.org/TR/ws-cdl-10/`
2. Bergstra, J.A., Bethke, I., Ponse, A.: Process algebra with iteration. Technical Report Report CS-R9314, Programming Research Group, University of Amsterdam (1993)

3. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
4. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
5. Bravetti, M., Lanese, I., Zavattaro, G.: Contract-driven implementation of choreographies. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 1–18. Springer, Heidelberg (2009)
6. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
7. Bravetti, M., Zavattaro, G.: Contract compliance and choreography conformance in the presence of message queues. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 37–54. Springer, Heidelberg (2009)
8. Caires, L., Vieira, H.T.: Conversation types. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 285–300. Springer, Heidelberg (2009)
9. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
10. Castagna, G., Padovani, L.: Contracts for mobile processes. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 211–228. Springer, Heidelberg (2009)
11. Chao, C., Zongyan, Q.: An approach to check choreography with channel passing in WS-CDL. In: Proceedings of ICWS 2008, pp. 700–707. IEEE Computer Society, Los Alamitos (2008)
12. Deniélou, P.-M., Yoshida, N.: Dynamic multirole session types. In: Proceedings of POPL 2011, pp. 435–446 (2011)
13. Dezani-Ciancaglini, M., de'Liguoro, U., Yoshida, N.: On progress for structured communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)
14. Genest, B., Muscholl, A., Peled, D.A.: Message sequence charts. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 537–558. Springer, Heidelberg (2004)
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of POPL 2008, pp. 273–284. ACM Press, New York (2008)
17. Milner, R.: A complete inference system for a class of regular behaviours. Journal of Computer and System Sciences 28(3), 439–466 (1984)
18. Parikh, R.J.: On context-free languages. Journal of the Association for Computing Machinery 13(4), 570–581 (1966)
19. Qiu, Z., Zhao, X., Cai, C., Yang, H.: Towards the theoretical foundation of choreography. In: Proceedings WWW 2007, pp. 973–982. ACM Press, New York (2007)
20. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
21. Yoshida, N., Deniélou, P.-M., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 128–145. Springer, Heidelberg (2010)

Linear-Time and May-Testing in a Probabilistic Reactive Setting

Lucia Acciai*, Michele Boreale, and Rocco De Nicola

Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy
{lucia.acciai,michele.boreale,rocco.denicola}@unifi.it

Abstract. We consider reactive probabilistic labelled transition systems (RPLTS), a model where internal choices are refined by probabilistic choices. In this setting, we study the relationship between linear-time and may-testing semantics, where an angelic view of nondeterminism is taken. Building on the model of *d-trees* of Cleaveland et al., we first introduce a clean model of probabilistic may-testing, based on simple concepts from measure theory. In particular, we define a probability space where statements of the form "*p may pass test o*" naturally correspond to measurable events. We then obtain an observer-independent characterization of the may-testing preorder, based on comparing the probability of *sets* of traces, rather than of *individual* traces. This entails that may-testing is strictly finer than linear-time semantics. Next, we characterize the may-testing preorder in terms of the probability of satisfying safety properties, expressed as languages of infinite *trees* rather than traces. We then identify a significative subclass of RPLTS where linear and may-testing semantics do coincide: these are the *separated* RPLTS, where actions are partitioned into probabilistic and nondeterministic ones, and at each state only one type is available.

Keywords: probabilistic transition systems, linear time, testing equivalence, safety.

1 Introduction

In a classical nondeterministic setting, it is well-known that trace equivalence is totally insensitive to points of choice in time. This makes trace equivalence a *linear*-time semantics, as opposed to the various *branching*-time semantics of van Glabbeek's spectrum [13], ranging from bisimilarity to failure equivalence. The insensitiveness to points of choice makes linear time the ideal framework when one is interested in analyzing properties that can be expressed as prefix-closed sets of traces, like Safety.

In this context, the *testing equivalence* approach [9] is conceptually important, for it provides a clean observational justification of linear time. Indeed, in the setting of CCS and labelled transition systems, trace equivalence does coincide with *may*-testing equivalence, which deems two processes equivalent when no system (observer) running in parallel may possibly note any difference between them (*must*-testing, on the other

* Corresponding author: Lucia Acciai, DSI - Università di Firenze, Viale Morgagni 65, 50134 Firenze. Work partially supported by the EU project ASCENS under the FET open initiative in FP7.

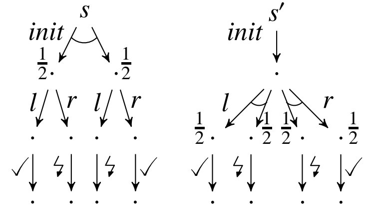
R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 29–43, 2011.
© IFIP International Federation for Information Processing 2011

hand, gives rise to failure semantics, see [8]). However expected, this coincidence result should not be taken for granted in other settings. For example, the result breaks down as soon as one moves from a synchronous to an *a*synchronous communication mechanism (see [3]).

In this paper, we study linear time and may-testing in a setting where internal choices are refined by probabilistic ones, the *reactive probabilistic labelled transition systems* (RPLTS for short, reviewed in Section 3) as described e.g. in [5]. RPLTS's are equivalent to the *Markov Decision Processes* used in probabilistic verification [2]. The reason for choosing this model, which only features external nondeterminism, is that the probabilistic version of linear-time semantics would not make sense in the presence of internal nondeterminism (see Section 3).

The motivation of our study is twofold. First, we are interested in formulating a *clean* probabilistic adaption of the original proposal of testing semantics [9]. This will allow us to check if, or under what assumptions, may-testing still provides an observational justification for (probabilistic) linear-time semantics. Second, with our model at hand, we hope to shed some light in some issues raised by existing probabilistic testing theories, particularly the issue of *overestimation* of probabilities (see e.g. [11] and references therein). These motivations are further discussed below.

The need for a clean model of probabilistic testing can be illustrated by the following example. Consider the two processes s and s' on the right. The two processes and the environment can synchronize over the actions *init*, l, r, ↯ and ✓. In both s and s', after an initial synchronization on *init*, the environment is offered a choice between l and r, then either ✓ or ↯ may be offered, also depending on the previous choice (l or r) of the environment. The interpretation of probabilistic choice is that a fair coin is internally tossed to decide which branch will be made available to the environment. The probability of any individual trace – that is, the probability that any given sequence of synchronizations becomes available to the environment – is the same for both s and s'. Hence the probabilistic linear-time semantics deems s and s' as equivalent. Indeed, following Georgievska and Andova [11], one can argue that not equating s and s' would imply overestimating the probability of success for an external observer, allowing him to observe some points of choice. On the other hand, consider the case of s and s' being two candidate implementations of a safety-critical system; here ↯ represents some catastrophic event, like a system crash. In this case, one is more interested in the chance that, by resolving the internal choices, the mere possibility of ↯ *is ruled out*, whatever the behaviour of the environment and of the scheduler. In other words, one is interested in the probability that *none* of the dangerous traces in the set $\{init \cdot l \cdot ↯,\ init \cdot r \cdot ↯\}$ becomes available. Now, s assigns to this event probability 0, while in the case of s', the choice of the left branch of l *and* of the right branch of r, an event that happens with probability $\frac{1}{4}$, will rule out the possibility of ↯. In this sense, s' might be considered as safer than, and not equivalent to, s - despite the fact that this implies observing a point of choice.



An assessment of this and similar issues should rest on a conceptually clean model of testing. Over the years, many models of probabilistic testing have been put forward by a number of authors [17,21,19], up to and including the recent work by Deng et al. [6,7]. A common paradigm is compose-and-schedule, by which the nondeterminism resulting from the synchronized composition of the process and the observer[1] is resolved by employing - implicitly or explicitly - *schedulers* that make the system fully probabilistic. We see two difficulties with schedulers. First, schedulers can look at the states of the synchronized system, including "ghost" states that result from purely probabilistic choices. This may lead to an unrealistic observation power - the issue of overestimation discussed e.g. by [11]. Some models rectify this by hiding the random choices from the scheduler [4,11]. But then it may become problematic to decide which points of choice should remain observable and which should not (see the example above). Second, the outcome of testing a process with an observer is a *range* of success probabilities, one for each possible scheduler. Comparing two systems on the basis of these ranges is in general awkward. Say one deems system A safer than system B if for each scheduler of A there is a scheduler of B that will lead to crash with a greater probability (see e.g. [7]). The relation "A safer than B" thus established may be of no use in a real-world context, where both the behaviour of the environment and the actual scheduling policy are unpredictable to a large extent. This is of course a drawback when analyzing safety-critical systems. To sum up, in these approaches taking schedulers explicitly into account makes the very concept of *passing a test* awkward, and somehow spoils the clarity of the original testing proposal [9].

In this paper we face these issues from a different perspective and put forward a model that abstracts away from schedulers. The basic idea is that one should first resolve probabilistic choices, then treat the resulting nondeterministic system angelically (if an event may happen, it will happen). Informally, resolving the probabilistic choices out of a process p and of an observer o yields a pair of nondeterministic systems, T and U, with certain associated probabilities, $\Pr(T)$ and $\Pr(U)$. Here, T may or may not satisfy U in a traditional sense. Approximately, the probability that p may pass test o could then be expressed as a sum

$$\Pr(p \text{ may pass } o) \;=\; \sum\nolimits_{T,U:T \text{ may pass } U} \Pr(T)\cdot\Pr(U)\,. \tag{1}$$

We formalize this intuition building on simple concepts from measure theory (reviewed in Section 2) and on the model of *d-trees* of Cleaveland et al. [5] (reviewed in Section 4). In particular, we introduce a probability space where the statements "p may pass o" naturally correspond to measurable events. In general, the sum (1) becomes a proper integral in this space. Going back to the example above, s and s' are distinguished in this model by the observer $o = init.(l.\lightning.\omega \;+\; r.\lightning.\omega)$ (here ω is the success action): indeed, the probability that s may pass o is 1, while it is $\frac{3}{4}$ for s'.

With this model at hand, we investigate the relationships existing among may-testing, linear-time semantics and safety properties. In summary, we offer the following contributions:

[1] Of course, the nondeterminism arising from this composition is always of *internal* type, despite the fact that the system and the process alone may only feature external nondeterminism.

- a clean model of probabilistic may-testing for RPLTS (Subsection 4.1);
- an observer-independent characterization of the may-testing preorder, based on comparing the probability of *sets* of traces, rather than of *individual* traces (Subsection 4.2);
- a comparison of may testing with both linear-time and tree-unfolding semantics (Subsection 4.3);
- a characterization of the may-testing preorder in terms of safety properties, expressed as sets of *infinite trees* rather than traces (Section 5).
- sufficient conditions on RPLTS's and observers guaranteeing that linear and may-testing semantics do coincide. This leads to the class of *separated* RPLTS, where probabilistic and nondeterministic transitions do not mix up (Section 6).

We end the paper with a few considerations on further and related work (Section 7). For lack of space most proofs are omitted in this short version; they can be found in the full version available online [1].

2 Background on Measure Theory

We recall some notions from elementary measure theory. A classical reference is [14]. Let X be any nonempty set. A *sigma-algebra*, or *measurable space*, on X is a pair $(X, \mathcal{A})$ such that $\emptyset \neq \mathcal{A} \subseteq 2^X$ is closed under countable unions and complementation. A *measure* over $(X, \mathcal{A})$ is a function $\mu : \mathcal{A} \to \mathbb{R}^+ \cup \{\infty\}$ satisfying additivity under countable disjoint unions and such that $\mu(\emptyset) = 0$. It is a *probability* measure if $\mu(X) = 1$. The triple $(X, \mathcal{A}, \mu)$ is called *measure space*; if μ is a probability measure, it is also called a *probability space*. Let $(X, \mathcal{A}, \mu_1)$ and $(Y, \mathcal{B}, \mu_2)$ be two measure spaces. The product sigma-algebra $\mathcal{A} \times \mathcal{B}$ is defined to be the sigma-algebra on the cartesian product $X \times Y$ generated by the subsets of the form $A \times B$, with $A \in \mathcal{A}$ and $B \in \mathcal{B}$.

Given two sigma-finite [14] measure spaces $(X, \mathcal{A}, \mu_1)$ and $(Y, \mathcal{B}, \mu_2)$, the product measure $\mu^{\mathcal{A}\times\mathcal{B}}$ is defined to be the unique measure on the measurable space $(X\times Y, \mathcal{A}\times\mathcal{B})$ satisfying the following condition

$$\mu^{\mathcal{A}\times\mathcal{B}}(A \times B) = \mu_1(A) \cdot \mu_2(B) \tag{2}$$

for all $A \in \mathcal{A}$ and $B \in \mathcal{B}$. If μ_1 and μ_2 are probability measures, so is their product $\mu^{\mathcal{A}\times\mathcal{B}}$, hence in this case $(X \times Y, \mathcal{A} \times \mathcal{B}, \mu^{\mathcal{A}\times\mathcal{B}})$ forms a probability space.

3 Reactive Probabilistic Labeled Transition Systems

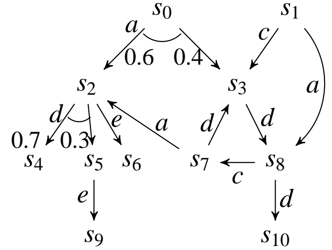
This section introduces the object of our study, Reactive Probabilistic Labeled Transition Systems (RPLTS for short) and the linear-time and tree-unfolding semantics. The relationship between the linear-time and tree-unfolding preorders and the may-testing preorder will be investigated in the next section.

3.1 RPLTS

Let us fix a nonempty set *Act* of actions, ranged over by $a, b, \ldots$. We will let $w, v, \ldots$ range over Act^*. A *Reactive Probabilistic Labeled Transition System* [5,12,18] is basically a finite-branching probabilistic LTS's over the set *Act*. Labels on transitions record the interactions the system may engage in with the environment: at each state, any given action may or may not be available for interaction. Internal nondeterminism is refined by probabilistic choices: if available, a given action can lead to different states depending on probabilities attached to transitions.

Definition 1 (RPLTS). *A* reactive probabilistic labeled transition system *L is a triple (S, δ, P), such that:*

- *S is an at most countable set of states;*
- *$\delta \subseteq S \times Act \times S$ is a transition relation such that for each $s \in S$ there exists a finite number of transitions of the form $(s, \cdot, \cdot)$ in δ (i.e. δ is finitely-branching);*
- *$P : \delta \to (0, 1]$ is a transition probability distribution such that for each $s \in S$ and $a \in Act$: $\sum_{s' : (s,a,s') \in \delta} P(s, a, s') \in \{0, 1\}$;*



A RPLTS can be depicted as a graph, as shown on the right. Let us now introduce some terminology. Let $L = (S, \delta, P)$ be a RPLTS. We will often write $s \xrightarrow{a} s'$ to mean that $(s, a, s') \in \delta$, if the underlying L is clear from the context.

A *computation* of L is any sequence σ of the form $s_0 a_1 s_1 a_2 \cdots a_n s_n \in S \cdot (Act \cdot S)^*$, where $n \geq 0$ and for each $0 \leq i < n$ it holds that $s_i \xrightarrow{a_{i+1}} s_{i+1}$. We will denote by $\mathrm{fst}(\sigma)$ and $\mathrm{lst}(\sigma)$, respectively, the initial and the final state of σ and by $\lambda(\sigma)$ the sequence of labels in σ, that is $\lambda(\sigma) = a_1 a_2 \cdots a_n$. We define the *weight* of σ as $\mathrm{wt}(\sigma) \triangleq \Pi_{i=0}^{n-1} P(s_i, a_{i+1}, s_{i+1})$. Let us fix a RPLTS $L = (S, \delta, P)$ and any state s of S. In what follows, we will denote by $\mathfrak{C}^L$ the set of all computations over L, and by $\mathfrak{C}^L_s$ the set of all computations σ over L such that $\mathrm{fst}(\sigma) = s$.

A computation σ' is said to be a *prefix* of σ if σ' is a prefix of σ as a string. A set of computations $D \subseteq \mathfrak{C}^L$ is said to be *prefix-closed* if for every $\sigma \in D$ and σ' prefix of σ, $\sigma' \in D$. A set of computations $D \subseteq \mathfrak{C}^L$ is said to be *deterministic* if whenever $\sigma, \sigma' \in D$, with $\sigma = \sigma'' as$ and $\sigma' = \sigma'' a' s'$, then either $a \neq a'$ or $s = s'$.

Definition 2 (d-trees). *Let L be a* RPLTS. *Then $\emptyset \neq D \subseteq \mathfrak{C}^L$ is a* d-tree *if the following hold:*

(1) *there is an $s \in S$ such that $D \subseteq \mathfrak{C}^L_s$;* (2) *$D$ is prefix-closed;* (3) *D is deterministic.*

We say that a d-tree D of L is *rooted at* s if $D \subseteq \mathfrak{C}^L_s$ and let $\mathcal{T}^L$ and $\mathcal{T}^L_s$ denote, respectively all d-trees of L and all d-trees of L rooted at s. We will write $\mathcal{F}^L$ for the set of all finite d-trees of L and $\mathcal{F}^L_s$ for the subset of those rooted at s. Finally, we define the weight of a $D \in \mathcal{F}^L_s$ as

$$\mathrm{wt}(D) \triangleq \prod\nolimits_{\sigma as \in D} P(\mathrm{lst}(\sigma), a, s)\,.$$

Note that if D is given as the prefix-closure of some computation σ, then $\mathrm{wt}(D) = \mathrm{wt}(\sigma)$. Finally, given any d-tree $D \subseteq \mathfrak{C}_s^L$, we set $\lambda(D) \triangleq \{\lambda(\sigma) \mid \sigma \in D\}$ and use $D \stackrel{w}{\rightarrow}$ as an abbreviation of $w \in \lambda(D)$.

3.2 Linear-Time Semantics of RPLTS

Definition 3 (probabilistic linear-time preorder). *Let L be a* RPLTS. *For any state s, the function* $f_s^L : Act^* \rightarrow [0,1]$ *is defined thus*

$$\textit{for each } w \in Act^*, \qquad f_s^L(w) \triangleq \sum\nolimits_{\sigma \in \mathfrak{C}_s^L \,:\, \lambda(\sigma)=w} \mathrm{wt}(\sigma)\,. \tag{3}$$

For any two states s and s′, we write $s \leq_{\mathrm{lin}} s'$ *if and only if for each* $w \in Act^*$, $f_s^L(w) \leq f_{s'}^L(w)$.

Note that the sum in (3) is finite, as L is finitely branching. Functions of the type $Act^* \rightarrow \mathbb{K}$ are classically known as *formal power series* in Automata Theory: they represent a generalization of the set-theoretic notion of language to a setting where weights of transitions are not just 0/1 (absence/presence), but elements of a semiring, $\mathbb{K}$ (in our case, the reals). In our scenario, a natural interpretation of "$f_s^L(w) = p$" is that, starting at s, with probability p a sequence of synchronizations along w will be available to an observer. Note that when applied to general PLTS[2], also featuring internal nondeterminism, this definition would not make sense: indeed, one might end up having $f_s^L(w) > 1$.

3.3 Tree-Unfolding Semantics of RPLTS

Some terminology on trees is in order. A *tree* θ over Act is a nonempty, prefix-closed subset of Act^*. In what follows, we shall call $\mathfrak{T}^f$ the set of finite trees over Act^* and use the letter t to range over finite trees.

Definition 4 (probabilistic tree-unfolding preorder). *Let L be a* RPLTS. *For any state s, the function* $\varphi_s^L : \mathfrak{T}^f \rightarrow [0,1]$ *is defined thus*

$$\textit{for each } t \in \mathfrak{T}^f, \qquad \varphi_s^L(t) \triangleq \sum\nolimits_{D \subseteq \mathfrak{C}_s^L \,:\, \lambda(D)=t} \mathrm{wt}(D)\,. \tag{4}$$

For any two states s and s′, we write $s \leq_{\mathrm{tree}} s'$ *if and only if for each* $t \in \mathfrak{T}^f$, $\varphi_s^L(t) \leq \varphi_{s'}^L(t)$.

Note that the sum in (4) is finite, as L is finitely branching and t is finite. Functions of type $\mathfrak{T}^f \rightarrow \mathbb{K}$ are known as *formal tree series* in Automata Theory (see e.g. [10]) and represent a generalization of formal power series to trees.

By Definition 3 and 4 it follows that the tree-unfolding preorder is included in the linear-time preorder. The example in the Introduction witnesses the fact that this inclusion is strict: the linear-time semantics deems s and s' as equivalent, while the tree-unfolding semantics does not. Indeed, $\varphi_s^L(t) > \varphi_{s'}^L(t)$ and $\varphi_s^L(t') < \varphi_{s'}^L(t')$, with $t = \{\epsilon, init, init{\cdot}l, init{\cdot}r, init{\cdot}l{\cdot}\checkmark, init{\cdot}r{\cdot}\lightning\}$ and $t' = \{\epsilon, init, init{\cdot}l, init{\cdot}r, init{\cdot}l{\cdot}\checkmark, init{\cdot}r{\cdot}\checkmark\}$. We sum the above discussion in the following:

[2] These can be obtained from Definition 1 by replacing the condition of the third item with just $\sum_{s':(s,a,s')\in\delta} P(s,a,s') \in \mathbb{N}$.

Proposition 1. *The preorder* $\leq_{\text{tree}}$ *is strictly included in* $\leq_{\text{lin}}$.

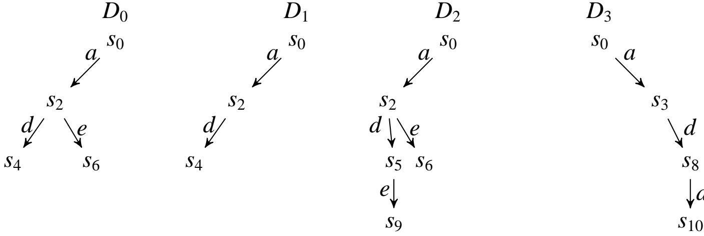
4 May Testing for RPLTS

In the first part, we review the probabilistic model of d-trees introduced by Cleaveland et al. [5]. In the second part, we introduce the testing scenario and the definition of may-testing preorder. We then establish a trace-based, observer-independent characterization of this preorder.

4.1 The Sigma-Algebra of d-trees

The material in this subsection is borrowed from [5]. Some additional terminology is in order. In what follows, we fix a generic RPLTS $L = (S, \delta, P)$. Given any $D, D' \subseteq \mathfrak{C}^L$ we say that D' is a prefix of D if $D' \subseteq D$. A d-tree is said *maximal* if it is not prefix of any other d-tree; we write $\mathcal{M}^L$ and $\mathcal{M}^L_s$, respectively, for the set of all maximal d-trees of L and of all maximal d-trees of L rooted at s. In what follows, we let T and U range over maximal d-trees.

Example 1. Consider the RPLTS L depicted in Section 3; the d-trees below belong to $\mathcal{F}^L_{s_0}$.



In the picture, each path from the root to a node of the tree - either leaf or internal node - represents a computation in the corresponding d-tree. Notice also that D_1 is a prefix of D_0, therefore it does not belong to $\mathcal{M}^L_s$, while D_0 and D_2 do.

Following [5], we consider the d-trees in $\mathcal{M}^L$ as the possible "outcomes" of observing L, and according to this intuition, define a probability space over $\mathcal{M}^L_s$, for a given state s of L. The construction of this space is based on the concept of "basic cylindrical sets" of maximal d-trees: subsets of $\mathcal{M}^L_s$ containing all d-trees sharing a given finite prefix. The measure of each cylinder is defined, as expected, as the product of the probabilities associated to edges of the common prefix. Formal definitions of basic cylindrical sets and of their measure are given below.

Definition 5 (basic cylindrical set). *Let* $L = (S, \delta, P)$ *be a* RPLTS*, let* $s \in S$ *and* $D \in \mathcal{F}^L_s$. *The* basic cylindrical set *(with common prefix D)* $B_D \subseteq \mathcal{M}^L_s$ *is defined as:* $B_D \triangleq \{T \in \mathcal{M}^L_s \mid D \subseteq T\}$.

We let $\mathcal{B}_s$ be the sigma-algebra generated by the collection of basic cylindrical sets B_D, for $D \in \mathcal{F}^L_s$. $\mathcal{B}_s$ is obviously sigma-finite. We let $\mu^L_s : \mathcal{B}_s \to [0, 1]$ be the unique measure satisfying $\mu^L_s(B_D) = \text{wt}(D)$ for each $D \in \mathcal{F}^L_s$.

For any s, μ^L_s is a probability measure over $\mathcal{B}_s$, therefore $(\mathcal{M}^L_s, \mathcal{B}_s, \mu^L_s)$ is a probability space. In the following, we will omit the superscript L from μ^L_s when the underlying RPLTS is clear from the context.

4.2 The May-Testing Preorder

Let us fix a generic RPLTS L. An *observer* O is a RPLTS over the set of actions $Act \cup \{\omega\}$, where $\omega \notin Act$ is a distinct *success* action. For any state o of O and $H \in \mathcal{T}^O_o$, we let $\Omega(H) \triangleq \{w \in Act^* \cdot \{\omega\} \mid H \xrightarrow{w} \}$. The set of all possible sequences of actions leading o to success is then $\Omega(o) \triangleq \bigcup_{H \in \mathcal{T}^O_o} \Omega(H)$. A state o of O is said to be *finite* if $\mathfrak{C}^O_o$ is a finite set. In other words, o is finite if the RPLTS reachable from o is finite-state and acyclic. In the following we write W° for the set $\{w \mid w\omega \in W\}$.

Definition 6. *Let s and o be states of L and of O, respectively. For any $D \in \mathcal{T}^L_s$ and $H \in \mathcal{T}^O_o$, we say that D* may *H if there is $w \in Act^*$ such that $D \xrightarrow{w}$ and $H \xrightarrow{w\omega}$. The set of maximal d-trees of L rooted at s that satisfy o is defined as*

$$\mathrm{sat}(s, o) \triangleq \{(T, U) \in \mathcal{M}^L_s \times \mathcal{M}^O_o \mid T \; may \; U\}.$$

Before introducing the may-testing preorder, we have to fulfill one obligation: proving that $\mathrm{sat}(s, o)$ is measurable in an appropriate sigma-algebra. To this purpose, we first generalize the concept of maximal d-tree. Informally, a W-maximal d-tree, with $W \subseteq Act^*$, is a d-tree D such that $D \xrightarrow{w}$ for at least one $w \in W$. Moreover, D is non redundant with respect to W, in the sense that it cannot be extended (resp. pruned) to match more (resp. the same) elements of W. These conditions, plus a requirement of local maximality on nodes, ensure that distinct W-maximal d-trees generate disjoint basic cylindrical sets.

Definition 7 (W-maximal d-tree). *Let L be a* RPLTS, *s a state and let $W \subseteq Act^*$. $D \in \mathcal{T}^L_s$ is said to be* locally-maximal *if whenever $\sigma a s_1 \in D$ and $\sigma b s_2 \in \mathfrak{C}^L_s$ then there is s_3 s.t. $\sigma b s_3 \in D$. $D \in \mathcal{T}^L_s$ is said to be W*-maximal *if it is locally-maximal and satisfies the following conditions (the inclusions below are strict):*

1. *$\lambda(D) \cap W \neq \emptyset$;*
2. *for each locally-maximal $D' \in \mathcal{T}^L_s$, $D \subset D'$ implies $\lambda(D') \cap W = \lambda(D) \cap W$;*
3. *for each locally-maximal $D' \in \mathcal{T}^L_s$, $D' \subset D$ implies $\lambda(D') \cap W \subset \lambda(D) \cap W$.*

This definition is extended to observers by letting W range over subsets of $Act^* \cdot \{\omega\}$.

It is worthwhile to note that the Act^*-maximal d-trees (rooted at s) are exactly the maximal d-trees (rooted at s).

Example 2. Consider again the RPLTS in Section 3 and the d-trees D_0, D_1, D_2 and D_3 from Example 1. Let $W = \{ade, ad\}$. Then:

- D_1 is not locally-maximal, hence not W-maximal: it does not contain the transition $s_2 \xrightarrow{e} s_6$;
- D_3 is not W-maximal: the transition $s_8 \xrightarrow{d} s_{10}$ is unnecessary;
- D_0 and D_2 are W-maximal.

The following key result motivates the introduction of W-maximal d-trees. It follows from the definition of basic cylindrical set, locally- and W-maximal d-tree.

Lemma 1. *Let $D, D' \in \mathcal{T}_s^L$ be W-maximal d-trees, for some $W \subseteq Act^*$. If $D \neq D'$ then $B_D \cap B_{D'} = \emptyset$*

We come to show that $\mathrm{sat}(s, o)$ is measurable.

Proposition 2. *The set* $\mathrm{sat}(s, o)$ *is measurable in the product sigma-algebra $\mathcal{B}_s \times \mathcal{B}_o$. Moreover, if o is finite then* $\mathrm{sat}(s, o) = \biguplus_{(D,H):\, \substack{H \in \mathcal{F}_o^O\ \Omega(o)\text{-maximal} \\ D \in \mathcal{F}_s^L\ \Omega(H)^\circ\text{-maximal}}} B_D \times B_H$.

Consider the probability spaces $(\mathcal{M}_s^L, \mathcal{B}_s, \mu_s)$ and $(\mathcal{M}_o^O, \mathcal{B}_o, \mu_o)$ and let $\mu_{(s,o)}$ denote the product probability measure over $\mathcal{B}_s \times \mathcal{B}_o$. As a corollary of Proposition 2, of the definition of product sigma-algebras and product measures (2), we get the following.

Corollary 1. *For a finite o,* $\mu_{(s,o)}(\mathrm{sat}(s, o)) = \sum_{(D,H):\, \substack{H \in \mathcal{F}_o^O\ \Omega(o)\text{-maximal} \\ D \in \mathcal{F}_s^L\ \Omega(H)^\circ\text{-maximal}}} \mathrm{wt}(D) \cdot \mathrm{wt}(H)$.

The classical definition of may testing preorder [9] is extended to the present probabilistic setting by taking into account the probability that two states satisfy any given observer. Note that the preorder thus defined only relates states of the same RPLTS. In case one wants to relate states belonging to two different RPLTS's, or even relate two rooted RPLTS's, one may work with the disjoint union of the two RPLTS's.

Definition 8 (may testing preorder). *Let $L = (S, \delta, P)$ be a* RPLTS*, let $s, s' \in S$, and $\mathcal{O}$ be a set of observers. We define $s \sqsubseteq^{L,\mathcal{O}} s'$ if and only if for any observer $O \in \mathcal{O}$ and any state o in O, it holds that* $\mu_{(s,o)}(\mathrm{sat}(s, o)) \leq \mu_{(s',o)}(\mathrm{sat}(s', o))$.

When $\mathcal{O}$ is the whole class of observers with actions in $Act \cup \{\omega\}$, we abbreviate $s \sqsubseteq^{L,\mathcal{O}} s'$ just as $s \sqsubseteq^{L} s'$, and call this just the *may-testing* preorder. The superscript L will be omitted when clear from the context.

We can now define a trace-based, observer-independent characterization of the may-testing preorder. Let us fix a generic RPLTS $L = (S, \delta, P)$ and take $s \in S$ and any $W \subseteq Act^*$. We define $(s \stackrel{W}{\Longrightarrow}) \triangleq \{T \in \mathcal{M}_s^L \mid T \stackrel{w}{\rightarrow}$ for some $w \in W\}$ and let $(s \stackrel{w}{\Longrightarrow})$ stand for $(s \stackrel{W}{\Longrightarrow})$ with $W = \{w\}$.

Theorem 1 (observer-independent characterization). *For each s and s' states of L, $s \sqsubseteq s'$ if and only if for every $W \subseteq_{fin} Act^*$, one has $\mu_s(s \stackrel{W}{\Longrightarrow}) \leq \mu_{s'}(s' \stackrel{W}{\Longrightarrow})$.*

Proof: In the proof we will use the following facts (proofs can be found in [1, Appendix A]):

(a) The set $(s \stackrel{W}{\Longrightarrow})$ is measurable in $\mathcal{B}_s$. In particular, if W is finite, one has $(s \stackrel{W}{\Longrightarrow}) = \biguplus_{D \in \mathcal{F}_s^L :\, D \text{ is } W\text{-maximal}} B_D$.

(b) Let L be a RPLTS, O be an observer and s, o be states of L and O, respectively. For each $U \in \mathcal{M}_o^O$ define $E_{s,U} \triangleq \{T \in \mathcal{M}_s^L \mid (T, U) \in \mathrm{sat}(s, o)\}$. Then $\mu_{(s,o)}(\mathrm{sat}(s, o)) = \int_{\mathcal{M}_o^O} \mu_s(E_{s,U})\, d\mu_o(U)$.

Assume $s \sqsubseteq s'$. Fix any $W \subseteq_{\text{fin}} Act^*$. One can build a *deterministic* observer O_W such that for some finite state o in O_W, one has $\Omega(o) = W \cdot \{\omega\}$. Since O_W is deterministic, one has that $\mathcal{M}_o^{O_W}$ consists of a single d-tree, say H, which is also the unique $W \cdot \{\omega\}$-maximal d-tree. Moreover, $\text{wt}(H) = 1$ by definition. Now we have

$$\begin{aligned}
\mu_{(s,o)}(\text{sat}(s,o)) &= \textstyle\sum_{D \in \mathcal{F}_s^L,\, D\ W\text{-maximal}} \text{wt}(D) \cdot \text{wt}(H) && \text{(by Corollary 1)} \\
&= \textstyle\sum_{D \in \mathcal{F}_s^L,\, D\ W\text{-maximal}} \text{wt}(D) && \text{(by wt}(H) = 1) \\
&= \mu_s\big(\biguplus_{D \in \mathcal{F}_s^L,\, D\ W\text{-maximal}} B_D \big) && \text{(by } \mu_s(B_D) = \text{wt}(D) \text{ and additivity)} \\
&= \mu_s(s \stackrel{W}{\Longrightarrow}) && \text{(by (a)).}
\end{aligned}$$

Finally, $\mu_{(s,o)}(\text{sat}(s,o)) \le \mu_{(s',o)}(\text{sat}(s',o))$ implies $\mu_s(s \stackrel{W}{\Longrightarrow}) \le \mu_{s'}(s' \stackrel{W}{\Longrightarrow})$.

Assume now that for every $W \subseteq_{\text{fin}} Act^*$ one has $\mu_s(s \stackrel{W}{\Longrightarrow}) \le \mu_{s'}(s' \stackrel{W}{\Longrightarrow})$. Take any observer O and any state o of O. For every $U \in \mathcal{M}_o^O$, let $V = \Omega(U)^\circ \subseteq Act^*$. The – possibly infinite – set V can be written as $V = \bigcup_{i \ge 0} V_i$, where each V_i is the subset of V containing sequences of length $\le i$. By the properties of measures, for any r, $\mu_r(r \stackrel{V}{\Longrightarrow}) = \lim_{i\to\infty} \mu_r(r \stackrel{V_i}{\Longrightarrow})$. Since for each i, $\mu_s(s \stackrel{V_i}{\Longrightarrow}) \le \mu_{s'}(s' \stackrel{V_i}{\Longrightarrow})$, on the limit we get $\mu_s(E_{s,U}) = \mu_s(\stackrel{V}{\Longrightarrow}) \le \mu_{s'}(s' \stackrel{V}{\Longrightarrow}) = \mu_{s'}(E_{s',U})$. Therefore, by integrating the two functions $U \mapsto \mu_s(E_{s,U})$ and $U \mapsto \mu_{s'}(E_{s',U})$ over $\mathcal{M}_o^O$, it follows that

$$\int_{\mathcal{M}_o^O} \mu_s(E_{s,U})\, d\mu_o(U) \le \int_{\mathcal{M}_o^O} \mu_{s'}(E_{s',U})\, d\mu_o(U)\,.$$

This is equivalent to $\mu_{(s,o)}(\text{sat}(s,o)) \le \mu_{(s',o)}(\text{sat}(s',o))$, by (b). Since O and o are arbitrary, it follows that $s \sqsubseteq s'$. □

4.3 On the Relationship among $\sqsubseteq$, $\le_{\text{lin}}$ and $\le_{\text{tree}}$

We can finally make precise the relationship between the may-testing preorder $\sqsubseteq$, the linear and tree-unfolding preorder, $\le_{\text{lin}}$ and $\le_{\text{tree}}$ respectively. In Proposition 1 we have already shown that $\le_{\text{tree}}$ is strictly included in $\le_{\text{lin}}$. It remains to establish a relationship between $\sqsubseteq$ and $\le_{\text{lin}}$ and between $\sqsubseteq$ and $\le_{\text{tree}}$. We start by considering the first pair of preorders and by introducing a simple result that is at the basis of Theorem 2 below.

Lemma 2. *For each $s \in S$ and $w \in Act^*$, $\mu_s(s \stackrel{w}{\Longrightarrow}) = f_s(w)$.*

Theorem 2. *The preorder $\sqsubseteq$ is strictly included in $\le_{\text{lin}}$.*

Remark 1 (on canonical observers). The proof of the above Theorem 1 shows that the class of finite, deterministic and *non*-probabilistic observers of the form O_W ($W \subseteq_{fin} Act^*$) is powerful enough to induce the may testing preorder $\sqsubseteq$.

On the other hand, we also note that "linear" observers, i.e. observers composed by a single successful sequence, that are sufficient in the classical case, are not sufficient here. It is quite easy to see that they induce the preorder $\le_{\text{lin}}$.

Consider now the RPLTS L depicted in Fig. 1. It witnesses the fact that neither $\sqsubseteq\ \subseteq\ \le_{\text{tree}}$ nor $\le_{\text{tree}}\ \subseteq\ \sqsubseteq$. As a matter of fact, Theorem 1 guarantees that $s \not\sqsubseteq s'$, indeed $\mu^L_{s'}(s' \stackrel{\{ab,ac\}}{\Longrightarrow}) = 0.6 < \mu^L_s(s \stackrel{\{ab,ac\}}{\Longrightarrow}) = 0.7$, while, for what concerns

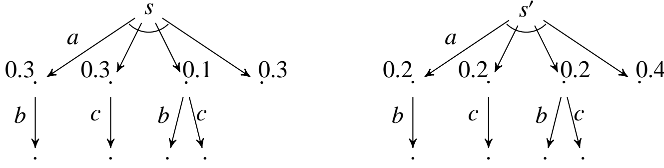
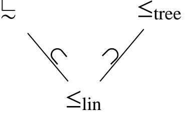


Fig. 1. $s \not\lesssim s'$ and $s' \not\leq_{\text{tree}} s$

the tree-unfolding preorder, we get the opposite: $s' \not\leq_{\text{tree}} s$. Indeed, $\varphi_s^L(t) = 0.1 < \varphi_{s'}^L(t) = 0.2$, where t is the tree represented by $\{\epsilon, a, ab, ac\}$.



To conclude, we pictorially represent on the right the inclusion relationships among the three preorders thus established.

5 May-Testing and the Safety Properties of Infinite Trees

A more satisfactory understanding of a behavioural relation can be obtained by looking at it in terms of the class of properties satisfied by equivalent processes. A famous example is the Hennessy-Milner theorem [16], asserting that two processes are bisimilar exactly when they satisfy the same formulae of the HM logic. Another example, in a probabilistic setting, is the characterization of probabilistic bisimulation in the work of Larsen and Skou [18]. In our case, the alternative characterization in terms of sets of traces obtained in the previous section suggests looking at properties of trees. In fact, we shall characterize the may testing preorder in terms of the probability, for two given states, of satisfying safety properties of trees.

Some additional terminology on strings and trees is in order. Recall that a (possibly infinite) tree θ is a prefix-closed subset of Act^*. Let us indicate by $\prec$ the usual prefix partial order on strings. The set of *leaves* of θ, denoted by leaves(θ), is the set of strings in θ that are $\prec$-maximal in θ. We say a tree is *maximal* if leaves(θ) $= \emptyset$; note that a maximal tree is necessarily infinite. We call $\mathfrak{T}$ the set of maximal trees. In what follows, we shall use the letter τ to range over $\mathfrak{T}$. There is a natural partial ordering on trees given by the following

$$\theta \leq \theta' \text{ iff } \theta \subseteq \theta' \text{ and whenever } w \in \theta' \setminus \theta \text{ then there is } u \in \text{leaves}(\theta) \text{ s.t. } u \prec w\,.$$

What this means is that θ' can be obtained from θ by expanding into trees the leaves of θ. If $\theta \leq \theta'$ we also say θ is a prefix of θ'. Let us call Θ the sigma-algebra of maximal trees generated by the basic cylindrical sets C_t, where t ranges over all finite trees and $C_t \triangleq \{\tau \mid t \leq \tau\}$.

Let us now fix a RPLTS L. We shall assume that L has *no dead state*, that is, for each state s there is always at least one transition from s. This assumption allows for a simpler treatment in the following, but is not really restrictive: any RPLTS can be turned into one with no dead states by adding, where necessary, dummy self-loops labelled by a distinct action. For any state s in L, recall that $\mathcal{B}_s$ is the sigma-algebra of maximal d-trees on L rooted at s (Section 4.1). Note that the label-extracting function $\lambda : \mathcal{B}_s \to \Theta$

maps each $T \in \mathcal{B}_s$ into a maximal tree $\tau = \lambda(T) \in \Theta$. Also note that whenever C_t and $C_{t'}$ are disjoint, so are $\lambda^{-1}(C_t)$ and $\lambda^{-1}(C_{t'})$. As a consequence, $\lambda^{-1}(\biguplus_{t \in I} C_t) = \biguplus_{t \in I} \lambda^{-1}(C_t)$. Moreover, $\lambda^{-1}(C_t^c) = (\lambda^{-1}(C_t))^c$. Another property of λ we shall rely upon is the following:

Lemma 3. *For any t, $\lambda^{-1}(C_t)$ is measurable in $\mathcal{B}s$.*

The previous properties of λ allow us to define measures on Θ as follows.

Definition 9. *Let s be a state of L. The measure ν_s on Θ is defined by setting for the basic cylindrical sets $\nu_s(C_t) \triangleq \mu_s(\lambda^{-1}(C_t))$.*

With the above definitions, $(\mathfrak{T}, \Theta, \nu_s)$ is a probability space, for each s in L. The following lemma is a consequence of the additivity of measures μ_s and ν_s and of the fact that λ^{-1} preserves disjointness.

Lemma 4. *Let $R = \biguplus_{t \in I} C_t$, for some index set I. Then $\nu_s(R) = \mu_s(\lambda^{-1}(R))$.*

The elements of Θ we are interested in are the *safety* properties of the form $Safe_W = \{\tau | \tau \cap W = \emptyset\}$, where W is any finite or infinite subset of Act^*. For example, if $W = \{w \in Act^* |$ action *crash* occurs in $w\}$, $Safe_W$ corresponds to the property that action *crash* is never executed. We have to make sure in the first place that the sets $Safe_W$ are measurable. We need the following lemma.

Lemma 5. *For each $W \subseteq Act^*$, $Safe_W$ is measurable in Θ. Moreover, if W is* finite*, $Safe_W$ can be written as a disjoint union of basic cylindrical sets.*

Corollary 2. *Let s be a state of L and $W \subseteq Act^*$. It holds that $\nu_s(Safe_W) = 1 - \mu_s(s \stackrel{W}{\Longrightarrow})$.*

As a corollary of the previous result, we get the following, which characterize $\sqsubseteq$ in terms of probability of satisfying safety properties.

Theorem 3. *Let s, s' be states of L and suppose L has no dead states. We have $s \sqsubseteq s'$ if and only if for each $W \subseteq Act^*$, $\nu_s(Safe_W) \geq \nu_{s'}(Safe_W)$.*

Of course, $\sqsubseteq$ can be also characterized in terms of *reachability* properties of the form $\{\tau | \tau \cap W \neq \emptyset\}$. In this case, the inequality between s and s' gets reversed.

6 Testing Separated RPLTS

In a separated system, actions can be partitioned into two sets: a set of actions Σ that are probabilistically controlled by the system, and a set of nondeterministic actions A that are under the control of the environment (observer). Accordingly, actions that are probabilistic for processes are nondeterministic for observers, and vice-versa. Importantly, the two types of actions do not mix up: the set of states as well gets partitioned into a set of states where only probabilistic choices are possible, and a set of states where only nondeterministic choices are possible. Nondeterministic actions can be modelled as actions that, if available, have probability one. These informal considerations lead to the next definition.

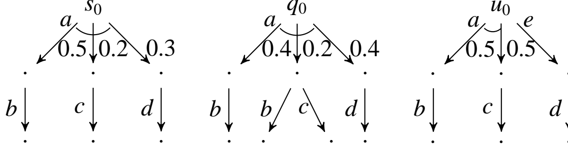


Fig. 2. Two separated RPLTS (left and center) and a non-separated one (right)

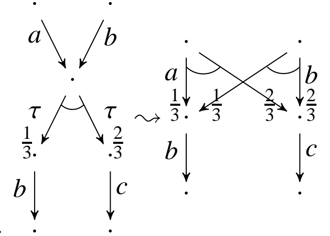
Definition 10 (separated processes and observers). *Let (Σ, A) form a partition of Act, that is $\Sigma \cap A = \emptyset$ and $\Sigma \cup A = Act$. We say a* RPLTS $L = (S, \delta, P)$ *is a* (Σ, A)-separated process *if there is a partition of the sates, $S = G \cup R$, such that*

– *for each $s \in G$ and $(s, a, s') \in \delta$ one has $a \in \Sigma$; moreover, if $(s, b, s'') \in \delta$, for some b and s'', then $a = b$;*
– *for each $s \in R$ and $(s, a, s') \in \delta$ one has $a \in A$; moreover, if $(s, a, s'') \in \delta$, for some s'', then $s' = s''$.*

A (Σ, A)-separated observer *is a $(A \cup \{\omega\}, \Sigma)$-separated* RPLTS, *where ω is the distinct success action, $\omega \notin Act$.*

Example 3. Let $A = \{b, c, d\}$ and $\Sigma = \{a, e\}$. An example of (Σ, A)-separated and one of non (Σ, A)-separated processes are depicted in Fig. 2.

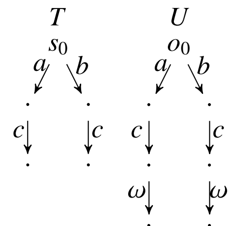
Remark 2. Separated RPLTS are reminiscent of Hansson and Jonsson's *alternating model* [15], where probabilistic and nondeterministic states are strictly alternated with one another. In the alternating model, probabilistic choices are labeled by the silent action τ. In our model we do not have silent actions; but one can get rid of those τ's by absorbing them into incoming edges of probabilistic states (see picture on the right). Modulo this transformation, separated RPLTS can be seen as a proper extension of the alternating model.



In what follows we fix a generic (Σ, A)-separated process L and a generic (Σ, A)-separated observer O. We let s be a state of L and o be a state of O. The proof of the main result in this section rests on the following crucial lemma.

Lemma 6. *Consider $T \in \mathcal{M}^L_s$ and $U \in \mathcal{M}^O_o$. There is at most one $w \in Act^*$ such that $T \xrightarrow{w}$ and $U \xrightarrow{w\omega}$.*

It is worthwhile to note that the above lemma fails to hold for non-separated system. As an example, consider T and U depicted on the right. Clearly, either T belongs to a $(\{c\}, \{a, b\})$-separated RPLTS or U belongs to a $(\{a, b\}, \{c\})$-separated observer (a and b are used in both as nondeterministic actions) and they violate Lemma 6.



Recall that $f^L_s : Act^* \to [0, 1]$ denotes the formal power series associated with the RPLTS L at state s. Similarly, $f^O_o : (Act \cup \{\omega\})^* \to [0, 1]$ is associated with the RPLTS O at state o.

Corollary 3. $\mu_{(s,o)}(\mathrm{sat}(s,o)) = \sum_{w\in Act^*} f^L_s(w)\cdot f^O_o(w\omega)$.

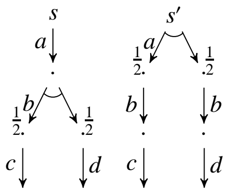
As a consequence of Theorem 1 and Corollary 3 we get the following result.

Corollary 4 (coincidence of linear-time and separated may-testing semantics). *Let* $\sqsubseteq^{(\Sigma,A)}$ *be the may preorder on the states of L generated by* (Σ, A)*-separated observers. For each s and s′, one has* $s \sqsubseteq^{(\Sigma,A)} s'$ *if and only if* $f^L_s \le f^L_{s'}$.

Example 4. Consider the RPLTS composed by the trees depicted in the left and in the center of Fig. 2. This is a $(\{a\},\{b,c,d\})$-separated RPLTS. It is easy to check that $s_0 \le_{\mathrm{lin}} q_0$. From Corollary 4 we get $s_0 \sqsubseteq^{(\{a\},\{b,c,d\})} q_0$.

7 Conclusion and Related Works

There exist many formalisms for the specification of probabilistic processes and as many variants of probabilistic semantics. The conceptual difference between our approach and other testing theories has been discussed in the Introduction. Concerning the set-theoretical comparison between behavioural relations, we restrict our attention to one of the most recent proposals [7], and refer for the other approaches to the papers mentioned therein. Consider the PCSP processes $P = a.(b.c\,{}_{\frac{1}{2}}\!\oplus b.d)$ and $Q = a.b.c\,{}_{\frac{1}{2}}\!\oplus a.b.d$. The picture on the right is the description of P and Q's operational semantics in terms RPLTS's (the operational model of [7] is in fact different from ours, because transitions lead to probability distributions on states, rather than states). P and Q are discriminated by the may preorder of [7], as witnessed by the test $s' = a.b.c.\omega \sqcap a.b.d.\omega$, which tells us that Q is not smaller than P. On the other hand, P and Q are equated by our may-preorder, which can be established by resorting to Theorem 1. This example shows that the presence of internal choice in [7] increases the distinguishing power of observers. Indeed, several of the axioms usually valid for classical CSP (like, e.g., distributivity of prefixing w.r.t. internal choice) are no longer valid in PCSP. We conjecture that if the internal choice operator were taken out from CSP, the may preorder of [7] would coincide with ours.



As one would expect, our may-testing equivalence is coarser than probabilistic bisimulation [18]. Indeed, any set W in the alternative characterization (or equivalently, any canonical observer O_W, see Theorem 1) can be characterized by a formula of the Probabilistic Modal Logic, which induces probabilistic bisimulation [18]. That the inclusion is strict is witnessed by processes s and s' above, which can be distinguished by probabilistic bisimulation.

As for future work, an obvious next-step is the study of *must*-testing behavioural relations. Also, decidability and algorithmic issues for the considered semantics deserve further investigation. Currently, the only known facts concern the linear-time setting: in the context of Rabin's probabilistic finite-state automata, which are equivalent to RPLTS, it is known that the preorder is undecidable, while the induced equivalence *is* decidable in polynomial time (see [20] and references therein).

References

1. Acciai, L., Boreale, M., De Nicola, R.: Linear-Time and May-Testing in a Probabilistic Reactive Setting. Full version, http://rap.dsi.unifi.it/~acciai/papers/prob-may.pdf
2. Baier, C.: On the algorithmic verification of probabilistic systems. Universität Mannheim, Habilitation Thesis (1998)
3. Boreale, M., De Nicola, R., Pugliese, R.: Trace and Testing Equivalence in Asynchronous Processes. Information and Computation 172, 139–164 (2002)
4. Chatzikokolakis, K., Palamidessi, C.: Making random choices invisible to the scheduler. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 42–58. Springer, Heidelberg (2007)
5. Cleaveland, R., Iyer, S.P., Narasimha, M.: Probabilistic temporal logics via the modal mu-calculus. Theor. Comput. Sci. 342(2-3) (2005)
6. Deng, D., van Glabbeek, R., Hennessy, M., Morgan, C.: Testing finitary probabilistic processes. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 274–288. Springer, Heidelberg (2009)
7. Deng, D., van Glabbeek, R., Hennessy, M., Morgan, C.: Characterising testing preorders for finite probabilistic processes. Logical Methods in Computer Science 4(4), 1–33 (2008)
8. De Nicola, R.: Extensional equivalences for transition systems. Acta Informatica 24(2), 211–237 (1987)
9. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34, 83–133 (1984)
10. Ésik, Z., Kuich, W.: Formal Tree Series. Journal of Automata, Languages and Combinatorics 8(2), 219–285 (2003)
11. Georgievska, S., Andova, S.: Retaining the Probabilities in Probabilistic Testing Theory. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 79–93. Springer, Heidelberg (2010)
12. van Glabbeek, R., Smolka, S., Steffen, B., Tofts, C.: Reactive, generative, and stratified models of probabilistic processes. Information and Computation 121(1), 59–80 (1995)
13. van Glabbeek, R.J.: The linear time-branching time spectrum. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 278–297. Springer, Heidelberg (1990)
14. Halmos, P.: Measure theory. Litton Educational Publishing, Inc. (1950)
15. Hansson, H., Jonsson, B.: A Calculus for Communicating Systems with Time and Probabilities. In: Proc. of IEEE Real-Time Systems Symposium, pp. 278–287 (1990)
16. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. Journal of the ACM 32(1), 137–161 (1985)
17. Jonsson, B., Yi, W.: Testing Preorders for Probabilistic Processes can be Characterized by Simulations. TCS 282(1), 33–51 (2002)
18. Larsen, K.G., Skou, A.: Bisimulation through Probabilistic Testing. Inf. and Comp. 94(1), 1–28 (1991)
19. Segala, R.: Testing Probabilistic Automata. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 299–314. Springer, Heidelberg (1996)
20. Tzeng, W.-G.: A polynomial time algorithm for the equivalence of probabilistic automata. SIAM Journal on Computing 21(2), 216–227 (1992)
21. Wang, Y., Larsen, K.G.: Testing Probabilistic and Nondeterministic Processes. In: Proc. of PSTV, IFIP Transactions C, vol. 8, pp. 47–61 (1992)

A Model-Checking Tool for Families of Services

Patrizia Asirelli[1], Maurice H. ter Beek[1], Alessandro Fantechi[1,2], and Stefania Gnesi[1]

[1] Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, Pisa, Italy
{asirelli,terbeek,gnesi}@isti.cnr.it
[2] Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy
fantechi@dsi.unifi.it

Abstract. We propose a model-checking tool for on-the-fly verification of properties expressed in a branching-time temporal logic based on a deontic interpretation of classical modal and temporal operators over modal transition systems. We apply this tool to the analysis of variability in behavioural descriptions of families of services.

1 Introduction

We present our ongoing research on an emerging topic in the engineering of distributed systems, right on the crossroads between (software) product line engineering and service computing [10, 23, 24]. Our aim is the development of rigorous modelling techniques as well as analysis and verification support tools for assisting organisations to plan, optimise, and control the quality of software service provision. To this aim, we foresee a flexible engineering methodology according to which software service line organisations can develop novel classes of service-oriented applications easily adaptable to customer requirements as well as to changes in the context in which they execute.

Product Line Engineering (PLE) is an approach to develop product families using a common platform and mass customisation [33]. It aims to lower production costs of individual products by letting them share an overall reference model of a product family, while allowing them to differ w.r.t. particular features in order to serve, e.g., different markets. As a result, the production process in PLE is organised so as to maximise commonalities of the products and at the same time minimise the cost of variations. Product variants can be derived from the product family, which allows for reuse and differentiation of a family's products. Variety of products from a single product platform is achieved by identifying variation points as places in design artifacts where a specific decision is reduced to the choice among several *features* but the feature to be chosen for a particular product is left open (like optional, mandatory or alternative features). Software Product Line Engineering (SPLE) is a paradigm for developing a diversity of software products and software-intensive systems based on the underlying architecture of an organisation's product platform [9, 36]. Variability management is the key aspect differentiating SPLE from 'conventional' software engineering.

Service-Oriented Computing (SOC) is a distributed computing paradigm [35]. Services are autonomous, distributed, and platform-independent computational

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 44–58, 2011.
© IFIP International Federation for Information Processing 2011

elements capable of solving specific tasks, which all need to be described, published, categorised, discovered, and then dynamically and loosely coupled in novel ways (orchestrated). They can be used to create distributed, interoperable, and dynamic applications and business processes which span organisational boundaries as well as computing platforms. In the end, SOC systems deliver application functionalities as services to either end-user applications or other services. Their underlying infrastructures are called Service-Oriented Architectures (SOAs).

We recently launched a research effort to, on the one hand, investigate the most promising existing modelling structures that allow (behavioural) variability to be described and product derivation to be defined, and, on the other hand, develop a proper temporal logic that can be interpreted over such structures and which can express interesting properties over families and products alike. In [3], we defined such a temporal logic and an efficient model-checking algorithm.

This paper is a first step towards extending our results to service families. We present a model checker based on a formal framework of vaCTL (a variability and action-based branching-time temporal logic) with its natural interpretation structure (MTS, Modal Transition System). Product derivation is defined in our framework and logical formulae are used as variability constraints as well as behavioural properties to be verified for families and products alike. We apply our tool to analyse variability in behavioural descriptions of service families.

After presenting a simple running example in Sect. 2, we formally define MTSs in Sect. 3. In Sect. 4, we introduce vaCTL and show how we can manage advanced variability in Sect. 5. In Sect. 6, we describe our tool and apply it to the running example. Related work is discussed in Sect. 7, before Sect. 8 concludes.

2 Running Example: Travel Agency

As motivating example, consider a software company developed a package on sale for those interested in starting a travel agency service (e.g. on the web). This company provides a choice among products of a family with different prices and features. All products provide the features *hotel*, *flight*, and *train* reservation: the coordination component uses *predefined* external services (one per business sector) to retrieve a list of quotes. These products can be enhanced in two ways:

1. By adding as *alternative* feature the possibility to choose, only for flights and hotels, from *multiple* external services in order to retrieve the best quotes through more than one service. This means that proper coordination addresses more hotel and flight services, and proper data fusion is done with the data received by the contacted external services.
2. By adding as *optional* feature the possibility for a customer to book a *leisure tour* during his/her stay at a hotel. This is achieved through an additional component that interacts with an external leisure tour service. However, as the provided tour packages may include a hotel in a different location for a subset of nights, a tour reservation *requires* interaction with the hotel service to offer a feature that allows to *cancel part* of the room reservations at the main hotel location for such nights. A coordination model variant can do so.

When combined, this choice of features leads to the following 8 different products.

features		variability	products							
			1	2	3	4	5	6	7	8
train reservation	predefined services	mandatory	√	√	√	√	√	√	√	√
hotel reservation	predefined services	alternative	√	√			√	√		
	multiple services				√	√			√	√
flight reservation	predefined services	alternative	√			√	√			√
	multiple services			√	√			√	√	
leisure tour reservation		optional ↰					√	√	√	√
cancel part reservation		required ┘					√	√	√	√

3 A Behavioural Model for Product Families

Modal Transition Systems (MTSs) [28] and variants are now an accepted formal model for defining behavioural aspects of product families [14, 27, 12, 29, 3]. An MTS is a Labelled Transition System (LTS) with a distinction between may and must transitions, seen as *optional* or *mandatory* features for a family's products. For a given product family, an MTS can model

- its *underlying behaviour*, shared among all products, and
- its *variation points*, differentiating between products.

An MTS cannot model advanced variability constraints regarding *alternative* features (only one of them may be present) nor those regarding inter-feature relations (a feature's presence *requires* or *excludes* that of another feature) [3]. We will formalise such advanced variability constraints by means of an associated set of logical formulae expressed in the variability and action-based branching-time temporal logic vaCTL that we will define in Sect. 4.

We now formally define MTSs and — to begin with — their underlying LTSs.

Definition 1. *A* Labelled Transition System *(LTS) is a quadruple* $(Q, A, \overline{q}, \delta)$*, with set* Q *of states, set* A *of actions, initial state* $\overline{q} \in Q$*, and transition relation* $\delta \subseteq Q \times A \times Q$*. We also write* $q \xrightarrow{a} q'$ *for* $(q, a, q') \in \delta$*.* □

To model behaviour of product families, we must define the evolution of time.

Definition 2. *Let* $(Q, A, \overline{q}, \delta)$ *be an LTS and* $q \in Q$*. Then* σ *is a* path *from* q *if* $\sigma = q$ *(an empty path) or* σ *is a (possibly infinite) sequence* $q_1 a_1 q_2 a_2 q_3 \cdots$ *such that* $q_1 = q$ *and* $q_i \xrightarrow{a_i} q_{i+1}$*, for all* $i > 0$*. A* full path *is a path that cannot be extended further, i.e., it is infinite or ends in a state with no outgoing transitions. The set of all full paths from* q *is denoted by* $path(q)$*.*

If $\sigma = q_1 a_1 q_2 a_2 q_3 \cdots$*, then its i-th state* q_i *is denoted by* $\sigma(i)$ *and its i-th action* a_i *is denoted by* $\sigma\{i\}$*.* □

In an MTS, transitions are defined to be possible (*may*) or mandatory (*must*).

Definition 3. *A Modal Transition System (MTS) is a quintuple* $(Q, A, \overline{q}, \delta^\Box, \delta^\Diamond)$ *such that the quadruple* $(Q, A, \overline{q}, \delta^\Box \cup \delta^\Diamond)$ *is an LTS, called its* underlying *LTS.*

An MTS has two transition relations: $\delta^\Diamond \subseteq Q \times A \times Q$ *is the* may *transition relation, expressing* possible *transitions, while* $\delta^\Box \subseteq Q \times A \times Q$ *is the* must *transition relation, expressing* mandatory *transitions. By definition,* $\delta^\Box \subseteq \delta^\Diamond$*.*

We also write $q \xrightarrow{a}_\Box q'$ *for* $(q, a, q') \in \delta^\Box$ *and* $q \xrightarrow{a}_\Diamond q'$ *for* $(q, a, q') \in \delta^\Diamond$*.* □

The inclusion $\delta^\Box \subseteq \delta^\Diamond$ formalises that mandatory transitions must also be possible. Reasoning on the existence of transitions is thus like reasoning with a 3-valued logic with the truth values *true*, *false*, and *unknown*: mandatory transitions ($\delta^\Box$) are *true*, possible but not mandatory transitions ($\delta^\Diamond \setminus \delta^\Box$) are *unknown*, and impossible transitions ($(q, a, q') \notin \delta^\Box \cup \delta^\Diamond$) are *false* [18].

The transition relations of MTSs allow the distinction of special type of paths.

Definition 4. *Let* $\mathcal{F}$ *be an MTS and* $\sigma = q_1 a_1 q_2 a_2 q_3 \cdots$ *a full path in its underlying LTS. Then* σ *is a* must path *(from* q_1*) in* $\mathcal{F}$*, denoted by* $\sigma^\Box$*, if* $q_i \xrightarrow{a_i}_\Box q_{i+1}$*, for all* $i > 0$*. The set of all must paths from* q_1 *is denoted by* $\Box$*-path*(q_1)*.* □

Recall that features are often used for compact representations of a family's products. To model such product family representations as MTSs one thus needs a 'translation' from features to actions (not necessarily a one-to-one mapping) and the introduction of a behavioural relation (temporal ordering) among them. A family's products are then considered to differ w.r.t. the actions they are able to perform in any given state of the MTS. This means that the MTS of a product family has to accommodate all the possibilities desired for each derivable product, predicating on the choices that make a product belong to that family.

The MTS in Fig. 1 models the Travel Agency product family of Sect. 2: edges labelled may($\cdot$) are possible but not mandatory transitions, whereas those labelled must($\cdot$) are mandatory.

Given an MTS description of a product family, an MTS describing a subfamily is obtained by preserving at least all must transitions and turning some of the may transitions (that are not must transitions) into must transitions as well as removing some of the remaining may transitions.

Definition 5. *Let* $\mathcal{F} = (Q, A, \overline{q}, \delta^\Box, \delta^\Diamond)$ *be an MTS specifying a product family. A* subfamily *specified as an MTS* $\mathcal{F}_s = (Q_s, A, \overline{q}, \delta^\Box_s, \delta^\Diamond_s)$ *is derived by considering* $\delta^\Box_s = \delta^\Box \cup R$*, with* $R \subseteq \delta^\Diamond$*, and* $\delta^\Diamond_s \subseteq \delta^\Diamond$*, defined over a set* $Q_s \subseteq Q$ *of states, so that* $\overline{q} \in Q_s$*, and every* $q \in Q_s$ *is reachable from* $\overline{q}$ *via transitions from* $\delta^\Box_s \cup \delta^\Diamond_s$*.*

More precisely, we say that $\mathcal{F}_s$ is a subfamily of $\mathcal{F}$*, denoted by* $\mathcal{F}_s \vdash \mathcal{F}$*, iff* $\overline{q}_s \vdash \overline{q}$*, where* $q_s \vdash q$ *holds, for some* $q_s \in Q_s$ *and* $q \in Q$*, iff:*

- *whenever* $q \xrightarrow{a}_\Box q'$*, for some* $q' \in Q$*, then* $\exists\, q'_s \in Q_s :\ q_s \xrightarrow{a}_\Box q'_s$ *and* $q'_s \vdash q'$*, and*
- *whenever* $q_s \xrightarrow{a}_\Diamond q'_s$*, for some* $q'_s \in Q_s$*, then* $\exists\, q' \in Q :\ q \xrightarrow{a}_\Diamond q'$ *and* $q'_s \vdash q'$*.* □

An LTS describing a product can be seen (i.e., obtained from an MTS description of a product family) as a subfamily containing only must transitions. Formally:

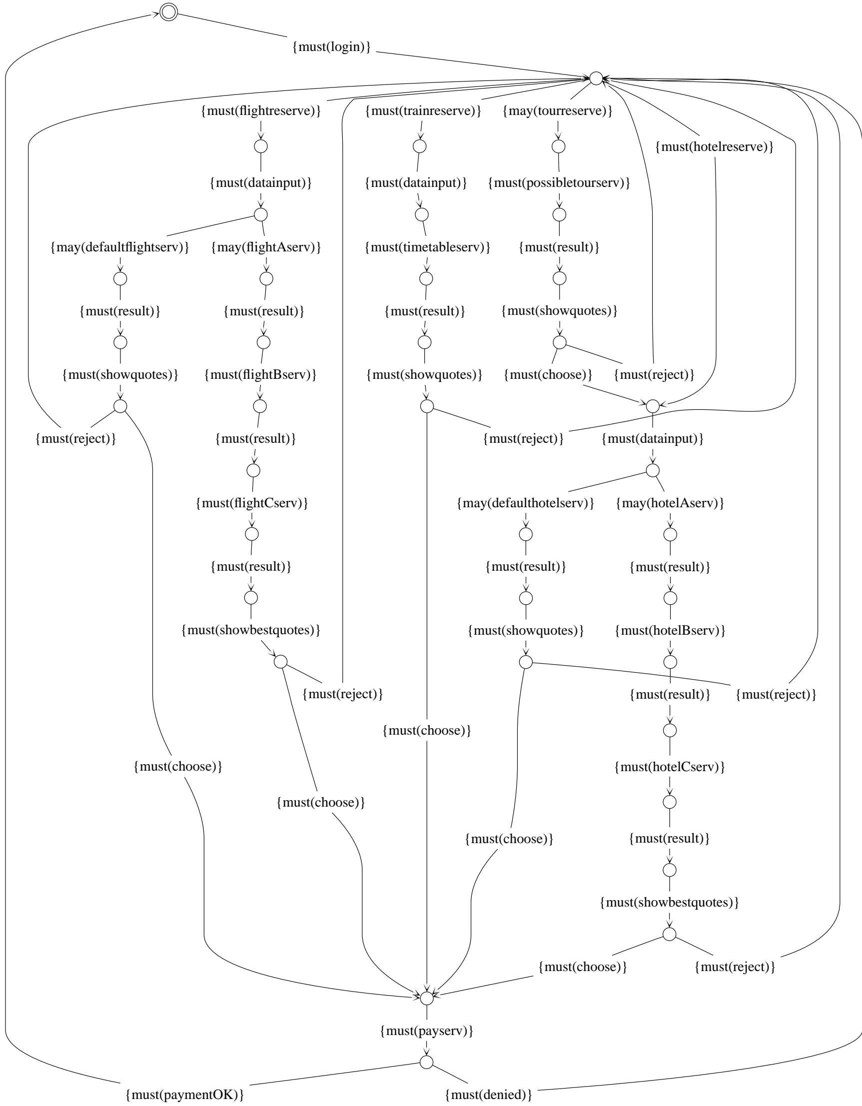


Fig. 1. MTS of the Travel Agency product family as produced by FMC

Definition 6. *Let $\mathcal{F} = (Q, A, \overline{q}, \delta^{\Box}, \delta^{\Diamond})$ be an MTS specifying a product family.*

A set of products *specified as a set of LTSs $\{ \mathcal{P}_i = (Q_i, A, \overline{q}, \delta_i) \mid i > 0 \}$ is derived by considering each transition relation δ_i to be $\delta^{\Box} \cup R$, with $R \subseteq \delta^{\Diamond}$, defined over a set of states $Q_i \subseteq Q$, so that $\overline{q} \in Q_i$, and every $q \in Q_i$ is reachable from $\overline{q}$ via transitions from δ_i.*

More precisely, we say that $\mathcal{P}_i$ is a product of *$\mathcal{F}$, denoted by $\mathcal{P}_i \vdash \mathcal{F}$, iff $\overline{q}_i \vdash \overline{q}$, where $q_i \vdash q$ holds, for some $q_i \in Q_i$ and $q \in Q$, iff:*

– *whenever* $q \xrightarrow{a}_{\Box} q'$*, for some* $q' \in Q$*, then* $\exists\, q'_i \in Q_i : q_i \xrightarrow{a}_i q'_i$ *and* $q'_i \vdash q'$*, and*
– *whenever* $q_i \xrightarrow{a}_i q'_i$*, for some* $q'_i \in Q_i$*, then* $\exists\, q' \in Q : q \xrightarrow{a}_{\Diamond} q'$ *and* $q'_i \vdash q'$*.* □

The subfamilies and products derived by Def. 5 and Def. 6 obviously might not satisfy the aforementioned advanced variability constraints that MTSs cannot model. However, as said before, we will show in Sect. 5 how to use the variability and action-based branching-time temporal logic vaCTL that we will define in Sect. 4 to express those constraints. Moreover, in [3] we outlined an algorithm to derive from an MTS all products that are valid w.r.t. constraints expressed in a temporal logic and we recently adapted this algorithm to vaCTL.

4 Logics for MTSs

In this section, we first introduce a minor extension of a well-known logic from the literature, after which we thoroughly extend the resulting logic into an action-based branching-time temporal logic with a semantics that is specifically well suited for capturing the aforementioned advanced variability constraints.

4.1 HML+UNTIL

HML+UNTIL extends the classical Hennessy–Milner logic with Until by incorporating existential and universal state operators (quantifying over paths) from CTL [5]. As such, HML+UNTIL is derived from the logics defined in [18,25,26].[1]

HML+UNTIL is a logic of state formulae (denoted by ϕ) and path formulae (denoted by π) defined over a set of atomic actions $A = \{a, b, \ldots\}$.

Definition 7. *The syntax of HML+UNTIL is:*

$$\phi ::= true \mid \neg\,\phi \mid \phi \wedge \phi' \mid \langle a \rangle\, \phi \mid [a]\, \phi \mid E\, \pi \mid A\, \pi$$
$$\pi ::= \phi\; U\, \phi'$$
□

While formally interpreted over MTSs, the semantics of HML+UNTIL does not actually consider the different type of transitions typical of MTSs. In fact, the informal meaning of the nonstandard operators of HML+UNTIL is the following.

– $\langle a \rangle\, \phi$: a next state exists, reachable by a *may* transition executing action a, in which ϕ holds
– $[a]\, \phi$: in all next states, reachable by a *may* transition executing a, ϕ holds
– $E\, \pi$: there exists a full path on which π holds
– $A\, \pi$: on all possible full paths, π holds
– $\phi\; U\, \phi'$: in a state of a path, ϕ' holds, whereas ϕ holds in all preceding states

The HML+UNTIL semantics is thus interpreted over MTSs as if they were LTSs.

[1] These logics use recursion defined by fixed points to extend HML into a temporal logic; we prefer to directly include the Until operator.

Definition 8. *Let* $(Q, A, \overline{q}, \delta^{\Box}, \delta^{\Diamond})$ *be an MTS, with* $q \in Q$ *and* σ *a full path. The satisfaction relation* $\models$ *of HML+UNTIL over MTSs is defined as follows:*

- $q \models true$ *always holds*
- $q \models \neg\, \phi$ *iff not* $q \models \phi$
- $q \models \phi \wedge \phi'$ *iff* $q \models \phi$ *and* $q \models \phi'$
- $q \models \langle a \rangle\, \phi$ *iff* $\exists\, q' \in Q$ *such that* $q \xrightarrow{a}_{\Diamond} q'$*, and* $q' \models \phi$
- $q \models [a]\, \phi$ *iff* $\forall\, q' \in Q$ *such that* $q \xrightarrow{a}_{\Diamond} q'$*, we have* $q' \models \phi$
- $q \models E\, \pi$ *iff* $\exists\, \sigma' \in path(q)$ *such that* $\sigma' \models \pi$
- $q \models A\, \pi$ *iff* $\forall\, \sigma' \in path(q)$ *such that* $\sigma' \models \pi$
- $\sigma \models \phi\; U\, \phi'$ *iff* $\exists\, j \geq 1 \colon \sigma(j) \models \phi'$ *and* $\forall\, 1 \leq i < j \colon \sigma(i) \models \phi$ □

A number of further operators can now be derived in the usual way: *false* abbreviates $\neg\, true$, $\phi \vee \phi'$ abbreviates $\neg(\neg\phi \wedge \neg\phi')$, $\phi \Longrightarrow \phi'$ abbreviates $\neg\phi \vee \phi'$. Moreover, $F\, \phi$ abbreviates $(true\ U\, \phi)$: there exists a future state in which ϕ holds. Finally, $AG\, \phi$ abbreviates $\neg EF\, \neg\phi$: in every state on every path, ϕ holds.

4.2 Variability and Action-Based CTL: vaCTL

We now introduce the variability and action-based logic vaCTL. It extends HML+UNTIL by implicitly incorporating two of the most classical *deontic* [2] modalities, namely O (*it is obligatory that*) and P (*it is permitted that*), as well as an action-based Until operator, both with and without a deontic interpretation.

vaCTL defines state formulae (denoted by ϕ) and path formulae (denoted by π), but also action formulae (boolean compositions of actions, denoted by φ, with the usual semantics, taken from [11]) over a set of atomic actions $A = \{a, b, \ldots\}$.

Definition 9. *The syntax of* vaCTL *is:*

$$\phi ::= true \mid \neg\, \phi \mid \phi \wedge \phi' \mid \langle a \rangle\, \phi \mid [a]\, \phi \mid \langle a \rangle^{\Box}\, \phi \mid [a]^{\Box}\, \phi \mid E\, \pi \mid A\, \pi$$
$$\pi ::= \phi\ \{\varphi\}\, U\, \{\varphi'\}\ \phi' \mid \phi\ \{\varphi\}\, U^{\Box}\, \{\varphi'\}\ \phi'$$
□

The informal meaning of the operators we added to HML+UNTIL is as follows.[2]

- $\langle a \rangle^{\Box}\, \phi$: a next state exists, reachable by a *must* transition executing a, in which ϕ holds
- $[a]^{\Box}\, \phi$: in all next states, reachable by a *must* transition executing a, ϕ holds
- $\phi\ \{\varphi\}\, U\, \{\varphi'\}\ \phi'$: in a state of a path reached by an action satisfying φ', ϕ' holds, whereas ϕ holds in all preceding states and all actions executed meanwhile along the path satisfy φ
- $\phi\ \{\varphi\}\, U^{\Box}\, \{\varphi'\}\ \phi'$: in a state of a path reached by an action satisfying φ', ϕ' holds, whereas ϕ holds in all preceding states and the path leading to that state is a *must* path along which all actions executed meanwhile satisfy φ

Also the formal semantics of vaCTL is given over MTSs, but in a deontic way by taking into account an MTS's different type of transitions.

[2] The operators $\langle a \rangle^{\Box}$ and $[a]^{\Box}$ represent the classical deontic modalities O and P, resp.

Definition 10. *Let* $(Q, A, \overline{q}, \delta^{\Box}, \delta^{\Diamond})$ *be an MTS, with* $q \in Q$ *and* σ *a full path. Then the satisfaction relation* $\models$ *of* vaCTL *over MTSs is defined as follows:*

- $q \models true$ *always holds*
- $q \models \neg\, \phi$ *iff not* $q \models \phi$
- $q \models \phi \wedge \phi'$ *iff* $q \models \phi$ *and* $q \models \phi'$
- $q \models \langle a \rangle\, \phi$ *iff* $\exists\, q' \in Q$ *such that* $q \xrightarrow{a}_{\Diamond} q'$, *and* $q' \models \phi$
- $q \models [a]\, \phi$ *iff* $\forall\, q' \in Q$ *such that* $q \xrightarrow{a}_{\Diamond} q'$, *we have* $q' \models \phi$
- $q \models \langle a \rangle^{\Box}\, \phi$ *iff* $\exists\, q' \in Q$ *such that* $q \xrightarrow{a}_{\Box} q'$, *and* $q' \models \phi$
- $q \models [a]^{\Box}\, \phi$ *iff* $\forall\, q' \in Q$ *such that* $q \xrightarrow{a}_{\Box} q'$, *we have* $q' \models \phi$
- $q \models E\, \pi$ *iff* $\exists\, \sigma' \in path(q)$ *such that* $\sigma' \models \pi$
- $q \models A\, \pi$ *iff* $\forall\, \sigma' \in path(q)$ *such that* $\sigma' \models \pi$
- $\sigma \models \phi\ \{\varphi\}\, U\, \{\varphi'\}\ \phi'$ *iff* $\exists\, j \geq 1 : \sigma(j) \models \phi'$, $\sigma\{j\} \models \varphi'$, *and* $\sigma(j+1) \models \phi'$, *and* $\forall 1 \leq i < j : \sigma(i) \models \phi$ *and* $\sigma\{i\} \models \varphi$
- $\sigma \models \phi\ \{\varphi\}\, U^{\Box}\, \{\varphi'\}\ \phi'$ *iff* σ *is a must path* $\sigma^{\Box}$ *and* $\sigma^{\Box} \models \phi\ \{\varphi\}\, U\, \{\varphi'\}\ \phi'$ □

Again, further operators can be derived in the usual way. $F\phi$ abbreviates $(true\ \{true\}\, U\, \{true\}\ \phi)$: there exists a future state in which ϕ holds; $AG\, \phi$ abbreviates $\neg EF\, \neg\phi$: in all states on all paths, ϕ holds. $F\, \{\varphi\}\ true$ abbreviates $true\ \{true\}\, U\, \{\varphi\}\ true$: there exists a future state reached by an action satisfying φ; $F^{\Box}\{\varphi\}\ true$ abbreviates $true\, \{true\}\, U^{\Box}\, \{\varphi\}\ true$: there exists a future state of a must path reached by an action satisfying φ; $F^{\Box}\phi$ abbreviates $true\, \{true\}\, U^{\Box}\, \{true\}\, \phi$: there exists a future state of a must path in which ϕ holds; $AG^{\Box}\, \phi$ abbreviates $\neg EF^{\Box}\, \neg\phi$: in all states on all must paths, ϕ holds.

5 Advanced Variability Management

vaCTL can complement the behavioural description of an MTS by expressing the constraints over possible products of a family that an MTS cannot model, i.e. regarding *alternative* features and *requires* and *excludes* inter-feature relations.

We formalise these three types of constraints as follows as vaCTL templates.

Template ALT: Features F1 and F2 are *alternative*:

$$(EF^{\Box}\, \{F1\}\ true \vee EF^{\Box}\, \{F2\}\ true) \wedge \neg(EF\, \{F1\}\ true \wedge EF\, \{F2\}\ true)$$

Template EXC: Feature F1 *excludes* feature F2:

$$((EF\, \{F1\}\ true) \Longrightarrow (AG\, \neg\langle F2\rangle\ true)) \wedge ((EF\, \{F2\}\ true) \Longrightarrow (AG\, \neg\langle F1\rangle\ true))$$

Both these vaCTL templates combine constraints represented by a deontic interpretation with behavioural relations among actions expressed by a temporal part.

Template REQ: Feature F1 *requires* feature F2:

$$(EF\, \{F1\}\ true) \implies (EF^{\Box}\, \{F2\}\ true)$$

Note that this vaCTL template does not imply any ordering among the related features: a product allowing F1 to be performed before F2 cannot be excluded as member of the product family on the basis of this formula (expressing a static relation among features). It is the duty of the behavioural LTS (MTS) description of a product (family) to impose orderings, which can consequently be verified by vaCTL formulae such as the ones we present in the next section.

6 Model Checking a Family of Services

In [3], we defined a global model-checking algorithm for a deontic extension of HML+UNTIL by extending classical algorithms for HML and (A)CTL [5, 6, 34, 11]. Recently, we actually implemented an on-the-fly model-checking algorithm of linear complexity for vaCTL as a particularization of the FMC model checker [31] for ACTL [11] over networks of automata, specified in a CCS-like language [17].

The MTS of Fig. 1 was automatically generated by FMC from the following CCS-like specification of the Travel Agency product family described in Sect. 2:

```
TravAgFam = must(login).Menu

Menu = must(trainreserve).TrainRes + must(flightreserve).FlightRes
     + must(hotelreserve).HotelRes + may(tourreserve).TourRes

TrainRes = must(datainput).must(timetableserv).must(result).
           must(showquotes).(must(choose).Pay + must(reject).Menu)

FlightRes = must(datainput).
          ( may(flightAserv).must(result).must(flightBserv).must(result).
            must(flightCserv).must(result).must(showbestquotes).
            (must(choose).Pay + must(reject).Menu)
          + may(defaultflightserv).must(result).must(showquotes).
            (must(choose).Pay + must(reject).Menu) )

HotelRes = must(datainput).
         ( may(hotelAserv).must(result).must(hotelBserv).must(result).
           must(hotelCserv).must(result).must(showbestquotes).
           (must(choose).Pay + must(reject).Menu)
         + may(defaulthotelserv).must(result).must(showquotes).
           (must(choose).Pay + must(reject).Menu) )

TourRes = must(possibletourserv).must(result).must(showquotes).
          (must(choose).HotelRes + must(reject).Menu)

Pay = must(payserv).(must(paymentOK).TravAgFam + must(denied).Menu)
```

In this TravAgFam specification of the Travel Agency family we actually use typed actions to implement the distinction between may and must transitions.

The above specification is extracted from the requirements of Sect. 2, based on the following assumptions on constraints and behaviour:

1. Only service orchestration is modelled, ignoring data exchange (for instance, a cart maintaining unpaid reservations could be used);
2. Services are invoked by a simple request-response interaction interface: requests are modelled by actions suffixed with ‘serv’, responses by correlated actions like ‘result’, ‘paymentOK’, and ‘denied’ actions;
3. All remaining actions model interaction with the client;

4. The alternative of contacting multiple reservation services is modelled by a sequential invocation of three services. A parallel invocation would be more realistic: while our framework allows this, it would only make our example more complex with no added information concerning family-related aspects.

The rest of this section illustrates the use of FMC. The property "A travel agency service always provides a selection of best quotes" can be formalised in vACTL as

$$AF\ \langle must(showbestquotes)\rangle\ true$$

This property clearly does not hold for TravAgFam, as contacting multiple services for best quotes is an alternative. Moreover, it is only available for flight and hotel reservations. Indeed, upon verifying this property FMC produces the result shown in the screenshot in Fig 2: the formula is false and a path through the MTS is given as counterexample (a successfully completed train reservation).

Fig. 2. FMC screenshot of result of model checking $AF\ \langle must(showbestquotes)\rangle\ true$

Since this alternative is not available for train reservations, which is a service that is mandatory, the above formula is also false for any product of the family. If we were to change it into the following existential path formula (expressing the property "A travel agency service may provide a selection of best quotes")

$$EF\ \langle must(showbestquotes)\rangle\ true$$

then this formula would hold for the family. However, due to the distinction between may and must transitions in MTSs and in Def. 6 in particular, not

all properties (expressed as vaCTL formulae) that hold for a product family (modelled as an MTS) continue to hold for all its products (modelled as LTSs).

We can experiment also this in FMC. Consider for instance the product that is obtained by removing the alternative of contacting multiple services from flight and hotel reservations. Recall from Def. 6 that a product can be obtained from a family by preserving at least all must transitions and turning some of the may transitions that are not must transitions into must transitions as well as removing all of the remaining may transitions. Doing so, we obtain the product shown in the FMC screenshot in Fig. 3. If we now verify the latter formula, FMC produces the result shown in the screenshot in Fig 4: the formula is false as there is no path through the MTS on which $must(showbestquotes)$ is executed.

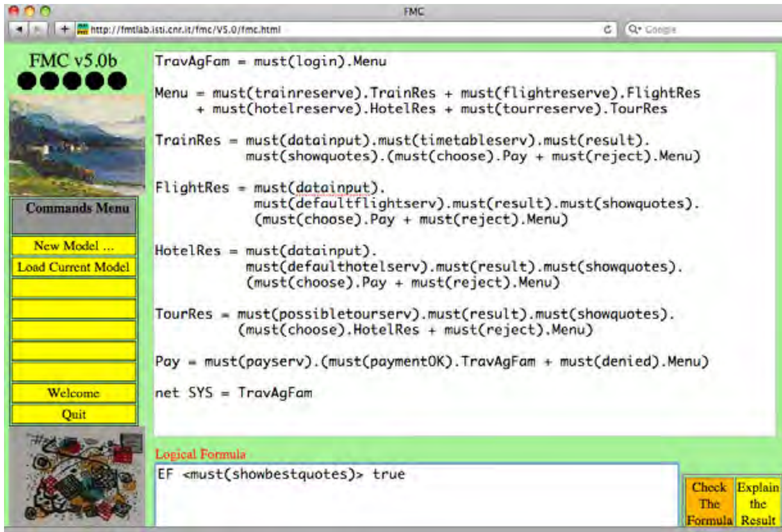


Fig. 3. FMC screenshot of a product obtained from TravAgFam

Now consider the property "A travel agency service must always provide the possibility to reject a proposal (of quotes)". In vaCTL this can be formalised as

$$AG\ [must(result)]\ AF^{\Box}\ \{must(reject)\}\ true$$

FMC can show it holds for the family. Moreover, universal formulae of this form dictate the existence of a must path with certain characteristics, which by Def. 6 are necessarily found in all products. This property thus holds for all products.

Finally, consider the variability constraint that contacting multiple reservation services and contacting a single reservation service are alternative features. This can be formalised in vaCTL by instantiating the Template ALT of Sect. 5:

$$(EF^{\Box}\ \{may(hotelAserv)\}\ true \vee EF^{\Box}\ \{may(defaulthotelserv)\}\ true) \wedge \\ \neg(EF\ \{may(hotelAserv)\}\ true \wedge EF\ \{may(defaulthotelserv)\}\ true)$$

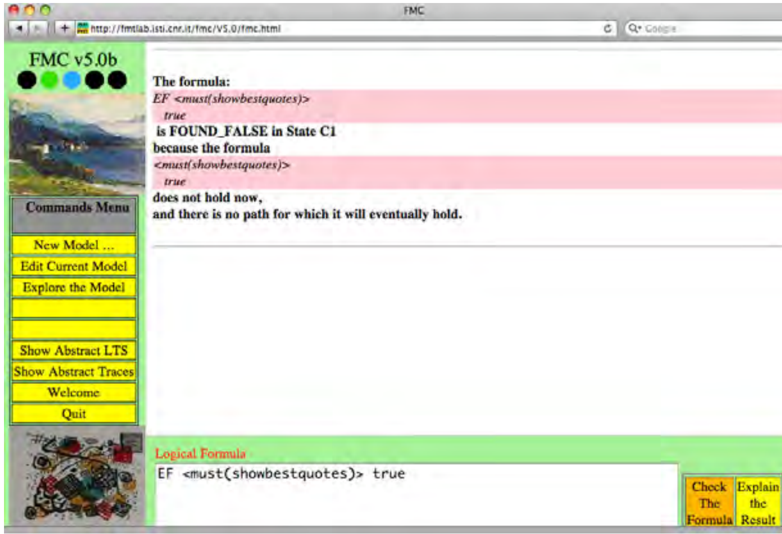


Fig. 4. FMC screenshot of result of model checking $EF \langle must(showbestquotes) \rangle \; true$

As expected, FMC can show this formula is false for the family, but true for the product shown in the FMC screenshot in Fig. 3 (i.e., this product satisfies the above alternative constraint and might hence be a valid product of the family).

FMC can thus be used to experiment with products, using vACTL to guide the derivation of valid products that satisfy the advanced variability constraints by construction. As said before, we recently adapted the algorithm designed in [3] to vACTL, thus allowing us to automatically derive, from an MTS description of a product family *and* an associated set of vACTL formulae expressing further constraints for this family, *all* valid products (a set of LTS descriptions of products, each correct w.r.t. all vACTL constraints). The algorithm's complexity is bounded by $O(n \times m \times w)$, with n the number of transitions of the MTS (family), m the number of LTSs (products) returned, and w the maximum width of may transitions with labels actually referred to in the vACTL formulae (constraints). The actual implementation of this algorithm is ongoing work.

7 Related Work

In [14,27,12,29,3], MTS variants are used to model and analyse product families. Part of our recent work was described previously (cf. also [3] and its references). In [12], we extended MTSs to model advanced variability constraints regarding alternative features. In [27], modal I/O automata were defined as one of the first attempts at behavioural modelling in SPLE. In [19,20], an algebraic approach to behavioural modelling and analysis of product families was developed. In [32], Feature Petri Nets were defined to model the behaviour of product families with a high degree of variability. We discuss some approaches close to ours in detail.

In [14], the authors present an algorithm for checking conformance of LTSs against MTSs according to a given branching relation, i.e. checking conformance of the behaviour of a product against that of its product family. It is a fixed-point algorithm that starts with the Cartesian product of the states and iteratively eliminates pairs that are invalid according to the given relation. The algorithm is implemented in a tool that allows one to check whether or not a given LTS conforms to a given MTS according to a number of different branching relations.

In [29], variable I/O automata are introduced to model product families, together with a model-checking approach to verify conformance of products w.r.t. a family's variability. This is achieved by using variability information in the model-checking algorithm (while exploring the state space an associated variability model is consulted continuously). Properties expressed in CTL [5] are verified by explicit-state model checking, progressing one state at a time.

In [7], an explicit-state model-checking technique to verify linear-time temporal properties over Featured Transition Systems (FTSs) is defined. This results in a means to check that whenever a behavioural property is satisfied by an FTS modelling a product family, then it is also satisfied by every product of that family, and whenever a property is violated, then not only a counterexample is provided but also the products violating the property. In [8], this approach is improved by using symbolic model checking, examining sets of states at a time, and a feature-oriented version of CTL.

In business process modelling, configurable process models were introduced to capture a family of related business process models in a single artifact. Inspired by methods from SPLE [4,30,22,21], alternatives are defined as variation points (cf. [38, 37, 1] and their references). Tool support allows the selection of valid configurations. Such correctness is related only to the static definition of a family. Our research goes beyond this static view: we adopt a specific logic interpreted over MTSs and use it to model check behavioural aspects of families of services.

8 Conclusions and Future Work

We addressed model checking families of services, starting from the addition of variability to a simple action-based branching-time temporal logic interpreted over a basic form of variable transition systems. Services are traditionally modelled with richer transition systems, which need to be addressed in future work. In particular, FMC is closely related to the CMC model checker for SocL (a service-oriented logic) formulae over the process algebra COWS (Calculus for Orchestration of Web Services) [13]. Adding variability management to this tool will allow us to handle families of services that use more complex mechanisms typical of SOC, like asynchronous communication, compensation and correlation.

Acknowledgments. The research reported in this paper was partially funded by two Italian projects: D-ASAP (MIUR–PRIN 2007) and XXL (CNR–RSTL).

We thank F. Mazzanti for stimulating discussions on the vaCTL logic and for his help in using FMC for model checking vaCTL formulae over MTSs.

References

1. van der Aalst, W.M.P., Dumas, M., Gottschalk, F., ter Hofstede, A.H.M., La Rosa, M., Mendling, J.: Preserving correctness during business process model configuration. Formal Asp. Comput. 22(3-4), 459–482 (2010)
2. Åqvist, L.: Deontic Logic. In: Gabbay, D., Guenthner, F. (eds.) Handbook of Philosophical Logic, 2nd edn., vol. 8, pp. 147–264. Kluwer, Dordrecht (2002)
3. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Logical Framework to Deal with Variability. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 43–58. Springer, Heidelberg (2010)
4. Bertolino, A., Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Use Case Description of Requirements for Product Lines. In: [16], pp. 12–18 (2002)
5. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT, Cambridge (1999)
7. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., Raskin, J.-F.: Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In: Proceedings 32nd International Conference on Software Engineering (ICSE 2010), pp. 335–344. ACM Press, New York (2010)
8. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A.: Symbolic Model Checking of Software Product Lines. To appear in: Proceedings 33rd International Conference on Software Engineering (ICSE 2011). ACM Press, New York (2011)
9. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2002)
10. Cohen, S.G., Krut, R.W. (eds.): Proceedings 1st Workshop on Service-Oriented Architectures and Software Product Lines: What is the Connection? (SOAPL 2007). Technical Report CMU/SEI-2008-SR-006, Carnegie Mellon University (2008)
11. De Nicola, R., Vaandrager, F.W.: Three Logics for Branching Bisimulation. J. ACM 42(2), 458–487 (1995)
12. Fantechi, A., Gnesi, S.: Formal Modelling for Product Families Engineering. In: [15], pp. 193–202 (2008)
13. Fantechi, A., Lapadula, A., Pugliese, R., Tiezzi, F., Gnesi, S., Mazzanti, F.: A Logical Verification Methodology for Service-Oriented Computing. To appear in ACM Trans. Softw. Eng. Methodol. (2011)
14. Fischbein, D., Uchitel, S., Braberman, V.A.: A Foundation for Behavioural Conformance in Software Product Line Architectures. In: Hierons, R.M., Muccini, H. (eds.) Proceedings ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA 2006), pp. 39–48. ACM Press, New York (2006)
15. Geppert, B., Pohl, K. (eds.): Proceedings 12th Software Product Lines Conference (SPLC 2008). IEEE Press, Los Alamitos (2008)
16. Geppert, B., Schmid, K. (eds.): Proceedings International Workshop on Requirements Engineering for Product Lines (REPL 2002). Technical Report ALR-2002-033, Avaya Labs Research (2002)
17. Gnesi, S., Mazzanti, F.: On the Fly Verification of Networks of Automata. In: Arabnia, H.R., et al. (eds.) Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 1999), pp. 1040–1046. CSREA Press, Athens (2009)
18. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)

19. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and Model Checking Software Product Lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008)
20. Gruler, A., Leucker, M., Scheidemann, K.D.: Calculating and Modelling Common Parts of Software Product Lines. In: [15], pp. 203–212 (2008)
21. Halmans, G., Pohl, K.: Communicating the Variability of a Software-Product Family to Customers. Software and System Modeling 2(1), 15–36 (2003)
22. John, I., Muthig, D.: Tailoring Use Cases for Product Line Modeling. In: [16], pp. 26–32 (2002)
23. Krut, R.W., Cohen, S.G. (eds.): Proceedings 2nd Workshop on Service-Oriented Architectures and Software Product Lines: Putting Both Together (SOAPL 2008). In: Thiel, S., Pohl, K. (eds.) Workshop Proceedings 12th Software Product Lines Conference (SPLC 2008), Lero Centre, University of Limerick, pp. 115–147 (2008)
24. Krut, R.W., Cohen, S.G. (eds.): Proceedings 3rd Workshop on Service-Oriented Architectures and Software Product Lines: Enhancing Variation (SOAPL 2009). In: Muthig, D., McGregor, J.D. (eds.) Proceedings 13th Software Product Lines Conference (SPLC 2009), pp. 301–302. ACM Press, New York (2009)
25. Larsen, K.G.: Modal Specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1990)
26. Larsen, K.G.: Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. Theor. Comput. Sci. 72(2–3), 265–288 (1990)
27. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O Automata for Interface and Product Line Theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)
28. Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: Proceedings 3rd Annual Symposium on Logic in Computer Science (LICS 1988), pp. 203–210. IEEE Press, Los Alamitos (1988)
29. Lauenroth, K., Pohl, K., Töhning, S.: Model Checking of Domain Artifacts in Product Line Engineering. In: Proceedings 24th International Conference on Automated Software Engineering (ASE 2009), pp. 269–280. IEEE Press, Los Alamitos (2009)
30. von der Maßen, T., Lichter, H.: Modeling Variability by UML Use Case Diagrams. In: [16], pp. 19–25 (2002)
31. Mazzanti, F.: FMC v5.0b (2011), `http://fmt.isti.cnr.it/fmc`
32. Muschevici, R., Clarke, D., Proenca, J.: Feature Petri Nets. In: Schaefer, I., Carbon, R. (eds.) Proceedings 1st International Workshop on Formal Methods in Software Product Line Engineering (FMSPLE 2010). University of Lancaster (2010)
33. Meyer, M.H., Lehnerd, A.P.: The Power of Product Platforms: Building Value and Cost Leadership. The Free Press, New York (1997)
34. Müller-Olm, M., Schmidt, D.A., Steffen, B.: Model-Checking: A Tutorial Introduction. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 330–354. Springer, Heidelberg (1999)
35. Papazoglou, M., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: State of the art and research challenges. IEEE Comput. 40(11), 38–45 (2007)
36. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005)
37. Razavian, M., Khosravi, R.: Modeling Variability in Business Process Models Using UML. In: Proceedings 5th International Conference on Information Technology: New Generations (ITNG 2008), pp. 82–87. IEEE Press, Los Alamitos (2008)
38. Rosemann, M., van der Aalst, W.M.P.: A configurable reference modelling language. Inf. Syst. 32(1), 1–23 (2007)

Partial Order Methods for Statistical Model Checking and Simulation*

Jonathan Bogdoll, Luis María Ferrer Fioriti,
Arnd Hartmanns, and Holger Hermanns

Saarland University – Computer Science, Saarbrücken, Germany

Abstract. Statistical model checking has become a promising technique to circumvent the state space explosion problem in model-based verification. It trades time for memory, via a probabilistic simulation and exploration of the model behaviour—often combined with effective a posteriori hypothesis testing. However, as a simulation-based approach, it can only provide sound verification results if the underlying model is a stochastic process. This drastically limits its applicability in verification, where most models are indeed variations of nondeterministic transition systems. In this paper, we describe a sound extension of statistical model checking to scenarios where nondeterminism is present. We focus on probabilistic automata, and discuss how partial order reduction can be twisted such as to apply statistical model checking to models with spurious nondeterminism. We report on an implementation of this technique and on promising results in the context of verification and dependability analysis of distributed systems.

1 Introduction

Model checking and simulation are complementary techniques in model-based system design. In broad terms, model checkers aim at finding bugs or inconsistencies in a design, and do so by an exhaustive analysis of the system under study. That system is usually modelled as a nondeterministic transition system, which is derived from some program notation. Nondeterminism may be present as a result of concurrent interleaving, or to represent incomplete knowledge about certain implementation decisions at design time, or to reflect the openness of the system to environmental stimuli. When complex designs are modelled by abstract transition systems, nondeterminism is a means to let a small abstract transition system overapproximate the behaviour of the larger concrete design.

Simulation also starts from a model, which is explored in a random manner to get insights into the behaviour of the system. There are many successful simulation environments based on this approach. In artificial intelligence, especially in the planning community, simulation is used on nondeterministic systems with

* This work has been supported by the European Union FP7-ICT project Quasimodo, contract no. 214755, by the DFG as part of SFB/TR 14 AVACS and by the DFG/NWO Bilateral Research Programme ROCKS.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 59–74, 2011.
© IFIP International Federation for Information Processing 2011

success: Markov chain Monte Carlo techniques just walk through a huge state space in a randomised manner, and find solutions for planning problems, such as games like Go [14], often in the form of a tree or trace through the state space. However, while probabilities are used in the analysis, they are meaningless for the model, and no attempt is made to deduce that a trace or tree has a certain probability to appear.

This is different in discrete-event simulation, where the model is a stochastic process, so the future behaviour at each particular state is determined probabilistically. This makes it possible to perform statistical analysis on many simulation runs to arrive at quantitative insights about the system behaviour [15]. This way of analysing a system model is routinely applied in the systems world, but sometimes the rigidity of the modelling is questionable, and comes with notorious suspicions about hidden assumptions that affect the simulation studies [2,8].

In the probabilistic verification world, discrete-event simulation is more and more applied as a vehicle in statistical model checking [5,24,25]. While conventional probabilistic model checkers solve numerical problems in the size of the entirety of the state space, statistical model checkers perform random sampling to walk through the state space, and perform a statistical analysis of the results to provide an estimate for the property under study. This analysis either uses classical estimates such as confidence intervals, or works with hypothesis testing to validate or refute a probabilistic property. Hypothesis testing can be very fast if the actual probability in the system is very far from the required bound, but it is slow if the two values are close. The basic algorithmic machinery is the same for both simulation approaches however. In this paper, we are discussing this basic machinery, and therefore use the terms statistical model checking and simulation interchangeably.

While some studies [23,13] have made efforts to compare the effectiveness of simulation vs. model checking empirically, such a comparison—which we do not focus on in this paper—is inherently problematic. Superficially, the choice between the two seems to be mainly a tradeoff between memory consumption and time: Model checkers generally need to represent the entirety of a state space in memory (or an encoding thereof, e.g. up to a limited depth), but give highly accurate and precise (or at least safe) answers, while simulation walks through the state space, needing only constant memory, and even applies to infinite-size models. But the accuracy and precision of the simulation results depends on the system parameters and especially (that is, logarithmically) on the number of paths explored. Here, theoretical complexity is practical complexity, and as a result, simulation is most competitive time-wise for low accuracy analysis.

Unfortunately, discrete-event simulation faces the additional problem of requiring the model to be a stochastic process, thus free of nondeterminism. This makes it quite far-fetched to assume that it can be applied inside the ordinary verification trajectory, where nondeterminism is not a bug, but a feature, e.g. resulting from the asynchronous, interleaved execution of components in a distributed system. There is no way to resolve nondeterministic choices without introducing additional assumptions. As it is, simulation can thus far only be

used for deterministic models. While this is an inherent limitation of the approach, this paper aims to show ways of mitigating the problem for a practically relevant class of models. The main contribution is an adaptation of partial order reduction methods [3,10,19,22] that enables statistical model checking of probabilistic automata [20,21] while preserving the low memory demands typical for simulation. We investigate how to apply partial order techniques symbolically and find that ignoring variable valuations renders the method impractical. We therefore adapt the existing techniques to work on-the-fly during simulation, and show the correctness of these modifications.

We report on a selection of case studies that provide empirical evidence concerning the potential of the method and our implementation. The case studies also look at continuous timed probabilistic models from the area of architectural dependability evaluation. In fact, the original motivation for our work stems from the observation that the compositional semantics of the architectural design notation Arcade [7,16] gives rise to nondeterminism which is spurious (i.e., irrelevant) *by construction*. Still, simulative approaches were not applicable to it, because of nondeterminism. The present paper overcomes this deficiency, and as such enables—for the first time—a memory-efficient analysis.

2 Preliminaries

We will use networks of probabilistic automata (PA) with variables as the model on which to explain our techniques in this paper. We will later sketch how to extend our techniques to stochastic timed automata (STA) [6], but the additional features of STA—continuous probability distributions and real time—are largely orthogonal to the issues and techniques we focus on.

2.1 Probabilistic Automata

The model of PA [20,21]—which is akin to Markov decision processes (MDPs)—combines nondeterministic and probabilistic choices. Informally, a PA consists of states from which action-labelled transitions lead to distributions over target states. In every state, there is thus a nondeterministic choice over transitions followed by a probabilistic choice over target states.

Formally, a PA is a 4-tuple $P = (S, s_0, \Sigma, \rightarrow)$ where S is a countable set of states, s_0 is the initial state, Σ is a finite set of actions, $\rightarrow \subseteq S \times \Sigma \times \mathsf{Distr}(S)$ is the transition relation and for a set X, $\mathsf{Distr}(X)$ is the set of probability distributions, i.e. functions $X \rightarrow [0, 1]$ such that the sum of the probabilities of all elements of X is 1. $T(s)$ denotes the set of outgoing transitions of a state. We say that P is finite if S and $\rightarrow$ are finite, and P is deterministic if $|T(s)| \leq 1$ for all states s (i.e., the only choices are probabilistic ones).

2.2 Networks of PA

It is often useful to describe complex and, in particular, distributed systems as the parallel composition of several independently specified interacting components.

PA are closed under the interleaving semantics of parallel composition. Using a CSP-style mandatory synchronisation on the actions of the shared alphabet, we define the parallel composition of PA as follows [21]: Given two PA $P_i = (S_i, s_{0_i}, \Sigma_i, \rightarrow_i)$ for $i \in \{1, 2\}$, the parallel composition $P_1 \parallel P_2$ is

$$P_1 \parallel P_2 = (S_1 \times S_2, (s_{0_1}, s_{0_2}), \Sigma_1 \cup \Sigma_2, \rightarrow)$$

where $((s_1, s_2), a, \mu) \in \rightarrow$ if and only if

$$\begin{aligned} & a \in \Sigma_1 \backslash \Sigma_2 \wedge \exists\, (s_1, a, \mu_1) \in \rightarrow_1 \colon \mu = \mu_1 \cdot \{(s_2, 1)\} \\ \text{or } & a \in \Sigma_2 \backslash \Sigma_1 \wedge \exists\, (s_2, a, \mu_2) \in \rightarrow_2 \colon \mu = \mu_2 \cdot \{(s_1, 1)\} \\ \text{or } & a \in \Sigma_1 \cap \Sigma_2 \wedge \exists\, (s_i, a, \mu_i) \in \rightarrow_i \text{ for } i = 1, 2 \colon \mu = \mu_1 \cdot \mu_2. \end{aligned} \tag{1}$$

and $\cdot$ is defined as $(\mu_1 \cdot \mu_2)((s_1, s_2)) = \mu_1(s_1) \cdot \mu_2(s_2)$. A special *silent* action τ is often assumed to be part of all alphabets, but τ-labelled transitions never synchronise. This binary parallel composition operator is associative and commutative; its extension to sets of PA is thus straightforward. We use the term *network of PA* to refer to both a set of PA and its parallel composition.

In Section 4, we will need to identify transitions that appear, in some way, to be the same. For this purpose, we define an equivalence relation $\equiv$ on transitions and denote the equivalence class of t under $\equiv$ by $[t]_\equiv$; this notation can naturally be lifted to sets of transitions. In our setting of a network of PA, it is natural to identify those transitions in the product automaton that result from the same (set of) transitions in the component automata.

2.3 PA with Variables

It is common to extend transition systems with variables that take values in some discrete domain D (see e.g. [4, Chapter 2] for the general recipe), and this can be lifted to PA. We call this class of variable decorated automata VPA. In a VPA P_V with a set of variables V, the transition relation is extended with a guard g—a Boolean expression over the variables that determines whether the transition is enabled—and a transition's target becomes a distribution in $\mathsf{Distr}(S \times \mathsf{Assgn}(V))$ where $\mathsf{Assgn}(V) = \mathit{Val}(V) \rightarrow \mathit{Val}(V)$ is the set of assignment functions and $\mathit{Val}(V) = V \rightarrow D$ the set of valuations for the variables. Let $V(A)$ for $A \in \mathsf{Assgn}(V)$ denote the set of variables that are modified by A in some valuation. For a function f and $S \subseteq \mathit{Dom}(f)$, we write $f|_S$ to denote f with domain restricted to S.

A VPA P_V can be transformed into a *concrete* PA P by unrolling the variables' values into the states: Every state $s \in S$ of P_V is replaced by a state $(s, v) \in S \times \mathit{Val}(V)$, the guards are precomputed according to the states' valuations, and assignments to discrete variables redirect edges to states with matching valuations. Note that P is finite iff P_V is finite and the ranges of all discrete variables used are finite. In this case, we say that the VPA is finitary.

Extending parallel composition to VPA is straightforward. Guards of synchronising transitions are the conjunctions of the original transitions' guards,

while the assignments are the union of the original assignments. We allow global variables, which can be used in all component automata. With global variables and synchronising transitions, it is possible to obtain inconsistent assignments (that assign different values to the same variable at the same instant). Such an assignment is no longer a function, and we consider this to be a modelling error.

By using networks of PA with variables, we choose a model that is compositional, but whose semantics is still a PA—so all results for PA (and MDPs) from the literature apply—and which features a distinction between control (via states) and data (via variables). It is the model underlying, among others, PRISM [18], Modest [6], and prCRL [12].

2.4 Paths, Schedulers and Probabilities

The behaviour of a PA is characterised by its paths or traces. Paths are words from $\rightarrow^\omega \cup \rightarrow^*$ where the start state of every transition must occur with positive probability in the preceding transition's distribution (or be s_0 for the first one), and finite paths end in deadlock states. In a state-based verification setting, where states are labelled with atomic propositions, a *visible transition* is a transition whose execution changes the set of valid atomic propositions.

In order to be able to reason about probabilities for sets of paths, nondeterminism has to be resolved first by a *scheduler* (or *adversary*). This yields a Markov chain for which a probability measure on paths is induced. Since different schedulers lead to different chains, the answer to questions such as "What is the probability of reaching an error state?" is actually a set of probabilities, which always is a singleton for deterministic models. For nondeterministic models, one is typically interested in maximum and minimum probabilities over all possible resolutions of nondeterminism.

For Markov chains, there exists the notion of terminal strongly connected components. The corresponding notion for PA is that of end components: pairs (S_e, T_e) where $S_e \subseteq S$ and $T_e\colon S \to (\rightarrow)$ with $T_e(s) \subseteq T(s)$, such that for all $s \in S_e$ and transitions $(s, a, \mu) \in T_e(s)$, we have $\mu(s') > 0 \Rightarrow s' \in S_e$ and the underlying directed graph of (S_e, T_e) is strongly connected.

3 Nondeterminism in Models for Simulation

Probabilistic model checkers such as PRISM [18] derive probabilistic quantities by first exploring the complete (reachable) state space of the model and afterwards using numerical techniques such as solvers for systems of linear equations or value iteration to compute minimum and maximum probabilities.

If no nondeterminism is present in the model, it is also possible to perform simulation. In this approach, a large number of concrete, finite paths of the model are explored, using random-number generators to resolve probabilistic choices. Only one state of the model is in memory at any time, and for every path explored, we only need to postprocess which of the properties of interest were satisfied and which were not, and this is evaluated with statistical means.

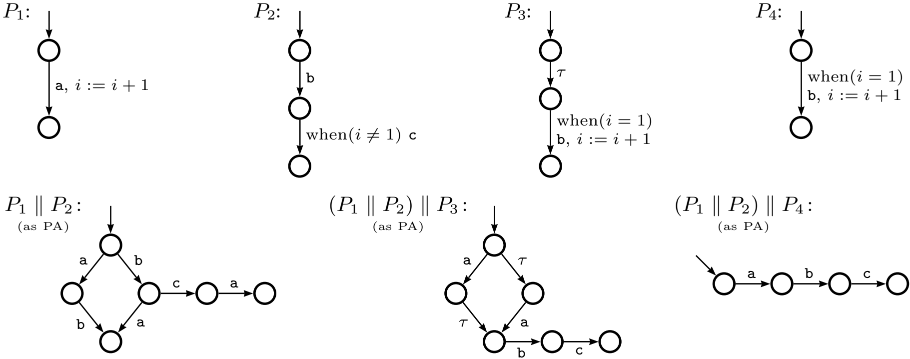


Fig. 1. Nondeterminism and parallel composition

The simulation approach fails in the presence of nondeterminism, because of the absence of a well-defined probability measure. However, the interleaving semantics employed for networks of VPA may result in nondeterminism that is spurious in the sense that the Markov chains induced by arbitrary schedulers all result in the same probability for the property under study to hold. The effects of parallel composition are manifold, as can be seen in the following example.

Example 1. The examples in Figure 1 illustrate possible effects of parallel composition on (non)determinism. i is a global variable that is initially 0. P_1 through P_4 are deterministic VPA, but the parallel composition $P_1 \parallel P_2$ contains a nondeterministic choice. If we ask for the probability of eventually seeing action c, the minimum and maximum values w.r.t. nondeterminism will be different. If we add P_3 to the composition, the result is still nondeterministic, but the minimum and the maximum probabilities are now 1. If we use P_4 instead of P_3 (where P_4 just omits the internal preprocessing step of P_3), the probabilities remain the same, but the final model is fully deterministic. In this case, the parallel composition with P_4 has removed the nondeterminism present in $P_1 \parallel P_2$.

We discuss ways to safely deal with the problem of simulating models with nondeterminism that is introduced by parallel composition in the next section.

4 Partial Order Techniques for Simulation

When simulating nondeterministic models, we need to be sure that the results obtained from a batch of simulation runs, i.e. a set of paths through the PA under study, were not affected by any nondeterministic choices. A sufficient condition for this would be that the model is equivalent—according to some equivalence notion that preserves the properties of interest—to a deterministic one. However, since only a part of the model is explored during the simulation runs, we use the following conditions:

Given a PA $P = (S, s_0, \Sigma, \rightarrow)$ and the set of finite paths $\Pi = \{\pi_1, \ldots, \pi_n\}$, all starting in s_0, encountered during a number of simulation runs, let

$$P|_\Pi = (S_\Pi, s_0, \Sigma, \cup_{i=1}^n \pi_i)$$

where $S_\Pi = \{s_0\} \cup \{s \mid \exists \pi \in \Pi \colon (s', a, \mu) \in \pi \wedge \mu(s) > 0\}$ denote the sub-PA of P explored by the simulation runs. For brevity, we identify a path $\pi \in \rightarrow^*$ with the set of transitions it contains, i.e. implicitly project to $\pi \subseteq \rightarrow$. Let

$$P_\Pi = (S, s_0, \Sigma, \rightarrow_\Pi)$$

where $t = (s, a, \mu) \in \rightarrow_\Pi$ iff $(s \notin S'_\Pi \wedge t \in \rightarrow) \vee (s \in S'_\Pi \wedge \exists \pi \in \Pi \colon t \in \pi)$ and $S'_\Pi = \{s \mid \exists \pi \in \Pi \colon (s, a, \mu) \in \pi\}$ denote P restricted to the decisions taken during the simulation runs. Note that P_Π's behaviour is not restricted for states that were not encountered during simulation.

In order to be sure that the result computed from a number of simulation runs was not influenced by nondeterminism, we require that

C1: $P|_\Pi$ is deterministic, and
C2: $P \sim P_\Pi$ for a relation $\sim$ that preserves the properties we consider.

Useful candidates for $\sim$ would be trace equivalence, simulation or bisimulation relations. In the proofs later in this section, $\sim$ will be stutter equivalence, which preserves quantitative (i.e., the probabilities of) $\text{LTL}_{\setminus\text{X}}$ properties [3].

The central practical question we face is how these conditions can be ensured before or during simulation without negating the memory advantages of the simulation approach. In the area of *model checking*, an efficient method already exists to reduce a model containing spurious nondeterminism resulting from the interleaving of parallel processes to smaller models that contain only those paths of interleavings necessary to not affect the end result, namely partial order reduction [3,10,19,22]. In the remainder of this section, we will first recall how partial order reduction for PA works, and then present approaches to harvest partial order reduction to ensure conditions C1 and C2 for *simulation*.

4.1 Partial Order Reduction for PA

The aim of partial order techniques for model checking is to avoid building the full state space corresponding to a model. Instead, a smaller state space is constructed and analysed where the spurious nondeterministic choices resulting from the interleaving of independent actions are resolved deterministically. The reduced system is not necessarily deterministic, but smaller, which increases the performance and reduces the memory demands of model checking (if the reduction procedure is less expensive than analysing the full model right away).

Partial order reduction techniques for PA [3] are extensions of the *ample set method* [19] for nonprobabilistic systems. The essence is to identify an ample set of transitions $\mathsf{ample}(s)$ for every state $s \in S$ of the PA $P = (S, s_0, \Sigma, \rightarrow)$, yielding a reduced PA $\hat{P} = (\hat{S}, s_0, \Sigma, \hat{\rightarrow})$—where $\hat{S}$ is the smallest set that

Table 1. Conditions for the ample sets

A0 For all states $s \in S$, $\mathsf{ample}(s) \subseteq T(s)$.

A1 If $s \in \hat{S}$ and $\mathsf{ample}(s) \neq T(s)$, then no transition in $\mathsf{ample}(s)$ is visible.

A2 For every path $(t_1 = (s, a, \mu), \ldots, t_n, t, t_{n+1}, \ldots)$ in P where $s \in \hat{S}$ and t is dependent on some transition in $\mathsf{ample}(s)$, there exists an index $i \in \{1, \ldots, n\}$ such that $t_i \in [\mathsf{ample}(s)]_\equiv$.

A3 In each end component (S_e, T_e) of $\hat{P}$, there exists a state $s \in S_e$ that is fully expanded, i.e. $\mathsf{ample}(s) = T(s)$.

A4 If $\mathsf{ample}(s) \neq T(s)$, then $|\mathsf{ample}(s)| = 1$.

satisfies $s_0 \in \hat{S}$ and $(\exists s \in \hat{S} \colon (s, a, \mu) \in \mathsf{ample}(s) \wedge \mu(s') > 0) \Rightarrow s' \in \hat{S}$, and $\hat{\rightarrow} = \cup_{s \in \hat{S}}(\mathsf{ample}(s))$—such that conditions A0-A4 (Table 1) are satisfied.

For partial order reduction, the notion of *(in)dependent* transitions[1] (rule A2) is crucial. Intuitively, the order in which two independent transitions are executed is irrelevant in the sense that they do not disable each other (forward stability) and that executing them in a different order from a given state still leads to the same states with the same probability distribution (commutativity). Formally, two equivalence classes $[t_1']_\equiv \neq [t_2']_\equiv$ of transitions of P are independent iff for all states $s \in S$ with $t_1, t_2 \in T(s)$, $t_1 = (s, a_1, \mu_1) \in [t_1']_\equiv$, $t_2 = (s, a_2, \mu_2) \in [t_2']_\equiv$,

I1: $\mu_1(s') > 0 \Rightarrow t_2 \in [T(s')]_\equiv$ and vice-versa *(forward stability)*, and then also
I2: $\forall s' \in S \colon \sum_{s'' \in S} \mu_1(s'') \cdot \mu_2^{s''}(s') = \sum_{s'' \in S} \mu_2(s'') \cdot \mu_1^{s''}(s')$ *(commutativity)*,

where $\mu_i^{s''}$ is the single element of $\{\mu \mid (s'', a_i, \mu) \in T(s'') \cap [t_i]_\equiv\}$. Checking dependence by testing these conditions on all pairs of transitions for all states of the product automaton is impractical. Instead, sufficient and easy-to-check conditions on the VPA level that rely on the fact that the automaton under study results from a network of VPA are typically used, such as the following (where the g_i are the guards and $\mu_i \in \mathsf{Distr}(S \times \mathsf{Assgn}(V))$):

J1: The sets of VPA that t_1 and t_2 originate from (according to the rules of (1), Section 2.2) must be disjoint, and
J2: $\forall v \colon \mu_1(s', A_1) > 0 \wedge \mu_2(s'', A_2) > 0 \Rightarrow (g_2(v) \Rightarrow g_2(A_1(v)) \wedge A_2(v)|_{V(A_2)} = A_2(A_1(v))|_{V(A_2)})$ and vice-versa.

J1 ensures that the only way for the two transitions to influence each other is via global variables, and J2 makes sure that this does actually not happen, i.e., each transition modifies variables only in ways that do not change the other's assignments or disable its guard. This check can be implemented on a syntactical level for the guards and the expressions occurring in assignments.

Using the ample set method with conditions A0-A4 and I1-I2 (or J1-J2) gives the following result:

[1] By abuse of language, we use the word "transition" when we actually mean "equivalence class of transitions under $\equiv$".

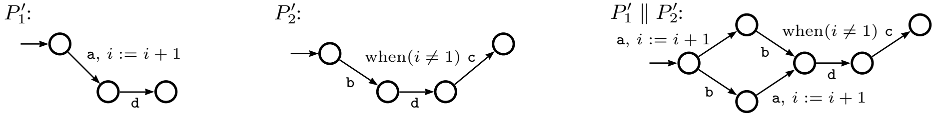


Fig. 2. Spuriousness that can be detected on the VPA level

Theorem 1 ([3]). *If a PA P is reduced to a PA $\hat{P}$ using the ample set method, then $P \sim \hat{P}$ where $\sim$ is stutter equivalence [3].*

For simulation, we are not particularly interested in smaller state spaces, but we can use partial order reduction to distinguish between spurious and actual nondeterminism. A first, naïve approach would be to expand the model given as a network of VPA into a product PA, use partial order reduction to reduce it, and check the result for nondeterminism. However, this neither preserves the low (constant) memory demand typical of simulation nor its general ability to deal with infinite-state systems.

4.2 Deciding Spuriousness on VPA Symbolically

We first investigate the practicality of applying partial order reduction to identify nondeterminism 'symbolically', that is, on the VPA level. By treating data in an abstract manner (as in conditions J1-J2, or using more advanced checks for the independence of operations à la Godefroid [10]), we could hope to avoid the blowup introduced by unrolling the variable valuations. Because synchronisation over shared actions becomes explicit and may remove actual nondeterminism, all nondeterminism that remains may already be spurious.

Example 2. Figure 2 shows an example similar to the ones presented in Figure 1 where this approach works. $P_1' \parallel P_2'$ is shown as a VPA. According to J1-J2, the only (nontrivially) dependent transitions are those labelled a and c. Choosing any singleton ample set for the initial state of $P_1' \parallel P_2'$ satisfies conditions A0 to A4, so the reduced system is deterministic, thus the nondeterminism is spurious.

The advantage of this approach is that it is a preprocessing step that will not induce a performance penalty in the simulation itself. It can also deal with infinite-data systems, but not infinite control. Yet, whenever an intricate but clever use of variables and guards is responsible for the removal of actual nondeterminism, this method will usually fail.

Unfortunately, such models are not uncommon in practice. They appear e.g. as the PA underlying specifications given in a guarded command language—such as that of PRISM—where the state information is entirely encoded in variables.

Example 3. In the two parallel compositions from Figure 1 that did not exhibit actual nondeterminism, the spuriousness cannot be detected by the symbolic approach because of the dependency between the assignment $i := i + 1$ and the guard $i = 1$. In both cases, the problematic transitions would go away once we look at the actually possible variable valuations.

Table 2. On-the-fly conditions for every state s encountered during simulation

A0 For all states $s \in S$, $\mathsf{ample}(s) \subseteq T(s)$.

A1 If $s \in \hat{S}$ and $\mathsf{ample}(s) \neq T(s)$, then no transition in $\mathsf{ample}(s)$ is visible.

A2' Every path in P starting in s has a finite prefix $(t_1, \ldots, t_n)$ of length at most k (i.e. $n \leq k$) such that $t_n \in [\mathsf{ample}(s)]_{\equiv}$ and for all $i \in \{1, \ldots, n-1\}$, t_i is independent of all transitions in $\mathsf{ample}(s)$.

A3' If more than l states have been explored, one of the last l states was fully expanded.

A4' For all states $s \in \hat{S}$, either $|\mathsf{ample}(s)| = 1$ or $T(s) = \varnothing$.

4.3 Bounded On-the-Fly Partial Order Checking

The examples above show that in order to detect more nondeterminism as spurious, we have to move from the (symbolic) VPA to the (concrete) PA level. However, we do not want to build the concrete model in its entirety before simulation. The goal of our second approach is thus to keep the general advantage of simulation in terms of memory demand, but trade in some simulation performance to obtain a spuriousness check that will work better in practice.

The idea is as follows: We simulate the model as usual on the PA level and without any preprocessing. For every state we encounter, we try to identify a valid singleton ample set. If successful, we can use that transition as the next simulation step. If there is more than one singleton ample set, we deterministically select the first set according to a total order on transitions. If all valid ample sets contain more than one transition, we cannot conclude that the nondeterminism between these is spurious, and simulation is aborted with an error message. To keep memory usage constant, the ample sets are not stored.

The ample set construction relies on conditions A0 through A4, but looking at their formulation in Table 1, conditions A2 and A3 cannot be checked on-the-fly without possibly exploring and storing lots of states—potentially the entire PA. To bound this number of states and ensure termination for infinite-state systems, we instead use the conditions shown in Table 2, which are parametric in k and l. Condition A2 is replaced by A2', which bounds the lookahead inherent in A2 to paths of length at most k. Notably, choosing $k = 1$ is equivalent to not checking for spuriousness at all but aborting on the first nondeterministic choice. Instead of checking for end components as in Condition A3, we use A3' that replaces the notion of an end component with the notion of a set of at least l states.

Conditions A0, A1, A2', A3' and A4' enable us to construct ample sets 'on-the-fly' during simulation. This can be proven correct, i.e. conditions C1 and C2 hold whenever the procedure terminates without raising an error:

Lemma 1. *If simulation with on-the-fly partial order checking applied to a PA P encounters the set of paths Π without aborting, then $P|_\Pi$ is deterministic* (C1).

Lemma 2. *If simulation with on-the-fly partial order checking applied to a PA P encounters the set of paths Π without aborting and all $\pi \in \Pi$ end in a fully expanded state, then $P \sim P_\Pi$ where $\sim$ is stutter equivalence* (C2)*.*

Example 4. For the parallel composition $(P_1 \parallel P_2) \parallel P_3$ in Figure 1, all nondeterminism will correctly be identified as spurious for $k \geq 2$ and $l \geq 2$.

Requiring all runs to end in a fully expanded state is a technical requirement that can always be satisfied if all nondeterminism is spurious, and is satisfied by construction for typical simulation run termination criteria.

For finite systems and large enough k and l, this on-the-fly approach will always be superior to the symbolic approach discussed in Section 4.2 in terms of detecting spuriousness—but if k needs to be increased to detect all spurious nondeterminism as such, the performance in terms of runtime and memory demand will degrade. Note, though, that it is not the actual user-chosen value of k that is relevant for the performance penalty, but what we denote k_{min}, the smallest value of k necessary for condition A2' to succeed in the model at hand—if a larger value is chosen for k, A2' will still only cause paths of length k_{min} to be explored[2]. The size of l actually has no performance impact since A3' can be implemented by just counting the distance to the last fully expanded state.

More precisely, the memory usage of this approach is bounded by $b \cdot k_{min}$ where b is the maximum fan-out of the PA; for a given model (i.e., a fixed b) and small k_{min}, we can consider this as constant, thus the approach still has the same flavour as standard simulation with respect to memory demand. Regarding runtime, exploring parts of the state space that are not part of the path currently being explored (up to $b^{k_{min}}$ states per invocation of A2') induces a performance penalty. The magnitude of this penalty is highly dependent on the structure of the model. In practice, however, we expect small values for k_{min}, which limits the penalty, and this is evidenced in our case studies.

The on-the-fly approach also naturally extends to infinite-state systems, both in terms of control and data. In particular, the kind of behaviour that condition A3 is designed to detect—the case of a certain choice being continuously available, but also continuously discarded—can, in an infinite system, also come in via infinite-state "end components", but since A3' strengthens the notion of end components by the notion of sufficiently large sets of states, this is no problem.

To summarise: For large enough k and l, this approach will allow us to use simulation for any network of VPA where the nondeterminism introduced by the parallel composition is spurious. Nevertheless, if conditions J1 and J2 are used to check for independence instead of I1 and I2, nondeterministic choices *internal to the component VPA*, if present, must be removed by synchronization (via shared variables or actions). While avoiding internal nondeterminism is manageable during the modelling phase, parallel composition and the nondeterminism it creates naturally occur in models of distributed or component-based systems.

[2] Our implementation therefore uses large defaults for k and l so the user usually need not worry about these parameters. If simulation aborts, the cause and its location is reported, including how it was detected, which may be that k or l was exceeded.

5 Implementation

A prototype of the on-the-fly approach presented above has been implemented in version 1.3 of modes, a discrete-event simulator for the Modest language with a sound treatment of nondeterminism in simulation. Modest is a high-level compositional modelling formalism for stochastic timed systems [6]. modes can be downloaded at www.modestchecker.net.

The full semantic model underlying Modest is that of stochastic timed automata (STA), which generalise VPA by adding time via clock variables as in timed automata (TA, [1]) as well as infinite and continuous probability distributions, and some of the cases we consider later use this expressiveness. There are, to our knowledge, no results on partial order reduction techniques that deal with real time or continuous probabilistic choices[3]. However, we can treat the additional features of STA in an orthogonal way: The passage of *time* introduces an implicit synchronisation over all component automata by incrementing the values of all clocks. modes thus provides a separate choice of scheduler for time and treats resulting non-zero deterministic delay steps like visible transitions. Nondeterminism can thus be detected as spurious only if the interleaving happens in zero time. In order to correctly detect the spuriousness of nondeterminism in presence of assignments with *continuous probability distributions* (like $x := \mathit{Exponential}(2 \cdot y)$), modes overapproximates by treating them like a nondeterministic assignment to some value from the distribution's support.

6 Case Studies

This section reports on experimental studies with modes on some practically relevant cases. As mentioned in the introduction, we do not aim for a comparison of simulative and numerical model checking approaches in this paper, but instead focus on a comparative study of standard simulation vs. the on-the-fly approach; the existing results (e.g., [23]) then still apply to this new approach modulo its overhead, which we investigate in this section.

Strictly speaking, though, such a comparison is impossible: Standard simulation cannot be applied if the model is nondeterministic. If instead it is deterministic, the on-the-fly approach will have virtually no overhead, because $k_{min} = 1$. To get a meaningful comparison, we use models that are provably equivalent to Markov chains, but contain manifold spurious nondeterminism. We then adjust standard simulation to use uniform distributions to resolve any nondeterministic choices as the zero-overhead baseline for comparison. Due to spuriousness, any resolution would do, and would lead to the same statistical results. For all experiments, modes was run on a 64-bit Core 2 Duo T9300 system; the values for time are for 1000 actual simulation runs, while for memory, the bytes after garbage collections during simulation were measured.

[3] All approaches for TA (see e.g. Minea [17] for one approach and an overview of related work) rely on modified semantics or more restricted models or properties. We are not aware of any approaches for models with continuous distributions.

Table 3. Results for PRISM and modes on the IEEE 802.3 BEB protocol

state space gen.		m.-check	model			uniform sim.		partial order sim.			
states	time	memory	K	N	H	time	memory	time	memory	l	k_{min}
5 371	0 s	1 MB	4	3	3	1 s	1.5 MB	1 s	1.8 MB	16	4
$3.3 \cdot 10^7$	7 s	856 MB	8	7	4	1 s	1.5 MB	2 s	1.4 MB	16	5
$3.8 \cdot 10^{12}$	1052 s	$>$ 100 GB	16	15	5	1 s	1.5 MB	7 s	1.8 MB	16	6
$8.8 \cdot 10^{14}$	4592 s	$>$ 100 GB	16	15	6	1 s	1.6 MB	94 s	3.2 MB	16	7

6.1 Binary Exponential Backoff

We first deal with a model of the IEEE 802.3 Binary Exponential Backoff (BEB) protocol for a set of hosts on a network, adapted from the PRISM model used in [9]. It gives rise to a network of PA, and this makes it possible to use model checking for Modest [11] in conjunction with PRISM (on a 100 GB RAM system). We report the total number of states, the time it took to generate the state space, and the memory necessary to model check a simple property with PRISM[4] using the default "hybrid" engine in Table 3 (left three columns). The model under study is determined by K, the maximum backoff counter value, i.e. the maximum number of slots to wait, by N, the number of times each host tries to seize the channel, and by H, the number of hosts. The remaining columns give results for uniformly resolved nondeterminism and our on-the-fly approach with parameters l and $k = k_{min}$. The number of component VPA is $H + 1$: the hosts plus a global timer process.

We observe that for larger models, model checking with PRISM is hopeless, while simulation scales smoothly. Simulation runs ended when a host seized the channel. In all cases the nondeterminism is identified as spurious for small values of k and l. The values for uniform resolution show that these models are extremely simple to simulate, while the on-the-fly partial order approach induces a time overhead. We see a clear dependence between the number of components and k_{min}, with $k_{min} = H + 1$. In line with our expectations concerning the performance impact from Section 4.3, we see a moderate increase of memory usage, while runtime is affected more drastically by increasing H.

6.2 Arcade Dependability Models

As a second case study, we focus on Arcade models and their translation into Modest [16]. Arcade is a compositional dependability framework based on a semantics in terms of I/O-IMCs [7]. The Arcade models under study are known to be weakly bisimilar to Markov chains; yet they exhibit nondeterminism. We studied two simple examples and two very large case studies from the literature:

1bc-1ruded: One basic component with a dedicated repair unit.
2bc-1rufcfs: Two basic components with one first-come-first-serve repair unit.

[4] PRISM needs more memory for model checking than just for state space generation.

Table 4. Results for modes on Arcade models

model	$\|\|$	t	uniform	partial order	l	k_{min}
1bc-1ruded	3	100	1 s / 1.3 MB	1 s / 1.5 MB	16	2
2bc-1rufcfs	4	100	2 s / 1.1 MB	4 s / 1.1 MB	16	3
dda-scaled	41	15	41 s / 1.5 MB	379 s / 2.6 MB	16	6
rcs-scaled	36	15	24 s / 1.6 MB	132 s / 2.0 MB	16	4

dda-scaled: A distributed disk architecture: 24 disks in 6 clusters, 4 controllers, and 2 processors; repairs are performed in a first-come-first-serve manner.
rcs-scaled: The model of a reactor cooling system with a heat exchanger, two pump lines and a bypass system.

The rates in the last two models are scaled to obtain more frequent events, in a variation of importance sampling. Table 4 shows the results; column $\|\|$ indicates the number of concurrently running component automata of the respective model and t is the simulation time bound (model time). We see that the value of k_{min} varies, but again stays relatively small, even for the models consisting of many components. The impact of enabling the on-the-fly approach is again in line with Section 4.3: Memory usage increases moderately, but the time necessary to complete all runs does increase significantly, though not as drastically as in the previous case study. Overall, these are encouraging results. Improving the performance of the lookahead procedure is the first point on our future work agenda for this currently prototypical implementation.

7 Conclusion

This paper has presented an on-the-fly, partial order reduction-based approach that enables statistical model checking and simulation of probabilistic automata with spurious nondeterminism arising from the parallel composition of largely independent, distributed, and intermittently synchronising components. The tool modes has been shown to be able to apply this technique successfully to two very different case studies. In fact, the work on connecting Arcade to a simulation engine was the nucleus for the work presented here: a method was looked for to certify that simulation results obtained on these models were not affected by any actual nondeterminism.

Acknowledgments. We are grateful to Christel Baier (TU Dresden) for the initial inspiration to combine partial order methods with simulation and to Pedro R. D'Argenio (UNC Cordoba) for fruitful discussions and insights.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. TCS 126(2), 183–235 (1994)
2. Andel, T.R., Yasinsac, A.: On the credibility of MANET simulations. IEEE Computer 39(7), 48–54 (2006)

3. Baier, C., D'Argenio, P.R., Größer, M.: Partial order reduction for probabilistic branching time. Electr. Notes Theor. Comput. Sci. 153(2), 97–116 (2006)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 32–46. Springer, Heidelberg (2010)
6. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MoDeST: A compositional modeling formalism for hard and softly timed systems. IEEE Transactions on Software Engineering 32(10), 812–830 (2006)
7. Boudali, H., Crouzen, P., Haverkort, B.R., Kuntz, M., Stoelinga, M.: Architectural dependability evaluation with Arcade. In: DSN, pp. 512–521. IEEE Computer Society Press, Los Alamitos (2008)
8. Cavin, D., Sasson, Y., Schiper, A.: On the accuracy of MANET simulators. In: POMC, pp. 38–43. ACM, New York (2002)
9. Giro, S., D'Argenio, P.R., Ferrer Fioriti, L.M.: Partial order reduction for probabilistic systems: A revision for distributed schedulers. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 338–353. Springer, Heidelberg (2009)
10. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
11. Hartmanns, A., Hermanns, H.: A Modest approach to checking probabilistic timed automata. In: QEST, pp. 187–196. IEEE Computer Society, Los Alamitos (2009)
12. Katoen, J.P., van de Pol, J., Stoelinga, M., Timmer, M.: A linear process algebraic format for probabilistic systems with data. In: ACSD, pp. 213–222. IEEE Computer Society, Los Alamitos (2010)
13. Katoen, J.P., Zapreev, I.S.: Simulation-based CTMC model checking: An empirical evaluation. In: QEST, pp. 31–40. IEEE Computer Society, Los Alamitos (2009)
14. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
15. Law, A.M., Kelton, D.W.: Simulation Modelling and Analysis. McGraw-Hill Education, Europe (2000)
16. Maaß, S.: Translating Arcade models into MoDeST code. B.Sc. Thesis (May 2010)
17. Minea, M.: Partial order reduction for model checking of timed automata. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 431–446. Springer, Heidelberg (1999)
18. Parker, D.: Implementation of Symbolic Model Checking for Probabilistic Systems. Ph.D. thesis, University of Birmingham (2002)
19. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
20. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. Ph.D. thesis. MIT, Cambridge (1995)
21. Stoelinga, M.: Alea jacta est: Verification of Probabilistic, Real-Time and Parametric Systems. Ph.D. thesis. Katholieke U. Nijmegen, The Netherlands (2002)

22. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
23. Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. Statistical probabilistic model checking: An empirical study. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 46–60. Springer, Heidelberg (2004)
24. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002)
25. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to simulink/stateflow verification. In: HSCC, pp. 243–252. ACM, New York (2010)

Counterexample Generation for Markov Chains Using SMT-Based Bounded Model Checking*

Bettina Braitling[1], Ralf Wimmer[1], Bernd Becker[1], Nils Jansen[2], and Erika Ábrahám[2]

[1] Albert-Ludwigs-University Freiburg, Germany
{braitlin,wimmer,becker}@informatik.uni-freiburg.de
[2] RWTH Aachen University, Germany
{abraham,nils.jansen}@informatik.rwth-aachen.de

Abstract. Generation of counterexamples is a highly important task in the model checking process. In contrast to, e. g., digital circuits where counterexamples typically consist of a single path leading to a critical state of the system, in the probabilistic setting counterexamples may consist of a large number of paths. In order to be able to handle large systems and to use the capabilities of modern SAT-solvers, bounded model checking (BMC) for discrete-time Markov chains was established.

In this paper we introduce the usage of SMT-solving over linear real arithmetic for the BMC procedure. SMT-solving, extending SAT with theories in this context on the one hand leads to a convenient way to express conditions on the probability of certain paths and on the other hand allows to handle Markov reward models. We use the former to find paths with high probability first. This leads to more compact counterexamples. We report on some experiments, which show promising results.

1 Introduction

The verification of formal systems has gained great importance both in research and industry. *Model checking* proves or refutes automatically (i. e. without user interaction) that a system exhibits a given set of properties (see, e. g., [1]). In many cases model checking also provides helpful diagnostic information; in case of a defective system a *counterexample* in form of a witnessing run is returned. The usage of symbolic representations like ordered binary decision diagrams (OBDDs) [2] assures the usability of model checking for many kinds of large systems. However, there are classes of practically relevant systems for which even these OBDD-based methods fail. To fill this gap, *bounded model checking* (BMC) was developed [3]. Thereby the existence of a path of fixed length that refutes the property under consideration is formulated as a satisfiability (SAT) problem.

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and the DFG-Project "CEBug – CounterExample Generation for Stochastic Systems using Bounded Model Checking".

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 75–89, 2011.
© IFIP International Federation for Information Processing 2011

As the size of the corresponding formula is relatively small and as modern SAT-solvers have strongly evolved over the last 15 years, it is not surprising that this approach is very successful.

To model real-life scenarios it is often necessary to cope with specific uncertainties using probabilities. For instance, properties can be formulated in *probabilistic computation tree logic* (PCTL) [4] for models called *discrete-time Markov chains* (DTMCs). The classical model checking approach for this setting is based on the solution of a linear equation system [4]. However, it lacks the generation of counterexamples, since the solution of the equation system yields only the mere probabilities without further information.

To provide a user with such diagnostic information for probabilistic systems, there have been some recent developments during the last few years [5,6,7,8,9]. Contrary to, e. g., LTL model checking for digital systems, counterexamples for a PCTL property may consist of a large number of paths to reach certain probability bounds. Various techniques have been proposed to obtain *compact representations*: incrementally adding the paths with the highest probability [6,8], reducing the strongly connected components of the underlying digraph of a DTMC [5,9], and using regular expressions to describe whole sets of counterexamples [6]. All these techniques rely on an explicit representation of the state space.

Bounded model checking for probabilistic systems in the context of counterexample generation was introduced in [7], which represents the state space symbolically. This procedure can be used to refute probabilistic reachability problems. They can be formulated in PCTL [4] as $\mathcal{P}_{\leq p}(a \,\mathcal{U}\, b)$ with atomic propositions a and b. The meaning of this formula is that the probability to reach a b-state, passing only a-states, may be at most p. The refutation is shown by using a SAT-solver to find paths satisfying the property whose joint probability measure exceeds the bound p. The input of the solver are propositional formulae that are satisfied iff the assignment of the variables corresponds to a sequence of states of the DTMC that represents a path to a target state. This process is significantly accelerated by a loop-detection mechanism, which is used to handle sets of paths which differ only in the number of unfoldings of the same loops.

A drawback of the state-of-the-art BMC procedure for DTMCs is that paths are found in an arbitrary order while for the size of counterexamples it is often advantageous to start with paths of high probability. Moreover, it is desirable to use this procedure for *Markov reward models* (MRMs), which extend DTMCs by the possibility to model costs (or dually rewards) of operations. MRMs allow to verify properties like "The probability to finish the computation with costs larger than c is at most 10^{-3}."

In this paper we therefore extend stochastic BMC in order to handle these problems. Instead of using a SAT-solver, we use the more powerful approach of SMT-solving. SMT stands for *satisfiability modulo theories* and is a generalization of SAT [10]. It allows to express propositional formulae representing paths to target states together with conditions on the probability of such paths. Furthermore, real numbers that are allocated to the states by a cost-function can be added up and restrictions on the accumulated costs of paths can be imposed.

We will also show how this counterexample generation can be combined with minimization techniques for Markov chains in order not only to speed up the counterexample generation but also to obtain more abstract counterexamples.

Organization of the paper. At first, we give a brief introduction to the foundations of DTMCs, counterexamples, Markov reward models, and bisimulation minimization. Section 3 then explains the concept of SMT-solving for this context. In Sect. 4 we describe the results of some experiments we did on well-known test cases. In Sect. 5 we draw a short conclusion and give an outlook to future work on this approach.

2 Foundations

In this section we take a brief look at the basics of discrete-time Markov chains, Markov reward models, and bisimulation minimization.

2.1 Stochastic Models

Definition 1. *Let* AP *be a set of atomic propositions. A* ***discrete-time Markov chain*** *(DTMC) is a tuple $M = (S, s_I, P, L)$ such that S is a finite, non-empty set of states; $s_I \in S$, the initial state; $P : S \times S \to [0, 1]$, the matrix of the one-step transition probabilities; and $L : S \to 2^{\mathrm{AP}}$ a labeling function that assigns each state the set of atomic propositions which hold in that state.*

P has to be a stochastic matrix that satisfies $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$. A (finite or infinite) *path* π is a (finite or infinite) sequence $\pi = s_0 s_1 \dots$ of states such that $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. A finite path $\pi = s_0 s_1 \dots s_n$ has length $|\pi| = n$; for infinite paths we set $|\pi| = \infty$. For $i \leq |\pi|$, π^i denotes the ith state of π, i.e., $\pi^i = s_i$. The ith prefix of a path π is denoted by $\pi\uparrow^i = s_0 s_1 \dots s_i$. The set of finite (infinite) paths starting in state s is called $\mathrm{Path}_s^{\mathrm{fin}}$ ($\mathrm{Path}_s^{\mathrm{inf}}$).

In order to obtain a probability measure for sets of infinite paths we first need to define a probability space for DTMCs:

Definition 2. *Let $M = (S, s_I, P, L)$ be a DTMC and $s \in S$. We define a* ***probability space*** *$\Psi_s = (\mathrm{Path}_s^{\mathrm{inf}}, \Delta_s, \mathrm{Pr}_s)$ such that*

- *Δ_s is the smallest σ-algebra generated by $\mathrm{Path}_s^{\mathrm{inf}}$ and the set of basic cylinders of the paths from $\mathrm{Path}_s^{\mathrm{fin}}$. Thereby, for a finite path $\pi \in \mathrm{Path}_s^{\mathrm{fin}}$, the basic cylinder over π is defined as $\Delta(\pi) = \{\lambda \in \mathrm{Path}_s^{\mathrm{inf}} \mid \lambda\uparrow^{|\pi|} = \pi\}$.*
- *Pr_s is the uniquely defined probability measure that satisfies the equation $\mathrm{Pr}_s(\Delta(s s_1 s_2 \dots s_n)) = P(s, s_1) \cdot P(s_1, s_2) \cdot \dots \cdot P(s_{n-1}, s_n)$ for all basic cylinders $\Delta(s s_1 s_2 \dots s_n)$.*

The properties we want to consider are formulated in PCTL [4] and are of the form $\mathcal{P}_{\leq p}(a \,\mathcal{U}\, b)$ with $a, b \in \mathrm{AP}$. This means that the probability to walk along a path from the initial state to a state in which b holds, with all intermediate states satisfying a, is less than or equal to p. More formally: A path π satisfies

$a\,\mathcal{U}\,b$, written $\pi \vDash a\,\mathcal{U}\,b$, iff $\exists i \geq 0 : \big(b \in L(\pi^i) \wedge \forall 0 \leq j < i : a \in L(\pi^j)\big)$. A state $s \in S$ satisfies the formula $\mathcal{P}_{\leq p}(a\,\mathcal{U}\,b)$ (written $s \vDash \mathcal{P}_{\leq p}(a\,\mathcal{U}\,b)$) iff $\mathrm{Pr}_s\big(\{\pi \in \mathrm{Path}_s^{\mathrm{inf}} \mid \pi \vDash a\,\mathcal{U}\,b\}\big) \leq p$.

Let us assume that such a formula $\mathcal{P}_{\leq p}(a\,\mathcal{U}\,b)$ is violated by a DTMC M. That means $\mathrm{Pr}_{s_I}\big(\{\pi \in \mathrm{Path}_s^{\mathrm{inf}} \mid \pi \vDash a\,\mathcal{U}\,b\}\big) > p$. In this case we want to compute a counterexample which certifies that the formula is indeed violated. Hence a counterexample is a set of paths that satisfy $a\,\mathcal{U}\,b$ and whose joint probability measure exceeds the bound p.

Definition 3. *Let $M = (S, s_I, P, L)$ be a discrete-time Markov chain for which the property $\varphi = \mathcal{P}_{\leq p}(a\,\mathcal{U}\,b)$ is violated in state s_I. An **evidence** for φ is a finite path $\pi \in \mathrm{Path}_{s_I}^{\mathrm{fin}}$ such that $\pi \vDash a\,\mathcal{U}\,b$, but no proper prefix of π satisfies this formula. A **counterexample** is a set $C \subseteq \mathrm{Path}_{s_I}^{\mathrm{fin}}$ of evidences such that $\mathrm{Pr}_{s_I}(C) > p$.*

Han and Katoen have shown that there is always a finite counterexample if $\mathcal{P}_{\leq p}(a\,\mathcal{U}\,b)$ is violated [11].

In order to be able to express properties like *"The probability to reach a target state with costs larger than c is at most p"*, Markov chains have to be extended by so-called reward functions.

Definition 4. *A **Markov reward model** (MRM) is a pair $(M, \boldsymbol{R})$ where $M = (S, s_I, P, L)$ is a DTMC and $\boldsymbol{R} : S \to \mathbb{R}$ a real-valued reward function.*

Rewards can be used to model costs of certain operations, to count steps or to measure given gratifications. The variant of rewards we use here are *state rewards*. One could also assign reward values to transitions instead of states (so-called *transition rewards*). We restrict ourselves to state rewards; all techniques that are developed in this paper can also be applied to transition rewards.

Let $\pi \in \mathrm{Path}_s^{\mathrm{fin}}$ be a finite path and $\mathbf{R}$ a reward function. The *accumulated reward* of π is given by $\mathbf{R}(\pi) = \sum_{i=0}^{|\pi|-1} \mathbf{R}(\pi^i)$. Note that the reward is granted when leaving a state.

We extend the PCTL-properties from above to Markov reward models. For a (possibly unbounded) interval $\mathcal{I} \subseteq \mathbb{R}$, a path π satisfies the property $a\,\mathcal{U}^{\mathcal{I}}\,b$, written $\pi \vDash a\,\mathcal{U}^{\mathcal{I}}\,b$, if there is $0 \leq i \leq |\pi|$ such that $b \in L(\pi^i)$, $\mathbf{R}(\pi{\uparrow}^i) \in \mathcal{I}$, and $a \in L(\pi^j)$ for all $0 \leq j < i$. A state $s \in S$ satisfies $\mathcal{P}_{\leq p}(a\,\mathcal{U}^{\mathcal{I}}\,b)$ if $\mathrm{Pr}_s\big(\{\pi \in \mathrm{Path}_s^{\mathrm{inf}} \mid \pi \vDash a\,\mathcal{U}^{\mathcal{I}}\,b\}\big) \leq p$. Our techniques can be extended in a straightforward way to $\mathcal{I}$ being the union of a finite set of intervals. Please note that the bounded until operator of PCTL is a special case of a reward condition.

Our goal is to compute counterexamples to refute such reward properties using bounded model checking (BMC).

2.2 Bisimulation Minimization

For the generation of counterexamples we work with a symbolic representation of the DTMC and the reward function. Symbolic representations have the advantage that the size of the representation is not directly correlated with the

size of the represented system. The representation can be smaller by orders of magnitude. By using algorithms whose running time only depends on the size of the representation, very large state spaces can be handled.

But even if the symbolic representation of a system is small, the number of paths that are needed for a counterexample can be very large. In order to reduce the number of paths, we first compute the smallest system that has the same behavior w. r. t. probabilities and rewards as the original one. This in general not only reduces the number of states, but also the number of paths, because all paths that stepwise consist of equivalent states are mapped onto a single path in the minimized system.

The methods for computing such minimal systems are based on bisimulation relations. A bisimulation groups states whose behavior is indistinguishable into equivalence classes [12]:

Definition 5. *Let $M = (S, s_I, P, L)$ be a DTMC and $\boldsymbol{R} : S \to \mathbb{R}$ a reward function for M. A partition $\boldsymbol{P}$ of S is a **bisimulation** if the following holds for all equivalence classes C of $\boldsymbol{P}$ and for all $s, t \in S$ such that s and t are contained in the same block of $\boldsymbol{P}$:*

$$L(s) = L(t), \quad \boldsymbol{R}(s) = \boldsymbol{R}(t), \quad \textit{and} \quad P(s, \mathrm{C}) = P(t, \mathrm{C}),$$

*where $P(s, \mathrm{C}) = \sum_{s' \in \mathrm{C}} P(s, s')$. Two states $s, t \in S$ are **bisimilar** ($s \sim t$) if there exists a bisimulation $\boldsymbol{P}$ such that s and t are contained in the same block of $\boldsymbol{P}$.*

The equivalence classes of bisimilarity and the coarsest bisimulation partition coincide. The equivalence class of a state $s \in S$ w. r. t. a bisimulation $\mathbf{P}$ is denoted by $[s]_P$. The classes of the bisimulation now become the states of a new DTMC.

Definition 6. *Let $M = (S, s_I, P, L)$ be a DTMC, $\boldsymbol{R}$ a reward function for M, and $\boldsymbol{P}$ be a bisimulation. The **bisimulation quotient** is the DTMC $(\boldsymbol{P}, \mathrm{C}_I, P', L')$ with reward function $\boldsymbol{R}'$ such that*

- *For all $\mathrm{C}, \mathrm{C}' \in \boldsymbol{P}$: $P'(\mathrm{C}, \mathrm{C}') = P(s, \mathrm{C}')$ for an arbitrary $s \in \mathrm{C}$,*
- *C_I is the block that contains the initial state s_I of M,*
- *$\forall\, \mathrm{C} \in \boldsymbol{P} : L'(\mathrm{C}) = L(s)$ for an arbitrary state $s \in \mathrm{C}$, and*
- *$\forall\, \mathrm{C} \in \boldsymbol{P} : \boldsymbol{R}'(\mathrm{C}) = \boldsymbol{R}(s)$ for an arbitrary state $s \in \mathrm{C}$.*

The quotient can be considerably smaller than the original system. At the same time it still satisfies the same PCTL and reward properties. All analyses can therefore be carried out on the quotient system instead of the larger original Markov model. For symbolic algorithms to compute the bisimulation quotient of a DTMC or MRM, we refer the reader to, e. g., [13,14].

3 SMT-Based Bounded Model Checking for Counterexample Generation

In this section we show how counterexamples can be computed using an SMT-based formulation of bounded model checking (BMC). BMC has already been

applied in [7] for this purpose but in a purely propositional variant. The main drawback of the existing approach is that the propositional BMC formula only contains information about the mere existence of a path, but not about its probability measure. Hence there is no direct possibility to control the SAT-solver such that paths with high probability measure are preferred.

Here we propose the usage of a more powerful formulation than purely propositional logic. The *satisfiability modulo theories* (SMT) problem is a generalization of the propositional satisfiability (SAT) problem. It combines propositional logic with arithmetic reasoning. Using SMT we can create a formula that is only satisfied by paths with a certain minimal probability. This enables us to apply binary search to find paths first whose probability measures differ from the probability of the most probable path of the current unrolling depth by at most a constant $\varepsilon > 0$. Furthermore, this formulation allows us to compute counterexamples for Markov reward models.

Since often even counterexamples of minimal size are too large to be useful, we minimize the DTMC or MRM before determining a counterexample. The counterexample obtained for the minimized system is much more compact since all equivalent evidences of the original system are merged into a single path of the minimized system. Given a counterexample of the minimized system, it can easily be translated back to the original system – either resulting in an ordinary counterexample or in one representative evidence per equivalence class.

We first describe the SMT-based BMC formula and compare it to the SAT-based approach of [7]. Then we show how to handle Markov reward models with an arbitrary number of reward functions. Finally we demonstrate how minimization can be applied to make the counterexample generation more efficient.

3.1 SMT-Based and SAT-Based SBMC

Stochastic bounded model checking (SBMC) as proposed in [7] is based on the formulation as a (propositional) satisfiability problem that a path of a certain length exists which exhibits a given property. To handle formulae of the form $\mathcal{P}_{\leq p}(a\,\mathcal{U}\,b)$, states that satisfy either $\neg a$ or b are made absorbing by removing all out-going edges. This reduces the problem to reaching a b-state.

After this preprocessing step, SBMC uses a SAT-solver to determine satisfying solutions for the BMC formula, which has the following structure:

$$\mathrm{SBMC}(k) := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T_{\mathrm{SAT}}(s_i, s_{i+1}) \wedge L_b(s_k). \tag{1}$$

$I(s_0)$ is a formula that is satisfied iff the assignment of s_0 corresponds to the initial state s_I. Accordingly, $T_{\mathrm{SAT}}(s_i, s_{i+1})$ represents the transition from a state s_i to a successor s_{i+1}, such that $T_{\mathrm{SAT}}(s_i, s_{i+1})$ is satisfied iff $P(s_i, s_{i+1}) > 0$, and $L_b(s_k)$ is the property which holds for s_k iff $b \in L(s_k)$. Each satisfying assignment of formula (1) corresponds to a path of length k that is an evidence for $a\,\mathcal{U}\,b$. The authors of [7] describe how this formula can efficiently be constructed from a BDD-based representation of the Markov chain. First they construct an

OBDD for the underlying directed graph by mapping positive probabilities in the MTBDD representation of the probability matrix to 1. Then they apply Tseitin-transformation [15] to the OBDDs to construct the different parts of the BMC formula. Fig. 1 gives a rough overview of the BMC procedure for a DTMC $M = (S, s_I, P, L)$ and a formula $\varphi = \mathcal{P}_{\leq p}(a\,\mathcal{U}\,b)$.

The probability measure of a path, however, is not considered within this formula. The probability measure has to be computed after the path has been found. Using an SMT-based formulation we can create a modified version of the BMC formula that allows us to put restrictions on the probability measure of evidences.

We define an extended transition formula $T_{\mathrm{SMT}}(s_i, s_{i+1}, \hat{p}_i)$ that is satisfied iff $P(s_i, s_{i+1}) > 0$ and the variable $\hat{p}_i$ is assigned the logarithm of this probability. Following [6], the logarithm is used in order to turn the multiplication of the probabilities along a path into a summation. This leads to SMT formulae over linear real arithmetic that can be decided efficiently. This variant of the transition predicate can also be generated from an MTBDD representation of the matrix $P(s, s')$ of transition probabilities using Tseitin-transformation. In contrast to the propositional case, where 'true' and 'false' are used as atomic formulae for the terminal nodes, we use '$\hat{p} = \log v$' for leaves with value $v > 0$ and 'false' for the leaf 0.

```
procedure COUNTEREXAMPLE
    preprocess(M, φ);
    C ← ∅;
    k ← 0;
    Pr(C) ← 0;
    φ ← BMCformula(k, M, φ);
    while Pr(C) ≤ p do
        solution ← solve(φ)
        if solution = UNSAT then
            k ← k + 1;
            φ ← BMCformula(k, M, φ);
        else
            π ← CreatePath(solution);
            C ← C ∪̇ {π};
            Pr(C) ← Pr(C) + Pr(π);
            φ ← φ ∧ ExcludePath(π);
        end if
    end while
    return C
end procedure
```

Fig. 1. Counterexample generation

To let the solver find only evidences with a probability measure of at least $p_t \in [0, 1]$, we add the condition $\sum_{i=0}^{k-1} \hat{p}_i > \log p_t$ to the BMC formula:

$$\mathrm{SSBMC}(k) := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T_{\mathrm{SMT}}(s_i, s_{i+1}, \hat{p}_i) \wedge L_b(s_k) \wedge \left(\sum_{i=0}^{k-1} \hat{p}_i \geq \log p_t \right). \quad (2)$$

This formula is given to an SMT-solver. If the solver returns SAT, the satisfying assignment represents an evidence with a probability measure of at least p_t. If we get UNSAT, there is no such path of the current length.

This enables us to do a binary search for evidences with a high probability measure in $\mathrm{Path}^{\mathrm{fin}}_{s_I}$. In principle we could determine the most probable path of the current unrolling depth first, then the one with the second highest probability, and so on. For efficiency reasons we apply a different strategy: First, we look for paths which already exceed our probability threshold p. If this fails, we search

for paths with a probability greater or equal to $p/2$. If we have found all existing paths with such a probability and the accumulated probability mass is still less than p, we start looking for paths with a probability greater or equal $p/4$, and so on. The value p_t is decreased, until either all paths with length k have been found or the accumulated probability of the set of evidences is high enough. If the latter is not the case, we proceed to depth $k+1$.

The optimizations for SBMC – exploiting loops and improvements of the clauses to exclude paths from the search space –, which are proposed in [7], work for SSBMC as well and without further modification.

3.2 Counterexamples for MRMs

With the proposed method, we are not only able to handle DTMCs, but to handle MRMs as well. To consider the reward structure of an MRM during the search for paths, we need to integrate the rewards into the transition relation. In the preprocessing step, which is needed to turn the PCTL-Until property $\mathcal{P}_{\leq p}(a\,\mathcal{U}^{\mathcal{I}}\,b)$ into a reachability property, we must not cut the transitions from all b-states. There must be the possibility to extend a path if its accumulated reward is not within $\mathcal{I}$. Thus we cut the transitions only from states s with $a \notin L(s)$.

After that we extend the transition formula by a formula for the rewards in a similar way as we have done for the probabilities. For each time frame $0 \leq i < k$ we introduce a variable $\hat{r}_i$ such that the formula $R(s_i, \hat{r}_i)$ is satisfied iff $\hat{r}_i$ carries the value $\mathbf{R}(s_i)$. This formula can be created from an MTBDD representation of the reward function using Tseitin-transformation. The resulting SMT-formula, which takes rewards into account, has the following structure:

$$\begin{aligned}\text{R-SSBMC}(k) := \text{SSBMC}(k) \\ \wedge \bigwedge_{i=0}^{k-1} R(s_i, \hat{r}_i) \wedge \left[\min(\mathcal{I}) \leq \left(\sum_{i=0}^{k-1} \hat{r}_i\right) \leq \max(\mathcal{I})\right]. \end{aligned} \tag{3}$$

Since b-states are no longer absorbing when using this formula, we have to make sure that we do not find paths of which we have already found a proper prefix. This can be guaranteed by adding clauses to the formula that exclude all paths that were found in previous iterations.

Using this technique it is possible to handle an arbitrary number of reward functions at the same time. We just add distinct variables for each reward function and build several reward sums which are checked against the corresponding intervals.

3.3 Bisimulation Minimization

We can use bisimulation minimization (cf. Sec. 2.2) as a further preprocessing step after cutting unnecessary transitions, but before constructing a counterexample. Since in many cases the quotient system is considerably smaller than the original system, fewer paths are needed for a counterexample.

Every path in the bisimulation quotient represents a set of paths in the original system. To be more precise, let $\pi, \pi' \in \text{Path}^{\text{fin}}(s)$ be two evidences in a DTMC $M = (S, s_I, P, L)$. They are equivalent if $|\pi| = |\pi'|$ and for all $0 \leq i \leq |\pi|$: $\pi^i \sim \pi'^i$. All equivalent paths correspond to the same path in the quotient system, namely to the path $\pi_Q = [\pi^0]_\sim [\pi^1]_\sim \ldots [\pi^{|\pi|}]_\sim$. The probability of π_Q is the sum of the probabilities of the represented original paths that start in s_I. Therefore in general fewer paths are needed in the quotient system for a counterexample.

Once a counterexample has been determined for the quotient DTMC, its paths can be translated back to the original system. The result is a tree that contains all evidences that are stepwise equivalent to the given path. Let us assume that $\pi_Q = C_0 C_1 \ldots C_n$ with $C_0 = [s_I]_\sim$ is such a path in the quotient system. We set $\text{succ}(s) = \{s' \in S \mid P(s, s') > 0\}$. The root of the tree is the initial state s_I that corresponds to C_0 on π_Q. If s_i is a node of the tree that corresponds to C_i on π_Q, its successor nodes in the tree are $\text{succ}(s) \cap C_{i+1}$. They correspond to C_{i+1}. The probability measures of the resulting tree and of π_Q coincide.

In the next section we show the effectiveness of SMT-based counterexample generation and of this optimization on a set of example benchmarks.

4 Experimental Results

We have implemented the SSBMC-tool in C++ with the refinements we presented above, including the optimizations from [7]. We used Yices [16] as the underlying SMT-solver.

Our benchmarks are instances of the following four case studies:

(1) The *contract signing protocol* [17,18] (`contract`) provides a fair exchange of signatures between two parties A and B over a network. If B has obtained the signature of A, the protocol ensures that A will receive the signature of B as well. In our experiments we examine the violation of this property.

(2) The task of the *crowds protocol* [19] (`crowds`) is to provide a method for anonymous web browsing. The communication of a single user is hidden by routing it randomly within a group of similar users. The model contains corrupt group members who try to discern the source of a message. We explore the probability that these corrupt members succeed.

(3) The *leader election protocol* [20] (`leader`) consists of N processors in a synchronous ring. Every processor chooses a random number from $\{0, \ldots, M\}$. The numbers are passed around the ring, the processor with the highest number becomes the leader if this number is unique. If this fails, a new round starts. We provide a certificate that the probability to eventually elect a leader exceeds a given bound.

(4) The *self-stabilizing minimal spanning tree algorithm* [21] (`mst`) computes a minimal spanning tree in a network with N nodes. Due to, e. g., communication failures the system may leave the state where a result has been computed, and recover after some time. For our experiments we explore the probability that the algorithm does not recover within a given number of k steps. This model

is of particular interest to us, because it has a large number of paths, but only a few have a notable probability mass. Because SAT-based BMC does not find paths with high probability first, the hope is that the SMT approach finds much smaller counterexamples in less time.

We used the probabilistic model checker PRISM [22] to generate symbolic representations of these case studies. All experiments were run on a Dual Core AMD Opteron processor with 2.4 GHz per core and 4 GB of main memory. Any computation which took more than two hours ("– TO –") or needed more than 2 GB of memory ("– MO –") was aborted.

In order to compare the results we ran the same benchmarks under the same conditions also with the SBMC-tool from [7].

4.1 Counterexamples for DTMCs

In this section we present the results for the SSBMC procedure. The evaluation of bisimulation minimization and counterexamples for MRMs follows in subsequent sections.

Table 1 contains the results for counterexample generation for DTMCs using the SMT- and the SAT-approach. The first and the second column contain the names of the model and the probability threshold p. In the third column the maximal unrolling depth $k_{\max}$ is listed. The blocks titled "SSBMC" and "SBMC" present the results for SMT-based approach and the SAT-based approach, respectively. The columns "#paths", "time", and "memory" contain the number of paths in the counterexample, the needed computation time in seconds, and the memory consumption in megabytes.

Both tools are able to solve a subset of the instances within the given limits. The SSBMC-tool was aborted due to the memory limit for one instance of the `contract` benchmark, most instances of the `crowds` benchmark, and some instances of the `leader` protocol. The reason is the high memory consumption of the SMT-solver YICES compared to the SAT-solver MINISAT, which is used in SBMC. The `contract` and `leader` benchmarks exhibit the property that all evidences of the same length have the same probability. Therefore the number of paths coincide for both tools. The running time of SSBMC for these instances is slightly higher due to the modified binary search (cf. Sec. 3.1) which increases the number of calls to the solver. The search strategy is also the reason why SSBMC may need slightly more evidences than SBMC, which is the case here for `crowds02_07`, where the probabilities of the different evidences differ only by a small amount.

Notable are the `mst` instances. They contain a large number of evidences with very different probabilities. SBMC is not able to compute a counterexample within the given time limit, while SSBMC returns less than 5000 paths in under 3 minutes. A more detailed look showed that SBMC had computed 633 729 paths for `mst-14` before the timeout occurred. This is because the SMT-approach is able to compute the more probable evidences first, only few of which are needed for a counterexample.

Table 1. Counterexample generation for DTMCs using SMT- and SAT-based BMC

			SSBMC			SBMC		
Name	p	$k_{\max}$	#paths	time	mem.	#paths	time	mem.
contract05_03	0.500	37	513	14.85	124.72	513	25.04	74.84
contract05_08	0.500	87	513	134.92	889.32	513	399.08	694.71
contract06_03	0.500	44	2049	70.21	388.81	2049	140.36	124.79
contract06_08	0.500	92		– MO –		2049	2620.12	1181.76
contract07_03	0.500	51	8193	474.59	1510.28	8193	627.56	198.27
crowds02_07	0.100	39	21306	1006.05	1394.26	21116	417.89	96.11
crowds04_06	0.050	34		– MO –		80912	1468.94	278.17
crowds04_07	0.050	34		– MO –		80926	1773.83	286.38
crowds05_05	0.050	36		– MO –			– MO –	
crowds10_05	0.025	27		– MO –		52795	1654.83	188.51
leader03_04	0.990	8	276	0.70	27.45	276	0.76	14.55
leader03_08	0.990	8	1979	28.21	101.41	1979	25.76	66.48
leader04_03	0.990	20		– MO –		347454	3959.88	720.56
leader05_02	0.900	42		– MO –			– MO –	
leader06_02	0.900	84		– MO –			– TO –	
mst14	0.990	14	426	11.64	42.99		– TO –	
mst15	0.049	15	4531	98.58	148.82		– TO –	
mst16	0.047	16	4648	107.27	158.25		– TO –	
mst18	0.036	18	4073	109.26	164.24		– TO –	
mst20	0.034	20	452	19.57	58.21		– TO –	

4.2 Bisimulation

In Table 2 we evaluate the impact of bisimulation minimization on running time and memory consumption for SSBMC and SBMC. In Table 3 we show the results of bisimulation with subsequent back-translation of the abstract evidences. In the latter case, we converted the minimized paths completely, i. e., we obtained *all* original paths which were represented by an abstract path. The running time listed in Table 2 and Table 3 include the time for bisimulation minimization, counterexample generation, and, in Table 3, path conversion. Again, the block titled "SSBMC" ("SBMC") contains the result for the SMT-based (SAT-based) approach.

As we can see, bisimulation minimization causes in most cases a significant decrease in computation time and required memory of SSBMC and SBMC. This effect is somewhat alleviated when the paths are translated back to the original system, although not dramatically. Some instances of the contract and the `crowds` protocol which could not be solved by SSBMC within the given time and memory bounds become actually solvable, even with path conversion.

However, there is also an exception to this trend: The path conversion for the minimal spanning tree benchmark cannot be done within the given memory bounds. This is due to the fact that one abstract path in these benchmarks represents a great number of original paths, too many to convert them all. While the SMT-based approach without bisimulation minimization can pick the paths

Table 2. Counterexample generation with bisimulation minimization

			SSBMC			SBMC		
Name	p	$k_{\max}$	#paths	time	mem.	#paths	time	mem.
contract05_03	0.500	37	7	23.41	61.42	7	48.50	54.55
contract05_08	0.500	87	7	467.88	179.29	7	829.43	94.72
contract06_03	0.500	44	8	57.75	84.94	8	144.06	64.56
contract06_08	0.500	97	8	1205.37	209.47	8	2213.94	120.83
contract07_03	0.500	51	9	169.37	123.93	9	407.63	79.13
crowds02_07	0.100	39	21069	629.24	633.34	21279	191.31	101.34
crowds04_06	0.050	34	3459	106.17	164.95	3624	44.19	48.76
crowds04_07	0.050	34	3459	123.32	167.61	3555	46.97	50.67
crowds05_05	0.050	36	6347	184.06	251.70	8435	55.20	50.47
crowds10_05	0.025	27	251	12.74	71.20	347	10.22	47.84
leader03_04	0.990	8	2	0.12	21.68	2	0.10	12.06
leader03_08	0.990	8	2	0.64	33.93	2	0.68	24.31
leader04_03	0.990	20	4	0.31	25.12	4	0.32	15.50
leader05_02	0.900	42	7	0.24	22.94	7	0.24	12.05
leader06_02	0.900	84	12	0.79	26.19	12	0.89	14.09
mst14	0.990	14	9	2.28	38.10	396	0.78	16.85
mst15	0.049	15	13	1.85	39.36	1648	1.40	17.92
mst16	0.047	16	13	2.19	39.73	5632	4.00	25.99
mst18	0.036	18	9	3.57	43.64	27475	30.82	69.69
mst20	0.034	20	7	5.02	44.91	20290	25.83	56.63

with the highest probability, leading to a small counterexample, this is not possible after bisimulation minimization. If we compute the most probable paths in the minimized system, they can correspond to huge numbers of paths in the original system each of which has only negligible probability.

4.3 Rewards

For the reward model checking we integrated rewards into the `leader` election protocol. A reward of 1 is granted whenever a new round starts, i. e., when each processor chooses a new ID. We want to analyze how high the probability measure is that at least three rounds are needed until a leader has been elected. For the experiments we restricted our search to a maximal path length of $\text{depth}_{\max}$. We computed all paths with the given property up to this length.

The results are shown in Table 4. The first column contains the name of the model, the second the maximal depth $\text{depth}_{\max}$. The subsequent columns contain the accumulated probability measure p, the number of found paths, the computation time (in seconds) and the required memory (in megabytes).

Compared to the results in Section 4.1 we need more and longer paths to get a noteworthy probability measure. The computation time and the amount of consumed memory are higher accordingly.

We also integrated bisimulation minimization for Markov reward models with state rewards. In this case only an appropriate initial partition has to be provided

Table 3. Counterexample generation with bisimulation and path conversion

			SSBMC			SBMC		
Name	p	$k_{\max}$	#paths	time	mem.	#paths	time	mem.
contract05_03	0.500	37	520	27.87	72.87	520	50.86	54.36
contract05_08	0.500	87	520	859.34	182.64	520	1259.94	98.06
contract06_03	0.500	44	2064	72.27	91.44	2064	168.28	75.50
contract06_08	0.500	97	2064	4181.45	224.31	2064	5927.5	131.68
contract07_03	0.500	51	8224	230.69	149.60	8224	450.20	103.13
crowds02_07	0.100	39	21069	812.51	699.80	21279	313.99	168.42
crowds04_06	0.050	34	81227	408.69	406.16	81138	218.81	289.62
crowds04_07	0.050	34	81227	426.69	409.29	80705	221.80	290.00
crowds05_05	0.050	36		– MO –			– MO –	
crowds10_05	0.025	27	54323	198.30	194.38	53507	119.83	168.93
leader03_04	0.990	8	300	0.21	21.68	300	0.16	12.06
leader03_08	0.990	8	4536	4.12	33.93	4536	3.67	24.31
leader04_03	0.990	20	583440	483.99	1123.74	583440	300.24	1108.83
leader05_02	0.900	42		– MO –			– MO –	
leader06_02	0.900	84		– MO –			– MO –	
mst14	0.990	14		– MO –			– MO –	
mst15	0.049	15		– MO –			– MO –	
mst16	0.047	16		– MO –			– MO –	
mst18	0.036	18		– MO –			– MO –	
mst20	0.034	20		– MO –			– MO –	

Table 4. Counterexample generation for MRMs using SMT-based BMC

Model	$\text{depth}_{\max}$	p	#paths	time	mem.
leader03_04	23	0.00391	20160	290.62	434.45
leader03_05	19	0.00160	18000	290.16	379.25
leader03_06	19	0.00077	52920	2134.05	1147.73
leader03_08	15	0.00024	32256	1050.96	709.98
leader04_02	25	0.21875	37376	912.11	1110.54
leader04_03	19	0.04979	26460	589.94	761.34
leader05_02	23	0.14771	4840	40.16	163.06
leader06_02	25	0.12378	32448	907.33	1360.11
leader08_02	28		– MO –		

for bisimulation computation. The results are shown in Table 5. For the `leader` benchmarks bisimulation minimization results in a enormous compression of the state space and a respective reduction of the number of evidences. Since the back-translation can be done efficiently and yields for the `leader` benchmark the same counterexample as the version without minimization, using bisimulation minimization as a preprocessing and back-translation as a postprocessing step reduces the overall computation time and memory consumption.

Table 5. Counterexample generation for MRMs with bisimulation minimization

		without conv.				with conv.			
Model	$\text{depth}_{\text{max}}$	p	#paths	time	mem.	p	#paths	time	mem.
leader03_04	23	0.00391	3	0.15	21.70	0.00391	20160	9.37	59.36
leader03_05	19	0.00160	2	0.25	24.84	0.00160	18000	10.68	54.93
leader03_06	19	0.00077	2	0.38	26.96	0.00077	52920	34.30	119.10
leader03_08	15	0.00024	1	0.66	33.96	0.00024	32256	25.95	69.56
leader04_02	25	0.21875	3	0.11	21.29	0.21875	37376	20.43	92.57
leader04_03	19	0.04979	1	0.29	25.15	0.04979	26460	18.55	72.67
leader05_02	23	0.14771	1	0.18	22.12	0.14771	4840	3.23	31.45
leader06_02	25	0.12378	1	0.30	23.74	0.12378	32448	31.71	87.96
leader08_02	28	0.05493	1	0.98	31.33	– MO –			

5 Conclusion

In our paper we showed how SMT and BMC can be combined to efficiently generate counterexamples for DTMCs. Our SSBMC method can handle systems which could not be handled with the previously presented SAT-based approach [7]. With SSBMC it is also possible to analyze Markov reward models with an arbitrary number of reward functions. This enables us to refute reachability properties which impose restrictions on the accumulated reward of paths.

Furthermore we presented bisimulation minimization as a preprocessing step for SSBMC. It reduces the number of evidences needed for a counterexample by merging equivalent paths. This way the counterexample generation is accelerated and the memory consumption is reduced. We are able to convert these minimized paths back to the original ones.

As future work we will carry out a more detailed experimental evaluation of our methods on appropriate models. Furthermore, there are still many possible optimizations for our tool. So far, reward model checking and bisimulation minimization only work without the loop detection optimization from [7]. These combinations have to be implemented.

We plan to optimize the search for paths with higher probabilities. We want to include the BDD-based method from [23], which applies Dijkstra's shortest path algorithm to compute the most probable evidences, into our tool as another preprocessing step. The advantage of this method is that it yields counterexamples of minimal size. Preliminary experiments have shown that this method is efficient as long as the number of paths is small. Since the BDD sizes grow with each path that has been found, memory consumption and running time grow accordingly. We want to combine this approach with the SMT-approach by using the BDD-based method as long as it is efficient and switch to the SMT-approach when the BDD-approach becomes too expensive.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)

3. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
4. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
5. Andrés, M.E., D'Argenio, P., van Rossum, P.: Significant diagnostic counterexamples in probabilistic model checking. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 129–148. Springer, Heidelberg (2009)
6. Han, T., Katoen, J.P., Damman, B.: Counterexample generation in probabilistic model checking. IEEE Trans. on Software Engineering 35(2), 241–257 (2009)
7. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time markov chains using bounded model checking. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 366–380. Springer, Heidelberg (2009)
8. Aljazzar, H., Leue, S.: Directed explicit state-space search in the generation of counterexamples for stochastic model checking. IEEE Trans. on Software Engineering 36(1), 37–60 (2010)
9. Ábrahám, E., Jansen, N., Wimmer, R., Katoen, J.P., Becker, B.: DTMC model checking by SCC reduction. In: Proc. of QEST, IEEE CS, pp. 37–46 (2010)
10. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems (TOPLAS) 1, 245–257 (1979)
11. Han, T., Katoen, J.-P.: Counterexamples in probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 72–86. Springer, Heidelberg (2007)
12. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.N.: Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007)
13. Derisavi, S.: Signature-based symbolic algorithm for optimal Markov chain lumping. In: Proc. of QEST, IEEE CS, pp. 141–150 (2007)
14. Wimmer, R., Derisavi, S., Hermanns, H.: Symbolic partition refinement with automatic balancing of time and space. Perf. Eval. 67(9), 815–835 (2010)
15. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic Part 2, 115–125 (1970)
16. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
17. Even, S., Goldreich, O., Lempel, A.: A randomized protocol for signing contracts. Communications of the ACM 28(6), 637–647 (1985)
18. Norman, G., Shmatikov, V.: Analysis of probabilistic contract signing. Journal of Computer Security 14(6), 561–589 (2006)
19. Reiter, M., Rubin, A.: Crowds: Anonymity for web transactions. ACM Transactions on Information and System Security (TISSEC) 1(1), 66–92 (1998)
20. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Information and Computation 88(1), 60–87 (1990)
21. Collin, Z., Dolev, S.: Self-stabilizing depth-first search. Information Processing Letters 49(6), 297–301 (1994)
22. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
23. Günther, M., Schuster, J., Siegle, M.: Symbolic calculation of k-shortest paths and related measures with the stochastic process algebra tool Caspa. In: Int'l Workshop on Dynamic Aspects in Dependability Models for Fault-Tolerant Systems (DYADEM-FTS), pp. 13–18. ACM Press, New York (2010)

Adaptable Processes (Extended Abstract)*

Mario Bravetti[1], Cinzia Di Giusto[2], Jorge A. Pérez[3], and Gianluigi Zavattaro[1]

[1] Laboratory FOCUS (Università di Bologna / INRIA), Italy
[2] INRIA Rhône-Alpes, Grenoble, France
[3] CITI - Dept. of Computer Science, FCT New University of Lisbon, Portugal

Abstract. We propose the concept of *adaptable processes* as a way of overcoming the limitations that process calculi have for describing patterns of dynamic *process evolution*. Such patterns rely on direct ways of controlling the behavior and location of *running* processes, and so they are at the heart of the *adaptation* capabilities present in many modern concurrent systems. Adaptable processes have a location and are sensible to actions of *dynamic update* at runtime. This allows to express a wide range of evolvability patterns for processes. We introduce a core calculus of adaptable processes and propose two verification problems for them: *bounded* and *eventual adaptation*. While the former ensures that at most k consecutive errors will arise in future states, the latter ensures that if the system enters into an error state then it will eventually reach a correct state. We study the (un)decidability of these two problems in different fragments of the calculus. Rather than a specification language, our calculus intends to be a basis for investigating the fundamental properties of evolvable processes and for developing richer languages with evolvability capabilities.

1 Introduction

Process calculi aim at describing formally the behavior of concurrent systems. A leading motivation in the development of process calculi has been properly capturing the *dynamic character* of concurrent behavior. In fact, much of the success of the π-calculus can be fairly attributed to the way it departs from CCS [16] so as to describe mobile systems in which communication topologies can change dynamically. Subsequent developments can be explained similarly. For instance, the introduction of the Ambient calculus can be justified by the need of describing mobility with considerations of space/context awareness, frequent in distributed systems. A commonality to these developments is that dynamic behavior in a system is realized through a number of *local changes* in its topology: mobility in the π-calculus arises from local changes in single linkages, while spatial mobility in the Ambient calculus arises from local changes in the containment relations in the hierarchy of ambients. This way, the combined effect of local changes suffices to explain dynamic behavior at the global (system) level.

There are, however, interesting forms of dynamic behavior that cannot be satisfactorily described as a combination of local changes, in the sense just discussed. These are

* Supported by the French project ANR-2010-SEGI-013 - AEOLUS, the EU integrated project HATS, the Fondation de Coopération Scientifique Digiteo Triangle de la Physique, and FCT / MCTES - Carnegie Mellon Portugal Program, grant NGN-44-2009-12 - INTERFACES.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 90–105, 2011.
© IFIP International Federation for Information Processing 2011

behavioral patterns which concern change at the *process* level, and thus describe *process evolution* along time. In general, forms of process evolvability are characterized by an enhanced control/awareness over the current behavior and location of running processes. Remarkably, this increased control is central to the *adaptation* capabilities by which processes modify their behavior in response to exceptional circumstances in their environment. As a simple example, consider the behavior of a process scheduler in an operating system, which executes, suspends, and resumes a given set of threads. In order to model the scheduler, the threads, and their evolution, we would need mechanisms for *direct* process manipulation, which appear quite difficult to represent by means of link mobility only. More precisely, it is not clear at all how to represent the *intermittent evolution* of a thread under the scheduler's control: that is, precise ways of describing that its behavior "disappears" (when the scheduler suspends the thread) and "appears" (when the scheduler resumes the thread). Emerging applications and programming paradigms provide challenging examples of evolvable processes. In workflow applications, we would like to be able to replace or update a running activity, without affecting the rest of the workflow. We might also like to suspend the execution of a set of activities, or even suspend and relocate the whole workflow. Similarly, in component-based systems we would like to reconfigure parts of a component, a whole component, or even groups of components. Also, in cloud computing scenarios, applications rely on scaling policies that remove and add instances of computational power at runtime. These are context-aware policies that dynamically adapt the application to the user's demand. (We discuss these examples in detail in Section 3.) At the heart of these applications are patterns of process evolution and adaptation which we find very difficult (if not impossible) to represent in existing process calculi.

In an attempt to address these shortcomings, in this paper we introduce the concept of *adaptable processes*. Adaptable processes have a location and are sensible to actions of *dynamic update* at runtime. While locations are useful to designate and structure processes into hierarchies, dynamic update actions implement a sort of built-in adaptation mechanism. We illustrate this novel concept by introducing $\mathcal{E}$, a process calculus of adaptable processes (Section 2). The $\mathcal{E}$ calculus arises as a variant of CCS without restriction and relabeling, and extended with primitive notions of location and dynamic update. In $\mathcal{E}$, $a[P]$ denotes the adaptable process P located at a. Name a acts as a *transparent locality*: P can evolve on its own as well as interact freely with its surrounding environment. Localities can be nested, and are sensible to interactions with *update actions*. An update action $\widetilde{a}\{U\}$ decrees the update of the adaptable process at a with the behavior defined by U, a *context* with zero or more holes, denoted by $\bullet$. The *evolution* of $a[P]$ is realized by its interaction with the update action $\widetilde{a}\{U\}$, which leads to process $U\{^P/_\bullet\}$, i.e., the process obtained by replacing every hole in U by P.

Rather than a specification language, the $\mathcal{E}$ calculus intends to be a basis for investigating the fundamental properties of evolvable processes. In this presentation, we focus on two *verification problems* associated to $\mathcal{E}$ processes and their (un)decidability. The first one, k*-bounded adaptation* (abbreviated $\mathcal{BA}$) ensures that, given a finite k, at most k consecutive error states can arise in computations of the system—including those reachable as a result of dynamic updates. The second problem, *eventual adaptation* (abbreviated $\mathcal{EA}$), is similar but weaker: it ensures that if the system enters into an

error state then it will eventually reach a correct state. In addition to error occurrence, the correctness of adaptable processes must consider the fact that the number of modifications (i.e. update actions) that can be applied to the system is typically *unknown*. For this reason, we consider $\mathcal{BA}$ and $\mathcal{EA}$ in conjunction with the notion of *cluster* of adaptable processes. Given a system P and a set M of possible updates that can be applied to it at runtime, its associated cluster considers P together with an arbitrary number of instances of the updates in M. This way, a cluster formalizes adaptation and correctness properties of an initial system configuration (represented by an *aggregation* of adaptable processes) in the presence of arbitrarily many sources of update actions.

A summary of our main results follows. $\mathcal{E}$ is shown to be Turing complete, and both $\mathcal{BA}$ and $\mathcal{EA}$ are shown to be *undecidable* for $\mathcal{E}$ processes (Section 4). Turing completeness of $\mathcal{E}$ says much on the expressive power of update actions. In fact, it is known that fragments of CCS without restriction can be translated into finite Petri nets (see for instance the discussion in [7]), and so they are not Turing complete. Update actions in $\mathcal{E}$ thus allow to "jump" from finite Petri nets to a Turing complete model. We then move to $\mathcal{E}^-$, the fragment of $\mathcal{E}$ in which *update patterns*—the context U in $\widetilde{a}\{U\}$—are restricted in such a way that holes $\bullet$ cannot appear behind prefixes. In $\mathcal{E}^-$, $\mathcal{BA}$ turns out to be *decidable* (Section 5), while $\mathcal{EA}$ remains *undecidable* (Section 6). Interestingly, $\mathcal{EA}$ is already undecidable in two fragments of $\mathcal{E}^-$: one fragment in which adaptable processes cannot be neither removed nor created by update actions—thus characterizing systems in which the topology of nested adaptable processes is *static*—and another fragment in which update patterns are required to have exactly one hole—thus characterizing systems in which running processes cannot be neither deleted nor replicated.

The undecidability results are obtained via encodings of Minsky machines [17]. While the encoding in $\mathcal{E}$ *faithfully* simulates the behavior of the Minsky machine, this is not the case for the encoding in $\mathcal{E}^-$, in which only finitely many steps are wrongly simulated. The decidability of $\mathcal{BA}$ in $\mathcal{E}^-$ is proved by resorting to the theory of *well-structured transition systems* [1,13] and its associated results. In our case, such a theory must be coupled with Kruskal's theorem [15] (so as to deal with terms whose syntactical tree structure has an unbounded depth), and with the calculation of the predecessors of target terms in the context of trees with unbounded depth (so as to deal with arbitrary aggregations and dynamic updates that may generate new nested adaptable processes). This combination of techniques and results proved to be very challenging.

In Section 7 we give some concluding remarks and review related work. Detailed definitions and proofs can be found in [5].

2 The $\mathcal{E}$ Calculus: Syntax, Semantics, Adaptation Problems

The $\mathcal{E}$ calculus extends the fragment of CCS without restriction and relabeling (and with replication instead of recursion) with *update prefixes* $\widetilde{a}\{U\}$ and a primitive notion of *adaptable processes* $a[P]$. As in CCS, in $\mathcal{E}$ processes can perform actions or synchronize on them. We presuppose a countable set $\mathcal{N}$ of names, ranged over by $a, b \ldots$, possibly decorated as $\overline{a}, \overline{b} \ldots$ and $\widetilde{a}, \widetilde{b} \ldots$. As customary, we use a and $\overline{a}$ to denote atomic input and output actions, respectively.

$$\text{COMP}\quad a[P] \xrightarrow{a[P]} \star \qquad \text{SUM}\quad \textstyle\sum_{i\in I} \alpha_i.\, P_i \xrightarrow{\alpha_i} P_i \qquad \text{REPL}\quad !\alpha.\, P \xrightarrow{\alpha} P \parallel !\alpha.\, P$$

$$\text{ACT1}\ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1' \parallel P_2} \qquad \text{TAU1}\ \frac{P_1 \xrightarrow{a} P_1' \qquad P_2 \xrightarrow{\overline{a}} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'}$$

$$\text{LOC}\ \frac{P \xrightarrow{\alpha} P'}{a[P] \xrightarrow{\alpha} a[P']} \qquad \text{TAU3}\ \frac{P_1 \xrightarrow{a[Q]} P_1' \qquad P_2 \xrightarrow{\widetilde{a}\{U\}} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1'\{U\{Q/\bullet\}/\star\} \parallel P_2'}$$

Fig. 1. LTS for $\mathcal{E}$. Symmetric counterparts of ACT1, TAU1, and TAU3 have been omitted

Definition 1. *The class of $\mathcal{E}$ processes, is described by the following grammar:*

$$P ::= a[P] \quad | \quad P \parallel P \quad | \quad \sum_{i\in I} \pi_i.\, P_i \quad | \quad !\pi.\, P \qquad\qquad \pi ::= a \quad | \quad \overline{a} \quad | \quad \widetilde{a}\{U\}$$

where the U in $\widetilde{a}\{U\}$ represents a context, *i.e., a process with zero or more holes $\bullet$.*

In $\mathcal{E}$, $a[P]$ denotes the *adaptable process* P located at a. Notice that a acts as a *transparent* locality: process P can evolve on its own, and interact freely with external processes. Given $a[P]$, we often refer to P as the process *residing at* a. Adaptable processes can be *updated*: when an update prefix $\widetilde{a}\{U\}$ is able to interact with the adaptable process $a[P]$, the current state of the adaptable process at a is used to fill the holes in context U (see below). The rest of the syntax follows standard lines. Parallel composition $P \parallel Q$ decrees the concurrent execution of P and Q. A process $\pi.\, P$ performs prefix π and then behaves as P. Given an index set $I = \{1, .., n\}$, the guarded sum $\sum_{i\in I} \pi_i.\, P_i$ represents an exclusive choice over $\pi_1.\, P_1, \ldots, \pi_n.\, P_n$. As usual, we write $\pi_1.\, P_1 + \pi_2.\, P_2$ if $|I| = 2$, $\pi_1.\, P_1$ if $|I| = 1$, and $\mathbf{0}$ if I is empty. Process $!\,\pi.\, P$ defines (guarded) replication, by which infinitely many copies of P are triggered by prefix π.

The operational semantics of $\mathcal{E}$ is given in terms of a Labeled Transition System (LTS). It is denoted $\xrightarrow{\alpha}$, and defined by the rules in Figure 1. There are five kinds of actions; we use α to range over them. In addition to the standard input, output, and τ actions, we consider two complementary actions for process update: $\widetilde{a}\{U\}$ and $a[P]$. The former represents the offering of an update U for the adaptable process at a; the latter expresses the fact that the adaptable process at a, with current state P, is ready to update. We often use $\longrightarrow$ to denote $\xrightarrow{\tau}$. Intuitions for some rules of the LTS follow. Rule COMP represents the contribution of a process at a in an update operation; we use $\star$ to denote a unique placeholder. Rule LOC formalizes the above mentioned transparency of localities. Process evolvability is formalized by rule TAU3 and its symmetric. The update action offers a process U for updating the process at a which, by virtue of rule COMP, is represented in P_1' by $\star$. Process Q—the current state of a— is then used to fill the holes in U that do not appear inside other update prefixes: we use $U\{Q/\bullet\}$ to denote the process U in which every occurrence of $\bullet$ has been replaced by Q in this way. The update action is completed by replacing all occurrences of $\star$ in P_1' with $U\{Q/\bullet\}$.

Notice that nested update prefixes are allowed in our language and treated consistently by the semantics. This way, for instance $\widetilde{a}\{\bullet \parallel \widetilde{b}\{\bullet\}\}$ is an allowed update prefix,

and the holes at a and at b are actually different. In fact, and as detailed above, the hole inside b is not instantiated in case of an update operation at a.

We move on to formally define the adaptation problems that we shall consider for $\mathcal{E}$ processes. They formalize the problem of reaching error states in a system with adaptable processes. Both problems are stated in terms of observability predicates, or *barbs*. Our definition of barbs is parameterized on the number of repetitions of a given signal.

Definition 2 (Barbs). *Let P be an $\mathcal{E}$ process, and let α be an action in $\{a, \overline{a} \mid a \in \mathcal{N}\}$. We write $P \downarrow_\alpha$ if there exists a P' such that $P \xrightarrow{\alpha} P'$. Moreover:*

- *Given $k > 0$, we write $P \Downarrow_\alpha^k$ iff there exist $Q_1, \ldots, Q_k$ such that $P \longrightarrow^* Q_1 \longrightarrow \ldots \longrightarrow Q_k$ with $Q_i \downarrow_\alpha$, for every $i \in \{1, \ldots, k\}$.*
- *We write $P \Downarrow_\alpha^\omega$ iff there exists an infinite computation $P \longrightarrow^* Q_1 \longrightarrow Q_2 \longrightarrow \ldots$ with $Q_i \downarrow_\alpha$ for every $i \in \mathbb{N}$.*

We use $\not\Downarrow_\alpha^k$ and $\not\Downarrow_\alpha^\omega$ to denote the negation of $\Downarrow_\alpha^k$ and $\Downarrow_\alpha^\omega$, with the expected meaning.

We consider two instances of the problem of reaching an error in an aggregation of terms, or *cluster*. A *cluster* is a process obtained as the parallel composition of an initial process P with an arbitrary set of processes M representing its possible subsequent modifications. That is, processes in M may contain update actions for the adaptable processes in P, and therefore may potentially lead to its modification (evolution).

Definition 3 (Cluster). *Let P be an $\mathcal{E}$ process and $M = \{P_1, \ldots, P_n\}$. The set of clusters is defined as: $\mathcal{CS}_P^M = \{P \parallel \prod^{m_1} P_1 \parallel \cdots \parallel \prod^{m_n} P_n \mid m_1, \ldots, m_n \in \mathbb{N}\}$.*

The *adaptation* problems below formalize correctness of clusters with respect to their ability for recovering from errors by means of update actions. More precisely, given a set of clusters $\mathcal{CS}_P^M$ and a barb e (signaling an error), we would like to know if all computations of processes in $\mathcal{CS}_P^M$ (1) have *at most* k consecutive states exhibiting e, or (2) *eventually* reach a state in which the barb on e is no longer observable.

Definition 4 (Adaptation Problems). *The* bounded adaptation *problem ($\mathcal{BA}$) consists in checking whether given an initial process P, a set of processes M, a barb e, and $k > 0$, for all $R \in \mathcal{CS}_P^M$, $R \not\Downarrow_e^k$ holds.*

Similarly, the eventual adaptation *problem ($\mathcal{EA}$) consists in checking whether given an initial process P, a set of processes M and a barb e, for all $R \in \mathcal{CS}_P^M$, $R \not\Downarrow_e^\omega$ holds.*

3 Adaptable Processes, by Examples

Next we discuss concrete instances of adaptable processes in several settings.

Mode Transfer Operators. In [3], dynamic behavior at the process level is defined by means of two so-called *mode transfer* operators. Given processes P and Q, the *disrupt* operator starts executing P but at any moment it may abandon P and execute Q. The *interrupt* operator is similar, but it returns to execute P once Q emits a termination signal. We can represent similar mechanisms in $\mathcal{E}$ as follows:

$$\mathsf{disrupt}_a(P, Q) \stackrel{\mathsf{def}}{=} a[P] \parallel \widetilde{a}\{Q\} \qquad \mathsf{interrupt}_a(P, Q) \stackrel{\mathsf{def}}{=} a[P] \parallel \widetilde{a}\{Q \parallel t_Q.\bullet\}$$

Assuming that P evolves on its own to P', the semantics of $\mathcal{E}$ decrees that $\mathsf{disrupt}_a(P, Q)$ may evolve either to $a[P'] \parallel \widetilde{a}\{Q\}$ (as locality a is transparent) or to Q (which represents disruption at a). By assuming that P was able to evolve into P'' just before being interrupted, process $\mathsf{interrupt}_a(P, Q)$ evolves to $Q \parallel t_Q.\, P''$. Notice how defining P as an adaptable process at a is enough to formalize its potential disruption/interruption. Above, we assume that a is not used in P and Q, and that termination of Q is signaled at the designated name t_Q.

Dynamic Update in Workflow Applications. Designing business/enterprise applications in terms of *workflows* is a common practice nowadays. A workflow is a conceptual unit that describes how a number of *activities* coordinate to achieve a particular task. A workflow can be seen as a container of activities; such activities are usually defined in terms of simpler ones, and may be software-based (such as, e.g., "retrieve credit information from the database") or may depend on human intervention (such as, e.g., "obtain the signed authorization from the credit supervisor"). As such, workflows are typically long-running and have a transactional character. A workflow-based application usually consists of a *workflow runtime engine* that contains a number of workflows running concurrently on top of it; a *workflow base library* on which activities may rely on; and of a number of *runtime services*, which are application dependent and implement things such as transaction handling and communication with other applications. A simple abstraction of a workflow application is the following $\mathcal{E}$ process:

$$App \stackrel{\mathsf{def}}{=} \mathsf{wfa}\Big[\, \mathsf{we}\big[\mathtt{WE} \parallel \mathsf{W}_1 \parallel \cdots \parallel \mathsf{W}_k \parallel \mathsf{wbl}[\mathtt{BL}]\big] \parallel \mathtt{S}_1 \parallel \cdots \parallel \mathtt{S}_j \,\Big]$$

where the application is modeled as an adaptable process wfa which contains a workflow engine we and a number of runtime services $\mathtt{S}_1, \ldots, \mathtt{S}_j$. In turn, the workflow engine contains a number of workflows $\mathsf{W}_1, \ldots, \mathsf{W}_k$, a process $\mathtt{WE}$ (which represents the engine's behavior and is left unspecified), and an adaptable process wbl representing the base library (also left unspecified). As described before, each workflow is composed of a number of activities. We model each W_i as an adaptable process w_i containing a process $\mathtt{WL}_i$ —which specifies the workflow's logic—, and n activities. Each of them is formalized as an adaptable process a_j and an *execution environment* env_j:

$$\mathsf{W}_i = \mathsf{w}_i\Big[\, \mathtt{WL}_i \parallel \prod_{j=1}^{n} \big(\mathsf{env}_j[\mathtt{P}_j] \parallel \mathsf{a}_j\big[!u_j.\, \widetilde{\mathsf{env}}_j\{\mathsf{env}_j[\bullet \parallel \mathtt{A}_j]\}\big]\big) \,\Big]$$

The current state of the activity j is represented by process $\mathtt{P}_j$ running in env_j. Locality a_j contains an update action for env_j, which is guarded by u_j and always available. As defined above, such an update action allows to add process $\mathtt{A}_j$ to the current state of the execution environment of j. It can also be seen as a procedure that is yet not active, and that becomes active only upon reception of an output at u_j from, e.g., $\mathtt{WL}_i$. Notice that by defining update actions on a_j (inside $\mathtt{WL}_i$, for instance) we can describe the evolution of the execution environment. An example of this added flexibility is the process

$$U_1 = !\,\mathsf{replace}_j.\, \widetilde{\mathsf{a}_j}\big\{a_j[!u_j.\, \widetilde{\mathsf{env}}_j\{\mathsf{env}_j[\bullet \parallel \mathtt{A}_j^2]\}]\big\}$$

Hence, given an output at $\mathsf{replace}_j$, process $a_j[!u_j.\, \widetilde{\mathsf{env}}_j\{\mathsf{env}_j[\bullet \parallel \mathtt{A}_j]\}] \parallel U_1$ evolves to $a_j[!u_j.\, \widetilde{\mathsf{env}}_j\{\mathsf{env}_j[\bullet \parallel \mathtt{A}_j^2]\}]$ thus discarding $\mathtt{A}_j$ in a *future* evolution of env_j. This kind

of *dynamic update* is available in commercial workflow engines, such as the Windows Workflow Foundation (WWF). Above, for simplicity, we have abstracted from lock mechanisms that keep consistency between concurrent updates on env_j and a_j.

In the WWF, dynamic update can also take place at the level of the workflow engine. This way, e.g., the engine may *suspend* those workflows which have been inactive for a certain amount of time. This optimizes resources at runtime, and favors active workflows. We can implement this policy as part of the process $\mathtt{WE}$ as follows:

$$U_2 = !\,\mathsf{suspend}_i.\,\widetilde{\mathsf{w}_i}\big\{!\,\mathsf{resume}_i.\,\mathsf{w}_i[\bullet]\big\}$$

This way, given an output signal at $\mathsf{suspend}_i$, process $\mathsf{w}_i[\mathtt{W}_i] \parallel U_3$ evolves to the persistent process $!\,\mathsf{resume}_i.\,\mathsf{w}_i[\mathtt{W}_i]$ which can be reactivated at a later time.

Scaling in Cloud Computing Applications. In the cloud computing paradigm, Web applications are deployed in the infrastructure offered by external providers. Developers only pay for the resources they consume (usually measured as the processor time in remote *instances*) and for additional services (e.g., services that provide performance metrics). Central to this paradigm is the goal of optimizing resources for both clients and provider. An essential feature towards that goal is *scaling*: the capability that cloud applications have for expanding themselves in times of high demand, and for reducing themselves when the demand is low. Scaling can be appreciated in, e.g., the number of running instances supporting the web application. Tools and services for *autoscaling* are provided by cloud providers such as Amazon's Elastic Cloud Computing (EC2) and by vendors who build on the public APIs cloud providers offer.

Here we draw inspiration from the autoscaling library provided by EC2. For scaling purposes, applications in EC2 are divided into *groups*, each defining different scaling policies for different parts of the application. This way, e.g., the part of the application deployed in Europe can have different scaling policies from the part deployed in the US. Each group is then composed of a number of identical instances implementing the web application, and of active processes implementing the scaling policies. This scenario can be abstracted in $\mathcal{E}$ as the process $App \stackrel{\mathsf{def}}{=} G_1 \parallel \cdots \parallel G_n$, with

$$G_i = g_i\big[\,I \parallel \cdots \parallel I \parallel S_{dw} \parallel S_{up} \parallel \mathtt{CTRL}_i\,\big]$$

where each group G_i contains a fixed number of running instances, each represented by $I = \mathtt{mid}[\mathsf{A}]$, a process that abstracts an instance as an adaptable process with unique identification $\mathtt{mid}$ and state A. Also, S_{dw} and S_{up} stand for the processes implementing scaling down and scaling up policies, respectively. Process $\mathtt{CTRL}_i$ abstracts the part of the system which controls scaling policies for group i. In practice, this control relies on external services (such as, e.g., services that monitor cloud usage and produce appropriate *alerts*). A simple way of abstracting scaling policies is the following:

$$S_{dw} = s_d\big[\,!\,\mathsf{alert}^{\mathsf{d}}.\prod^{j}\widetilde{\mathtt{mid}}\{\mathbf{0}\}\,\big] \qquad S_{up} = s_u\big[\,!\,\mathsf{alert}^{\mathsf{u}}.\prod^{k}\widetilde{\mathtt{mid}}\big\{\mathtt{mid}[\bullet] \parallel \mathtt{mid}[\bullet]\big\}\,\big]$$

Given proper alerts from $\mathtt{CTRL}_i$, the above processes modify the number of running instances. In fact, given an output at $\mathsf{alert}^{\mathsf{d}}$ process S_{dw} destroys j instances. This is achieved by leaving the inactive process as the new state of locality $\mathtt{mid}$. Similarly, an output at $\mathsf{alert}^{\mathsf{u}}$ process S_{up} spawns k update actions, each creating a new instance.

Autoscaling in EC2 also includes the possibility of *suspending* and *resuming* the scaling policies themselves. To represent this, we proceed as we did for process U_2 above. This way, for the scale down policy, one can assume that $\mathtt{CTRL}_i$ includes a process $U_{dw} = !\,\mathsf{susp}_{\mathsf{down}}.\,\widetilde{s_d}\{!\,\mathsf{resume}_{\mathsf{dw}}.\,s_d[\bullet]\}$ which, provided an output signal on $\mathsf{susp}_{\mathsf{down}}$, captures the current policy, and evolves into a process that allows to resume it at a later stage. This idea can be used to enforce other modifications to the policies (such as, e.g., changing the number of instances involved).

4 Bounded and Eventual Adaptation Are Undecidable in $\mathcal{E}$

We prove that $\mathcal{BA}$ and $\mathcal{EA}$ are undecidable in $\mathcal{E}$ by defining an encoding of Minsky machines (MMs) into $\mathcal{E}$ which satisfies the following: a MM terminates if and only if its encoding into $\mathcal{E}$ evolves into at least $k > 0$ processes that can perform a barb e.

A MM [17] is a Turing complete model composed of a set of sequential, labeled instructions, and two registers. Registers r_j $(j \in \{0,1\})$ can hold arbitrarily large natural numbers. Instructions $(1 : I_1), \ldots, (n : I_n)$ can be of three kinds: $\mathtt{INC}(r_j)$ adds 1 to register r_j and proceeds to the next instruction; $\mathtt{DECJ}(r_j, s)$ jumps to instruction s if r_j is zero, otherwise it decreases register r_j by 1 and proceeds to the next instruction; $\mathtt{HALT}$ stops the machine. A MM includes a program counter p indicating the label of the instruction to be executed. In its initial state, the MM has both registers set to 0 and the program counter p set to the first instruction.

The encoding, denoted $[\![\cdot]\!]_1$, is given in Table 1. A register j with value m is represented by an adaptable process at r_j that contains the encoding of number m, denoted $(\!|\ m\ |\!)_j$. In turn, $(\!|\ m\ |\!)_j$ consists of a sequence of m output prefixes on name u_j ending with an output action on z_j (the encoding of zero). Instructions are encoded as replicated processes guarded by p_i, which represents the MM when the program counter $p = i$. Once p_i is consumed, each instruction is ready to interact with the registers. To encode the increment of register r_j, we enlarge the sequence of output prefixes it contains. The adaptable process at r_j is updated with the encoding of the incremented value (which results from putting the value of the register behind some prefixes) and then the next instruction is invoked. The encoding of a decrement of register j consists of an exclusive choice: the left side implements the decrement of the value of a register, while the right one implements the jump to some given instruction. This choice is indeed exclusive: the encoding of numbers as a chain of output prefixes ensures that both an input prefix on u_j and one on z_j are never available at the same time. When the MM reaches the $\mathtt{HALT}$ instruction the encoding can either exhibit a barb on e, or set the program counter again to the $\mathtt{HALT}$ instruction so as to pass through a state that exhibits e at least $k > 0$ times. The encoding of a MM into $\mathcal{E}$ is defined as follows:

Definition 5. *Let N be a MM, with registers $r_0 = 0$, $r_1 = 0$ and instructions $(1 : I_1) \ldots (n : I_n)$. Given the encodings in Table 1, the encoding of N in $\mathcal{E}$ (written $[\![N]\!]_1$) is defined as $[\![r_0 = 0]\!]_1 \parallel [\![r_1 = 0]\!]_1 \parallel \prod_{i=1}^{n} [\![(i : I_i)]\!]_1 \parallel \overline{p_1}$.*

It can be shown that a MM N terminates iff its encoding has at least k consecutive barbs on the distinguished action e, for every $k \geq 1$, i.e. $[\![N]\!]_1 \Downarrow_e^k$. By considering the cluster $\mathcal{CS}^{\emptyset}_{[\![N]\!]_1} = \{[\![N]\!]_1\}$, we can conclude that $\mathcal{BA}$ is undecidable for $\mathcal{E}$ processes.

Table 1. Encoding of MMs into $\mathcal{E}$

REGISTER r_j $\quad [\![r_j = n]\!]_1 = r_j[(\!|\, n \,|\!)_j]$ where $(\!|\, n \,|\!)_j = \begin{cases} \overline{z_j} & \text{if } n = 0 \\ \overline{u_j}.\,(\!|\, n-1 \,|\!)_j & \text{if } n > 0. \end{cases}$

INSTRUCTIONS $(i : I_i)$

$[\![(i : \mathtt{INC}(r_j))]\!]_1 = !p_i.\,\widetilde{r_j}\{r_j[\overline{u_j}.\,\bullet]\}.\,\overline{p_{i+1}}$

$[\![(i : \mathtt{DECJ}(r_j, s))]\!]_1 = !p_i.\,(u_j.\,\overline{p_{i+1}} + z_j.\,\widetilde{r_j}\{r_j[\overline{z_j}]\}.\,\overline{p_s})$

$[\![(i : \mathtt{HALT})]\!]_1 = !p_i.\,(e + \overline{p_i})$

Moreover, since the number of consecutive barbs on e can be unbounded (i.e., there exists a computation where $[\![N]\!]_1 \Downarrow_e^\omega$), we can also conclude that $\mathcal{EA}$ is undecidable.

Theorem 1. *$\mathcal{BA}$ and $\mathcal{EA}$ are undecidable in $\mathcal{E}$.*

5 The Subcalculus $\mathcal{E}^-$ and Decidability of Bounded Adaptation

Theorem 1 raises the question as whether there are fragments of $\mathcal{E}$ in which the problems are decidable. A natural way of restricting the language is by imposing limitations on *update patterns*, the behavior of running processes as a result of update actions. We now consider $\mathcal{E}^-$, the fragment of $\mathcal{E}$ in which update prefixes are restricted in such a way that the hole $\bullet$ cannot occur in the scope of prefixes. More precisely, $\mathcal{E}^-$ processes are those $\mathcal{E}$ processes in which the context U in $\widetilde{a}\{U\}$ respects the following grammar:

$$U ::= P \;\mid\; a[U] \;\mid\; U \parallel U \;\mid\; \bullet$$

In [5], we have shown that there exists an algorithm to determine whether there exists $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^k$ holds. We therefore have the following result.

Theorem 2. *$\mathcal{BA}$ is decidable in $\mathcal{E}^-$.*

We now provide intuitions on the proof of Theorem 2; see [5] for details. The algorithm checks whether one of such R appears in the (possibly infinite) set of processes from which it is possible to reach a process that can perform α at least k consecutive times. The idea is to introduce a *preorder* (or, equivalently, quasi-order) $\preceq$ on processes so as to characterize such a set by means of a so-called *finite basis*: finitely many processes that generate the set by upward closure, i.e., by taking all processes which are greater or equal to some process (wrt $\preceq$) in the finite basis.

The proof appeals to the theory of well-structured transition systems [1,13]. We define the preorder $\preceq$ by resorting to a tree representation, in such a way that: (i) $\preceq$ is a *well-quasi-order*: given any infinite sequence $x_i, i \in \mathbb{N}$, of elements there exist $j < k$ such that $x_j \preceq x_k$; (ii) $\preceq$ is *strongly compatible* wrt reduction in $\mathcal{E}$: for all $x_1 \preceq y_1$ and all reductions $x_1 \longrightarrow x_2$, there exists y_2 such that $y_1 \longrightarrow y_2$ and $x_2 \preceq y_2$; (iii) $\preceq$ is *decidable*; and (iv) $\preceq$ has an *effective pred-basis*, i.e., for any x it is possible to compute a basis of the set of states y such that there exists $y' \longrightarrow x'$ with $y' \preceq y$ and $x \preceq x'$. It is known [1,13] that given any target set I which is characterizable by a finite basis, an algorithm exists to compute a finite basis FB for the set of processes from which it is possible to reach a process belonging to I.

The algorithm to determine if there exists $R \in \mathcal{CS}_P^M$ such that $R \Downarrow_\alpha^k$ consists of three steps: (1) We restrict the set of terms that we consider to those reachable by any $R \in \mathcal{CS}_P^M$. We characterize this set by (a) considering the *sequential subterms* in $\mathcal{CS}_P^M$, i.e., terms not having parallel or locations as their topmost operator, included in the term P or in the terms in M and (b) introducing $\preceq$ over terms with the above properties. (2) We then show that it is possible to compute I (i.e. the finite basis of the set mentioned before) such that it includes all processes that expose α at least k consecutive times. (3) Finally, we show that it is possible to determine whether or not some $R \in \mathcal{CS}_P^M$ is included in the set generated by the finite basis FB.

Intuitions on these three steps follow. As for (1), we exploit Kruskal's theorem on well-quasi-orderings on trees [15]. Unlike other works appealing to well-structured transition systems for obtaining decidability results (e.g. [9]), in the case of $\mathcal{E}^-$ it is not possible to find a bound on the "depth" of processes. Consider, for instance, process $R = a[P] \parallel !\widetilde{a}\{a[a[\bullet]]\}$. One possible evolution of R is when it is always the innermost adaptable process which is updated; as a result, one obtains a process with an unbounded number of nested adaptable processes: $a[a[\ldots a[P]]]$. Nevertheless, some regularity can be found also in the case of $\mathcal{E}^-$, by mapping processes into trees labeled over location names and sequential subterms in $\mathcal{CS}_P^M$. The tree of a process P is built as follows. We set a root node labeled with the special symbol ϵ, and which has as many children nodes as parallel subterms in P. For those subterms different from an adaptable process, we obtain leaf nodes labeled with the subterms. For each subterm $a[P']$, we obtain a node which is labeled with a and has as many children nodes as parallel subterms in P'; tree construction then proceeds recursively on each of these parallel subterms. By mapping processes into this kind of trees, and by using Kruskal's ordering over them, it can be shown that our preorder $\preceq$ is a well-quasi-ordering with strong compatibility, and has an effective pred-basis. The pred-basis of a process P is computed by taking all the *minimal* terms that reduce to P in a single reduction. These terms are obtained by extending the tree of P with at most two additional subtrees, which represent the subprocess(es) involved in a reduction.

As for (2), we proceed backwards. We first determine the finite basis FB' of the set of processes that can *immediately* perform α. This corresponds to the set of sequential subterms of $\mathcal{CS}_P^M$ that can immediately perform α. If $k > 1$ (otherwise we are done) we do a backward step by applying the effective pred-basis procedure described above to each term of FB', with a simple modification: if an element Q of the obtained basis cannot immediately perform α, we replace it with all the terms $Q \parallel Q'$, where Q' is a sequential subterm of $\mathcal{CS}_P^M$ that can immediately perform α. To ensure minimality of the finite basis FB', we remove from it all R' such that $R \preceq R'$, for some R already in FB'. We repeat this procedure $k-1$ times to obtain FB—the finite basis of I.

As for (3), we verify if there exists a $Q \in FB$ for which the following check succeeds. Let $Par(Q)$ be the multiset of terms Q_i such that Q is obtained as the parallel composition of such Q_i (notice that it could occur that $Par(Q) = \{Q\}$). We first remove from $Par(Q)$ all those Q_i such that there exists $T \in M$ with $Q_i \preceq T$, thus obtaining the multiset $Par'(Q)$. Let S be the parallel composition of the processes in $Par'(Q)$. Then, we just have to check whether $S \preceq P$ or not.

Table 2. Encoding of MMs into $\mathcal{E}^-$ - Static case

$$\textsc{Control} = !a.\,(\overline{f} \parallel \overline{b} \parallel \overline{a}) \parallel \overline{a}.\,a.\,(\overline{p_1} \parallel e) \parallel\ !h.\,(g.\,\overline{f} \parallel \overline{h})$$

$\textsc{Register}\ r_j$

$$[\![r_j = m]\!]_2 = \begin{cases} r_j[!inc_j.\,\overline{u_j} \parallel \overline{z_j}] & \text{if } m = 0 \\ r_j[!inc_j.\,\overline{u_j} \parallel \prod_1^m \overline{u_j} \parallel \overline{z_j}] & \text{if } m > 0. \end{cases}$$

$\textsc{Instructions}\ (i : I_i)$

$$\begin{aligned} [\![(i : \mathtt{INC}(r_j))]\!]_2 &= !p_i.\,f.\,(\overline{g} \parallel b.\,\overline{inc_j}.\,\overline{p_{i+1}}) \\ [\![(i : \mathtt{DECJ}(r_j, s))]\!]_2 &= !p_i.\,f.\,\big(\overline{g} \parallel (u_j.\,(\overline{b} \parallel \overline{p_{i+1}}) + z_j.\,\widetilde{r_j}\{r_j[!inc_j.\,\overline{u_j} \parallel \overline{z_j}]\}.\,\overline{p_s})\big) \\ [\![(i : \mathtt{HALT})]\!]_2 &= !p_i.\,\overline{h}.\,h.\,\widetilde{r_0}\{r_0[!inc_0.\,\overline{u_0} \parallel \overline{z_0}]\}.\,\widetilde{r_1}\{r_1[!inc_1.\,\overline{u_1} \parallel \overline{z_1}]\}.\,\overline{p_1} \end{aligned}$$

6 Eventual Adaptation Is Undecidable in $\mathcal{E}^-$

We show that $\mathcal{EA}$ is undecidable in $\mathcal{E}^-$ by relating it to termination in MMs. In contrast to the encoding given in Section 4, the encodings presented here are *non faithful*: when mimicking a test for zero, the encoding may perform a jump even if the tested register is not zero. Nevertheless, we are able to define encodings that repeatedly simulate finite computations of the MM, and if the repeated simulation is infinite, then we have the guarantee that the number of erroneous steps is finite. This way, the MM terminates iff its encoding has a non terminating computation. As during its execution the encoding continuously exhibits a barb on e, it then follows that $\mathcal{EA}$ is undecidable in $\mathcal{E}^-$.

We show that $\mathcal{EA}$ is already undecidable in two fragments of $\mathcal{E}^-$. While in the *static* fragment we assume that the topology of nested adaptable processes is fixed and cannot change during the computation, in the *dynamic* fragment we assume that such a topology can change, but that processes cannot be neither removed nor replicated.

Undecidability in the Static Case. The encoding relies on finitely many output prefixes acting as *resources* on which instructions of the MM depend in order to be executed. To repeatedly simulate finite runs of the MM, at the beginning of the simulation the encoding produces finitely many instances of these resources. When HALT is reached, the registers are reset, some of the consumed resources are restored, and a new simulation is restarted from the first instruction. In order to guarantee that an infinite computation of the encoding contains only finitely many erroneous jumps, finitely many instances of a second kind of resource (different from that required to execute instructions) are produced. Such a resource is consumed by increment instructions and restored by decrement instructions. When the simulation performs a jump, the tested register is reset: if it was not empty (i.e., an erroneous test) then some resources are permanently lost. When the encoding runs out of resources, the simulation will eventually block as increment instructions can no longer be simulated. We make two non restrictive assumptions. First, we assume that a MM computation contains at least one increment instruction. Second, in order to avoid resource loss at the end of a correct simulation run, we assume that MM computations terminate with both the registers empty.

We now discuss the encoding defined in Table 2. We first comment on $\textsc{Control}$, the process that manages the resources. It is composed of three processes in parallel. The first replicated process produces an unbounded amount of processes $\overline{f}$ and $\overline{b}$, which represent the two kinds of resources described above. The second process starts and

stops a resource production phase by performing $\overline{a}$ and a, respectively. Then, it starts the MM simulation by emitting the program counter $\overline{p_1}$. The third process is used at the end of the simulation to restore some of the consumed resources $\overline{f}$ (see below).

A register r_j that stores number m is encoded as an adaptable process at r_j containing m copies of the unit process $\overline{u_j}$. It also contains process $!inc_j.\overline{u_j}$ which allows to create further copies of $\overline{u_j}$ when an increment instruction is executed. Instructions are encoded as replicated processes guarded by p_i. Once p_i is consumed, increment and decrement instructions consume one of the resources $\overline{f}$. If such a resource is available then it is renamed as $\overline{g}$, otherwise the simulation blocks. The simulation of an increment instruction also consumes an instance of resource $\overline{b}$.

The encoding of a decrement-and-jump instruction is slightly more involved. It is implemented as a choice: the process can either perform a decrement and proceed with the next instruction, or to jump. In case the decrement can be executed (the input u_j is performed) then a resource $\overline{b}$ is restored. The jump branch can be taken even if the register is not empty. In this case, the register is reset via an update that restores the initial state of the adaptable process at r_j. Note that if the register was not empty, then some processes $\overline{u_j}$ are lost. Crucially, this causes a permanent loss of a corresponding amount of resources $\overline{b}$, as these are only restored when process $\overline{u_j}$ are present.

The simulation of the `HALT` instruction performs two tasks before restarting the execution of the encoding by reproducing the program counter p_1. The first one is to restore some of the consumed resources $\overline{f}$: this is achieved by the third process of CONTROL, which repeatedly consumes one instance of $\overline{g}$ and produces one instance of $\overline{f}$. This process is started/stopped by executing the two prefixes $\overline{h}.\, h$. The second task is to reset the registers by updating the adaptable processes at r_j with their initial state.

The full definition of the encoding is as follows.

Definition 6. *Let N be a MM, with registers r_0, r_1 and instructions $(1 : I_1) \dots (n : I_n)$. Given the* CONTROL *process and the encodings in Table 2, the encoding of N in $\mathcal{E}^-$ (written $[\![N]\!]_2$) is defined as $[\![r_0 = 0]\!]_2 \parallel [\![r_1 = 0]\!]_2 \parallel \prod_{i=1}^{n} [\![(i : I_i)]\!]_2 \parallel$* CONTROL.

The encoding has an infinite sequence of simulation runs if and only if the corresponding MM terminates. As the barb e is continuously exposed during the computation (the process e is spawn with the initial program counter and is never consumed), we can conclude that a MM terminates if and only if its encoding does not eventually terminate.

Lemma 1. *Let N be a MM. N terminates iff $[\![N]\!]_2 \Downarrow_e^\omega$.*

Exploiting Lemma 1, we can state the following:

Theorem 3. *$\mathcal{EA}$ is undecidable in $\mathcal{E}^-$.*

Note that the encoding $[\![\cdot]\!]_2$ uses processes that do not modify the topology of nested adaptable processes; update prefixes do not remove nor create adaptable processes: they simply remove the processes currently in the updated locations and replace them with the predefined initial content. One could wonder whether the ability to remove processes is necessary for the undecidability result: next we show that this is not the case.

Undecidability in the Dynamic Case. We now show that $\mathcal{EA}$ is still undecidable in $\mathcal{E}^-$ even if we consider updates that do not remove running processes. The proof relies

Table 3. Encoding of MMs into $\mathcal{E}^-$ - Dynamic case

REGISTER r_j

$[\![r_j = 0]\!]_3 = r_j[Reg_j \parallel c_j[\mathbf{0}]]$ with $Reg_j = !inc_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}.\, u_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}$

INSTRUCTIONS $(i : I_i)$

$[\![(i : \mathtt{INC}(r_j))]\!]_3 \quad = !p_i.\, f.\, (\overline{g} \parallel b.\, \overline{inc_j}.\, ack.\, \overline{p_{i+1}})$

$[\![(i : \mathtt{DECJ}(r_j, s))]\!]_3 = !p_i.\, f.\, \big(\overline{g} \parallel (\overline{u_j}.\, ack.\, (\overline{b} \parallel \overline{p_{i+1}}) + \widetilde{c_j}\{\bullet\}.\, \widetilde{r_j}\{r_j[Reg_j \parallel c_j[\bullet]]\}.\, \overline{p_s})\big)$

$[\![(i : \mathtt{HALT})]\!]_3 \quad = !p_i.\, \overline{h}.\, h.\, \widetilde{c_0}\{\bullet\}.\, \widetilde{r_0}\{r_0[Reg_0 \parallel c_0[\bullet]]\}.\, \widetilde{c_1}\{\bullet\}.\, \widetilde{r_1}\{r_1[Reg_1 \parallel c_1[\bullet]]\}.\, \overline{p_1}$

on a nondeterministic encoding of MMs, similar to the one presented before. In that encoding, process deletion was used to restore the initial state inside the adaptable processes representing the registers. In the absence of process deletion, we use a more involved technique based on the possibility of moving processes to a different context: processes to be removed are guarded by an update prefix $\widetilde{c_j}\{c_j[\bullet]\}$ that simply tests for the presence of a parallel adaptable process at c_j; when a process must be deleted, it is "collected" inside c_j, thus disallowing the possibility to execute such an update prefix.

The encoding is as in Definition 6, with registers and instructions encoded as in Table 3. A register r_j that stores number m is encoded as an adaptable process at r_j that contains m copies of the unit process $u_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}$. It also contains process Reg_j, which creates further copies of the unit process when an increment instruction is invoked, as well as the collector c_j, which is used to store the processes to be removed.

An increment instruction adds an occurrence of $u_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}$. Note that an output $\overline{inc}$ could synchronize with the corresponding input inside a collected process. This immediately leads to deadlock as the containment induced by c_j prevents further interactions. The encoding of a decrement-and-jump instruction is implemented as a choice, following the idea discussed for the static case. If the process guesses that the register is zero then, before jumping to the given instruction, it proceeds at disabling its current content: this is done by (i) removing the boundary of the collector c_j leaving its content at the top-level, and (ii) updating the register placing its previous state in the collector. A decrement simply consumes one occurrence of $u_j.\, \widetilde{c_j}\{c_j[\bullet]\}.\, \overline{ack}$. Note that as before the output $\overline{u_j}$ could synchronize with the corresponding input inside a collected process. Again, this immediately leads to deadlock. The encoding of HALT exploits the same mechanism of collecting processes to simulate the reset of the registers.

This encoding has the same properties of the one discussed for the static case. In fact, in an infinite simulation the collected processes are never involved, otherwise the computation would block. We can conclude that process deletion is not necessary for the undecidability of $\mathcal{EA}$ in $\mathcal{E}^-$. Nevertheless, in the encoding in Table 3 we need to use the possibility to remove and create adaptable processes (namely, the collectors c_j are removed and then reproduced when the registers must be reset). One could therefore wonder whether $\mathcal{EA}$ is still undecidable if we remove from $\mathcal{E}^-$ *both* the possibility to remove processes and to create/destroy adaptable processes. In the extended version of this paper [5] we have defined a fragment of $\mathcal{E}^-$ obtained by (i) disallowing creation and destruction of adaptable processes and (ii) eliminating the possibility of removing or relocating a running process to a different adaptable process. By resorting to the theory of Petri nets, we have proved that $\mathcal{EA}$ for processes in this fragment is *decidable*.

7 Concluding Remarks

We have proposed the concept of *adaptable process* as a way of describing concurrent systems that exhibit complex evolvability patterns at runtime. We have introduced $\mathcal{E}$, a calculus of adaptable processes in which processes can be updated/relocated at runtime. We also proposed the *bounded* and *eventual* adaptation problems, and provided a complete study of their (un)decidability for $\mathcal{E}$ processes. Our results shed light on the expressive power of $\mathcal{E}$ and on the verification of concurrent processes that may evolve at runtime. As for future work, it would be interesting to develop variants of $\mathcal{E}$ tailored to concrete application settings, to determine how the proposed adaptation problems fit in such scenarios, and to study how to transfer our decidability results to such variants.

Related Work. As argued before, the combination of techniques required to prove decidability of $\mathcal{BA}$ in $\mathcal{E}^-$ is non trivial. In particular, the technique is more complex than that in [2], which relies on a bound on the depth of trees, or that in [23], where only topologies with bounded paths are taken into account. Kruskal's theorem is also used in [7] for studying the decidability properties of calculi with exceptions and compensations. The calculi considered in [7] are *first-order*; in contrast, $\mathcal{E}$ is a *higher-order* calculus (see below). We showed the undecidability of $\mathcal{EA}$ in $\mathcal{E}^-$ by means of an encoding of MMs that does not reproduce faithfully the corresponding machine. Similar techniques have been used to prove the undecidability of repeated coverability in reset Petri nets [11], but in our case their application revealed much more complex. Notice that since in a cluster there is no a-priori knowledge on the number of modifications that will be applied to the system, the analysis needs to be *parametric*. Parametric verification has been studied, e.g., in the context of broadcast protocols in fully connected [12] and ad-hoc networks [10]. Differently from [12,10], in which the number of nodes (or the topology) of the network is unknown, we consider systems in which there is a known part (the initial system P), and there is another part composed of an unknown number of instances of processes (taken from the set of possible modifications M).

$\mathcal{E}$ is related to *higher-order* process calculi such as, e.g., the higher-order π-calculus [21], Kell [22], and Homer [8]. In such calculi, processes can be passed around, and so communication involves term instantiation, as in the λ-calculus. Update actions in $\mathcal{E}$ are a form of term instantiation: as we elaborate in [6], they can be seen as a streamlined version of the *passivation* operator of Kell and Homer, which allows to suspend a running process. The encoding given in Section 4 is inspired in the encoding of MMs into a core higher-order calculus with passivation presented in [19, Ch 5]. In [19], however, no adaptation concerns are studied. Also related to $\mathcal{E}$ are process calculi for distributed algorithms/systems (e.g., [4,20,18,14]) which feature located processes and notions of failure. In [4], a higher-order operation that defines *savepoints* is proposed: process $\mathsf{save}\langle P\rangle.\, Q$ defines the savepoint P for the current location; if the location crashes, then it will be restarted with state P. The calculus in [20] includes constructs for killing a located process, spawning a new located process, and checking the status of a location. In [18,14], the language includes a *failure detector* construct $\mathcal{S}(k).\, P$ which executes P if location k is *suspected* to have failed. Crucially, while in the languages in [4,20,18,14] the *post-failure* behavior is defined statically, in $\mathcal{E}$ it can be defined

dynamically, exploiting running processes. Moreover, differently from our work, neither of [4,20,18,14] addresses expressiveness/decidability issues, as we do here.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: Algorithmic analysis of programs with well quasi-ordered domains. Inf. Comput. 160(1-2), 109–127 (2000)
2. Acciai, L., Boreale, M.: Deciding safety properties in infinite-state pi-calculus via behavioural types. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 31–42. Springer, Heidelberg (2009)
3. Baeten, J., Bergstra, J.: Mode transfer in process algebra. Technical Report Report 00/01, Eindhoven University of Technology (2000)
4. Berger, M., Honda, K.: The two-phase commitment protocol in an extended pi-calculus. Electr. Notes Theor. Comput. Sci. 39(1) (2000)
5. Bravetti, M., Di Giusto, C., Pérez, J.A., Zavattaro, G.: Adaptable processes. Technical report, Univ. of Bologna, (2011), Draft, `http://www.cs.unibo.it/~perez/ap/`
6. Bravetti, M., Di Giusto, C., Pérez, J.A., Zavattaro, G.: Steps on the Road to Component Evolvability. Post-proceedings of FACS 2010. LNCS. Springer, Heidelberg (to appear, 2011)
7. Bravetti, M., Zavattaro, G.: On the expressive power of process interruption and compensation. Math. Struct. in Comp. Sci. 19(3), 565–599 (2009)
8. Bundgaard, M., Godskesen, J.C., Hildebrandt, T.: Bisimulation congruences for homer — a calculus of higher order mobile embedded resources. Technical Report TR-2004-52, IT University of Copenhagen (2004)
9. Busi, N., Gabbrielli, M., Zavattaro, G.: On the expressive power of recursion, replication and iteration in process calculi. Math. Struct. in Comp. Sci. 19(6), 1191–1222 (2009)
10. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 313–327. Springer, Heidelberg (2010)
11. Esparza, J.: Some applications of petri nets to the analysis of parameterised systems. Talk at WISP 2003 (2003)
12. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: Proc. of LICS, pp. 352–359 (1999)
13. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
14. Francalanza, A., Hennessy, M.: A fault tolerance bisimulation proof for consensus (Extended abstract). In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 395–410. Springer, Heidelberg (2007)
15. Kruskal, J.B.: Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. Transactions of the American Mathematical Society 95(2), 210–225 (1960)
16. Milner, R.: Comunication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
17. Minsky, M.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs (1967)
18. Nestmann, U., Fuzzati, R., Merro, M.: Modeling consensus in a process calculus. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 399–414. Springer, Heidelberg (2003)

19. Pérez, J.A.: Higher-Order Concurrency: Expressiveness and Decidability Results. PhD thesis, University of Bologna (2010), Draft, `http://www.japerez.phipages.com`
20. Riely, J., Hennessy, M.: Distributed processes and location failures. Theor. Comput. Sci. 266(1-2), 693–735 (2001); An extended abstract appeared in Proc. of ICALP 1997
21. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis CST–99–93, University of Edinburgh, Dept. of Comp. Sci. (1992)
22. Schmitt, A., Stefani, J.-B.: The kell calculus: A family of higher-order distributed process calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
23. Wies, T., Zufferey, D., Henzinger, T.A.: Forward analysis of depth-bounded processes. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)

A Framework for Verifying Data-Centric Protocols

Yuxin Deng[1,2,⋆], Stéphane Grumbach[3,⋆⋆], and Jean-François Monin[4]

[1] Department of Computer Science, Shanghai Jiao Tong University
[2] Laboratory of Computer Science, Chinese Academy of Sciences
[3] INRIA - LIAMA
[4] Université de Grenoble 1 - CNRS - LIAMA

Abstract. Data centric languages, such as recursive rule based languages, have been proposed to program distributed applications over networks. They simplify greatly the code, while still admitting efficient distributed execution. We show that they also provide a promising approach to the verification of distributed protocols, thanks to their data centric orientation, which allows us to explicitly handle global structures such as the topology of the network. We consider a framework using an original formalization in the Coq proof assistant of a distributed computation model based on message passing with either synchronous or asynchronous behavior. The declarative rules of the Netlog language for specifying distributed protocols and the virtual machines for evaluating these rules are encoded in Coq as well. We consider as a case study tree protocols, and show how this framework enables us to formally verify them in both the asynchronous and synchronous setting.

1 Introduction

Up to now, most efforts to formalize protocols or distributed algorithms and automate their verification relied on control-oriented paradigms. It is the case for instance of the "Formal Description Techniques" developed by telecom labs at the beginning of the 1980s in order to specify and verify protocols to be standardized at ITU and ISO. Two of the languages developed, Estelle and SDL, are based on asynchronous communicating automata, while LOTOS is a process algebra based on CCS and CSP extended with algebraic data types [36]. Several verification tools, ranging from simulation to model checking, have been developed and applied to different case studies [39,19,35,40,32,34,18,10,13].

For the verification of communication protocols based on process algebras, the idea has been to model both the implementation and the specification of a protocol as processes in a process algebra, and then to use automatic tools to

⋆ Supported by the Natural Science Foundation of China under grant No. 61033002 and the Opening Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University.

⋆⋆ Supported by the French Agence Nationale de la Recherche, under grant ANR-09-BLAN-0131-01.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 106–120, 2011.
© IFIP International Federation for Information Processing 2011

check whether the former is a refinement of the latter or if they are behaviorally equivalent [6,33]. Examples include the Concurrency Workbench [6], which is a verification tool based on CCS, FDR [33] which is based on CSP [17], ProVerif [2] which is based on the applied pi calculus [1]. Other approaches include input/output automata [27], or Unity and TLA, which combine temporal logic and transition-based specification [4,20], and may rely as well on proof assistant technology [31,16,5,21].

The common feature of all these approaches is their focus on control, in particular on how to deal with behaviors in a distributed framework. Typical issues include non-determinism, deadlock freedom, stuttering, fairness, distributed consensus and, more recently, mobility. Data is generally considered as an abstract object not really related to the behavior. If this is relevant for many low-level protocols, such as transport protocols, it does not suit the needs of applications which aim at building up distributed global information, such as topological information on the network (in a physical or a virtual sense), e.g. routing tables. Such protocols are qualified as *data-centric* in the sequel. Correctness proofs of data-centric protocols are even more complex than those for *control-centric* protocols.

Data-centric rule-based languages have been recently proposed to program network protocols and distributed applications [25,24,23]. This approach benefits from reasonably efficient implementations, by using methods developed in the field of databases for recursive languages à la Datalog. Programs are expressed at a very high level, typically two orders of magnitude shorter than code written in usual imperative languages.

Our claim is that this framework is promising not only for the design of distributed algorithms, but for their verification as well. Up to now, only a few partial results have been demonstrated. In [37], a declarative network verifier (DNV) was presented. Specifications written in the Network Datalog query language are mapped into logical axioms, which can be used in theorem provers like PVS to validate protocol correctness. The reasoning based on DNV is for Datalog specifications of (eventually distributed) algorithms, but not for distributed versions of Datalog such as the one proposed in this paper. In other words, only the highly abstract centralized behaviour of a network is considered. Therefore, deep subtleties on message passing, derivation of local facts and their relationship with the intended global behaviour are absent in [37].

In the present paper, we go one essential step further. We show that it is indeed feasible to reason about the distributed behaviour of individual nodes which together yield some expected global behaviour of the whole network. We consider the Netlog language [14], which relies on deductive rules of the form $head \leftarrow body$, which are installed on each node of the distributed system. The rules allow to derive new facts of the form "*head*", if their body is satisfied locally on the node. The facts derived might then be stored locally on the node or sent to other nodes in the network depending upon the rule.

Netlog admits a fixpoint semantics which interleaves local computation on the nodes and communication between neighboring nodes. On each node, a local

round consists of a computation phase followed by a communication phase. During the computation phase, the program updates the local data and produces messages to be sent. During the communication phase, messages are transmitted and become available to the destination node.

Our objective is to develop a framework to formally verify properties of Netlog programs. As to formal verification, there are roughly two kinds of approaches: *model checking* and *theorem proving*. Model checking explores the state space of a system model exhaustively to see if a desirable property is satisfied. It is largely automated and generates a counterexample if the property does not hold. The state explosion problem limits the potential of model checkers for large systems. The basic idea of theorem proving is to translate a system's specification into a mathematical theory and then construct a proof of a theorem by generating the intermediate proof steps. Theorem proving can deal with large or even infinite state spaces by using proof principles such as induction and co-induction.

We use the *proof assistant*, Coq, which is an interactive theorem prover, in which high level proof search commands construct formal proofs behind the scene, which are then mechanically verified. Coq has been successfully applied to ensure reliability of hardware and software systems in various fields, such as multiplier circuits [30], concurrent communication protocols [12], self-stabilizing population protocols [9], devices for broadband protocols [28], local computation systems [3] and compilers [22], to name a few.

We develop a Coq library necessary for our purposes, including (i) the formalization of the distributed system; (ii) the modeling of the embedded machine evaluating the Netlog programs; (iii) the translation of the Netlog programs; as well as (iv) a formalization of graphs and trees suitable to our needs.

As a proof of concept, we experimented the proposed framework on concrete protocols for constructing spanning trees over connected graphs. Such protocols have been shown to be correct on theoretical models. This is the case for instance of the well-known distributed algorithm for computing a minimum-weight spanning tree due to Gallager, Humblet and Spira [11]. The rigorous proofs made between 1987 and 2006 [38,15,29] are all intricate and very long (100 to 170 pages). Only [15] has been mechanically proof-checked.

Our objective is to carry on proofs for protocols written in a programming language (Netlog) which is implemented and runs on real distributed systems, and not only in theoretical models, and to concentrate on a data-centric approach. The protocols proceed in rounds, where one node (in the asynchronous model) or all nodes (in the synchronous model) perform some local computation, update their local data and then exchange data with their neighbors before entering the next round.

We have proved an initial simple protocol for defining spanning trees. Furthermore, in the synchronous message passing model, we show that we obtain a distributed version of the classical breadth-first search (BFS) algorithm. To show its correctness, the crucial ingredient is to formally prove the validity of the invariant that states the relationship between the centralized and the distributed version of the protocol, as well as the propagation of information on the

distributed version. This is non trivial and requires modeling how distributed tasks cooperate together to form the invariant. We claim that the proposed techniques establish foundations for proving more complex tree protocols such as GHS [11].

The paper is organized as follows. In Section 2, the distributed computation as formalized in Coq is presented. Section 3 is devoted to the presentation of the Netlog language. Section 4 contains the sketches of the proofs of the correctness of the tree protocol (a complete Coq script is available in [7]).

2 Distributed Computation Model of Netlog

In this section we introduce a distributed computation model based on the message passing mechanism, which is suitable both for synchronous and asynchronous execution. Corresponding formal definitions in Coq can be found in [7,8]. The distributed computation model described below does not depend on Netlog. In our formalization, we just assume that the states at nodes have a type *local_data* which can evolve using simple set-theoretic operations such as union.

A *distributed system* relies on a communication network whose topology is given by a *directed connected graph* $\mathcal{G} = (V_\mathcal{G}, G)$, where $V_\mathcal{G}$ is the set of nodes, and G denotes the set of *communication links* between nodes. For many applications, we can also assume that the graph is symmetric, that is $G(\alpha, \beta) \Leftrightarrow G(\beta, \alpha)$.

Each node has a unique *identifier*, *Id*, taken from $1, 2, \cdots, n$, where n is the number of nodes, and distinct local ports for distinct links incident to it. The control is fully distributed in the network, and there is no shared memory. In this high-level computation model, we abstract away detailed factors like node failures and lossy channels; if we were to formalize a more precise model, most of the data structures defined below would have to be refined.

All the nodes have the same architecture and the same behavior. Each node consists of three main components: (i) a router, handling the communication with the network; (ii) an engine, executing the local programs; and (iii) a local data store to maintain the information (data and programs) local to the node. It contains in particular the fragment of G, which relates a node to its neighbors. The router queues the incoming messages on the reception queue and the message to push produced by the engine on the emission queue.

We distinguish between *computation events*, performed in a node, and *communication events*, performed by nodes which cast their messages to their neighbors. On one node, a *computation phase* followed by a *communication phase* is called a *local round* of the distributed computation.

An *execution* is a sequence of alternating global configurations and rounds occurring on one node, in the case of an asynchronous system, or a sequence of alternating global configurations and rounds occurring simultaneously on each node, in the case of a synchronous system. In the latter case, the computation phase runs in parallel on all nodes, immediately followed by a parallel execution on all nodes of the corresponding communication phase.

The contents of node *loc* (the database of facts stored at *loc*) for a configuration *cnf* is denoted by $|loc|^{cnf}$, or just $|loc|$ when the configuration is clear from the context. Similarly, the set of messages arriving a node y from node x is denoted by $|x \rightarrow y|^{cnf}$ or $|x \rightarrow y|$.

A local round at node *loc* relates an actual configuration *pre* to a new configuration *mid* and a list *out* of messages emitted from *loc*. Furthermore, incoming edges are cleared – intuitively, this represents the consumption of messages, once their contents has been used to elaborate *mid* and *out*. The new data d to be stored on *loc* is defined by a relation *new_stores* given as a parameter, and we assume that d depends only on the data available at *loc* in *pre*, that is, $|loc|^{pre}$ and all the $|x \rightarrow loc|^{pre}$ such that there is an edge from x to *loc*. Intuitively, the relation *new_stores* expresses that d consists of new facts derived from facts available at *loc*. Similarly, *out* is defined by a relation *new_push* and satisfies similar requirements. Formally, a local round is defined by the following conjunction.

$$local_round(loc, pre, mid, out) \stackrel{\text{def}}{=} \begin{cases} \exists d, new_stores(pre, loc, d) \ \wedge \ |loc|^{mid} = |loc|^{pre} \cup d \\ new_push(pre, loc, out) \\ \forall \ x \in neighbors(loc), |x \rightarrow loc|^{mid} = \emptyset \end{cases}$$

For modeling asynchronous behaviors, we also need the notion of a trivial local round at *loc*, where the local data does not change and moreover incoming edges are not cleared either.

$$no_change_at(loc, pre, mid) \stackrel{\text{def}}{=} \begin{cases} |loc|^{mid} = |loc|^{pre} \\ \forall \ x \in neighbors(loc), |x \rightarrow loc|^{mid} = |x \rightarrow loc|^{pre} \end{cases}$$

A communication event at node *loc* specifies that the local data at *loc* does not change and that facts from *out* are appended on edges according to their destinations.

$$communication(loc, mid, post, out) \stackrel{\text{def}}{=} \begin{cases} |loc|^{post} = |loc|^{mid} \\ \forall \ y \in neighbors(loc), |loc \rightarrow y|^{post} = find(y, out) \ \cup \ |loc \rightarrow y|^{mid} \end{cases}$$

The function *find* returns the fact in *out* whose destination is y. Note that none of the previous three definitions specifies completely the next configuration in function of the previous one. They rather constrain a relation between two consecutive configurations by specifying what should happen at a given location. Combining these definitions in various ways allows us to define a complete transition relation between two configurations, with either a synchronous or an asynchronous behavior.

$$async_round(pre, post) \stackrel{\text{def}}{=} \exists \ loc \ mid \ out \begin{cases} local_round(loc, pre, mid, out) \\ \forall \ loc', loc \neq loc' \Rightarrow no_change_at(loc', pre, mid) \\ communication(loc, mid, post, out) \\ \forall \ loc', loc \neq loc' \Rightarrow communication(loc', mid, post, \emptyset) \end{cases}$$

An asynchronous round between two configurations *pre* and *post* is given by a node Id *loc*, an intermediate configuration *mid* and a list of messages *out* such that there is a local round relating *pre*, *mid* and *out* on *loc* while no change occurs on loc' different from *loc*, and a communication relates *mid* and *out* to *post* on *loc* while nothing is communicated on loc' different from *loc*.

$$sync_round(pre, post) \stackrel{\text{def}}{=} \exists\, mid,\ \forall\, loc,\ \exists\, out \begin{cases} local_round(loc, pre, mid, out) \\ communication(loc, mid, post, out) \end{cases}$$

A synchronous round between two configurations *pre* and *post* is given by an intermediate configuration *mid* such that for all node Id *loc*, there exists a list of messages *out* such that there is a local round relating *pre*, *mid* and *out* on *loc* and a communication relating *mid* and *out* to *post* on *loc*.

Now, given an arbitrary *trans* relation, which can be of the form *sync_round*, or *async_round*, or even of some alternative form, we can co-inductively define a run starting from a configuration. We have two cases: either there is a transition from configuration *pre* to configuration *post*, then any run from *post* yields a run from *pre*; or, in the opposite case, we have an empty run from *pre*. Altogether, a run from *pre* is either a finite sequence of transitions ended up with a configuration where no transition is available, or an infinite sequence of transitions, where consecutive configurations are related using *trans*. In order to prove properties on run, we define some temporal logic operators. In the examples considered below we need a very simple version of `always`, which is parametrized by a property P of configurations. In a more general setting, the parameter would be a property of runs. It is well known that a property which holds initially and is invariant is always satisfied on a run. This fact is easily proved in the very general setting provided by Coq.

3 Data Centric Protocols

In this section, we introduce the Netlog language through examples of simple protocols for defining trees. Only the main constructs of the language are presented. A more thorough presentation can be found in [14]. Netlog relies on Datalog-like recursive rules, of the form $head \leftarrow body$, which allow to derive the fact "*head*" whenever the "*body*" is satisfied.

We first recall classical Datalog, whose programs run in a centralized setting over relational structures, and which allow to define invariants that will be used as well in the proofs in the distributed setting. We assume that the language contains *negation* as well as *aggregation functions*, which can be used in the head of rules to aggregate over all values satisfying the body of the rule. For instance, the function *min* will be used in the next example.

Let us start with the program, **BFS-seq**, which computes BFS trees. It runs on an instance of a graph represented by a binary relation E, and a unique node satisfying $root(x)$. The derived relations $onST$, and ST, are such that $onST(\alpha)$

holds for a node α already on the tree, and $ST(\alpha, \beta)$ holds for an edge (α, β) already in the BFS tree.

BFS-seq in Datalog

$$onST(x) \leftarrow Root(x). \tag{1}$$

$$\left.\begin{array}{r} ST(\min(x), y) \\ onST(y) \end{array}\right\} \leftarrow E(x, y); onST(x); \neg onST(y). \tag{2}$$

The evaluation of the program is iterated in a inflationary manner, by accumulating the results till a fixpoint, which defines its semantics, is reached. At the first step, the root node is included in the relation *onST* using the first rule. At the nth step, nodes at distance $n - 1$ from the root are added in *onST*, and an arc of the tree is added in *ST* for each of them, by choosing the parent with minimal Id. The fixpoint is reached when all nodes are on the tree.

Minimum spanning trees can be defined in Datalog with arithmetic. Let us consider first the definition corresponding to Prim's algorithm [26]. We assume weighted graphs, $G = (V, E, \omega)$, where the weight $\omega : E \rightarrow \mathbb{R}^+$, satisfies $\omega(u, v) = \omega(v, u)$ for every edge $(u, v) \in E$. As usual, to simplify the algorithm, we assume that ω is a 1-1 mapping, so the weights of any pair of edges are distinct. Prim's algorithm starts from a (root) node, and construct successive fragments of the MST, by adding the minimal outgoing edge to the fragment at each step.

The sequential Datalog program can be written with three rules as follows. The symbol "!" denotes the consumption of the fact used in the body of the rule, which is deleted after the application of the rule.

MST-Prim-seq in Datalog

$$\left.\begin{array}{r} onST(x) \\ MWOE(min(m)) \end{array}\right\} \leftarrow Root(x); E(x, y, m). \tag{3}$$

$$\left.\begin{array}{r} ST(x, y) \\ onST(y) \end{array}\right\} \leftarrow onST(x); \neg onST(y); E(x, y, m); !MWOE(m). \tag{4}$$

$$MWOE(min(m)) \leftarrow onST(x); \neg onST(y); E(x, y, m); \neg MWOE(m). \tag{5}$$

The evaluation of this program alternates two phases, (i) computation of the minimal outgoing edge's weight, MWOE, and when it is obtained, (ii) addition of the corresponding unique edge.

Let us consider now, Netlog programs, which are installed on each node, where they run concurrently. The rules of a program are applied in parallel, and the results are computed by iterating the rules over the local instance of the node, using facts either stored on the node or pushed by a neighbor. In contrast with other approaches to concurrency, the focus is not primarily on monitoring events, but data (i.e. Datalog facts) contained in nodes.

The facts deduced from rules can be stored on the node, on which the rules run, or sent to other nodes. The symbol in the head of the rules means that the

result has to be either stored on the local data store ($\downarrow$), sent to neighbor nodes ($\uparrow$), or both ($\updownarrow$). The facts received on a node are used to trigger the rules, but do not get stored on that node.

The evaluation of the body of a rule is always performed on a given node. A fact is then considered to hold if and only if it occurs on this node. The *negation* of a fact holds if the fact does not occur on the node where the computation is performed.

The following program, which constructs a spanning tree over a distributed system, relies as above on three relation symbols: E, $onST$, and ST; E represents the edge relation; and at any stage of the computation, $onST(\alpha)$ (respectively $ST(\alpha,\beta)$) hold iff the node α (respectively the edge (α,β)) is already on the intended tree.

Spanning Tree Protocol in Netlog

$$\updownarrow onST(x) \quad \leftarrow @x = 0. \tag{6}$$

$$\left.\begin{array}{r} \updownarrow onST(y) \\ \downarrow ST(min(x), y) \end{array}\right\} \leftarrow E(x, @y); onST(x); \neg onST(y). \tag{7}$$

Rule (6) runs on the unique (rrot) node, say ρ, which satisfies the relation $\rho = 0$. It derives a fact $onST(\rho)$, which is stored on ρ and sent to its neighbors. Rule (7) runs on the nodes ($@y$) at the border of the already computed tree. It chooses one parent (the one with minimal Id) to join the tree. Two facts are derived, which are both locally stored. The fact $onST(y)$ is pushed to all neighbors. Each fact $E(x,y)$ is assumed to be initially stored on node y. As no new fact $E(x,y)$ can be derived from Rules (6) and (7), the consistency of E with the physical edge relation holds forever. This algorithm aims at constructing suitable distributed relations $onST$ and ST. In Section 4, we will prove that they actually define a tree; moreover, in the synchronous setting they define a BFS tree.

The translation of the sequential Datalog program defining a spanning tree, to a distributed program is almost trivial. It suffices to add communication instructions since the program runs locally. The translation to a distributed program is more complex for the **Minimal Spanning tree protocol**. Indeed, rules (4) and (5) are not local, and require communication between remote nodes. In a network, the root can orchestrate the distributed computation, by alternating phases of (i) computation of the MWOE by a convergecast into the current spanning tree, and (ii) addition of the new edge with minimal weight to the tree.

The next program (together with two simple rule modules for the convergecast and the edge addition) defines the minimal spanning tree in Netlog. The facts $AddEdge(x,m)$ and $GetMWOE(x)$ are triggering rule modules (of half a dozen rules) that perform respectively a traversal of the tree to add the edge with minimal outgoing weight, and a convergecast to obtain the new minimal outgoing weight. These rule modules (omitted for space reason) can be used in other protocols requiring non local actions. We have tested them in particular in a concurrent version of the Minimal Spanning Tree, the GHS, where there is no initial root and all nodes concurrently start building MST fragments.

Minimum Spanning Tree Protocol in Netlog

$$\left.\begin{array}{c} onST(x) \\ UpMWOE(x, min(m)) \\ GetMWOE(x) \end{array}\right\} \leftarrow Root(x); E(x, y, m). \tag{8}$$

$$\downarrow AddEdge(x, m) \leftarrow Root(x); !GetMWOE(x); !UpMWOE(x, m). \tag{9}$$

$$\downarrow GetMWOE(x) \leftarrow Root(x); !AddEdge(x, m); !UpEdge(x, m). \tag{10}$$

4 Verifying Tree Protocols

We conduct the verification in two settings. In the asynchronous case, we prove that the previous protocol for spanning tree eventually constructs a spanning tree, while in the synchronous case, we prove that this protocol constructs actually a spanning tree by doing a breadth-first search in the network. We briefly sketch the first case study and then give a more detailed discussion for the second one which involves a much more difficult proof.

In both cases we expect to show that the relation ST determines a spanning tree. However, this relation is distributed on the nodes and the Netlog protocol reacts only to a locally visible part of relations ST, $onST$ and E. The expected property is then stated in terms of the *union* of all ST facts available on the network.

4.1 Spanning Tree in the Asynchronous Case

We have to check that when adding a new fact $ST(x, y)$ at some node loc then x is already on the tree while y is not yet. This is basically entailed by the body of the last rule, but additional properties are needed in order to ensure this rigorously. We use the following ones:

1. The E relation corresponds exactly to the edges.
2. An $onST(z)$ fact arriving at a node y is already stored on the sender x.
3. If an $onST(x)$ fact is stored on a node loc, then $x = loc$.
4. The $onST$ relation grows consistently with ST ($onST$ is actually the engine of the algorithm), and these two relations define a tree.

The first three properties are separately proved to be invariant. The last property is included in a predicate `is_tree`(o, s), which intuitively means that the union of all $onST$ facts o and the union of all ST facts s are consistent and they define a tree. We prove that if at the beginning of a round the first three properties together with `is_tree`(o, s) hold, then at the end of the round `is_tree`(o, s) still holds. The conjunction of all the four properties then constitutes an invariant of the protocol.

We check that the initial configuration generates a tree, then we have that in all configurations of any asynchronous run starting from the initial configuration,

ST has the shape of a tree. This safety property is formalized in Coq (the script is available online [7]).

Liveness, i.e. each node is eventually a member of $onST$, can be easily proved, provided the graph is finite and connected, and a *fairness* property is assumed in order to discard uninteresting runs where an inactive node is continuously chosen for each local round, instead of another node having an enabled rule. The proof is by induction on the finite cardinality of the set $\overline{onST}$ of nodes which do not satisfy $onST$. If at some point of a run this set is non-empty, then at least one of its members is a neighbor of the current tree due to connectivity. By fairness, this node eventually performs a local round and is no longer in $\overline{onST}$. Formalizing such arguments involving liveness and fairness properties of infinite behaviors of distributed systems has already been done in Coq [9]. The issue of termination is simpler in the synchronous setting, since fairness is no more needed to remove fake stuttering steps.

4.2 BFS in the Synchronous Case

For our second case study, the correctness proof of the BFS protocol, we prove that in the synchronous setting, the union of ST facts is the same as the one which would be computed by a centralized algorithm $\mathcal{O}$ (the oracle) running rules (1) and (2) on a reference version of the global relations $onST$ and ST. This is subtler than one may expect at first sight, because decisions taken on a given node do not depend on the global relations $onST$ and ST, but only on the visible part, which is made of the locally stored facts and of the arriving messages. Moreover, the information contained in an arriving $onST(x)$ fact is ephemeral: this fact is not itself stored locally (only its consequences $onST(y)$ and $ST(m, y)$ are stored) and it will never be sent again. Indeed this information is available exactly at the right time. We therefore make a precise reasoning on the consistency of stored and transmitted facts with the computation that would be performed by the oracle $\mathcal{O}$.

We denote by $\mathcal{C}$ the database of facts managed by $\mathcal{O}$. Our main theorem states that a synchronous round in the distributed synchronous version corresponds to a step of computation performed by $\mathcal{O}$ on $\mathcal{C}$. The proof relies necessarily on a suitable characterization of the body of rule (7), which depends on the presence and the absence of facts $onST$. Therefore we need first to prove that facts $onST$, as computed by distributed rules (6) and (7), are the ones computed by $\mathcal{O}$ and conversely – this is respectively called correctness and completeness of $onST$ (definitions 2 and 3).

The first direction is not very hard (proposition 3). Completeness requires more attention. The issue is to ensure that, given an edge from x to y, such that $onST(x) \in \mathcal{C}$ but $onST(y) \notin \mathcal{C}$, the body of rule (7) holds at y in order to ensure that rule (7) will derive $onST(y)$ as expected by rule (2) at the next step. If we just assume correctness and completeness of $onST$, we get $onST(x)$ only on x, while we need it on y. Therefore a stronger invariant is needed. The key is the introduction of the notion of a *good* edge (definition 4) which says that if $onST(x)$ is stored at x, then $onST(y)$ is stored at y or $onST(x)$ is arriving

at y (both things can happen simultaneously as well). Here are the main steps. Additional properties, such as the establishment of the invariant in the initial configuration (actually: after one synchronous round) are available in [8,7].

Notation. Let φ be a fact; here φ can have the shape $E(x, y)$ or $onST(x)$ or $ST(x, y)$. The presence of a fact φ in a database d is denoted by $\varphi \in d$. The set of facts $ST(x, y)$ in d is denoted by d_{ST}, and use a similar convention for *onST* end E. The database of facts stored at node *loc* is denoted by $|loc|$. Similarly, the database of facts arriving a node y from node x is denoted by $|x{\rightarrow}y|$. Statements such as $onST(z) \in |loc|$ are about a given configuration *cnf* or even an extended configuration $\langle cnf, \mathcal{C}\rangle$, and should be written $cnf, \mathcal{C} \Vdash onST(z) \in |loc|$. In general *cnf* is clear from the context and we just write P instead of $cnf, \mathcal{C} \Vdash P$. When we consider a synchronous round, i.e., a transition between two consecutive configurations *pre* and *post*, we write $P \xrightarrow{sr} Q$ for $pre \Vdash P \;\Rightarrow\; post \Vdash Q$. Similarly, for oracle transitions and transitions between extended configurations, we write respectively $P \xrightarrow{o} Q$ for $\mathcal{C} \Vdash P \;\Rightarrow\; \mathcal{C}' \Vdash Q$ and $P \xrightarrow{sro} Q$ for $pre, \mathcal{C} \Vdash P \;\Rightarrow\; post, \mathcal{C}' \Vdash Q$.

Definition 1. *A configuration satisfies* received-onST-already-stored *if and only if for all edges $x{\rightarrow}y$, if $onST(z) \in |x{\rightarrow}y|$, then $z = x$ and $onST(z) \in |x|$.*

Proposition 1. *After a transition, a configuration always satisfies* received-onST-already-stored.

Proof. By inspection of store and push rules (6) and (7). □

Correctness of *onST*.

Definition 2. *An extended configuration $\langle cnf, \mathcal{C}\rangle$ satisfies* correct-onST *if and only if for all location loc of cnf, if some fact $onST(z)$ is visible at loc, then $onST(z) \in \mathcal{C}$.*

Proposition 2. *We have: $onST(0) \in \mathcal{C} \xrightarrow{o} onST(0) \in \mathcal{C}$.*

Proof. By inspection of oracle rules (1) and (2). □

Proposition 3. *We have: $onST(0) \in \mathcal{C}$, correct-onST $\xrightarrow{sro}$ correct-onST.*

Proof. By inspection of the consequences of rule (7) and using proposition 1. □

Completeness of *onST*. The notion of completeness needed is much more precise than the converse of correct-onST: the location where $onST(z)$ is stored has to be known. This is especially clear in the proof of lemma 1.

Definition 3. *An extended configuration $\langle cnf, \mathcal{C}\rangle$ satisfies* complete-onST-node *if and only if for all x, if $onST(x) \in \mathcal{C}$, then $onST(x)$ is stored at x.*

Definition 4. *An edge* $x \rightarrow y$ *is* good *in a given configuration if and only if, if* $onST(x) \in |x|$*, then* $onST(y) \in |y|$ *or* $onST(x) \in |x \rightarrow y|$*. A configuration satisfies* all-good *if and only if all its edges are good.*

The following proposition is about non-extended configurations, i.e. it is purely about the distributed aspect of the BFS algorithm.

Proposition 4. *We have: received-onST-already-stored, all-good* $\xrightarrow{sr}$ *all-good.*

The main use of goodness is the completeness of the evaluation of the body of rule (7).

Definition 5. *We say that an extended configuration* $\langle cnf, \mathcal{C} \rangle$ *is* ready *if and only if (i) it satisfies correct-onST, complete-onST-node and (ii) cnf satisfies all-good.*

Lemma 1. *Given an extended configuration satisfying ready, and an edge* $x \rightarrow y$ *such that* $onST(x) \in \mathcal{C}$ *but* $onST(y) \notin \mathcal{C}$*, the body of rule* (7) *holds at* y*.*

The propagation of the completeness of *onST* follows.

Proposition 5. *We have: ready* $\xrightarrow{sro}$ *complete-onST-node.*

Correctness and Completeness of *ST*.

Definition 6. *Let cnf be a given configuration. We say that* $\langle cnf, \mathcal{C} \rangle$ *satisfies* same-ST *if and only if the union of all ST facts contained in some node of cnf is the same as set of facts ST in* $\mathcal{C}$*.*

Proposition 6. *We have: ready, same-ST* $\xrightarrow{sro}$ *same-ST.*

Main Theorem. Our invariant is the following conjunction.

Definition 7. *An extended configuration* $\langle cnf, \mathcal{C} \rangle$ *satisfies* invar *if and only if it satisfies* $onST(0) \in \mathcal{C}$*, received-onST-already-stored, ready and same-ST.*

Theorem 1. *We have: invar* $\xrightarrow{sro}$ *invar.*

Proof. Immediate use of propositions 1, 2, 3, 5 and 6. □

Finally, we observe that invar is established after one synchronous round from the initial configuration, and that same-ST holds in the initial configuration. As a consequence, same-ST holds forever, as expected.

Besides this global property, one may wonder whether $ST(x, y)$ facts are located on relevant nodes, i.e. child nodes y in our case, so that this information could be used by a higher layer protocol for transmitting data towards the root. This is actually a simple consequence of Rules (6) and (7), since they ensure that $ST(x, y)$ can only be stored on y. This is formally proved in our framework.

5 Conclusion

We developed a framework for verifying data-centric protocols expressed in a rule-based language. We have shown that both the synchronous and the asynchronous models of communication can be formalized in very similar ways from common building blocks, that can be easily adapted to other communication models. Our framework includes a Coq library, which contains the formalization of the distributed computation environment with the communication network, as well as the embedded machine which evaluates the Netlog programs on each node. The Netlog programs are translated into straightforward Coq definitions. This framework allows us to state and prove formally expected properties of data-centric protocols expressed in Netlog. This is examplified on a topological property of a distributed data structure – a tree – constructed by a simple but subtle program: the proofs, sketched in the paper have been fully designed and formalized in Coq [8].

Figures on the size of our current Coq development are given in Table 1. The detail of justifications such as "by inspection of rules (6) and (7)" requires in general many bureaucratic proof steps. From previous experience with Coq, we know that most of them can be automated using dedicated tactics, so that the user can focus entirely on the interesting part of the proof. The representation of Netlog rules was obtained in a systematical way and could be automated as well, using a deep embedding.

The distributed algorithm considered here as a case study was not as trivial as it may appear at first sight, though it can be expressed in a few lines of Netlog. It was sufficient to cover essential issues of a data-centric distributed algorithm, in particular the relationship between local transformations and global properties. Such properties are difficult to handle and even to state in event-centric approaches to the verification of distributed programs.

The advantage of the techniques we have developed is that they constitute a natural and promising open framework to handle other distributed data-centric algorithms. We are currently working on proofs for minimum spanning trees, and plan to further verify protocols for routing, election, naming, and other fundamental distributed problems.

Table 1. Size of Coq scripts (in number of lines)

Distributed computation model	180
Netlog	1300
Tree definitions and properties	80
Translation of rules	50
Proofs on centralized ST algorithm (Rules (1) and (2))	360
Proofs on (asynchronous) ST (Rules (6) and (7))	1100
Proofs on (synchronous) BFS (Rules (6) and (7))	1300

References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL 2001, vol. 36, pp. 104–115. ACM, New York (2001)
2. Blanchet, B.: Automatic Verification of Correspondences for Security Protocols. Journal of Computer Security 17(4), 363–434 (2009)
3. Castéran, P., Filou, V.: Tasks, Types and Tactics for Local Computation Systems. Studia Informatica Universalis (to appear, 2011)
4. Chandy, K.M.: Parallel program design: a foundation. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1988)
5. Chetali, B.: Formal Verification of Concurrent Programs Using the Larch Prover. IEEE Transactions on Software Engineering 24, 46–62 (1998)
6. Cleaveland, R., Parrow, J., Steffen, B.: The Concurrency Workbench: A semantics-based tool for the verification of concurrency systems. ACM Transactions on Programming Languages and Systems 15(1), 36–72 (1993)
7. Deng, Y., Grumbach, S., Monin, J.-F.: Coq Script for Netlog Protocols, `http://www-verimag.imag.fr/~monin/Proof/NetlogCoq/netlogcoq.tar.gz`
8. Deng, Y., Grumbach, S., Monin, J.-F.: Verifying Declarative Netlog Protocols with Coq: a First Experiment. Research Report 7511, INRIA (2011)
9. Deng, Y., Monin, J.-F.: Verifying Self-stabilizing Population Protocols with Coq. In: TASE 2009, pp. 201–208. IEEE Computer Society, Los Alamitos (2009)
10. Fernandez, J.-C., Garavel, H., Mounier, L., Rasse, A., Rodriguez, C., Sifakis, J.: A toolbox for the verification of LOTOS programs. In: ICSE 1992, pp. 246–259. ACM, New York (1992)
11. Gallager, R.G., Humblet, P.A., Spira, P.M.: A Distributed Algorithm for Minimum-Weight Spanning Trees. ACM Trans. Program. Lang. Syst. 5(1), 66–77 (1983)
12. Giménez, E.: A Calculus of Infinite Constructions and its application to the verification of communicating systems. PhD thesis, ENS Lyon (1996)
13. Gotzhein, R., Bredereke, J. (eds.): Formal Description Techniques IX: Theory, application and tools, IFIP TC6 WG6.1, IFIP Conference Proceedings, vol. 69. Chapman and Hall, Boca Raton (1996)
14. Grumbach, S., Wang, F.: Netlog, a Rule-Based Language for Distributed Programming. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 88–103. Springer, Heidelberg (2010)
15. Hesselink, W.H.: The Verified Incremental Design of a Distributed Spanning Tree Algorithm: Extended Abstract. Formal Asp. Comput. 11(1), 45–55 (1999)
16. Heyd, B., Crégut, P.: A Modular Coding of UNITY in COQ. In: von Wright, J., Harrison, J., Grundy, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 251–266. Springer, Heidelberg (1996)
17. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
18. Jard, C., Monin, J.F., Groz, R.: Development of Veda, a Prototyping Tool for Distributed Algorithms. IEEE Trans. Softw. Eng. 14(3), 339–352 (1988)
19. Kirkwood, C., Thomas, M.: Experiences with specification and verification in LOTOS: a report on two case studies. In: WIFT 1995, p. 159. IEEE Computer Society Press, Los Alamitos (1995)
20. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)
21. Långbacka, T.: A HOL Formalisation of the Temporal Logic of Actions. In: TPHOL 1994, pp. 332–345. Springer, Heidelberg (1994)

22. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL 2006, pp. 42–54. ACM, New York (2006)
23. Liu, C., Mao, Y., Oprea, M., Basu, P., Loo, B.T.: A declarative perspective on adaptive manet routing. In: PRESTO 2008, pp. 63–68. ACM, New York (2008)
24. Loo, B.T., Condie, T., Garofalakis, M.N., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: language, execution and optimization. In: ACM SIGMOD 2006 (2006)
25. Loo, B.T., Hellerstein, J.M., Stoica, I., Ramakrishnan, R.: Declarative routing: extensible routing with declarative queries. In: ACM SIGCOMM 2005 (2005)
26. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1996)
27. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2, 219–246 (1989)
28. Monin, J.-F.: Proving a real time algorithm for ATM in Coq. In: Giménez, E. (ed.) TYPES 1996. LNCS, vol. 1512, pp. 277–293. Springer, Heidelberg (1998)
29. Moses, Y., Shimony, B.: A New Proof of the GHS Minimum Spanning Tree Algorithm. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 120–135. Springer, Heidelberg (2006)
30. Paulin-Mohring, C.: Circuits as Streams in Coq: Verification of a Sequential Multiplier. In: Berardi, S., Coppo, M. (eds.) TYPES 1995. LNCS, vol. 1158, pp. 216–230. Springer, Heidelberg (1996)
31. Paulson, L.C.: Mechanizing UNITY in Isabelle. ACM Trans. Comput. Logic 1(1), 3–32 (2000)
32. Regensburger, F., Barnard, A.: Formal Verification of SDL Systems at the Siemens Mobile Phone Department. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 439–455. Springer, Heidelberg (1998)
33. Roscoe, A.W.: Model-checking CSP, ch. 21. Prentice-Hall, Englewood Cliffs (1994)
34. Shahrier, S.M., Jenevein, R.M.: SDL Specification and Verification of a Distributed Access Generic optical Network Interface for SMDS Networks. Technical report, University of Texas at Austin (1997)
35. Törö, M., Zhu, J., Leung, V.C.M.: SDL specification and verification of universal personal computing: with Object GEODE. In: FORTE XI / PSTV XVIII 1998, pp. 267–282. Kluwer, B.V., Dordrecht (1998)
36. Turner, K.J.: Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL. John Wiley & Sons, Inc., Chichester (1993)
37. Wang, A., Basu, P., Loo, B.T., Sokolsky, O.: Declarative Network Verification. In: Gill, A., Swift, T. (eds.) PADL 2009. LNCS, vol. 5418, pp. 61–75. Springer, Heidelberg (2008)
38. Welch, J.L., Lamport, L., Lynch, N.: A lattice-structured proof of a minimum spanning. In: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing, PODC 1988, pp. 28–43. ACM, New York (1988)
39. Wu, J.-P., Chanson, S.T.: Translation from LOTOS and Estelle Specifications to Extended Transition System and its Verification. In: FORTE 1989, pp. 533–549. North-Holland Publishing Co., Amsterdam (1990)
40. Zhang, W.: Applying SDL Specifications and Tools to the Verification of Procedures. In: Reed, R., Reed, J. (eds.) SDL 2001. LNCS, vol. 2078, pp. 421–438. Springer, Heidelberg (2001)

Relational Concurrent Refinement: Timed Refinement

John Derrick[1] and Eerke Boiten[2]

[1] Department of Computer Science, University of Sheffield, Sheffield, S1 4DP, UK
[2] School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, UK

Abstract. Data refinement in a state-based language such as Z is defined using a relational model in terms of the behaviour of abstract programs. Downward and upward simulation conditions form a sound and jointly complete methodology to verify relational data refinements, which can be checked on an event-by-event basis rather than per trace. In models of concurrency, refinement is often defined in terms of sets of observations, which can include the events a system is prepared to accept or refuse, or depend on explicit properties of states and transitions. By embedding such concurrent semantics into a relational one, eventwise verification methods for such refinement relations can be derived. In this paper we continue our program of deriving simulation conditions for process algebraic refinement by considering how notions of time should be embedded into a relational model, and thereby deriving relational notions of timed refinement.

Keywords: Data refinement, Z, simulations, timed-refinement.

1 Introduction

The modelling and understanding of time is important in computer science. It plays an especially important role in refinement, where how time is modelled and how it is treated in a development step lead to important differences and subtleties in notions of refinement. These distinctions are more prominent in a process algebra or behavioural setting where many refinement preorders have been defined, reflecting different choices of what is taken to be observable and different choices of how time is modelled.

In a process algebra such as CSP [11] a system is defined in terms of actions (or events) which represent the *inter*actions between a system and its environment. The exact way in which the environment is allowed to interact with the system varies between different semantics. Typical semantics are set-based, associating one or more sets with each process, for example traces, refusals, divergences. Refinement is then defined in terms of set inclusions and equalities between the corresponding sets for different processes. A survey of many prominent untimed process algebraic refinement relations is given in [24,25]. The addition of time adds further complications, and there are important choices as to how time is modelled, and the assumptions one makes. These choices all affect any associated refinement relations.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 121–137, 2011.
© IFIP International Federation for Information Processing 2011

In a state-based system, e.g., one specified in Z, specifications are considered to define abstract data types (ADTs), consisting of an initialisation, a collection of operations and a finalisation. A program over an ADT is a sequential composition of these elements. Refinement is defined to be the subset relation over program behaviours for all possible programs, where what is deemed visible is the input/output relation. The accepted approach to make verification of refinements tractable is through downward and upward simulations which are sound and jointly complete [4]. Although there has been some work on modelling time relationally (eg see work on duration calculus and its integration with state-based languages [27]), there has been less work on associated refinement relations, and none on how to model the various choices that arise in a process algebraic setting.

In integrated notations, for both practical and theoretical reasons, it is important to understand how time is modelled, whether from the process algebraic or state-based angle, and how this impacts on refinement. Our ongoing research on relational concurrent refinement [6,3,7,2,8] contributes to this agenda, by explicitly recording in a relational setting the observations characterising process algebraic models. This allows the verification of concurrent refinement through the standard relational method of simulations, and to interpret relational formalisms like Z in a concurrency model.

We derived simulation rules for process algebraic refinement (such as trace, failures-divergences, readiness [6] etc), including also outputs and internal operations [3], for different models of divergence [2], as well as automaton-based refinements [8]. The current paper extends this by considering how time can be modelled in a relational context, and thus we derive simulation rules for some of the timed refinement preorders. In Section 2 we provide the basic definitions and background. In Section 3 we provide the simulation rules for a number of process algebraic preorders. In Section 4 we introduce time and timed refinements, and derive their relational simulation rules. We conclude in Section 5.

2 Background

The standard refinement theory of Z [26,5] is based on a relational model of data refinement where all operations are total, as described in [10]. However, the restriction to total relations can be dropped, see [9], and soundness and joint completeness of the same set of simulation rules in the more general case can be shown.

2.1 A Partial Relational Model

A program (defined here as a sequence of operations) is given as a relation over a global state G, implemented using a local state State. The *initialisation* of the program takes a global state to a local state, on which the operations act, a *finalisation* translates back from local to global. In order to distinguish between relational formulations (which use Z as a meta-language) and expressions in terms of Z schemas etc., we use the convention that expressions in the relational data types are typeset in a sans serif font.

Definition 1 (Data type)
A (partial) data type *is a quadruple* $(\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i \in J}, \mathsf{Fin})$*. The operations* $\{\mathsf{Op}_i\}$*, indexed by* $i \in J$*, are relations on the set* State*;* Init *is a total relation from* G *to* State*;* Fin *is a total relation from* State *to* G*. If the operations are all total relations, we call it a* total *data type.* □

Insisting that Init and Fin be total merely records the facts that we can always start a program sequence and that we can always make an observation.

Definition 2 (Program)
For a data type $\mathsf{D} = (\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i \in J}, \mathsf{Fin})$ *a* program *is a sequence over* J*. The meaning of a program* p *over* D *is denoted by* p_D*, and defined as follows. If* $\mathsf{p} = \langle \mathsf{p}_1, ..., \mathsf{p}_\mathsf{n} \rangle$ *then* $\mathsf{p}_\mathsf{D} = \mathsf{Init} \fatsemi \mathsf{Op}_{\mathsf{p}_1} \fatsemi ... \fatsemi \mathsf{Op}_{\mathsf{p}_\mathsf{n}} \fatsemi \mathsf{Fin}$. □

Definition 3 (Data refinement)
For data types A *and* C*,* C refines A*, denoted* $\mathsf{A} \sqsubseteq_{data} \mathsf{C}$ *(dropping the subscript if the context is clear), iff for each program* p *over* J*,* $\mathsf{p}_\mathsf{C} \subseteq \mathsf{p}_\mathsf{A}$. □

Downward and *upward* simulations form a sound and jointly complete [10,4] proof method for verifying refinements. In a simulation a step-by-step comparison is made of each operation in the data types, and to do so the concrete and abstract states are related by a retrieve relation.

Definition 4 (Downward and upward simulations)
Let $\mathsf{A} = (\mathsf{AState}, \mathsf{AInit}, \{\mathsf{AOp}_i\}_{i \in J}, \mathsf{AFin})$ *and* $\mathsf{C} = (\mathsf{CState}, \mathsf{CInit}, \{\mathsf{COp}_i\}_{i \in J}, \mathsf{CFin})$*. A* downward *simulation is a relation* R *from* AState *to* CState *satisfying*

$$\mathsf{CInit} \subseteq \mathsf{AInit} \fatsemi \mathsf{R}$$
$$\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin}$$
$$\forall\, i : J \bullet \mathsf{R} \fatsemi \mathsf{COp}_i \subseteq \mathsf{AOp}_i \fatsemi \mathsf{R}$$

An upward *simulation is a relation* T *from* CState *to* AState *such that*

$$\mathsf{CInit} \fatsemi \mathsf{T} \subseteq \mathsf{AInit}$$
$$\mathsf{CFin} \subseteq \mathsf{T} \fatsemi \mathsf{AFin}$$
$$\forall\, i : J \bullet \mathsf{COp}_i \fatsemi \mathsf{T} \subseteq \mathsf{T} \fatsemi \mathsf{AOp}_i$$

2.2 Refinement in Z

The definition of refinement in a specification language such as Z is usually based on the relational framework described above, using an additional intermediate step (not used in the rest of this paper) where partial relations are embedded into total relations ("totalisation", see [26,5] for details). Specifically, a Z specification can be thought of as a data type, defined as a tuple $(\mathit{State}, \mathit{Init}, \{\mathit{Op}_i\}_{i \in J})$. The operations Op_i are defined in terms of (the variables of) State (its before-state) and State' (its after-state). The initialisation is also expressed in terms of an

after-state $State'$. In addition to this, operations can also consume inputs and produce outputs. As finalisation is implicit in these data types, it only has an occasional impact on specific refinement notions. If specifications have inputs and outputs, these are included in both the global and local state of the relational embedding of a Z specification. See [5] for the full details on this – in this paper we only consider data types without inputs and outputs. In concurrent refinement relations, inputs add little complication; outputs particularly complicate refusals as described in [3].

In a context where there is no input or output, the global state contains no information and is a one point domain, i.e., $\mathsf{G} == \{*\}$, and the local state is $\mathsf{State} == \mathit{State}$. In such a context the other components of the embedding are as given below.

Definition 5 (Basic embedding of Z data types). *The Z data type* $(State, Init, \{Op_i\}_{i \in J})$ *is interpreted relationally as* $(\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i \in J}, \mathsf{Fin})$ *where*

$$\begin{array}{l} \mathsf{Init} == \{Init \bullet * \mapsto \theta State'\} \\ \mathsf{Op} == \{Op \bullet \theta State \mapsto \theta State'\} \\ \mathsf{Fin} == \{State \bullet \theta State \mapsto *\} \end{array}$$

Given these embeddings, we can translate the relational refinement conditions of downward simulations for totalised relations into the following refinement conditions for Z ADTs.

Definition 6 (Standard downward simulation in Z)
Given Z data types $A = (AState, AInit, \{AOp_i\}_{i \in J})$ *and* $C = (CState, CInit, \{COp_i\}_{i \in J})$. *The relation* R *on* $AState \wedge CState$ *is a* downward simulation *from* A *to* C *in the non-blocking model if*

$$\begin{array}{l} \forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R' \\ \forall\, i : J;\ AState;\ CState \bullet \mathrm{pre}\, AOp_i \wedge R \Rightarrow \mathrm{pre}\, COp_i \\ \forall\, i : J;\ AState;\ CState;\ CState' \bullet \mathrm{pre}\, AOp_i \wedge R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i \end{array}$$

In the blocking model, the correctness (last) condition becomes

$$\forall\, i : J;\ AState;\ CState;\ CState' \bullet R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i \qquad \square$$

The translation of the upward simulation conditions is similar, however this time the finalisation produces a condition that the simulation is total on the concrete state.

Definition 7 (Standard upward simulation in Z)
For Z data types A *and* C, *the relation* T *on* $AState \wedge CState$ *is an* upward simulation *from* A *to* C *in the non-blocking model if*

$$\begin{array}{l} \forall\, AState';\ CState' \bullet CInit \wedge T' \Rightarrow AInit \\ \forall\, i : J;\ CState \bullet \exists\, AState \bullet T \wedge (\mathrm{pre}\, AOp_i \Rightarrow \mathrm{pre}\, COp_i) \\ \forall\, i : J;\ AState';\ CState;\ CState' \bullet \\ \qquad\qquad (COp_i \wedge T') \Rightarrow (\exists\, AState \bullet T \wedge (\mathrm{pre}\, AOp_i \Rightarrow AOp_i)) \end{array}$$

In the blocking model, the correctness condition becomes

$$\forall\, i : J;\ AState';\ CState;\ CState' \bullet (COp_i \wedge T') \Rightarrow \exists\, AState \bullet T \wedge AOp_i \quad \square$$

3 Process Algebraic Based Refinement

The semantics of a process algebra [11,14,1] is often given by associating a labelled transition system (LTS) to each term. Equivalence, and preorders, can be defined over the semantics where two terms are identified whenever no observer can notice any difference between their external behaviours. Varying how the environment *interacts* with a process leads to differing observations and therefore different preorders (i.e., refinement relations) – an overview and comprehensive treatment is provided by van Glabbeek in [24,25]. We will need the usual notation for labelled transition systems:

Definition 8 (Labelled Transition Systems (LTSs))
A labelled transition system is a tuple $L = (States, Act, T, Init)$ where States is a non-empty set of states, $Init \subseteq States$ is the set of initial states, Act is a set of actions, and $T \subseteq States \times Act \times States$ is a transition relation. The components of L are also accessed as $states(L) = States$ and $init(L) = Init$. $\square$

Every state in the LTS itself represents a process – namely the one representing all possible behaviour from that point onwards. Specific notation needed includes the usual notation for writing transitions as $p \xrightarrow{a} q$ for $(p, a, q) \in T$ and the extension of this to traces (written $p \xrightarrow{tr} q$) and the set of enabled actions of a process which is defined as: $next(p) = \{a \in Act \mid \exists\, q \bullet p \xrightarrow{a} q\}$.

To relate refinements in process algebras to those in a relational model, the methodology (as described in earlier papers [6,3,7,8]) is as illustrated in the following subsection. As an example in an untimed context we give the embedding of the trace semantics and trace preorder as follows.

3.1 Trace Preorder

We first define the trace refinement relation.

Definition 9. *$\sigma \in Act^*$ is a trace of a process p if $\exists\, q \bullet p \xrightarrow{\sigma} q$. $\mathcal{T}(p)$ denotes the set of traces of p. The trace preorder is defined by $p \sqsubseteq_{tr} q$ iff $\mathcal{T}(q) \subseteq \mathcal{T}(p)$.* $\square$

We then define a relational embedding of the Z data type, that is, define a data type (specifically define the finalisation operation) so as to facilitate the proof that data refinement equals the event based semantics. The choice of finalisation is taken so that we observe the characteristics of interest. Thus in the context of trace refinement we are interested in observing traces, but in that of failures refinement we need to observe more. So here possible traces lead to the single global value; impossible traces have no relational image.

Definition 10 (Trace embedding)
A Z data type $(State, Init, \{Op_i\}_{i \in J})$ *has the following trace embedding into the relational model.*

$$\begin{aligned}
&\mathsf{G} == \{*\}\\
&\mathsf{State} == State\\
&\mathsf{Init} == \{Init \bullet * \mapsto \theta State'\}\\
&\mathsf{Op} == \{Op \bullet \theta State \mapsto \theta State'\}\\
&\mathsf{Fin} == \{State \bullet (\theta State, *)\}
\end{aligned}$$

To distinguish between the different embeddings we denote the trace embedding of a data type A *as* $\mathsf{A}\,\big|_{tr}$. *We drop the* $\big|_{tr}$ *if the context is clear.* □

We then describe how to calculate the relevant LTS aspect from the Z data type. For example, for trace refinement what denotes traces (as in Definition 9) in the Z data type.

Definition 11. *The* traces *of a Z data type* $(State, Init, \{Op_i\}_{i \in J})$ *are all sequences* $\langle i_1, \ldots, i_n \rangle$ *such that*

$$\exists\, State' \bullet Init \mathbin{⨾} Op_{i_1} \mathbin{⨾} \ldots \mathbin{⨾} Op_{i_n}$$

We denote the traces of an ADT A *by* $\mathcal{T}(A)$. □

We now prove that data refinement equals the relevant event based definition of refinement:

Theorem 1. *With the trace embedding, data refinement corresponds to trace preorder. That is, when Z data types* A *and* C *are embedded as* A *and* C*,*

$$\mathsf{A}\,\big|_{tr} \sqsubseteq_{data} \mathsf{C}\,\big|_{tr} \quad \text{iff } \mathcal{T}(C) \subseteq \mathcal{T}(A)$$

Finally, we extract a characterisation of refinement as simulation rules on the operations of the Z data type. These are of course the rules for standard Z refinement but omitting applicability of operations, as used also e.g., in Event-B.

$$\begin{aligned}
&\mathsf{CInit} \subseteq \mathsf{AInit} \mathbin{⨾} \mathsf{R}\\
&\mathsf{R} \mathbin{⨾} \mathsf{CFin} \subseteq \mathsf{AFin}\\
&\forall\, i : I \bullet \mathsf{R} \mathbin{⨾} \mathsf{COp}_i \subseteq \mathsf{AOp}_i \mathbin{⨾} \mathsf{R}
\end{aligned}$$

We thus have the following conditions for the trace embedding.

Definition 12 (Trace simulations in Z)
A relation R *on* $AState \wedge CState$ *is a* trace downward simulation *from* A *to* C *if*

$$\begin{aligned}
&\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R'\\
&\forall\, i \in J \bullet \forall\, AState;\ CState;\ CState' \bullet R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i
\end{aligned}$$

The relation T on $AState \wedge CState$ is a trace upward simulation *from A to C if it is total on $CState$ and*

$$\forall\, AState';\ CState' \bullet CInit \wedge T' \Rightarrow AInit$$
$$\forall\, i : J \bullet \forall\, AState';\ CState;\ CState' \bullet (COp_i \wedge T') \Rightarrow (\exists\, AState \bullet T \wedge AOp_i) \qquad \square$$

4 Timed Models

We now add in the consideration of time into our framework. We adapt our definitions of data type, refinement and simulations given above for the untimed case, and again consider different refinement relations based upon differing semantics.

4.1 A Timed Relational Model

We do not consider internal or silent actions, as their consideration complicates some of the timing based issues and these are discussed below. The model we define is essentially that of timed automaton [13], and we make the following assumptions:

- We use a continuous time domain $\mathbb{R}^+$, although we could parameterize the theory by a time domain $\mathcal{T}$.
- We have a single global clock.
- We impose no constraints about Zeno behaviour or its absence.
- We have no internal events.
- We have no timelocks, time can always progress.
- *Time additivity:* If time can advance by a particular amount d in two steps, then it can also advance by d in a single step. This corresponds to the axiom **S1** in [13].
- *Time interpolation:* which is the converse of time additivity, that is, for any time progression there is an intermediate state at any instant during that progression. This corresponds to the axiom **S2** in [13].

Additional constraints can be placed as is done in many models. For example, *constancy of offers* [23] states that the progression of time does not introduce any new events as possibilities, nor allows the withdrawal of any offers. Another axiom which is commonly included is that of *time determinism*, that is, that if time evolves but no internal or other visible action takes place then the new state is uniquely determined.

Our relational model will then be constructed with the following in mind:

- We assume the specifications contain a reserved variable t (of type $\mathbb{R}^+$) representing the current time which is part of the local state *State*, which also serves as the relational local state State.
- Atomic operations can refer to t but not change it, that is, they are instantaneous.

- The time passing operation Op_τ advances t by τ as well as possibly having other effects.
- A variable t representing time is also added to the global state G. Since time can always progress, these time passing operations are total.
- The finalisation Fin maps t in State to t in G, that is, it makes time visible at the end of a computation.
- Programs are then, as before, finite sequences of time passing or atomic operations.

With this construction (which is just an embedding of time into our relational model) Definitions 1 and 2 define notions of data type and program, and Definitions 3 and 4 give us definitions of refinement and simulations for use on the timed relational model. Furthermore, the assumption of a global clock holds for both abstract and concrete systems, and simulation conditions on the given finalisation imply that the retrieve relation has to be the identity as far as time t is concerned.

4.2 A Timed Behavioural Model

To relate the simulations with differing refinement preorders we need a timed semantics such as provided by timed automaton, e.g., [13], or the semantics for a timed process algebra, e.g., [20,23]. To do so we will augment our LTSs with visible time passing actions, and will not make a fundamental distinction between a transition due to time and one due to another atomic action as is done in some process algebraic semantics (i.e., we do not have two types of transitions). Similarly we do not, at this stage, make a distinction between visible time actions and other external actions as is done in, say, [23] (i.e., we do not have two types of actions or events), c.f. pp3 of [13].

Example 1. The automaton A in Figure 1 represents a system that places no timing constraints on the sequential ordering of the events a then b then *stop*. In this, and subsequent figures, we denote arbitrary time passing by the transition $\stackrel{d}{\longrightarrow}$.

In Figure 1 B adapts the above to introduce a specific time delay of at least 1 time units (assumed here to be secs) between a and b. This system corresponds

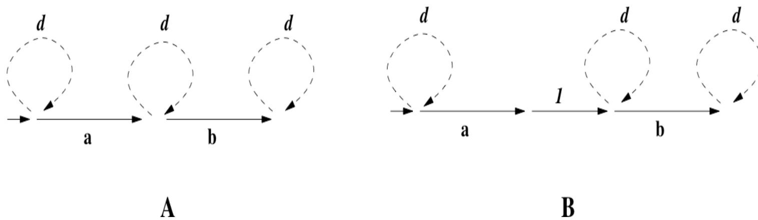


Fig. 1. Simple timed specifications

to the timed CSP behaviour of $a \xrightarrow{1} b \rightarrow stop$. A timeout can be modelled by allowing a state-change to occur at a certain point in time. Figure 2 corresponds to the timed CSP expression $(a \rightarrow stop) \overset{10}{\triangleright} (c \rightarrow stop)$. Note that in the presence of the requirement of constancy of offers it would be necessary to use an internal event to model the timeout at 10 secs; without such a requirement we can use a transition of 0 secs which represents a state change at that moment in time. □

Timed traces, which consist of a trace of action-time pairs (the time component recording the time of occurrence of the action), are often used as the representation of visible executions in a timed model (e.g., as in [13,20,23]). However, in order that we can reuse results from Section 3 we take timed traces to be simply traces over the visible actions, which here includes time and other atomic actions. Thus a timed trace here will be of the form $\langle t_1, a_1, t_2, a_2, \ldots \rangle$, where the strict alternation can be assumed without loss of generality. In[13] a mapping is defined between the set of traces given as action-time pairs and those consisting of traces over time and other atomic actions, thus there is no loss of generality in our assumption of the form of timed traces.

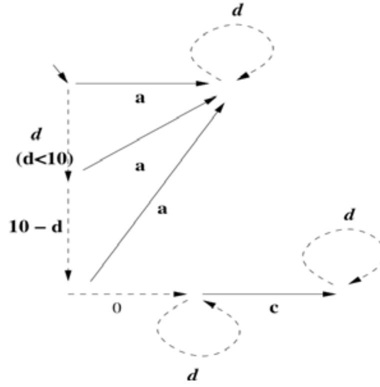


Fig. 2. A timeout

Example 2. The timed traces of A in Figure 1 include $\langle 4, a, 6.2, b \rangle$ and $\langle 0, 1, a, b, 4 \rangle$, those of B include $\langle 4, a, 1, b \rangle$, $\langle 3, a, 1, b, 7 \rangle$ etc. The relational model of B can be written as, where t_a is the time of occurrence of the action a:

$$
\begin{aligned}
& G == \mathbb{R}^+ \\
& State == [t : \mathbb{R};\ x : \{0, 1, 2\};\ t_a : \mathbb{R}] \\
& Init == \{t_0 : \mathbb{R}^+ \bullet t_0 \mapsto \langle\!\langle t = t_0, x = 0, t_a = 0 \rangle\!\rangle\} \\
& Op_\tau == [\Delta State \mid x' = x \wedge t'_a = t_a \wedge t' = t + \tau] \\
& a == [\Delta State \mid x = 0 \wedge x' = 1 \wedge t'_a = t \wedge t' = t] \\
& b == [\Delta State \mid x = 1 \wedge x' = 2 \wedge t \geq t_a + 1 \wedge t' = t] \\
& Fin == \lambda\, s : State \bullet s.t
\end{aligned}
$$

Timed trace preorder. We now consider the first preorder: the timed trace preorder.

Definition 13. *Let Act be the atomic (non-time passing) actions, and t–Act* $= Act \cup \{ Op_\tau \mid \tau \in \mathbb{R}^+ \}$. *Let us denote the set of timed traces of* p *by* $\mathcal{T}_t(p) \subseteq (t\text{–}Act)^*$. *The timed trace preorder,* $\sqsubseteq_{t-tr}$, *is defined by* $p \sqsubseteq_{t-tr} q$ *iff* $\mathcal{T}_t(q) \subseteq \mathcal{T}_t(p)$. □

Example 3. Figure 1 defines A and B with $A \sqsubseteq_{t-tr} B$ but $B \not\sqsubseteq_{t-tr} A$. However, A and B have the same underlying *untimed* behaviour even though $A \neq_{t-tr} B$. □

Because of the construction of our timed relational model, and in particular, the consideration of timed actions as visible actions, the embeddings and correspondence given in Definitions 10 and 11 and Theorem 1 carry over directly to the timed case. Thus all that remains to be done is to derive the consequences for the simulation rules. In fact, the derivations given in Definition 12 still hold providedt the quantification $\forall\, i : I$ includes quantification over the time passing events. All that remains is thus to articulate the precise consequences of this with regards to time. We consider these in turn.

Initialisation: the consequence here is, since the retrieve relation is the identity on time, the times at initialisation must be the same.

Non-time passing actions: the correctness condition falls apart into two parts: the usual correctness condition as in Definition 12 for the untimed aspects of the behaviour, with the additional consequence that

$$R \wedge \operatorname{pre} COp_i \Rightarrow \operatorname{pre} AOp_i$$

where the preconditions include reference to t – this in particular implies that in linked states, at the same time t, COp_i can only be enabled if AOp_i is.

Time passing actions: Since by assumption the time passing actions are total, the correctness condition in the blocking model thus becomes (for a downward simulation):

$$\forall\, \tau \in \mathbb{R}^+ \bullet \forall\, AState;\ CState;\ CState' \bullet R \wedge COp_\tau \Rightarrow \exists\, AState' \bullet R' \wedge AOp_\tau$$

if the time passing actions do not change the rest of the state space (i.e., we have time determinism), then this is vacuously satisfied.

Example 4. Figure 3 augments Figure 1 by the representation of the retrieve relation between the states of the two systems. With this retrieve relation it is easy to see that B is a timed-trace downward simulation of A. However, the reverse implication would not hold since

$$\operatorname{pre} AOp_i \Rightarrow \operatorname{pre} BOp_i$$

fails to hold at the state right after a in B. In a similar fashion the timed automaton corresponding to $P = (a \rightarrow stop)$ and $Q = (WAIT\ 2;\ a \rightarrow stop)$ have the same untimed behaviour and $P \sqsubseteq_{t-tr} Q$ but $Q \not\sqsubseteq_{t-tr} P$ since $\langle 1, a \rangle$ is a timed trace of P but not of Q. □

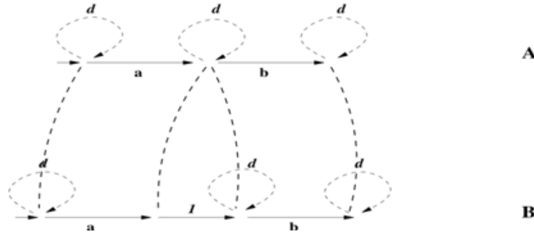


Fig. 3. A retrieve relation between timed specifications

The above simulations agrees with those presented in [13], where Lynch and Vaandrager derive simulations for timed automata in a fashion similar to those presented in [12] for untimed automata (see also results in [8]). Specifically, they present a notion of *timed refinement* and the corresponding *timed forward simulation* and *timed backward simulation*. As in the untimed case, their notion of timed refinement corresponds to timed trace preorder, and their timed simulations to those given by the simulations defined here.

Many of the results in [13] are derived via their untimed counterpart using the idea of a *closure automaton* which embeds a timed automaton into one with explicit time passing steps. Definitions and results on the closure automata can be translated directly to definitions and results on the underlying timed automata. In terms of our relational construction, such a closure corresponds to our basic definition since we consider time passing to be visible (in order that our results concerning simulations carry over to the timed case).

Our motivation for considering time passing to be a visible action here, though is twofold. Not only does it allow one to use the results of the preceding sections concerning simulations, but moreover, the view that time passing is indeed visible. Since we are less concerned with the parallel composition of two components in our relational framework, we do not need to stress the component-environment boundary and interface in a way that comes to the fore in, say, a process algebra. Thus we can either consider time to be visible, yet not under the control of the environment or even to be visible and under the control of an environment that will always offer time passing as one of its behaviours.

Completed timed trace preorder. In an untimed setting, $\sigma \in Act^*$ is a completed trace of a process p if $\exists\, q \bullet p \stackrel{\sigma}{\longrightarrow} q$ and $next(q) = \varnothing$. $\mathcal{CT}(p)$ denotes the set of completed traces of p. The completed trace preorder, $\sqsubseteq_{ctr}$, is defined by $p \sqsubseteq_{ctr} q$ iff $\mathcal{T}(q) \subseteq \mathcal{T}(p)$ and $\mathcal{CT}(q) \subseteq \mathcal{CT}(p)$.

To adapt the notion of the completed trace preorder we need to take completed traces as those that can do no more non-time passing actions (since time can always pass, we'd otherwise have no completed traces). Hence we define:

Definition 14. *σ is a completed timed trace of a process p if $\exists\, q \bullet p \stackrel{\sigma}{\longrightarrow} q$ and $q \stackrel{tr}{\longrightarrow}$ implies $tr \in \{Op_{\mathcal{T}}\}^*$. $\mathcal{CT}_t(p)$ denotes the set of completed timed*

traces of p*. The completed timed trace preorder,* $\sqsubseteq_{t-ctr}$*, is defined by* $p \sqsubseteq_{t-ctr} q$ *iff* $\mathcal{T}_t(q) \subseteq \mathcal{T}_t(p)$ *and* $\mathcal{CT}_t(q) \subseteq \mathcal{CT}_t(p)$*.* □

The basic relational embedding without time uses a global state that has been augmented with an additional element $\surd$, which denotes that the given trace is complete (i.e., no operation is applicable). Thus it uses the finalisation:

$$\mathsf{Fin} == \{State \bullet \theta State \mapsto *\} \cup \{State \mid (\forall\, i : J \bullet \neg\, \mathrm{pre}\, Op_i) \bullet \theta State \mapsto \surd\}$$

We have to adapt in a similar fashion in the presence of time, in particular, amend the finalisation so that we correctly record our completed timed traces. We thus change the global state to $\mathbb{R}^+ \times \{*, \surd\}$ and set

$$\begin{aligned}\mathsf{Fin} == &\{s : State \bullet s \mapsto (s.t, *)\}\\ &\cup\{State \mid (\forall\, i : I;\ \tau : \mathbb{R}^+ \bullet \neg\, \mathrm{pre}(Op_\tau \fatsemi Op_i)) \bullet \theta State \mapsto (\theta State.t, \surd)\}\end{aligned}$$

The effect on the downward simulation finalisation condition is that it becomes:

$$\begin{aligned}&\forall\, AState;\ CState \bullet R \land \forall\, i : I;\ l\tau : \mathbb{R}^+ \bullet \neg\, \mathrm{pre}(COp_\tau \fatsemi COp_i) \Rightarrow\\ &\qquad\qquad \forall\, i : I;\ \tau : \mathbb{R}^+ \bullet \neg\, \mathrm{pre}(AOp_\tau \fatsemi AOp_i)\end{aligned}$$

For an upward simulation, $\mathsf{CFin} \subseteq \mathsf{T} \fatsemi \mathsf{AFin}$ becomes

$$\begin{aligned}&\forall\, CState \bullet (\forall\, i : I;\ \tau : \mathbb{R}^+ \bullet \neg\, \mathrm{pre}(COp_\tau \fatsemi COp_i)) \Rightarrow\\ &\qquad \exists\, AState \bullet T \land \forall\, i : I;\\ &\qquad tau : \mathbb{R}^+ \bullet \neg\, \mathrm{pre}(AOp_\tau \fatsemi AOp_i)\end{aligned}$$

Furthermore, if one makes the additional assumption of constancy of offers, then these reduce to the untimed conditions [8], namely:

Downward simulations: $\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin}$ is equivalent to

$$\forall\, AState;\ CState \bullet R \land \forall\, i : J \bullet \neg\, \mathrm{pre}\, COp_i \Rightarrow \forall\, i : J \bullet \neg\, \mathrm{pre}\, AOp_i$$

Upward simulations: $\mathsf{CFin} \subseteq \mathsf{T} \fatsemi \mathsf{AFin}$ is equivalent to

$$\forall\, CState \bullet \forall\, i : J \bullet \neg\, \mathrm{pre}\, COp_i \Rightarrow \exists\, AState \bullet T \land \forall\, i : J \bullet \neg\, \mathrm{pre}\, AOp_i$$

Those conditions, together with the above for non-time passing actions:

$$R \land \mathrm{pre}\, COp_i \Rightarrow \mathrm{pre}\, AOp_i$$

then give the required timed simulations for completed timed trace preorder.

Timed failure preorder. Timed traces, as defined above, consist of traces where elements are either atomic or time passing actions. Our timed failures will be timed trace-refusals pairs, where refusals will be sets of actions which can be refused at the end of a trace. We thus use the following definition.

Definition 15. $(\sigma, X) \in (t\text{–}Act)^* \times \mathbb{P}(Act)$ *is a timed failure of a process* p *if there is a process* q *such that* $p \xrightarrow{\sigma} q$*, and* $next(q) \cap X = \varnothing$*.* $\mathcal{F}_t(p)$ *denotes the set of timed failures of* p*. The timed failures preorder,* $\sqsubseteq_{t-f}$*, is defined by* $p \sqsubseteq_{t-f} q$ *iff* $\mathcal{F}_t(q) \subseteq \mathcal{F}_t(p)$*.*

Example 5. The timed automata corresponding to $Q = (WAIT\ 2;\ a \rightarrow stop)$ is a timed trace refinement, but not a timed failure refinement, of that corresponding to $P = (a \rightarrow stop)$. That is, $P \sqsubseteq_{t-tr} Q$ but $P \not\sqsubseteq_{t-f} Q$.

The timed failures of P include $(\langle d \rangle, \varnothing)$ for any time d, $(\langle d_1, a, d_2 \rangle, \{a\})$ for any times d_1, d_2. Those of Q include $(\langle 1 \rangle, \{a\})$ and $(\langle d_1, a, d_2 \rangle, \{a\})$ for any d_2 and any $d_1 \geq 2$. Thus $(\langle 1 \rangle, \{a\}) \in \mathcal{F}_t(Q)$ but this is not a failure of P. For the converse since we do not have timed trace refinement we cannot have timed failure refinement, for example, $(\langle 0.5, a \rangle, \varnothing) \in \mathcal{F}_t(P)$ but this is not a failure of Q. □

Since we have no timelocks, we have defined the timed refusals to be a subset of $\mathbb{P}(Act)$. We can adapt the relational embedding in the obvious way, and extracting the simulations we get those of the timed trace preorder (see Section 4.2 above) plus those due to the finalisation conditions:

Downward simulations: $\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin}$ is equivalent to

$$\forall\, i : I \bullet \forall\, AState;\ CState \bullet R \wedge \text{pre}\, AOp_i \Rightarrow \text{pre}\, COp_i$$

Upward simulations: $\mathsf{CFin} \subseteq \mathsf{T} \fatsemi \mathsf{AFin}$ is equivalent to

$$\forall\, CState \bullet \exists\, AState \bullet \forall\, i : I \bullet\ T \wedge (\text{pre}\, AOp_i \Rightarrow \text{pre}\, COp_i)$$

However, note that the quantification over the state includes quantification over t, the variable representing time.

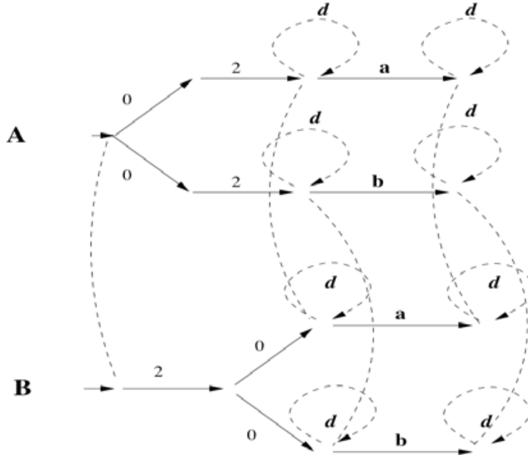


Fig. 4. A retrieve relation verifying a timed failure refinement

Example 6. Figure 4 represents (the principle parts of) a retrieve relation between the states of two systems. With this retrieve relation it is easy to see that B is a timed-failure downward simulation of A. □

5 Discussion

In this paper we have discussed how time might be modelled relationally, and thus derived simulations for relational embeddings of a number of refinement preorders found in timed process algebras. Following the basic methodology defined above one can define further embeddings and preorders. For example, we can define a timed failure trace preorder which considers refusal sets not only at the end of a timed trace, but also between each action in a timed trace, and adapting the appropriate definitions is straightforward.

The timed failure preorder defined above determines the refusals at the *end* of a trace, in a fashion similar to the definition of refusal sets for an untimed automata or the failures preorder in CSP. This is in contrast to a number of failures models given for timed CSP, where refusals are determined throughout the trace rather than simply at the end. Thus these models are closer to a timed failure trace semantics as opposed to a timed failure semantics. The need to do this arises largely due to the treatment of internal events and, specifically, their urgency due to maximal progress under hiding.

There are a number of variants of these models, perhaps reflecting the fact that the presence of time has some subtle interactions with the underlying process algebra. They include the infinite timed failures model discussed in [23,15], which as the name suggests includes infinite traces in its semantics, as well as the timed failures-stability model of Reed and Roscoe [20]. A number of different models are developed in [18,19,20], and a hierarchy described in [17].

The common aspect of these models is that refusals are recorded *throughout* the trace rather than just at the end. Thus, for example, considering P, Q and R defined by:

$$P = a \rightarrow \mathit{stop} \quad Q = b \rightarrow \mathit{stop} \quad R = c \rightarrow \mathit{stop}$$

In untimed CSP we have

$$P \sqcap (Q \Box R) = (P \sqcap Q) \Box (P \sqcap R)$$

Furthermore, in a timed setting using the timed failure preorder defined in Section 4.2 this equivalence still holds. For example, the timed failures of both sides of this equivalence include $(\langle 1, b \rangle, \{a, b, c\})$, reflecting an execution where 1 time unit passes followed by the occurrence of b, and afterwards all events are refused. However, if refusals are recorded throughout the trace, then these for the RHS include $(\langle 1, b \rangle, [0, 1) \times \{c\})$, whereas those of the process on the LHS do not - this refusal representing an execution where c was refused over the time interval $[0, 1)$ and b occurred at time 1.

Indeed, using the timed failure preorder defined in Section 4.2, the above law of untimed CSP still remains valid, but this does not hold in any of the

timed failures-stability models nor the infinite timed failures model [23,15]. The infinite timed failures model of Schneider et al [23,15] differs in two respects from the model in Section 4.2, namely the inclusion of infinite traces and the refusal information throughout the trace. The inclusion of infinite traces means that a better treatment can be given to divergence. Specifically, in a timed analysis a more precise treatment of an infinite sequence of internal events can be given, since an such an infinite sequence of internal events can now be classified as either: a timed divergence if they all occur at one instant in time; a Zeno divergence if they approach some finite time; or as a well-timed sequence if they take for ever for the whole sequence to occur [23]. Further, in timed CSP the first two possibilities are excluded from their notion of a well-timed process, thus nothing in timed CSP requires a treatment of divergence in the way that is necessary in untimed CSP.

The understanding of a failure (tr, X) in the infinite timed failures model is that after the trace *tr* the execution will eventually reach a point after which all events can be refused for the remainder of the execution. Even without consideration of refusals throughout the trace, this differs slightly from the model in Section 4.2, because there our traces observed the passing of time, and we do not need to say that the execution will *eventually* reach a point ..., since the quantification of 'eventually' is part of our timed traces. This model thus embeds the notion of constancy of offers since it requires that it will be possible to associate a timed refusal set of the form $[t, \infty) \times X$ with the execution. This restriction is not present in the model of Section 4.2 allowing time to change the events on offer.

It is argued in [23] that the inclusion of infinite traces in a timed failures model provides a more uniform treatment of unstable processes in the context of relating untimed and timed models of CSP, and indeed it allows a refinement theory (called *timewise refinement* [21,22]) to formally link the two models.

The alternative approaches use stability values, which denote the time by which any internal activity (including time) following the end of a trace must have ceased. Thus, the timed failures-stability model of Reed and Roscoe consists of triples (tr, X, α), where α is the stability. The use of stability can be seen in the following three processes:

$$P = a \rightarrow stop \quad Q = b \xrightarrow{3} stop \quad R = c \rightarrow loop \quad \text{where } loop = wait 1;\ loop$$

Here $\xrightarrow{3}$ denotes a delay of 3 time units before control passing to the subsequent process, and *wait* 1 delays 1 time unit before terminating successfully. Then (see [23]) these three processes have identical timed failures but can be distinguished by different stability values. For example, P has $(\langle(1, a)\rangle, [0, 3) \times \{b\}, 1)$ as a behaviour, whereas Q has $(\langle(1, a)\rangle, [0, 3) \times \{b\}, 4)$ and R $(\langle(1, a)\rangle, [0, 3) \times \{b\}, \infty)$. Further details of stability values and how they are calculated are given in [20].

The need for refusals to be recorded during a trace in the above models arises from the urgency of internal events upon hiding in timed CSP. That arises ultimately from the consideration of the system/environment interface and the subsequent notion of *maximal progress*: that events occur when all participants

are willing to engage in it. In the case of internal events, since the environment does not participate in them, they must occur when they become available. Now when an internal event occurs due to hiding, for example, as in the CSP process $P = (a \rightarrow stop) \setminus \{a\}$, the maximal progress condition implies such an internal event is *urgent*, that is, it occurs at the instance it becomes enabled.

One consequence of this urgency is that it forces negative premises in the transition rules for the hiding operator in timed CSP, specifically, time can only pass in $P \setminus A$ if no event from A is enabled in P.

This urgency also means we need more refusal information in order to maintain a *compositional semantics*. Reed and Roscoe give a number of examples of why hiding, urgency and a compositional semantics needs refusal information throughout the trace. One such example is the process $((a \rightarrow stop) \sqcap (wait\ 1;\ b \rightarrow stop)) \setminus \{a\}$. The urgency of the internal event upon hiding a means the non-deterministic choice is resolved in favour of $a \rightarrow stop$ and thus b cannot occur in a trace of this process. However, $\langle (1, b) \rangle$ is a trace of the process before hiding, meaning that more information is needed to regain compositionality. Further examples and discussion are given in [20].

However, in the context in which we began this discussion, namely a timed automata model, these issues are less relevant. Specifically, although a timed automata model can have internal events, a hiding operator is not defined, and therefore internal events do not have to be urgent. Furthermore, not all process algebraic models include urgency or maximal progress (see discussion in [16]). Additionally, one could argue that the urgency of internal events upon hiding is not entirely consistent with the informal semantics of untimed CSP. In particular, in untimed CSP the emphasis is on hidden events eventually occurring instantly: "a process exercises complete control over its internal events. ... [they] should not be delayed indefinitely once they are enabled" [23].

References

1. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra. Elsevier Science Inc., New York (2001)
2. Boiten, E., Derrick, J.: Modelling divergence in relational concurrent refinement. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 183–199. Springer, Heidelberg (2009)
3. Boiten, E.A., Derrick, J., Schellhorn, G.: Relational concurrent refinement II: Internal operations and outputs. Formal Aspects of Computing 21(1-2), 65–102 (2009)
4. de Roever, W.-P., Engelhardt, K.: Refinement: Model-Oriented Proof Methods and their Comparison, CUP (1998)
5. Derrick, J., Boiten, E.A.: Refinement in Z and Object-Z. Springer, Heidelberg (2001)
6. Derrick, J., Boiten, E.A.: Relational concurrent refinement. Formal Aspects of Computing 15(1), 182–214 (2003)
7. Derrick, J., Boiten, E.A.: More relational refinement: traces and partial relations. In: Electronic Notes in Theoretical Computer Science.Proceedings of REFINE 2008, Turku, vol. 214, pp. 255–276 (2008)

8. Derrick, J., Boiten, E.A.: Relational concurrent refinement: Automata. In: Electronic Notes in Theoretical Computer Science. Proceedings of the 14th BCS-FACS Refinement Workshop (REFINE 2009), December 2009, vol. 259, pp. 21–34 (2009); Extended version: Relational Concurrent Refinement III, submitted to Formal Aspects of Computing
9. Jifeng, H., Hoare, C.A.R.: Prespecification and data refinement. In: Data Refinement in a Categorical Setting. Technical Monograph, number PRG-90, Oxford University Computing Laboratory (November 1990)
10. Jifeng, H., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986)
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
12. Lynch, N., Vaandrager, F.: Forward and backward simulations I.: untimed systems. Information and Computation 121(2), 214–233 (1995)
13. Lynch, N., Vaandrager, F.: Forward and backward simulations Part II: timing-based systems. Information and Computation 128(1), 1–25 (1996)
14. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
15. Mislove, M.W., Roscoe, A.W., Schneider, S.A.: Fixed points without completeness. Theoretical Computer Science 138(2), 273–314 (1995)
16. Nicollin, X., Sifakis, J.: An overview and synthesis on timed process algebras. In: Proceedings of the Real-Time: Theory in Practice, REX Workshop, pp. 526–548. Springer, London (1992)
17. Reed, G.M.: A uniform mathematical theory for real-time distributed computing. PhD thesis (1988)
18. Reed, G.M., Roscoe, A.W.: A timed model for communicating sequential processes. In: Kott, L. (ed.) ICALP 1986. LNCS, vol. 226, pp. 314–323. Springer, Heidelberg (1986)
19. Reed, G.M., Roscoe, A.W.: A timed model for communicating sequential processes. Theor. Comput. Sci. 58(1-3), 249–261 (1988)
20. Reed, G.M., Roscoe, A.W.: The timed failures-stability model for CSP. Theoretical Computer Science 211(1-2), 85–127 (1999)
21. Schneider, S.: Timewise refinement for communicating processes. In: Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics, pp. 177–214. Springer, London (1994)
22. Schneider, S.: Timewise refinement for communicating processes. Sci. Comput. Program. 28(1), 43–90 (1997)
23. Schneider, S.: Concurrent and Real Time Systems: The CSP Approach. John Wiley & Sons, Inc., New York (1999)
24. van Glabbeek, R.J.: The linear time - branching time spectrum I. The semantics of concrete sequential processes. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 3–99. North-Holland, Amsterdam (2001)
25. van Glabbeek, R.J.: The linear time – branching time spectrum II; the semantics of sequential systems with silent moves (extended abstract)
26. Woodcock, J.C.P., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)
27. Yuan, X., Chen, J., Zheng, G.: Duration calculus in COOZ. SIGSOFT Softw. Eng. Notes 23 (May 1998)

Galois Connections for Flow Algebras

Piotr Filipiuk, Michał Terepeta, Hanne Riis Nielson, and Flemming Nielson

Technical University of Denmark
{pifi,mtte,riis,nielson}@imm.dtu.dk

Abstract. We generalise *Galois connections* from complete lattices to flow algebras. *Flow algebras* are algebraic structures that are less restrictive than idempotent semirings in that they replace distributivity with monotonicity and dispense with the annihilation property; therefore they are closer to the approach taken by Monotone Frameworks and other classical analyses. We present a generic framework for static analysis based on flow algebras and program graphs. Program graphs are often used in Model Checking to model concurrent and distributed systems. The framework allows to *induce* new flow algebras using Galois connections such that correctness of the analyses is preserved. The approach is illustrated for a *mutual exclusion* algorithm.

1 Introduction

In the classical approach to static analysis we usually use the notion of Monotone Frameworks [9,1] that work over flow graphs as an abstract representation of a program. A Monotone Framework, primarily used for data-flow analysis [10], consists of a complete lattice describing the properties of the system (without infinite ascending chains) and transfer functions over that lattice (that are monotone). When working with complete lattices, one can take advantage of Galois connections to induce new analysis or over-approximate them [6]. Recall that a Galois connection is a correspondence between two complete lattices that consists of an abstraction and concretisation functions. It is often used to move an analysis from a computationally expensive lattice to a less costly one and plays a crucial role in abstract interpretation [5].

In this paper we introduce a similar framework that uses flow algebras to define analyses. Flow algebras are algebraic structures consisting of two monoids [15] quite similar to idempotent semirings [7], which have already been used in software analysis [3,14]. However, flow algebras are less restrictive and allow to directly express some of the classical analysis, which is simply not possible with idempotent semirings. Furthermore as representation of the system under consideration we use *program graphs*, in which actions label the edges rather than the nodes. The main benefit of using program graphs is that we can model concurrent systems in a straightforward manner. Moreover since a model of a concurrent system is also a program graph, all the results are applicable both in the sequential as well as in the concurrent setting.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 138–152, 2011.
© IFIP International Federation for Information Processing 2011

We also define both the Meet Over all Paths (*MOP*) solution of the analysis as well as a set of constraints that can be used to obtain the Maximal Fixed Point (*MFP*) solution. By establishing that the solutions of the constraints constitute a Moore family, we know that there always is a least (i.e. best) solution. Intuitively the main difference between *MOP* and *MFP* is that the former expresses what we would like to compute, whereas the latter is sometimes less accurate but computable in some cases where *MOP* is not. Finally we establish that they coincide in case of distributive analyses.

We also extend the notion of Galois connections to flow algebras and program graphs. This allows us to easily create new analyses based on existing ones. In particular we can create Galois connections between the collecting semantics (defined in terms of our framework) and various analyses, which ensures their semantic correctness.

Finally we apply our results to a variant of the Bakery mutual exclusion algorithm [11]. By inducing an analysis from the collecting semantics and a Galois insertion, we are able to prove the correctness of the algorithm. Thanks to our previous developments we know that the analysis is semantically correct.

The structure of the paper is as follows. In Section 2 we introduce the flow algebras and then we show how Galois connections are defined for them in Section 3. We perform a similar development for program graphs by defining them in Section 4, presenting how to express analysis using them in Section 5 and then describing Galois connections for program graphs in Section 6. Finally we present a motivating example of our approach in Section 7 and conclude in Section 8.

2 Flow Algebra

In this section we introduce the notion of a flow algebra. As already mentioned, it is an algebraic structure that is less restrictive than an idempotent semiring. Recall that an idempotent semiring is a structure of the form $(S, \oplus, \otimes, 0, 1)$, such that $(S, \oplus, 0)$ is an idempotent commutative monoid, $(S, \otimes, 1)$ is a monoid, where $\otimes$ distributes over $\oplus$ and 0 is an annihilator with respect to multiplication. In contrast to idempotent semirings flow algebras do not require the distributivity and annihilation properties. Instead we replace the first one with a monotonicity requirement and dispense with the second one. A flow algebra is formally defined as follows.

Definition 1. *A flow algebra is a structure of the form* $(F, \oplus, \otimes, 0, 1)$ *such that:*

- $(F, \oplus, 0)$ *is an idempotent commutative monoid:*
  - $(f_1 \oplus f_2) \oplus f_3 = f_1 \oplus (f_2 \oplus f_3)$
  - $0 \oplus f = f \oplus 0 = f$
  - $f_1 \oplus f_2 = f_2 \oplus f_1$
  - $f \oplus f = f$
- $(F, \otimes, 1)$ *is a monoid:*
  - $(f_1 \otimes f_2) \otimes f_3 = f_1 \otimes (f_2 \otimes f_3)$
  - $1 \otimes f = f \otimes 1 = f$

– *$\otimes$ is monotonic in both arguments:*
  - $f_1 \leq f_2 \;\Rightarrow\; f_1 \otimes f \leq f_2 \otimes f$
  - $f_1 \leq f_2 \;\Rightarrow\; f \otimes f_1 \leq f \otimes f_2$

where $f_1 \leq f_2$ if and only if $f_1 \oplus f_2 = f_2$.

In a flow algebra all finite subsets $\{f_1, \cdots, f_n\}$ have a least upper bound; it is given by $0 \oplus f_1 \oplus \cdots \oplus f_n$.

Definition 2. *A distributive flow algebra is a flow algebra $(F, \oplus, \otimes, 0, 1)$, where $\otimes$ distributes over $\oplus$ on both sides, i.e.*

$$f_1 \otimes (f_2 \oplus f_3) = (f_1 \otimes f_2) \oplus (f_1 \otimes f_3)$$
$$(f_1 \oplus f_2) \otimes f_3 = (f_1 \otimes f_3) \oplus (f_2 \otimes f_3)$$

We also say that a flow algebra is strict if $0 \otimes f = 0 = f \otimes 0$.

Fact 1. *Every idempotent semiring is a strict and distributive flow algebra.*

We consider flow algebras because they are closer to Monotone Frameworks, and other classical static analyses. Restricting our attention to semirings rather than flow algebras would mean restricting attention to strict and distributive frameworks. Note that the classical bit-vector frameworks [12] are distributive, but not strict; hence they are not directly expressible using idempotent semirings.

Definition 3. *A complete flow algebra is a flow algebra $(F, \oplus, \otimes, 0, 1)$, where F is a complete lattice; we write $\bigoplus$ for the least upper bound. It is affine [12] if for all non-empty subsets $F' \neq \emptyset$ of F*

$$f \otimes \bigoplus F' = \bigoplus \{f \otimes f' \mid f' \in F'\}$$

$$\bigoplus F' \otimes f = \bigoplus \{f' \otimes f \mid f' \in F'\}$$

Furthermore, it is completely distributive if it is affine and strict.

If the complete flow algebra satisfies the Ascending Chain Condition [12] then it is affine if and only if it is distributive. The proof is analogous to the one presented in Appendix A of [12].

Example 1. As an example let us consider a complete lattice $L \rightarrow L$ of monotone functions over the complete lattice L. Then we can easily define a flow algebra for forward analyses, by taking $(L \rightarrow L, \sqcup, ⨾, \lambda f.\bot, \lambda f.f)$ where $(f_1 ⨾ f_2)(l) = f_2(f_1(l))$ for all $l \in L$. It is easy to see that all the laws of a complete flow algebra are satisfied. If we restrict the functions in $L \rightarrow L$ to be distributive, we obtain a distributive and complete flow algebra. Note that it can be used to define forward data-flow analyses such as reaching definitions [1,12].

3 Galois Connections for Flow Algebras

Let us recall that *Galois connection* is a tuple (L, α, γ, M) such that L and M are complete lattices and α, γ are monotone functions (called abstraction and concretisation functions) that satisfy $\alpha \circ \gamma \sqsubseteq \lambda m.m$ and $\gamma \circ \alpha \sqsupseteq \lambda l.l$. A *Galois insertion* is a Galois connection such that $\alpha \circ \gamma = \lambda m.m$. In this section we will present them in the setting of flow algebras.

In order to extend the Galois connections for flow algebras, we need to define what it means for a flow algebra to be an upper-approximation of another flow algebra. In other words we need to impose certain conditions on $\otimes$ operator and 1 element of the less precise flow algebra. The requirements are presented in the following definition.

Definition 4. *For a Galois connection* (L, α, γ, M) *we say that the flow algebra* $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ *is an upper-approximation of* $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ *if*

$$\alpha(\gamma(m_1) \otimes_L \gamma(m_2)) \sqsubseteq_M m_1 \otimes_M m_2$$
$$\alpha(1_L) \sqsubseteq_M 1_M$$

If we have equalities in the above definition, then we say that the flow algebra $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ is *induced* from $(L, \oplus_L, \otimes_L, 0_L, 1_L)$.

Example 2. Assume that we have a Galois connection (L, α, γ, M) between complete lattices L and M. We can easily construct $(L \to L, \alpha', \gamma', M \to M)$ which is a Galois connection between monotone function spaces on those lattices (for more details about this construction please consult Section 4.4 of [12]), where α', γ' are defined as

$$\alpha'(f) = \alpha \circ f \circ \gamma$$
$$\gamma'(g) = \gamma \circ g \circ \alpha$$

When both $(M \to M, \oplus_M, \otimes_M, 0_M, 1_M)$ and $(L \to L, \oplus_L, \otimes_L, 0_L, 1_L)$ are forward analyses as in Example 1, we have

$$\begin{aligned}
\alpha'(\gamma'(g_1) \otimes_L \gamma'(g_2)) &= \alpha'((\gamma \circ g_1 \circ \alpha) \fatsemi (\gamma \circ g_2 \circ \alpha)) \\
&= \alpha'(\gamma \circ g_2 \circ \alpha \circ \gamma \circ g_1 \circ \alpha) \\
&\sqsubseteq \alpha'(\gamma \circ g_2 \circ g_1 \circ \alpha) \\
&= \alpha \circ \gamma \circ g_2 \circ g_1 \circ \alpha \circ \gamma \\
&\sqsubseteq g_2 \circ g_1 \\
&= g_1 \otimes_M g_2
\end{aligned}$$

$$\alpha'(1_L) = \alpha \circ \lambda l.l \circ \gamma = \alpha \circ \gamma \sqsubseteq \lambda m.m = 1_M$$

Hence a flow algebra over $M \to M$ is a upper-approximation of the flow algebra over $L \to L$. Note that in case of a Galois insertion the flow algebra over $M \to M$ is induced.

Definition 4 requires a bit of care. Given a flow algebra $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ and a Galois connection (L, α, γ, M) it is tempting to define $\otimes_M$ by $m_1 \otimes_M m_2 = \alpha(\gamma(m_1) \otimes_L \gamma(m_2))$ and 1_M by $1_M = \alpha(1_L)$. However, it is not generally the case that $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ will be a flow algebra. This motivates the following development.

Lemma 1. *Let $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ be a flow algebra, (L, α, γ, M) be a Galois insertion, define $\otimes_M$ by $m_1 \otimes_M m_2 = \alpha(\gamma(m_1) \otimes_L \gamma(m_2))$ and 1_M by $1_M = \alpha(1_L)$. If*

$$1_L \in \gamma(M) \quad \text{and} \quad \otimes_L : \gamma(M) \times \gamma(M) \rightarrow \gamma(M)$$

for all m_1, m_2 then $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ is a flow algebra (where $\oplus_M$ is $\sqcup_M$ and 0_M is $\perp_M$).

Proof. *We need to ensure that $\otimes_M$ is associative*

$$\begin{aligned}
(m_1 \otimes_M m_2) \otimes_M m_3 &= \alpha(\gamma(\alpha(\gamma(m_1) \otimes_L \gamma(m_2))) \otimes_L \gamma(m_3)) \\
&= \alpha(\gamma(\alpha(\gamma(m'))) \otimes_L \gamma(m_3)) \\
&= \alpha(\gamma(m_1) \otimes_L \gamma(m_2) \otimes_L \gamma(m_3)) \\
m_1 \otimes_M (m_2 \otimes_M m_3) &= \alpha(\gamma(m_1) \otimes_L \gamma(\alpha(\gamma(m_2) \otimes_L \gamma(m_3)))) \\
&= \alpha(\gamma(m_1) \otimes_L \gamma(\alpha(\gamma(m')))) \\
&= \alpha(\gamma(m_1) \otimes_L \gamma(m_2) \otimes_L \gamma(m_3))
\end{aligned}$$

and similarly we need to show that 1_M is a neutral element for $\otimes_M$

$$\begin{aligned}
1_M \otimes m &= \alpha(1_L) \otimes_M m \\
&= \alpha(\gamma(\alpha(1_L)) \otimes_L \gamma(m)) \\
&= \alpha(\gamma(\alpha(\gamma(m'))) \otimes_L \gamma(m)) \\
&= \alpha((\gamma(m') \otimes_L \gamma(m)) \\
&= \alpha((1_L \otimes_L \gamma(m)) \\
&= \alpha(\gamma(m)) \\
&= m
\end{aligned}$$

where $1_L = \gamma(m')$ for some m'. The remaining properties of flow algebra hold trivially. ☐

The above requirements can be expressed in a slightly different way. This is presented by the following two lemmas.

Lemma 2. *For flow algebras $(L, \oplus_L, \otimes_L, 0_L, 1_L)$, $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ and a Galois insertion (L, α, γ, M), the following are equivalent:*

1. $1_L = \gamma(1_M)$
2. $\alpha(1_L) = 1_M$ *and* $1_L \in \gamma(M)$

Lemma 3. *For flow algebras $(L, \oplus_L, \otimes_L, 0_L, 1_L)$, $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ and a Galois insertion (L, α, γ, M), the following are equivalent:*

1. $\forall m_1, m_2 : \gamma(m_1) \otimes_L \gamma(m_2) = \gamma(m_1 \otimes_M m_2)$
2. $\forall m_1, m_2 : \alpha(\gamma(m_1) \otimes_L \gamma(m_2)) = m_1 \otimes_M m_2$ *and* $\otimes_L : \gamma(M) \times \gamma(M) \rightarrow \gamma(M)$

4 Program Graphs

This section introduces program graphs, a representation of software (hardware) systems that is often used in model checking [2] to model concurrent and distributed systems. Compared to the classical flow graphs [10,12], the main difference is that in the program graphs the actions label the edges rather than the nodes.

Definition 5. *A program graph over a space S has the form*

$$(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, S)$$

where

- Q *is a finite set of states;*
- Σ *is a finite set of actions;*
- $\rightarrow \subseteq \mathsf{Q} \times \Sigma \times \mathsf{Q}$ *is a transition relation;*
- $\mathsf{Q}_I \subseteq \mathsf{Q}$ *is a set of initial states;*
- $\mathsf{Q}_F \subseteq \mathsf{Q}$ *is a set of final states; and*
- $\mathcal{A} : \Sigma \rightarrow S$ *specifies the meaning of the actions.*

A concrete program graph is a program graph where $S = \mathbf{Dom} \hookrightarrow \mathbf{Dom}$, where **Dom** *is the set of all configurations of a program, and $\mathcal{A} = \mathcal{T}$ where $\mathcal{T}$ is the semantic function. An abstract program graph is a program graph where S is a complete flow algebra.*

Now we can define the collecting semantics [5,12] of a concrete program graph in terms of a flow algebra. This can be used to establish the semantic correctness of an analysis by defining a Galois connection between the collecting semantics and the analysis.

Definition 6. *We define the collecting semantics of a program graph using the flow algebra $(\mathcal{P}(\mathbf{Dom}) \rightarrow \mathcal{P}(\mathbf{Dom}), \cup, \fatsemi, \lambda.\emptyset, \lambda d.d)$, by*

$$\mathcal{A}[\![a]\!](S) = \{\mathcal{T}[\![a]\!](s) \mid s \in S \wedge \mathcal{T}[\![a]\!](s) \textit{ is defined}\}$$

where **Dom** *is the set of all configurations of a program and $\mathcal{T}$ is the semantic function.*

Now let us consider a number of processes each specified as a program graph $PG_i = (\mathsf{Q}_i, \Sigma_i, \rightarrow_i, \mathsf{Q}_{I\,i}, \mathsf{Q}_{F\,i}, \mathcal{A}_i, S)$ that are executed independently of one another except that they can exchange information via shared variables. The combined program graph $PG = PG_1 \mid\mid\mid \cdots \mid\mid\mid PG_n$ expresses the interleaving between n processes.

Definition 7. *The interleaved program graph over S*

$$\mathsf{PG} = \mathsf{PG}_1 \mid\mid\mid \cdots \mid\mid\mid \mathsf{PG}_n$$

is defined by $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, S)$ *where*

- $\mathsf{Q} = \mathsf{Q}_1 \times \cdots \times \mathsf{Q}_n$*,*
- $\Sigma = \Sigma_1 \uplus \cdots \uplus \Sigma_n$ *(disjoint union),*
- $\langle q_1, \cdots, q_i, \cdots, q_n \rangle \xrightarrow{a} \langle q_1, \cdots, q_i', \cdots, q_n \rangle$ *if* $q_i \xrightarrow{a}_i q_i'$*,*
- $\mathsf{Q}_I = \mathsf{Q}_{I1} \times \cdots \times \mathsf{Q}_{In}$*,*
- $\mathsf{Q}_F = \mathsf{Q}_{F1} \times \cdots \times \mathsf{Q}_{Fn}$*, and*
- $\mathcal{A}[\![a]\!] = \mathcal{A}_i[\![a]\!]$ *if* $a \in \Sigma_i$*.*

Note that $\mathcal{A}_i : \Sigma_i \rightarrow S$ *for all* i *and hence* $\mathcal{A} : \Sigma \rightarrow S$*.*

Analogously to the previous definition, we say that a concrete interleaved program graph is an interleaved program graph where $S = \mathbf{Dom} \hookrightarrow \mathbf{Dom}$*, and* $\mathcal{A} = \mathcal{T}$ *where* $\mathcal{T}$ *is the semantic function. An abstract interleaved program graph is an interleaved program graph where* S *is a complete flow algebra.*

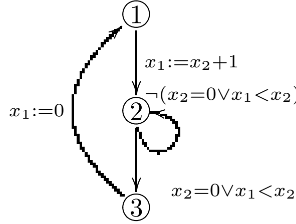
The application of this definition is presented in the example below, where we model the Bakery mutual exclusion algorithm. Note that the ability to create interleaved program graphs allows us to model concurrent systems using the same methods as in the case of sequential programs. This will be used to analyse and verify the algorithm in Section 7.

Example 3. As an example we consider a variant of the Bakery algorithm for two processes. Let P_1 and P_2 be the two processes, and x_1 and x_2 be two shared variables both initialised to 0. The algorithm is as follows

```
do true ->                              ||| do true ->
    x1 := x2 + 1;                       |||     x2 := x1 + 1;
    do ¬((x2 = 0) ∨ (x1 < x2)) ->       |||     do ¬((x1 = 0) ∨ (x2 < x1)) ->
        skip                            |||         skip
    od;                                 |||     od;
    critical section                    |||     critical section
    x1 := 0                             |||     x2 := 0
od                                      ||| od
```

The variables x_1 and x_2 are used to resolve the conflict when both processes want to enter the critical section. When x_i is equal to zero, the process P_i is not in the critical section and does not attempt to enter it — the other one can safely proceed to the critical section. Otherwise, if both shared variables are non-zero, the process with smaller "ticket" (i.e. value of the corresponding variable) can enter the critical section. This reasoning is captured by the conditions of busy-waiting loops. When a process wants to enter the critical section, it simply takes the next "ticket" hence giving priority to the other process.

The program graph corresponding to the first process is quite simple and is presented below (the program graph for the second process is analogous).



Now we can use the Definition 7 to obtain the interleaving of the two processes, which is depicted in Figure 1. Since the result is also a program graph, it can be analysed in our framework.

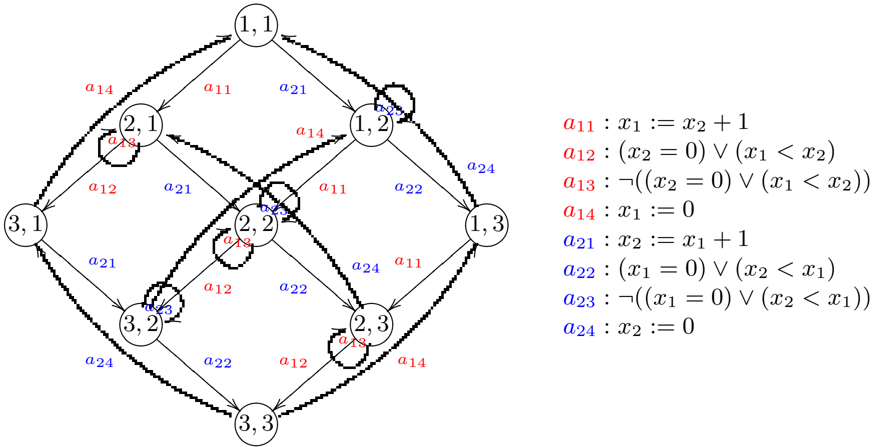


Fig. 1. Interleaved program graph

5 Flow Algebras over Program Graphs

Having defined flow algebras and program graphs, it remains to show how to obtain the analysis results. We shall consider two approaches, namely *MOP* and *MFP*. As already mentioned, these stand for Meet Over all Paths and Maximal Fixed Point, respectively. However, since we take a *join* (least upper bound) to merge information from different paths, in our setting these really mean join over all paths and least fixed point. However, we use the *MOP* and *MFP* acronyms for historical reasons.

We consider the *MOP* solution first, since it is more precise and captures what we would ideally want to compute.

Definition 8. *Given an abstract program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ *over a complete flow algebra* $(F, \oplus, \otimes, 0, 1)$*, and two sets* $\mathsf{Q}_\circ \subseteq \mathsf{Q}$ *and* $\mathsf{Q}_\bullet \subseteq \mathsf{Q}$ *we are interested in*

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \bigoplus_{\pi \in \mathsf{Path}(\mathsf{Q}_\circ, \mathsf{Q}_\bullet)} \mathcal{A}[\![\pi]\!]$$

where

$$\mathsf{Path}(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \{a_1 a_2 \cdots a_k \mid \exists q_0, q_1, \cdots q_k : q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} q_k, q_0 \in \mathsf{Q}_\circ, q_k \in \mathsf{Q}_\bullet\}$$

and

$$\mathcal{A}[\![a_1 a_2 \cdots a_k]\!] = 1 \otimes \mathcal{A}[\![a_1]\!] \otimes \mathcal{A}[\![a_2]\!] \otimes \cdots \otimes \mathcal{A}[\![a_k]\!]$$

Since the *MOP* solution is not always computable (e.g. for Constant Propagation), one usually uses the *MFP* one which only requires that the lattice satisfies the Ascending Chain Condition and is defined as the least solution to a set of constraints. Let us first introduce those constraints.

Definition 9. *Consider an abstract program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ *over a complete flow algebra* $(F, \oplus, \otimes, 0, 1)$. *This gives rise to a set* $Analysis_F$ *of constraints*

$$An_F^{\mathsf{Q}_\circ}(q) \sqsupseteq \begin{cases} \bigoplus\{An_F^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus 1_F & , \text{if } q \in \mathsf{Q}_\circ \\ \bigoplus\{An_F^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} & , \text{if } q \notin \mathsf{Q}_\circ \end{cases}$$

where $q \in \mathsf{Q}$, $\mathsf{Q}_\circ \subseteq \mathsf{Q}$.

We write $An_F^{\mathsf{Q}_\circ} \models Analysis_F$ whenever $An_F^{\mathsf{Q}_\circ} : \mathsf{Q} \rightarrow F$ is a solution to the constraints $Analysis_F$. Now we establish that there is always a least (i.e. best) solution of those constraints.

Lemma 4. *The set of solutions to the constraint system from Definition 9 is a Moore family (i.e. it is closed under* $\bigsqcap$*), which implies the existence of the least solution.*

Definition 10. *We define MFP to be the least solution to the constraint system from Definition 9.*

The following result states the general relationship between *MOP* and *MFP* solutions and shows in which cases the they coincide.

Proposition 1. *Consider the MOP and MFP solutions for an abstract program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ *defined over a complete flow algebra* $(F, \oplus, \otimes, 0, 1)$, *then*

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) \sqsubseteq \bigoplus_{q \in \mathsf{Q}_\bullet} MFP_F^{\mathsf{Q}_\circ}(q)$$

If the flow algebra is affine and either $\forall q \in \mathsf{Q} : \mathsf{Path}(\mathsf{Q}_\circ, \{q\}) \neq \emptyset$ *or the flow algebra is strict then*

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \bigoplus_{q \in \mathsf{Q}_\bullet} MFP_F^{\mathsf{Q}_\circ}(q)$$

This is consistent with the previous results, e.g. for Monotone Frameworks, where the *MOP* and *MFP* coincide in case of distributive frameworks and otherwise *MFP* is a safe approximation of *MOP* [9].

6 Galois Connections for Program Graphs

In the current section we show how the generalisation of Galois connections to flow algebras can be used to upper-approximate solutions of the analyses. Namely, consider a flow algebra $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ and a Galois connection (L, α, γ, M). Moreover, let $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ be a flow algebra that is an upper-approximation of the flow algebra over L. We show that whenever we have a solution for an analysis in M then, when concretised, it is an upper-approximation of the solution of an analysis in L. First we state necessary requirements for the analyses. Then we present the results for the *MOP* and *MFP* solutions.

6.1 Upper-Approximation of Program Graphs

Since analyses using abstract program graphs are defined in terms of functions specifying effects of different actions, we need to impose conditions on these functions.

Definition 11. *Consider a flow algebra* $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ *that is an upper-approximation of* $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ *by a Galois connection* (L, α, γ, M)*. A program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ *is an upper-approximation of another program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *if*

$$\forall a \in \Sigma : \alpha(\mathcal{A}[\![a]\!]) \sqsubseteq_M \mathcal{B}[\![a]\!]$$

It is quite easy to see that this upper-approximation for action implies one for paths.

Lemma 5. *If a program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ *is an upper-approximation of* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *by* (L, α, γ, M)*, then for every path* π *we have that*

$$\mathcal{A}[\![\pi]\!] \sqsubseteq_L \gamma(\mathcal{B}[\![\pi]\!])$$

Consider a flow algebra $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ induced from $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ by a Galois connection (L, α, γ, M). As in case of flow algebras, we say that a program graph $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ is induced from $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ if we change the inequality from Lemma 5 to

$$\forall a \in \Sigma : \alpha(\mathcal{A}[\![a]\!]) = \mathcal{B}[\![a]\!]$$

6.2 Preservation of the *MOP* and *MFP* Solutions

Now we will investigate what is the relationship between the solutions of an analysis in case of original program graph and its upper-approximation. Again we will first consider the *MOP* solution and then *MFP* one.

MOP. We want to show that if we calculate the *MOP* solution of the analysis $\mathcal{B}$ in M and concretise it (using γ), then we will get an upper-approximation of the *MOP* solution of $\mathcal{A}$ in L.

Lemma 6. *If a program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, M)$ *is an upper-approximation of* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *by* (L, α, γ, M) *then*

$$MOP_L(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) \sqsubseteq \gamma(MOP_M(\mathsf{Q}_\circ, \mathsf{Q}_\bullet))$$

Proof. The result follows from Lemma 5. □

MFP. Let us now consider the *MFP* solution. We would like to prove that whenever we have a solution An_M of the constraint system $Analysis_M$ then, when concretised, it also is a solution to the constraint system $Analysis_L$. This is established by the following lemma.

Lemma 7. *If a program graph given by* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, M)$ *is an upper-approximation of* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *by* (L, α, γ, M) *then*

$$An_M^{\mathsf{Q}_\circ} \models Analysis_M \implies \gamma \circ An_M^{\mathsf{Q}_\circ} \models Analysis_L$$

and in particular

$$MFP_L^{\mathsf{Q}_\circ} \sqsubseteq \gamma \circ MFP_M^{\mathsf{Q}_\circ}$$

Proof. We consider only the case where $q \in \mathsf{Q}_\circ$ *(the other case is analogous). From the assumption we have*

$$\begin{aligned}\gamma \circ An_M^{\mathsf{Q}_\circ} &\sqsupseteq \lambda q.\gamma(\bigoplus\{An_M^{\mathsf{Q}_\circ}(q') \otimes \mathcal{B}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus 1_M)\\ &\sqsupseteq \lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q') \otimes \mathcal{B}[\![a]\!]) \mid q' \xrightarrow{a} q\} \oplus \gamma(1_M)\end{aligned}$$

Now using the definition of upper-approximation of a flow algebra it follows that

$$\begin{aligned}&\lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q') \otimes \mathcal{B}[\![a]\!]) \mid q' \xrightarrow{a} q\} \oplus \gamma(1_M)\\ &\sqsupseteq \lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q')) \otimes \gamma(\mathcal{B}[\![a]\!]) \mid q' \xrightarrow{a} q\} \oplus 1_L\\ &\sqsupseteq \lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q')) \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus 1_L\end{aligned}$$

We also know that every solution $An_L^{\mathsf{Q}_\circ}$ *to the constraints* $Analysis_L$ *must satisfy*

$$An_L^{\mathsf{Q}_\circ} \sqsupseteq \lambda q.\bigoplus\{An_L^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus 1_L$$

and it is clear that $\gamma \circ An_M^{\mathsf{Q}_\circ}$ *is also a solution these constraints.* □

7 Application to the Bakery Algorithm

In this section we use flow algebras and Galois insertions to verify the correctness of the Bakery mutual exclusion algorithm. Although the Bakery algorithm is designed for an arbitrary number of processes, we consider the simpler setting with two processes, as in Example 3. For reader's convenience we recall the pseudo-code of the algorithm:

```
do true ->                                ||| do true ->
    x1 := x2 + 1;                          |||     x2 := x1 + 1;
    do ¬((x2 = 0) ∨ (x1 < x2)) ->          |||     do ¬((x1 = 0) ∨ (x2 < x1)) ->
        skip                               |||         skip
    od;                                    |||     od;
    critical section                       |||     critical section
    x1 := 0                                |||     x2 := 0
od                                         ||| od
```

We want to verify that the algorithm ensures mutual exclusion, which is equivalent to checking whether the state $(3, 3)$ (corresponding to both processes being in the critical section at the same time) is unreachable in the interleaved program graph. First we define the collecting semantics, which tells us the potential values of the variables x_1 and x_2. Since they can never be negative, we take the complete lattice to be the monotone function space over $\mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0})$. This gives rise to the flow algebra $\mathcal{C}$ of the form

$$(\mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}) \rightarrow \mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}), \cup, \fatsemi, \lambda ZZ.\emptyset, \lambda ZZ.ZZ)$$

The semantic function is defined as follows

$$\begin{aligned}
\mathcal{T}[\![x_1 := x_2 + 1]\!] &= \lambda ZZ.\{(z_2 + 1, z_2) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![x_2 := x_1 + 1]\!] &= \lambda ZZ.\{(z_1, z_1 + 1) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![x_1 := 0]\!] &= \lambda ZZ.\{(0, z_2) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![x_2 := 0]\!] &= \lambda ZZ.\{(z_1, 0) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![e]\!] &= \lambda ZZ.\{(z_1, z_2) \mid \mathcal{E}[\![e]\!](z_1, z_2) \wedge (z_1, z_2) \in ZZ\}
\end{aligned}$$

where $\mathcal{E} : \mathit{Expr} \rightarrow (\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0} \rightarrow \{\mathit{true}, \mathit{false}\})$ is used for evaluating expressions. Unfortunately, as the values of x_1 and x_2 may grow unboundedly, the underlying transition system of the parallel composition of two processes is infinite. Hence it is not possible to naively use it to verify the algorithm.

Therefore we clearly need to introduce some abstraction. Using our approach we would like to define an analysis that is an upper-approximation of the collecting semantics. This should allow us to compute the result and at the same time guarantee that the analysis is semantically correct. The only remaining challenge is to define a domain that is precise enough to capture the property of interest and then show that the analysis is an upper-approximation of the collecting semantics.

For our purposes it is enough to record when the conditions allowing to enter the critical section (e.g. $(x_2 = 0) \vee (x_1 < x_2)$) are true or false. For that we can

use the Sign Analysis. We take the complete lattice to be the monotone function space over $\mathcal{P}(\$ \times \$ \times \$)$, where $\$ = \{-, \mathit{0}, +\}$. The three components record the signs of variables x_1, x_2 and their difference i.e. $x_1 - x_2$, respectively. We define a Galois connection $(\mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}), \alpha, \gamma, \mathcal{P}(\$ \times \$ \times \$))$ by the extraction function

$$\eta(z_1, z_2) = (sign(z_1), sign(z_2), sign(z_1 - z_2))$$

where

$$sign(z) = \begin{cases} - \text{ if } z < 0 \\ \mathit{0} \text{ if } z = 0 \\ + \text{ if } z > 0 \end{cases}$$

Then α and γ are defined by

$$\alpha(ZZ) = \{\eta(z_1, z_2) \mid (z_1, z_2) \in ZZ\}$$
$$\gamma(SSS) = \{(z_1, z_2) \mid \eta(z_1, z_2) \in SSS\}$$

for $ZZ \subseteq \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}$ and $SSS \subseteq \$ \times \$ \times \$$.

However, note that the set $\mathcal{P}(\$ \times \$ \times \$)$ contains superfluous elements, such as $(0, 0, +)$. Therefore we reduce the domain of the Sign Analysis to the subset that contains only meaningful elements. For that purpose we use the already defined extraction function η. The resulting set $\mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}])$ is defined using

$$\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}] = \{\eta(z_1, z_2) \mid (z_1, z_2) \in \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}\}$$

It is easy to see that

$$\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}] = \left\{ \begin{array}{l} (\mathit{0}, \mathit{0}, \mathit{0}), (\mathit{0}, +, -), (+, \mathit{0}, +), \\ (+, +, \mathit{0}), (+, +, +), (+, +, -) \end{array} \right\}$$

This gives rise to a Galois insertion (recall Example 2)

$$(\mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}) \to \mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}), \alpha', \gamma', \mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}]) \to \mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}]))$$

where:

$$\alpha'(f) = \alpha \circ f \circ \gamma$$
$$\gamma'(g) = \gamma \circ g \circ \alpha$$

We next consider the flow algebra $\mathcal{S}$ given by

$$(\mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}]) \to \mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}]), \cup, \fatsemi, \lambda SSS.\emptyset, \lambda SSS.SSS)$$

and note that it is induced from the flow algebra $\mathcal{C}$ by the Galois insertion (for details refer to Example 2). Now we can induce transfer functions for the Sign Analysis. As an example let us consider the case of $x_2 := 0$ and calculate

$$\begin{aligned} \mathcal{A}[\![x_2 := 0]\!](SSS) &= \alpha(\mathcal{T}[\![x_2 := 0]\!](\gamma(SSS))) \\ &= \alpha(\mathcal{T}[\![x_2 := 0]\!](\{(z_1, z_2) \mid \eta(z_1, z_2) \in SSS\})) \\ &= \alpha(\{(z_1, 0) \mid \eta(z_1, z_2) \in SSS\}) \\ &= \{(s_1, 0, s_1) \mid (s_1, s_2, s) \in SSS\} \end{aligned}$$

Other transfer functions are induced in a similar manner and are omitted.

Clearly the program graph over flow algebra $\mathcal{S}$ is an upper-approximation of the collecting semantics. It follows that the Sign Analysis is semantically correct. Therefore we can safely use it to verify the correctness of the Bakery algorithm. For the actual calculations of the least solution for the analysis problem we use the *Succinct Solver* [13], in particular its latest version [8] that is based on Binary Decision Diagrams [4]. The analysis is expressed in $ALFP$ (i.e. the constraint language of the solver). The result obtained for the node $(3, 3)$ is the empty set, which means that the node is unreachable. Thus the mutual exclusion property is guaranteed.

8 Conclusions

In this paper we presented a general framework that uses program graphs and flow algebras to define analyses. One of our main contributions is the introduction of flow algebras, which are algebraic structures less restrictive than idempotent semirings. Their main advantage and our motivation for introducing them is the ability to directly express the classical analyses, which is clearly not possible when using idempotent semirings. Moreover the presented approach has certain advantages over Monotone Frameworks, such as the ability to handle both sequential and concurrent systems in the same manner. We also define both *MOP* and *MFP* solutions and establish that the classical result of their coincidence in case of distributive analyses carries over to our framework.

Furthermore we investigated how to use Galois connections in this setting. They are a well-known and powerful technique that is often used to "move" from a costly analysis to a less expensive one. Our contribution is the use of Galois connections to upper-approximate and induce flow algebras and program graphs. This allows inducing new analyses such that their semantic correctness is preserved.

Finally we applied our approach to a variant of the Bakery mutual exclusion algorithm. We verified its correctness by moving from a precise, but uncomputable analysis to the one that is both precise enough for our purposes and easily computable. Since the analysis is induced from the collecting semantics by a Galois insertion, we can be sure that the it is semantically correct.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools, 2nd edn., p. 1009. Pearson/Addison Wesley, Boston, MA, USA (2007)
2. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press, Cambridge (2008)
3. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. 14(4), 551 (2003)
4. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. 24(3), 293–318 (1992)

5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
7. Droste, M., Kuich, W.: Semirings and formal power series. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata. Monographs in Theoretical Computer Science. An EATCS Series, pp. 3–28. Springer, Heidelberg (2009)
8. Filipiuk, P., Nielson, H.R., Nielson, F.: Explicit versus symbolic algorithms for solving ALFP constraints. Electr. Notes Theor. Comput. Sci. 267(2), 15–28 (2010)
9. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Inf. 7, 305–317 (1977)
10. Kildall, G.A.: A unified approach to global program optimization. In: POPL, pp. 194–206 (1973)
11. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM 17(8), 453–455 (1974)
12. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus (1999)
13. Nielson, F., Seidl, H., Nielson, H.R.: A succinct solver for ALFP. Nord. J. Comput. 9(4), 335–372 (2002)
14. Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)
15. Rosen, B.K.: Monoids for rapid data flow analysis. SIAM J. Comput. 9(1), 159–196 (1980)

An Accurate Type System for Information Flow in Presence of Arrays

Séverine Fratani and Jean-Marc Talbot

Laboratoire d'Informatique Fondamentale de Marseille (LIF)
UMR6166 CNRS - Université de la Méditerranée - Université de Provence

Abstract. Secure information flow analysis aims to check that the execution of a program does not reveal information about secret data manipulated by this program. In this paper, we consider programs dealing with arrays; unlike most of existing works, we will not assume that arrays are homogeneous in terms of security levels. Some part of an array can be declared as secret whereas another part is public. Based on a pre-computed approximation of integer variables (serving as indices for arrays), we devise a type system such that typed programs do not leak unauthorized information. Soundness of our type system is proved by a non-interference theorem.

1 Introduction

Information flow analysis aims to check that data propagation within applications conforms some security requirements; the goal is to avoid that programs leak confidential information during their executions: observable/public outputs of a program must not disclose information about secret/confidential values manipulated by this program.

Data are labelled with security levels, usually H for secret/confidential/high security level values and L for observable/public/low security level variables (or more generally, by elements of a lattice of security levels [6]). The absence of illicit information flow is proved by the non-interference property: if two inputs of the program coincide on their public (observable) part then it is so for the outputs. Information flows arise from assignments (direct flow) or from the control structure of a program (implicit flow). For example, the code `l = h` generates a direct flow from `h` to `l`, while `if(h) then l=1 else l=0` generates an implicit flow. If `h` has security level H and `l` L, then the two examples are insecure and generate an illicit information flow, as confidential data can be deduced by the reader of `l`. Non-interference can be checked through typing or static analysis, leading to the automatic rejection of insecure programs. A considerable part of works on information flow control, based on static analysis, has been achieved in the last decades, *eg* [7,8,12,16,15].

Although some tools based on these works have been developed [12,14], it seems that they are not mature enough to properly address real-world problems. As advocated in [1], we believe that one of the bottlenecks is the current unability of the proposed works on information flow to address real-world programming languages, that is, languages including in particular, built-ins data structures (arrays, hash tables, ..). Another issue is the "real-world" programming style that is frequently used, in particular in the area of embedded software, to save resources (time, space, energy, ..).

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 153–167, 2011.
© IFIP International Federation for Information Processing 2011

In this paper, we focus on arrays, the most casual built-ins data structure. Two works [5,17] have already addressed this topic. Both are based on the Volpano, Smith and Irvine'approach (Volpano, Smith and Irvine proposed in their seminal paper a type system to guarantee the absence of illicit information flows [16]) and share the same feature: the security type of an array is uniform for the whole array; an array is either totally public or totally private. Unfortunately, this may be insufficient for real-world programs.

Let us consider the file given below on the left and the program next to it:

```
Doe
John
john.doe@yahoo.fr
Martin
Jean
jean.martin@mail.com
```

```
i=0
while (not(eof(f)))
  T[i] ::= read(f)
  i :=i+1
j :=0
while (j <i)
  if ((j mod 3) <> 2) then
    print(T[j])
  j :=j+1
```

The structure of the file is simple, one piece of information per line, and one after the other, a last name, a first name and an email. Suppose a policy security stating that first and last names are public but emails have to remain confidential. One may argue that this program is written by a programmer who is unware of security issues and should be rejected because of its programming style. But on the one hand, similar programs are rather frequent in embedded systems and on the other hand, this program does not disclose any confidential information and is thus correct wrt to secure information flows.

This example illustrates that considering access to arrays in fine-grained manner is crucial to obtain relevant results concerning security issues; this fact has motivated the work of Amtoft, Hatcliff and Rodríguez [1]: the authors proposed there an Hoare-style logic to reason about information flows, this logic being able to express properties about single array cells. Our approach is different and composed of two parts: first, an accurate information about the values of integer variables used as indices of arrays is considered. We will assume that these informations on integer variables are known (pre-computed or given by the programmer). Then, based on that, we define a type system such that typed programs do not leak unauthorized information. We prove the soundness of our approach by proving a non-interference theorem.

The paper is organized as follows: in Section 2, we describe the small programming language we consider within this paper: this language is a simple imperative language allowing to manipulate one-dimensional array. However, array aliases are not supported. As discussed earlier, accurate typing requires information about values taken by the integer variables during the execution of the program. These information allow us to consider arrays not only as a single piece of data but also quite sophisticated parts of them. Thus, Section 3 aims to described a framework for expressing approximation of integer variables. We also specify in which sense this framework is sound. As an example, we chose Presburger formulas as they form a quite expressive and well-known theory. Section 4 is devoted to our type system. This type systems relies on the approximation given in the previous section. In that section, we also give some examples of typed programs. Finally, in Section 5, we prove that the devised type system is correct: we show first that it satisfies the subject reduction property. As intermediate steps

to soundness, we prove the "classical" *simple security* property of expressions and the *confinement* property of commands. Finally, we prove the main result of the paper as a non-interference theorem: for inputs data that can not be distinguished on the lower level part, a well-typed program produces outputs coinciding on the lower level part.

2 Language and Semantics

The language we consider is a simple imperative language with arrays. We assume to be fixed a finite set of integer identifier $\mathcal{I} = \{x_1, \ldots, x_n\}$, a finite set of array identifiers $\mathcal{T} = \{T_1, T_2, \ldots\}$ and a finite set of labels $\mathcal{L}$.

The syntax of programs is the given by the following grammar:

$$
\begin{array}{lll}
(\text{EXPRESSIONS})\ e & ::= & x_i \mid n \mid T[e] \mid T.\texttt{length} \mid e_1 + e_2 \mid e_1 - e_2 \mid n * e \mid e/n \\
 & & e_1 \vee e_2 \mid e_1 \wedge e_2 \mid e_1 \neq e_2 \mid e_1 = e_2 \mid \ldots \\
(\text{COMMANDS})\ \ c & ::= & x_i :=^{\ell} e \mid T[e_1] :=^{\ell} e_2 \mid \texttt{allocate}^{\ell}\, T[e] \mid \texttt{skip}^{\ell} \mid \\
 & & \texttt{if}^{\ell}\, e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid \texttt{while}^{\ell}\, e\ \texttt{do}\ P \\
(\text{PROGRAMS})\ \ \ P & ::= & c \mid c; P
\end{array}
$$

Here, meta-variables x_i range over $\mathcal{I}$, n over integers, T over the set of array identifiers $\mathcal{T}$ and ℓ over the set of labels $\mathcal{L}$. The sequence operator ";" is implicitly assumed to be associative. Finally, we assume the labels appearing in a program to be pairwise distinct and we may write c^{ℓ} to emphasize the fact that the label of the command c is ℓ.

It should be noticed that we address only one dimensional arrays and do not support array aliases (expressions such as $T_1 := T_2$ are not allowed).

A program P is executed under a memory μ, which maps identifiers to values in the following way: for all x_i in $\mathcal{I}$, $\mu(x_i) \in \mathbb{Z}$ and for all $T \in \mathcal{T}$, $\mu(T)$ is an array of integers $\langle n_0, n_1, \ldots, n_{k-1}\rangle$, where $k > 0$. Additionally, we assume $\langle\rangle$, a special value for arrays. The length of $\langle\rangle$ is 0.

We assume that expressions are evaluated atomically and we denote by $\mu(e)$ the value of the expression e in the memory μ; the semantics of array, addition and division expressions is given on Fig. 1. The semantics of the others expressions is defined in an obvious way. Note that, as in C language, the result of the evaluation of an expression is always an integer. Moreover, following the lenient semantics proposed in [5], accessing an array with an out-of-bounds index yields 0, as well as division by 0. Hence, the programs we deal with are all error-free. The semantics of programs is given by a transition relation $\rightarrow$ on configurations. A configuration is either a pair (P, μ) or simply a memory μ: in the first case, P is the program yet to be executed, whereas in the second one, the command is terminated yielding the final memory μ. We write $\rightarrow^k$ for the k-fold self composition of $\rightarrow$ and $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$.

The semantics of array commands is defined in Fig. 2 (where $\mu[U := V]$ denotes a memory identical to μ except for the variable U whose value is V). Given an expression e, $\texttt{allocate}\ T[e]$ allocates a 0-initialized block of memory for the array T, the size of this array is given by the value of e (when strictly positive). The remaining executions of programs are given by a standard structural operational semantics (SOS) presented on Fig. 3. Remark that when a program is executed, the execution either terminates successfully or loops (it cannot get stuck) : for each configuration of the form (P, μ) there always exists a rule that may apply.

$$\text{(ARR-READ)}\ \frac{\mu(T) = \langle n_0, n_1, \ldots n_{k-1}\rangle,\ \mu(e) = i,\ 0 \le i < k}{\mu(T[e]) = n_i}$$

$$\frac{\mu(T) = \langle n_0, n_1, \ldots n_{k-1}\rangle,\ \mu(e) \notin [0, k-1]}{\mu(T[e]) = 0} \qquad \frac{\mu(T) = \langle\rangle}{\mu(T[e]) = 0}$$

$$\text{(GET-LGTH)}\ \frac{\mu(T) = \langle n_0, n_1, \ldots n_{k-1}\rangle}{\mu(T.\mathtt{length}) = k}$$

$$\text{(ADD)}\ \frac{\mu(e_1) = n_1,\ \mu(e_2) = n_2}{\mu(e_1 + e_2) = n_1 + n_2}$$

$$\text{(DIV)}\ \frac{\mu(e_1) = n_1, n_2 \ne 0}{\mu(e_1/n_2) = \lfloor n_1/n_2 \rfloor} \qquad \frac{\mu(e_1) = n_1,\ \mu(n_2) = 0}{\mu(e_1/n_2) = 0}$$

Fig. 1. Semantics of array, addition and division expressions

$$\text{(UPD-ARR)}\ \frac{\mu(T) = \langle n_0, \ldots n_{k-1}\rangle,\ \mu(e_1) = i \in [0, k-1],\ \mu(e_2) = n}{(T[e_1] :=^{\ell} e_2, \mu) \to \mu[T := \langle n_0, \ldots, n_{i-1}, n, n_{i+1}, \ldots n_{k-1}\rangle]}$$

$$\frac{\mu(T) = \langle n_0, \ldots n_{k-1}\rangle,\ \mu(e_1) \notin [0, k-1]}{(T[e_1] :=^{\ell} e_2, \mu) \to \mu} \qquad \frac{\mu(T) = \langle\rangle}{(T[e_1] :=^{\ell} e_2, \mu) \to \mu}$$

$$\text{(CALLOC)}\ \frac{\mu(e) > 0,\ \mu(T) = \langle\rangle}{(\mathtt{allocate}^{\ell}\ T[e], \mu) \to \mu[T := \langle \underbrace{0, 0, \ldots, 0}_{\mu(e)} \rangle]}$$

$$\frac{\mu(e) \le 0}{(\mathtt{allocate}^{\ell}\ T[e], \mu) \to \mu} \qquad \frac{\mu(T) \ne \langle\rangle}{(\mathtt{allocate}^{\ell}\ T[e], \mu) \to \mu}$$

Fig. 2. Semantics of array commands

$$\text{(UPDATE)}\ (x :=^{\ell} e, \mu) \to \mu[x := \mu(e)] \qquad \text{(NO-OP)}\ (\mathtt{skip}^{\ell}, \mu) \to \mu$$

$$\text{(BRANCH)}\ \frac{\mu(e) \ne 0}{(\mathtt{if}^{\ell}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2, \mu) \to (P_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\mathtt{if}^{\ell}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2, \mu) \to (P_2, \mu)}$$

$$\text{(LOOP)}\ \frac{\mu(e) \ne 0}{(\mathtt{while}^{\ell}\ e\ \mathtt{do}\ P, \mu) \to (P; \mathtt{while}^{\ell}\ e\ \mathtt{do}\ P, \mu)} \qquad \frac{\mu(e) = 0}{(\mathtt{while}^{\ell}\ e\ \mathtt{do}\ P, \mu) \to \mu}$$

$$\text{(SEQUENCE)}\ \frac{(c_1, \mu) \to \mu'}{(c_1; P_2, \mu) \to (P_2, \mu')} \qquad \frac{(c_1, \mu) \to (P, \mu')}{(c_1; P_2, \mu) \to (P; P_2, \mu')}$$

Fig. 3. Part of the SOS for programs

3 Over-Approximation of the Semantics

Our aim is to address security information for arrays in a non-uniform manner; this implies that slices/parts of the arrays have to considered on their own. As arrays are manipulated via indices stored in integer variables, we have to figure out what is the content of these variables during the execution of the program. Capturing the exact values taken by some variables during the execution of a program is, of course, an undecidable problem. Therefore, we require an approximation of the values of those variables. To do so, we associate with each command label ℓ an approximation α of the values of integer variables at this program point (just before the execution of the command labelled with ℓ); α is an approximation in the following sense: in any execution of the program reaching this program point, if the variable x_i has k for value, then α states that k is an admissible value for x_i at that point.

Note that such approximations can be obtained in various ways : tools like FAST [3] or TReX [2] performing reachability analysis of systems augmented with (unbounded) integer variables can be used. Indeed, since we do not need to evaluate values in array cells but just integer values, our programs can be safely abstracted as counter automata. Another possibility is to use tools for inferring symbolic loop invariants such as in [10].

Hence, our aim is not to compute approximations, neither to propose various formalisms to represent them. We will mainly present the set of conditions such approximations must satisfy and prove that this set of conditions is sufficient.

To make our approach more concrete we chose Presburger formulas both to express the required conditions on the approximation as well as the approximation itself.

We recall that a Presburger formula ψ is given by the following syntax:

$$\psi := \exists x\psi \mid \forall x\psi \mid x = x_1 + x_2 \mid x = n \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \mid n * x \mid x/n \mid x \mod n$$

x, x_1, x_2 being integer variables and n an integer.

We will use the following conventions. Given two Presburger formulas $\psi(\bar{x})$ and $\psi'(\bar{x})$, we write $\psi \subseteq \psi'$ if $\models \forall\bar{x}, \neg\psi(\bar{x}) \vee \psi'(\bar{x})$. Given a Presburger formula $\psi(y, \bar{x})$ and an integer n, we write $\psi(n, \bar{x})$ instead of $\psi(y, \bar{x}) \wedge y = n$.

For the approximation, we consider here programs labelled by Presburger formulas whose free variables are essentially $x_1, \ldots, x_n$, which corresponds to the integer variable identifiers[1]. A labelling is then a mapping λ associating with each label $\ell \in \mathcal{L}$, a Presburger formula $\psi(\bar{x})$.

Given a program P and a labelling λ, we define $\lambda(P)$ to be the label of the first command of P, that is, recursively, $\lambda(c^\ell) = \lambda(\ell)$ and $\lambda(P_1; P_2) = \lambda(P_1)$. Intuitively, $\lambda(P) = \psi(\bar{x})$ means "enter in P with a memory μ whose integer variables satisfy $\psi(\bar{x})$", i.e., $[x_1 \mapsto \mu(x_1), \ldots, x_n \mapsto \mu(x_n)] \models \psi(\bar{x})$.

We start by defining in Fig. 4 for every expression e, the formula $V_e^\psi(y, \bar{x})$ giving the possible values of e when the integer variables of the memory satisfy a formula

[1] In the rest of the paper, we will denote by $\bar{x}$ the sequence of variables $x_1, \ldots, x_n$.

(O-INT)	$V^{\psi}_{n}(y,\bar{x}) \stackrel{def}{=} (y = n) \wedge \psi(\bar{x})$
(O-INT-VAR)	$V^{\psi}_{x_i}(y,\bar{x}) \stackrel{def}{=} (y = x_i) \wedge \psi(\bar{x})$
(O-ARR-READ)	$V^{\psi}_{T[e]}(y,\bar{x}) \stackrel{def}{=} \psi(\bar{x})$
(O-GET-LGTH)	$V^{\psi}_{T.\mathtt{length}}(y,\bar{x}) \stackrel{def}{=} \psi(\bar{x})$
(O-ADD)	$V^{\psi}_{e_1+e_2}(y,\bar{x}) \stackrel{def}{=} \exists y_1, \exists y_2, V^{\psi}_{e_1}(y_1,\bar{x}) \wedge V^{\psi}_{e_2}(y_2,\bar{x}) \wedge y = y_1 + y_2$
(O-DIV)	$V^{\psi}_{e/0}(y,\bar{x}) \stackrel{def}{=} V^{\psi}_{0}(y,\bar{x})$
if $n \neq 0$	$V^{\psi}_{e/n}(y,\bar{x}) \stackrel{def}{=} [V^{\psi}_{e}(0,\bar{x}) \wedge y = 0] \vee [\exists y_1, V^{\psi}_{e}(y_1,\bar{x}) \wedge y = y_1/n]$

Fig. 4. Definition of $V^{\psi}_{e}(y,\bar{x})$ for relevant expressions e

ψ. $V^{\psi}_{e}(y,\bar{x})$ is then a formula fulfilling for all memories μ: $[y \mapsto n, \bar{x} \mapsto \mu(\bar{x})] \models V^{\psi}_{e}(y,\bar{x})$ iff $[\bar{x} \mapsto \mu(\bar{x})] \models \psi(\bar{x})$ and $\mu(e) = n$. Hence, for instance, for $x_1 + x_2$ and $\psi = x_1 \leq 3 \wedge x_2 = 4$, we have that $V^{\psi}_{x_1+x_2}(y, x_1, x_2)$ is the formula $x_1 \leq 3 \wedge x_2 = 4 \wedge y = x_1 + x_2$. Note also that for an expression $T[e]$, the resulting formula $V^{\psi}_{T[e]}(y,\bar{x})$ imposes no constraints on y and thus, gives no information on the possible value of such an expression (there is no analysis of array contents). Similarly, let us point out, because of the programming language we have chosen, that the formulas $V^{\psi}_{e}(\bar{x})$ are always Presburger formulas. It may not be the case if expressions such as $e_1 * e_2$ were allowed: however, as we simply require an approximative computation, we could have defined $V^{\psi}_{e_1 * e_2}(y,\bar{x})$ as $\psi(\bar{x})$, leading to coarse but correct approximation.

Definition 1. *Let μ be a memory and $\psi(\bar{x})$ be a Presburger formula. We say that $\psi(\bar{x})$ is an over-approximation of μ (denoted $\psi(\bar{x}) \geq \mu$) if $[x_1 \mapsto \mu(x_1), \ldots, x_n \mapsto \mu(x_n)] \models \psi(x_1, \ldots, x_n)$.*

Lemma 1. *Let μ be a memory and $\psi(\bar{x})$ be a Presburger formula. If $\psi(\bar{x}) \geq \mu$ then $V^{\psi}_{e}(\mu(e),\bar{x}) \geq \mu$ (that is, $\models V^{\psi}_{e}(\mu(e), \mu(x_1), \ldots, \mu(x_n))$).*

Given a program P, a labelling λ, and a Presburger formula $\psi(\bar{x})$, we write $P \vdash_{\lambda} \psi$ to say that if we execute P under a memory μ satisfying $\lambda(P)$, then we get a new memory satisfying ψ. Formally,

Definition 2. *A program P is well labelled by λ if there exists a Presburger formula $\psi(\bar{x})$ such that $P \vdash_{\lambda} \psi$, where $\vdash_{\lambda}$ is the relation defined inductively in Fig. 5 according to the possible shape of P.*

(O-UP-ARR) $\dfrac{\lambda(\ell) \subseteq \psi}{x[e_1] :=^{\ell} e_2 \vdash_\lambda \psi}$ (O-CALLOC) $\dfrac{\lambda(\ell) \subseteq \psi}{\texttt{allocate}^{\ell}\ x[e] \vdash_\lambda \psi}$

(O-UPDATE) $\dfrac{(\exists y, [\exists x_i, V_e^{\lambda(\ell)}(y, x_1, \ldots x_{i-1}, x_i, x_{i+1}, \ldots, x_n,)] \wedge x_i = y) \subseteq \psi}{x_i :=^{\ell} e \vdash_\lambda \psi}$

(O-NO-OP) $\dfrac{\lambda(\ell) \subseteq \psi}{\texttt{skip}^{\ell} \vdash_\lambda \psi}$

(O-BRANCH) $\dfrac{V_e^{\lambda(\ell)}(0, \bar{x}) \subseteq \lambda(P_2), \exists y \neq 0. V_e^{\lambda(\ell)}(y, \bar{x}) \subseteq \lambda(P_1) \quad P_2 \vdash_\lambda \psi_2,\ P_1 \vdash_\lambda \psi_1,\ \psi_2 \vee \psi_1 \subseteq \psi}{\texttt{if}^{\ell}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \vdash_\lambda \psi}$

(O-LOOP) $\dfrac{\exists y \neq 0. V_e^{\lambda(\ell)}(y, \bar{x}) \subseteq \lambda(P),\ P \vdash_\lambda \lambda(\ell),\ V_e^{\lambda(\ell)}(0, \bar{x}) \subseteq \psi}{\texttt{while}^{\ell}\ e\ \texttt{do}\ P \vdash_\lambda \psi}$

(O-SEQ) $\dfrac{P_1 \vdash_\lambda \lambda(P_2),\ P_2 \vdash_\lambda \psi}{P_1; P_2 \vdash_\lambda \psi}$

Fig. 5. Definition of $P \vdash_\lambda \psi$

Let us point out that from rule (O-LOOP), the label $\lambda(P)$ in a command $\texttt{while}^{\ell}\ e\ \texttt{do}\ P$ has to be an invariant for this loop.

We give an example for the rule (O-UPDATE): if $x_1 :=^{\ell} x_1 + x_2$ and $\lambda(\ell) = x_1 \leq 3 \wedge x_2 = 4$, we have that $V_{x_1+x_2}^{\psi}(y, x_1, x_2)$ is $x_1 \leq 3 \wedge x_2 = 4 \wedge y = x_1 + x_2$ and then $\exists y[\exists x_1, V_{x_1+x_2}^{\psi}(y, x_1, x_2)] \wedge x_1 = y$ is equivalent to $\exists y, \exists x_1', x_1' \leq 3 \wedge x_2 = 4 \wedge y = x_1' + x_2 \wedge x_1 = y$ i.e. to $x_2 = 4 \wedge x_1 \leq 7$. Then $x_i :=^{\ell} e \vdash_\lambda \psi$ for all ψ such that $(x_2 = 4 \wedge x_1 \leq 7) \subseteq \psi$.

Remark that for each line of a program, checking the label costs the test of the validity of a Presburger formula with at most $2 + k$ quantifier alternations, where k is the number of quantifier alternations of the label formula.

Lemma 2. *If $P \vdash_\lambda \psi$ and $(P, \mu) \rightarrow (P', \mu')$ then there exists ψ' s.t. $P' \vdash_\lambda \psi'$ and $\psi' \subseteq \psi$.*

Proposition 1. *If P is well-labelled by λ, $(P, \mu) \rightarrow^* (P', \mu')$ and $\lambda(P) \geq \mu$, then $\lambda(P') \geq \mu'$.*

4 A Type System for Information Flow in Arrays

In this section, we present the type system we devise to guarantee the absence of illicit information flows in programs manipulating arrays. We exemplify our approach giving the type of some programs.

The goal of this type system is to guarantee that the execution of the program does not disclose confidential information that is, the contents of public variables do not depend on the contents of private ones through explicit and implicit flows.

4.1 The Type System

Our type system relies on a labelling of the program with approximation of the values taken by integer variables. This labelling is just supposed to be a well-labelling in the sens of Definition 2.

Types for arrays are couples (ϕ_H, τ), where ϕ_H is a Presburger formula, meaning that for all $n \geq 0$, $\phi_H(n)$ holds iff the cell of index n in the array has type H (otherwise it has type L) and that the length of the array has type τ.

Here are the types used by our type system:

$$\begin{array}{ll} \text{(data types)} & \tau ::= L \mid H \\ \text{(phrase types)} & \rho ::= \tau \mid \tau\, var \mid \tau\, cmd \mid (\phi_H, \tau) \end{array}$$

Intuitively, τ is the security level for integer expressions, $\tau\, var$ is the one of integer variables and (ϕ_H, τ) is the security level for the cells and the length of arrays. For commands and programs, $\tau\, cmd$ stands for a program that can safely write in storage (variables, array cells) of level higher than τ.

Moreover, we adopt the following constraint of types : in any array type (ϕ_H, τ), we require that if $\tau = H$ then ϕ_H is the formula $True$ **(Array type constraint)**. Indeed, due to the lenient semantics of arrays, reading the content of `T[3]` after the execution of `allocate T[h]; T[3] := 2;` leaks information about `h` (if `T[3]` is zero then `h` is smaller than 3).

Our type system allows to prove judgments of the form $\Gamma, \psi \vDash e : \tau$ for expressions and of the form $\Gamma \vDash_\lambda P : \tau\, cmd$ for programs as well of subtyping judgments of the form $\rho_1 \subseteq \rho_2$. Here Γ denotes an environment, that is an identifier typing, mapping each integer identifier in $\mathcal{I}$ to a phrase type of the form $\tau\, var$ and each array identifier in $\mathcal{T}$ to a phrase type of the form (ϕ_H, τ), ψ is a Presburger formula, e is an expression, P is a program and λ is a labelling.

The subtyping rules are on Figure 6: informally, a low level expression can always be used in the high level situation (hence, constants can be assigned to high level variables) whereas high level commands can always be used in a low level context (if only private variables are affected then only variables greater than L have been affected).

$$\begin{array}{llll} \text{(BASE)} & L \subseteq H & \text{(CMD}^-\text{)} & \dfrac{\tau' \subseteq \tau}{\tau\, cmd \subseteq \tau'\, cmd} \\[2ex] \text{(REFLEX)} & \rho \subseteq \rho & \text{(TRANS)} & \dfrac{\rho_1 \subseteq \rho_2,\ \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3} \\[2ex] \text{(SUBSUMP)} & \dfrac{\Gamma, \psi \vDash e : \tau_1,\ \tau_1 \subseteq \tau_2}{\Gamma, \psi \vDash e : \tau_2} & & \dfrac{\Gamma \vDash_\lambda P : \rho_1,\ \rho_1 \subseteq \rho_2}{\Gamma \vDash_\lambda P : \rho_2} \end{array}$$

Fig. 6. Subtyping rules

The most relevant typing rules for expressions can be found in Fig. 7. Typing rules (INT), (R-VAL) and (QUOTIENT) are borrowed from the Volpano-Smith type system:

(INT) $\Gamma, \psi \vDash n : L$ (R-VAL) $\dfrac{\Gamma(x_i) = \tau\, var}{\Gamma, \psi \vDash x_i : \tau}$

(SUBSCR) $\dfrac{\Gamma, \psi \vDash e : \tau}{\Gamma, \psi \vDash T[e] : H}$

$$\frac{\Gamma(T) = (\phi_H, L),\ \Gamma, \psi \vDash e : L,\ \models \forall y, [(\exists \bar{x}, V_e^{\psi}(y, \bar{x})) \implies \neg \phi_H(y)]}{\Gamma, \psi \vDash T[e] : L}$$

(LENGTH) $\dfrac{\Gamma(T) = (\phi_H, \tau)}{\Gamma, \psi \vDash T.\mathtt{length} : \tau}$ (QUOTIENT) $\dfrac{(\Gamma, \psi) \vDash e_1 : \tau,\ \Gamma, \psi \vDash n : \tau}{\Gamma, \psi \vDash e_1/n : \tau}$

Fig. 7. Typing rules for expressions

the rule (INT) states that constants are from level L (and also, from level H thanks to subtyping). For variables, as usual, the environment Γ fixes the type of identifier. Finally, for arithmetical operations such as (QUOTIENT), the type of the parameters and of the result have to be the same. The rule (LENGTH) is simply the direct translation of the meaning of types for arrays in the spirit of (R-VAL).

The first rule for (SUBSCR) states that any cell of arrays can be typed as H. The second rule for (SUBSCR) states that the content of an array cell is L only if the expression used to address this cell is L and the possible values for this expression according to the approximation given by λ are set to be not H by the formula ϕ_H of the array type. Note that due to the "array type constraint", the fact that some cells being not of high level H implies that the length of the array is not high (and thus, low).

Typing rules for programs are described in Fig. 8. The types deduced for the commands control mainly the security level of the context in which they are executed; it aims to treat implicit control flows. The typing rules (ASSIGN), (SKIP),(IF), (WHILE), (COMPOSE) are borrowed from the Volpano-Smith type system: the last three simply force the components of a composed program to have the same type. The rule (ASSIGN) states that the assigned value and the variable must be of the same type. This prevents illicit direct flow of information. Finally, because of the rule (SKIP), the command `skip` is used in a high security context (but it can also be used in a low security context thanks to subtyping).

Let us point out that for array cells, just as scalar variables, we distinguish between the security level associated with a cell from the security level of the content of this cell.

The typing rule for (ALLOCATE) states that the type of an expression used in the declaration of an array must coincide with the type of the size of this array. Finally, for (ASSIGN-ARR), the rules aim to guarantee that, as for scalar variables, high level values can not be assigned to low level array cells (to address explicit flows); moreover, a command modifying a low level cell has to be a low level command (to address implicit flows).

In our typing approach, information on the security level for variables is fixed for the whole program; hence, the type of arrays are not modified during the execution according to updates.

$$\text{(ASSIGN)} \quad \frac{\Gamma(x) = \tau\, var,\ \Gamma, \lambda(\ell) \vDash e : \tau}{\Gamma \vDash_\lambda (x :=^\ell e) : \tau\, cmd} \qquad \text{(SKIP)} \quad \Gamma \vDash_\lambda (\mathtt{skip}^\ell) : H\, cmd$$

$$\text{(ASSIGN-ARR)} \quad \frac{\begin{array}{c}\Gamma, \lambda(\ell) \vDash e_1 : L,\ \Gamma, \lambda(\ell) \vDash e_2 : \tau,\ \Gamma(T) = (\phi_H, L), \\ \vDash \forall y,\ [(\exists \bar{x},\ V_{e_1}^{\lambda(\ell)}(y, \bar{x}) \implies \phi_H(y)]\end{array}}{\Gamma \vDash_\lambda (T[e_1] :=^\ell e_2) : H\, cmd}$$

$$\frac{\Gamma, \lambda(\ell) \vDash e_1 : H,\ \Gamma, \lambda(\ell) \vDash e_2 : \tau,\ \Gamma(T) = (\phi_H, H)}{\Gamma \vDash_\lambda (T[e_1] :=^\ell e_2) : H\, cmd}$$

$$\frac{\Gamma, \lambda(\ell) \vDash e_1 : L,\ \Gamma, \lambda(\ell) \vDash e_2 : L,\ \Gamma(T) = (\phi_H, L)}{\Gamma \vDash_\lambda (T[e_1] :=^\ell e_2) : L\, cmd}$$

$$\text{(ALLOCATE)} \quad \frac{\Gamma(T) = (\phi_H, \tau),\ \Gamma, \lambda(\ell) \vDash e : \tau}{\Gamma \vDash_\lambda (\mathtt{allocate}^\ell\ T[e]) : \tau\, cmd}$$

$$\text{(IF)} \quad \frac{\Gamma, \lambda(\ell) \vDash e : \tau,\ \Gamma \vDash_\lambda P_1 : \tau\, cmd,\ \Gamma \vDash_\lambda P_2 : \tau\, cmd}{\Gamma \vDash_\lambda (\mathtt{if}^\ell\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2) : \tau\, cmd}$$

$$\text{(WHILE)} \quad \frac{\Gamma, \lambda(\ell) \vDash e : \tau,\ \Gamma \vDash_\lambda P : \tau\, cmd}{\Gamma \vDash_\lambda (\mathtt{while}^\ell\ e\ \mathtt{do}\ P) : \tau\, cmd}$$

$$\text{(COMPOSE)} \quad \frac{\Gamma \vDash_\lambda c^\ell : \tau\, cmd,\ \Gamma \vDash_\lambda P : \tau\, cmd}{\Gamma \vDash_\lambda (c^\ell ; P) : \tau\, cmd}$$

Fig. 8. Typing rules for programs

Remark that for checking a given typing, at each line of the program, we have to test the validity of a Presburger formula with at most $2+max(k+l)$ quantifier alternations, where k is the number of quantifier alternations of the label formula, and l is the number of quantifier alternations of the formula typing the array occurring in the program line.

In fact, if we infer a typing for the program, we get array formulas containing at most $1+m$ quantifier alternations, where m is the maximum of the number of quantifier alternations of formulas labelling the program.

4.2 Example

To illustrate our approach, let us give now examples obtained with a prototype that we have developed which, from a program P, a labelling λ and some initial typing, determines if P is well-labelled and when possible types the program P respecting the initial typing by giving the minimal type. The first example shows how an array allocation can force the type of an array to be H:

```
[true]                                    x1:=T1.length;
[true]                                    x2:=T2.length;
[true]                                    x3:=x1+x2;
[x3=x1+x2]                               allocate(T0[x3]);
```

```
[x3=x1+x2]                                x4:=0;
[x3=x1+x2 and x4=0]                       x5:=0;
[x3=x1+x2 and 0<=x4<=x3+1 and x4=2x0]     while (x4 < x3) do
[x3=x1+x2 and 0<=x4<x3 and x4=2x0]            T0[x4]:=T1[x0];
[x3=x1+x2 and 0<=x4<x3 and x4=2x0]            T0[x4+1]:=T2[x0];
[x3=x1+x2 and 0<=x4<x3 and x4=2x0]            x4:=x4+2;
[x3=x1+x2 and 2<=x4<=x3+1 and x4=2x0+2]       x0:=x0+1;
```

If we impose the constaint `T2:(True,H)`, we get in a time less than one second:

```
T0: (True,High),T1: (False, Low),T2: (True, High)
x0: High, x1: Low, x2: High, x3:High, x4:High.
```

Indeed, line 2 and rule (ALLOCATE) imply `x2:High`, then line 3 and rule (ASSIGN) imply `x3:High` and then line 4 and rule (ALLOCATE) imply `T0:(True,High)`. If `T0` is allocated with a size typed by `Low`, then all the even indexes of `T0` can have the type `Low`, as shown in the following example:

```
[ True ]                                     allocate(T0[x1]);
[ True ]                                     x2 := 0;
[ x2=0 ]                                     x0 := 0;
[ 0 <= x2 <= 1+x1 and x2=2x0]          while ( x2 < x1 ) do
[ 0 <= x2 < x1 and x2=2x0]                 T0[x2] := T1[x0];
[ 0 <= x2 < x1 and x2=2x0]                 T0[x2 + 1] := T2[x0];
[ 0 <= x2 < x1 and x2=2x0]                 x2 := x2 + 2;
[ 2 <= x2<=1+x1 and x2=2+2x0]             x0 := x0 + 1;
```

If we impose the constaint `T1:(True,H)`, our prototype returns:

```
T0: (Ex x2,x0.(y=x2 and x2=2x0),Low),T1:(True,High),T2:(False,Low)
x0: Low, x1: Low, x2: Low.
```

Remark that since all labels are quantifier free, the formula obtained for the type of `T0` is existential.

5 Properties of the Type System

We prove that our type system guarantees noninterference for well labelled programs. The proofs of some lemmas below are complicated somewhat by subtyping. We therefore assume, without loss of generality, that all typing derivations end with a single (perhaps trivial) use of the rule (SUBSUMP).

Lemma 3 (Subject Reduction). *If* $\Gamma \vDash_\lambda P : \tau\, cmd$ *and* $(P,\mu) \rightarrow (P',\mu')$ *then* $\Gamma \vDash_\lambda P' : \tau\, cmd$.

Proof. By induction on the structure of P. There are just three kinds of programs that can take more than one step to terminate:

1. Case $\texttt{if}^\ell\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. By our assumption, the typing derivation for P must end with a use of the rule (IF) followed by a use of (SUBSUMP):

$$\frac{\dfrac{(\Gamma, \lambda(\ell)) \vDash e : \tau',\ \Gamma \vDash_\lambda P_1 : \tau'\, cmd,\ \Gamma \vDash_\lambda P_2 : \tau'\, cmd}{\Gamma \vDash_\lambda \texttt{if}^\ell\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 : \tau'\, cmd,\ \tau'\, cmd \subseteq \tau cmd}}{\Gamma \vDash_\lambda \texttt{if}^\ell\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 : \tau\, cmd}$$

Hence, by the rule (CMD$^-$), we must have $\tau \subseteq \tau'$. So by (SUBSUMP) we have $\Gamma \vDash_\lambda P_1 : \tau\, cmd$ and $\Gamma \vDash_\lambda P_2 : \tau\, cmd$. By the rule (BRANCH), P' can be either P_1 or P_2, therefore, we have $\Gamma \vDash_\lambda P' : \tau\, cmd$.

2. Case $\texttt{while}^\ell\ e\ \texttt{do}\ P_1$. By an argument similar to the one used in the previous case, we have $\Gamma \vDash_\lambda P_1 : \tau\, cmd$. By the rule (LOOP), P' is P_1, hence $\Gamma \vDash_\lambda P' : \tau\, cmd$ by rule (COMPOSE).
3. Case $P = {c_1}^\ell; P_2$. As above, we get $\Gamma \vDash_\lambda {c_1}^\ell : \tau\, cmd$ and $\Gamma \vDash_\lambda P_2 : \tau\, cmd$. By the rule (SEQUENCE) P' is either P_2 (if c_1 terminates in one step) or else $P_1; P$, where $({c_1}^\ell, \mu) \rightarrow (P_1, \mu')$. For the first case, we have then $\Gamma \vDash_\lambda P_2 : \tau\, cmd$. For the second case, we have $\Gamma \vDash_\lambda P_1 : \tau\, cmd$ by induction; hence $\Gamma \vDash_\lambda P_1; P_2 : \tau\, cmd$ by rule the (COMPOSE).

We also need an obvious lemma about the execution of a sequential composition.

Lemma 4. *If $((c^\ell; P), \mu) \rightarrow^j \mu'$ then there exist k and μ'' such that $0 < k < j$, $(c^\ell, \mu) \rightarrow^k \mu$", $((c^\ell; P), \mu) \rightarrow^k (P, \mu$") and $(P, \mu$")$\rightarrow^{j-k} \mu'$.*

Definition 3. *Memories μ and ν are equivalent with respect to Γ, written $\mu \sim_\Gamma \nu$, if*

- *μ and ν agree on all L variables,*
- *for all $T \in \mathcal{T}$, if $\Gamma(T) = (\phi_H, L)$, then there exists $k \geq 0$ such that $\mu(T) = \langle n_0, \ldots, n_{k-1} \rangle$ and $\nu(T) = \langle n'_0, \ldots, n'_{k-1} \rangle$ and for all $i \in [0, k-1]$, $\vDash \neg\phi_H(i)$ implies $n_i = n'_i$.*

Lemma 5 (Simple Security). *If $\Gamma, \psi \vDash e : L$ and $\mu \sim_\Gamma \nu$ and $\psi \geq \mu$, then $\mu(e) = \nu(e)$.*

Proof. By induction on the structure of e:

1. Case n. Obviously, $\mu(e) = \nu(e) = n$.
2. Case x_i. By the typing rule (r-val), $\Gamma(x_i) = L\, var$. Therefore, by memory equivalence, $\mu(x_i) = \nu(x_i)$.
3. Case $T[e]$. By the typing rule (subscr), we have $\Gamma(T) = (\phi_H, L)$, $\Gamma, \psi \vDash e : L$, and $\vDash \forall y[(\exists \bar{x}. V_e^\psi(y, \bar{x}) \implies \neg\phi_H(y)]$. Since $\psi \geq \mu$, from Lemma 1, $\vDash V_e^\psi(\mu(e), \mu(\bar{x}))$, then $\vDash \neg(\phi_H(\mu(e)))$. From $\Gamma, \psi \vDash e : L$ and induction hypothesis, there exists an integer i such that $\mu(e) = \nu(e) = i$. Suppose that $\mu(T) = \langle n_0, \ldots, n_{k-1} \rangle$ and $\nu(x) = \langle n'_0, \ldots, n'_{k-1} \rangle$ ($\mu(T)$ and $\nu(T)$ have the same length by memory equivalence). We consider then two cases:
   - case $i \in [0, k-1]$. Since $\vDash \neg\phi_H(i)$, by memory equivalence we get $n_i = n'_i$. Then by the SOS rule (ARR-READ), $\mu(T[e]) = \nu(T[e])$.
   - case $i \notin [0, k-1]$. By the SOS rule (ARR-READ), $\mu(T[e]) = \nu(T[e]) = 0$.
4. Case $T.\texttt{length}$. Suppose that $\Gamma, \psi \vDash T.\texttt{length} : L$, then $\Gamma(T) = (\phi_H, L)$ and from memory equivalence, there exists an integer $k \geq 0$ such that $\mu(T) = \langle n_0, \ldots, n_{k-1} \rangle$ and $\nu(x) = \langle n'_0, \ldots, n'_{k-1} \rangle$. From the SOS semantics(GET-LGTH), $\mu(T.\texttt{length}) = \nu(T.\texttt{length}) = k$.
5. Case e/n. By the typing rule (QUOTIENT), we have $\Gamma, \psi \vDash e_1 : L$ and $\Gamma, \psi \vDash n : L$. By induction, we have $\mu(e_1) = \nu(e_1)$.Then by the SOS rule (DIV), we have $\mu(e_1/n) = \nu(e_1/n)$.

Lemma 6 (Confinement). *If $\Gamma \vDash_\lambda P : H\,cmd$, $(P,\mu) \rightarrow (P',\mu')$ (or $(P,\mu) \rightarrow \mu'$) and $\lambda(P) \geq \mu$ then $\mu \sim_\Gamma \mu'$.*

Proof. By induction on the structure of c:

1. Case $x_i :=^\ell e$. From the SOS rule (UPDATE), $\mu' = \mu[x := \mu(e)]$, and from the typing rule (ASSIGN) $\Gamma(x) = H\,var$, so $\mu \sim_\Gamma \mu'$.
2. Case $T[e_1] :=^\ell e_2$. We consider two cases according to the type of e_1.
   - Case $\Gamma, \lambda(\ell) \vDash e_1 : L$, then from the typing rule (ASSIGN-ARR), $\Gamma(T) = (\phi_H, L)$ and $\models \forall y[(\exists \bar{x}, V_e^{\lambda(\ell)}(y,\bar{x}) \implies \phi_H(y)]$. Since $\lambda(\ell) \geq \mu$, we have from Lemma 1: $\models V_{e_1}^{\lambda(\ell)}(\mu(e_1), \mu(\bar{x}))$, and then $\models \phi_H(\mu(e_1))$. Let us suppose that $\mu(x) = \langle n_0, \ldots, n_{k-1}\rangle$, we consider two cases:
     - if $\mu(e_1) \in [0, k-1]$, by the SOS rule (UPD-ARRAY) $\mu' = \mu[T[\mu(e_1)] := \mu(e_2)]$ and since $\models \phi_H(i)$, $\mu \sim_\Gamma \mu'$.
     - else, by the SOS rule (UPD-ARRAY) $\mu' = \mu$ and then $\mu \sim_\Gamma \mu'$.
   - Case $\Gamma, \lambda(\ell) \vDash e_1 : H$, then $\Gamma(T) = (\phi_H, L)$ and $\mu \sim_\Gamma \mu'$.
3. Case $\texttt{allocate}^\ell\ T[e]$. From the typing rule (ALLOCATE), $\Gamma(T) = (\phi_H, H)$ and from the SOS rule (CALLOC), $\mu \sim_\Gamma \mu'$,
4. Cases $\texttt{skip}^\ell$, $\texttt{if}^\ell\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ and $\texttt{while}^\ell\ e\ \texttt{do}\ c$. These cases are trivial, because $\mu = \mu'$.
5. Case ${c_1}^\ell; P_2$. From the typing rule (COMPOSE), $\Gamma \vDash_\lambda {c_1}^\ell : H\,cmd$ and from the SOS rule (SEQUENCE), $({c_1}^\ell, \mu) \rightarrow \mu'$ or there exists P_1 such that $({c_1}^\ell, \mu) \rightarrow (P_1, \mu')$. Then $\lambda(c_1) = \lambda(P) \geq \mu$ and by induction, $\mu \sim_\Gamma \mu'$.

Corollary 1. *If P is well-labelled by λ, $\Gamma \vDash_\lambda P : H\,cmd$, $\mu : \Gamma$, $(P,\mu) \rightarrow^* \mu'$ and $\lambda(P) \geq \mu$ then $\mu \sim_\Gamma \mu'$.*

Proof. By induction on the length of the derivation $(P,\mu) \rightarrow^* \mu'$. The base case is proved by Confinement (Lemma 6). Let us prove the induction step. We suppose that $(P,\mu) \rightarrow (P_1, \mu_1) \rightarrow^* \mu'$. From Confinement, $\mu \sim_\Gamma \mu_1$, from Subject Reduction, $\Gamma \vDash_\lambda P_1 : H\,cmd$ and as P is well-labelled by λ, by Proposition 1, $\lambda(P_1) \geq \mu_1$. From Lemma 2, P_1 is well-labelled by λ, hence, by induction hypothesis, $\mu_1 \sim_\Gamma \mu'$, and then $\mu \sim_\Gamma \mu'$.

Theorem 1 (Noninterference). *Suppose that P is well-labelled by λ, $\mu \sim_\Gamma \nu$, $\Gamma \vDash_\lambda P : \tau\,cmd$, $\lambda(P) \geq \mu$ and $\lambda(P) \geq \nu$. If $(P,\mu) \rightarrow^* \mu'$ and $(P,\nu) \rightarrow^* \nu'$ then $\mu' \sim_\Gamma \nu'$.*

Proof. By induction on the length of the execution $(P,\mu) \rightarrow^* \mu'$. We consider the different forms of P.

1. Case $x :=^\ell e$. By the SOS rule (UPDATE), we have $\mu' = \mu[x := \mu(e)]$ and $\nu' = \nu[x := \nu(e)]$. If $\Gamma(x) = L\,var$, then by the typing rule (ASSIGN), we have $(\Gamma, \ell) \vDash_\lambda e : L$. So by Simple Security, $\mu(e) = \nu(e)$, and so $\mu' \sim_\Gamma \nu'$. If instead, $\Gamma(x) = H\,var$ then trivially $\mu' \sim_\Gamma \nu'$.
2. Case $T[e_1] :=^\ell e_2$. We consider the possible values of τ.
   - If $\tau = H$, then from Corollary 1, $\mu \sim_\Gamma \mu'$ and $\nu \sim_\Gamma \nu'$. So $\mu' \sim_\Gamma \nu'$.
   - If $\tau = L$, then $\Gamma, \lambda(\ell) \vDash_\lambda e_1 : L$ and $\Gamma, \lambda(\ell) \vDash_\lambda e_2 : L$ and $\Gamma(T) = (\phi_H, L)$. By Simple Security, there exist i, j s.t. $\mu(e_1)=\nu(e_1)=i$ and $\mu(e_2)=\nu(e_2)=j$. Since $\mu \sim_\Gamma \nu$, there exists $k \geq 0$ s.t $\mu(T)=\langle n_0, \ldots n_{k-1}\rangle$, $\nu(T)=\langle n'_0, \ldots n'_{k-1}\rangle$.

- if $i \in [0, k-1]$. By the SOS rule (UPDATE-ARR), $\mu' = \mu[T[i] := j]$ and $\nu' = \nu[T[i] := j]$. So $\mu' \sim_\Gamma \nu'$.
- else, from the SOS rule (UPDATE-ARR), $\mu' = \mu$ and $\nu' = \nu$. So $\mu' \sim_\Gamma \nu'$.

3. Case $\texttt{allocate}^\ell\ T[e]$. We consider two cases.
   - if $\tau = H$, then by Corollary 1, $\mu \sim_\Gamma \mu'$ and $\nu \sim_\Gamma \nu'$. So $\mu' \sim_\Gamma \nu'$.
   - if $\tau = L$, from the rule (ALLOCATE), $\Gamma, \lambda(\ell) \vDash e : L$ and from Simple Security $\mu(e)$=$\nu(e)$. So, by the SOS rule (CALLOC), $\mu'(T)$=$\nu'(T)$. So, $\mu' \sim_\Gamma \nu'$.
4. Case $\texttt{skip}^\ell$. From the SOS rule (NO-OP), $\mu = \mu'$ and $\nu = \nu'$. So, $\mu' \sim_\Gamma \nu'$.
5. Case $P = \texttt{if}^\ell\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. If $(\Gamma, \ell) \vDash_\lambda e : L$ then $\mu(e) = \nu(e)$ by Simple Security. If $\mu(e) \neq 0$ then $(\mu, P) \rightarrow (\mu, P_1) \rightarrow^* \mu'$ and $(\nu, P) \rightarrow (\nu, P_1) \rightarrow^* \nu'$. From the typing rule (IF), we have then $\Gamma \vDash_\lambda P_1 : L\, cmd$. In addition, from Proposition 1 , $\lambda(P_1) \geq \mu$ and $\lambda(P_1) \geq \nu$ and from Lemma 2, P_1 is well-labelled by λ, then by induction, $\mu' \sim_\Gamma \nu'$. The case $\mu(e) = 0$ is similar.
   If instead $\Gamma, \lambda(\ell) \vDash e : H$, then from the typing rule (IF), we have $\Gamma \vDash_\lambda P : H\, cmd$. Then by Corollary 1, $\mu \sim_\Gamma \mu'$ and $\nu \sim_\Gamma \nu'$. So $\mu' \sim_\Gamma \nu'$.
6. Case $\texttt{while}^\ell\ e\ \texttt{do}\ c'$. Similar to the $\texttt{if}$ case.
7. Case $P = c_1{}^\ell; P_2$. If $((c_1{}^\ell; P_2), \mu) \rightarrow^j \mu'$, then by Lemma 4 there exist k and μ" such that $0 < k < j$, $(c_1{}^\ell, \mu) \rightarrow^k \mu$", $(c_1{}^\ell; P_2, \mu) \rightarrow^k (P_2, \mu")$ and $(P_2, \mu") \rightarrow^{j-k} \mu'$. Similarly, if $((c_1{}^\ell; P), \nu) \rightarrow^{j'}, \nu'$, then there exist k' and ν" such that $0 < k' < j'$, $(c_1{}^\ell, \nu) \rightarrow^{k'} \nu$", $(c_1{}^\ell; P_2, \nu) \rightarrow^{k'} (P_2, \nu")$ and $(\nu", P_2) \rightarrow^{j'-k'} \nu'$. Since P is well-labelled by λ, by the rule (O-SEQ), c_1 and P_2 are well-labelled by λ. In addition, from Lemma 1, $\lambda(P_2) \geq \mu$" and $\lambda(P_2) \geq \nu$". By induction, $\mu" \sim_\Gamma \nu"$. So by induction again, $\mu' \sim_\Gamma \nu'$.

Corollary 2. *Suppose that P is well-labelled, $\Gamma \vDash_\lambda P : \tau\, cmd$ and $\mu \sim_\Gamma \nu$ and $lab(P) = true$ and $(c, \mu) \rightarrow^* \mu'$ and $(c, \nu) \rightarrow^* \nu'$ then $\mu' \sim_\Gamma \nu'$.*

6 Conclusion and Future Work

In this paper, we have proposed a type system for an accurate information flow analysis of programs with arrays. Our type system is based on a pre-computed approximation of integer variables manipulated by the program. We believe that our approach can be extended to array aliases as in [17] as well as multi-dimensional arrays.

We have developed a prototype which, from a program P a labelling λ and some initial typing, determines if P is well-labelled and when possible types the program P respecting the initial typing by giving the minimal type. To improve this prototype, the next step will be the integration of a module for automatic generation of a well-labelling: the main ingredient for this will be an invariant generator such as in [10].

The approach we presented is flow-insensitive (the relative order of commands is irrelevant for typing); it is known that flow sensitive approach are more accurate and allows to consider polymorphic data structures since a specific type is inferred for each variable at each program point [8,4]. However, our work is fully compatible with the framework proposed in [11] (as our types can trivially be organised as lattices), allowing to transform our type system into a flow-sensitive one.

As we explain, we based the well-labelling of programs only on the values of integer variables ignoring the values contained in array cells. Recent works try to address the issue of reasonning about array contents [9,13]. It could be interesting to know whether this kind of works can be combined directly with our type system.

References

1. Amtoft, T., Hatcliff, J., Rodríguez, E.: Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 43–63. Springer, Heidelberg (2010)
2. Annichini, A., Bouajjani, A., Sighireanu, M.: Trex: A tool for reachability analysis of complex systems. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 368–372. Springer, Heidelberg (2001)
3. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: Fast acceleration of symbolikc transition systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 118–121. Springer, Heidelberg (2003)
4. Bonelli, E., Compagnoni, A., Medel, R.: Information flow analysis for a typed assembly language with polymorphic stacks. In: Barthe, G., Grégoire, B., Huisman, M., Lanet, J.-L. (eds.) CASSIS 2005. LNCS, vol. 3956, pp. 37–56. Springer, Heidelberg (2006)
5. Deng, Z., Smith, G.: Lenient array operations for practical secure information flow. In: 17th IEEE Computer Security Foundations Workshop (CSFW-17 2004), p. 115. IEEE Computer Society Press, Los Alamitos (2004)
6. Denning, D.: A lattice model of secure information flow. Commun. ACM 19(5), 236–243 (1976)
7. Denning, D., Denning, P.: Certification of programs for secure information flow. Commun. ACM 20(7), 504–513 (1977)
8. Genaim, S., Spoto, F.: Information flow analysis for java bytecode. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 346–362. Springer, Heidelberg (2005)
9. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, pp. 339–348. ACM Press, New York (2008)
10. Henzinger, T., Hottelier, T., Kovács, L.: Valigator: A verification tool with bound and invariant generation. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 333–342. Springer, Heidelberg (2008)
11. Hunt, S., Sands, D.: On flow-sensitive security types. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 2006, pp. 79–90. ACM Press, New York (2006)
12. Myers, A.: Jflow: Practical mostly-static information flow control. In: POPL, pp. 228–241 (1999)
13. Perrelle, V., Halbwachs, N.: An analysis of permutations in arrays. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 279–294. Springer, Heidelberg (2010)
14. Pottier, F., Simonet, V.: Information flow inference for ml. ACM Trans. Program. Lang. Syst. 25(1), 117–158 (2003)
15. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21 (2003)
16. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security 4(2/3), 167–188 (1996)
17. Yao, J., Li, J.: Discretional array operations for secure information flow. Journal of Information and Computing Science 1(2), 67–77 (2007)

Analysis of Deadlocks in Object Groups*

Elena Giachino and Cosimo Laneve

Dipartimento di Scienze dell'Informazione, Università di Bologna

Abstract. Object groups are collections of objects that perform collective work. We study a calculus with object groups and develop a technique for the deadlock analysis of such systems based on abstract descriptions of method's behaviours.

1 Introduction

Object groups are collections of objects that perform collective work. The group abstraction is an encapsulation mechanism that is convenient in distributed programming in several circumstances. For example, in order to achieve continuous availability or load balancing through replication, or for retrieval services of distributed data. In these cases, in order to keep consistencies, the group abstraction must define suitable protocols to synchronize group members. As usual with synchronization protocols, it is possible that object groups may manifest deadlocks, which are particularly hard to discover in this context because of the two encapsulation levels of the systems (the object and the group levels).

Following the practice to define lightweight fragments of languages that are sufficiently small to ease proofs of basic properties, we define an object-oriented calculus with group operations and develop a technique for the analysis of deadlocks. Our object-oriented language, called `FJg`, is an imperative version of Featherweight Java [9] with method invocations that are asynchronous and group-oriented primitives that are taken from *Creol* [10] (*cf.* the JCoBoxes [19]).

In `FJg`, objects always belong to one group that is defined when they are created. Groups consist of multiple tasks, which are the running methods of the objects therein. Tasks are cooperatively scheduled, that is there is at most one task active at each time per group and the active task explicitly returns the control in order to let other tasks progress. Tasks are created by method invocation that are *asynchronous* in `FJg`: the caller activity continues after the invocation and the called code runs on a different task that may belong to a different group. The synchronization between the caller and the called methods is performed when the result is strictly necessary [10,21,3]. Technically, the decoupling of method invocation and the returned value is realized using *future variables* (see [6] and the references in there), which are pointers to values that may be not available yet. Clearly, the access to values of future variables may require waiting for the value to be returned.

In a model with object groups and cooperative scheduling, a typical deadlock situation occurs when two active tasks in different groups are waiting for each other to

* The authors are member of the joint FOCUS Research Team INRIA/Università di Bologna. This research has been funded by the EU project FP7-231620 HATS.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 168–182, 2011.
© IFIP International Federation for Information Processing 2011

return a value. This circular dependency may involve less or more than two tasks. For example, a case of circularity of size one is

```
Int fact(Int n){    if (n=0) then return 1 ;
                    else return n*(this!fact(n-1).get)   }
```

The above FJg code defines the factorial function (for the sake of the example we include primitive types Int and conditional into FJg syntax. See Section 2.1). The invocation of this!fact(n) deadlocks on the recursive call this!fact(n-1) because the caller does not explicitly release the group lock. The operation get is needed in order to synchronously retrieve the value returned by the invocation.

We develop a technique for the analysis of deadlocks in FJg programs based on *contracts*. Contracts are abstract descriptions of behaviours that retain the necessary informations to detect deadlocks [12,11]. For example, the contract of fact (assuming it belongs to the class Ops) is G(){ $\texttt{Ops.fact}^{g}$: G() }. This contract declares that, when fact is invoked on an object of a group G, then it will call recursively fact on an object of the same group G without releasing the control – a *group dependency*. With this contract, any invocation of fact is fated to deadlock because of the circularity between G and itself (actually this.fact(0) never deadlocks, but the above contract is not expressive enough to handle such cases).

In particular, we define an inference system for associating a contract to every method of the program and to the expression to evaluate. Then we define a simple algorithm – the dla algorithm – returning informations about group dependencies. The presence of circularities in the result of dla reveals the possible presence of deadlocked computations. Overall, our results show the possibility and the benefit of applying techniques developed for process calculi to the area of object-oriented programming.

The paper is organized as follows. Section 2 defines FJg by introducing the main ideas and presenting its syntax and operational semantics. Section 3 discusses few sample programs in FJg and the deadlocks they manifest. Section 4 defines contracts and the inference algorithm for deriving contracts of expressions and methods. Section 5 considers the problem of extracting dependencies from contracts, presents the algorithm dla, and discusses its enhancements. Section 6 surveys related works, and we give conclusions and indications of further work in Section 7.

Due to space limitations, the technical details are omitted. We refer the interested reader to the full paper in the home-pages of the authors.

2 The Calculus FJg

In FJg a program is a collection of class definitions plus an expression to evaluate. A simple definition in FJg is the class C in Table 1. This program defines a class C with a method m. When m is invoked, a new object of class C is created and returned. A distinctive feature of FJg is that an object belongs to a unique group. In the above case, the returned object belongs to a new group – created by the operator newg. If the new object had to belong to the group of the caller method, then we would have used the standard operation new.

Table 1. Simple classes in FJg

```
class C {    C m() { return newg C() ;}  }
class D extends C {    C n(D c) { return (c!m()).get ;}  }
```

Method invocations in FJg are asynchronous. For example, the class D in Table 1 defines an extension of C with method n. In order to emphasize that the semantics of method invocation is not as usual, we use the exclamation mark (instead of the dot notation). In FJg, when a method is invoked, the caller continues executing *in parallel with* the callee *without releasing its own group lock*; the callee gets the control by acquiring the lock of its group when it is free. This guarantees that, at each point in time, at most one task may be active per group. The get operation in the code of n constraints the method to wait for the return value of the callee before terminating (and therefore releasing the group lock).

In FJg, it is also possible to wait for a result without keeping the group lock. This is performed by the operation await that releases the group lock and leaves other tasks the chance to perform their activities until the value of the called method is produced. That is, x!m().await.get corresponds to a method invocation in standard object-oriented languages.

The decoupling of method invocation and the returned value is realized in FJg by using *future types*. In particular, if a method is declared to return a value of type C, then its invocations return values of type Fut(C), rather than values of type C. This means that the value is not available yet; when it will be, it is going to be of type C. The operation get takes an expression of type Fut(C) and returns C (as the reader may expect, await takes an expression of type Fut(C) and returns Fut(C)).

2.1 Syntax

The syntax of FJg uses four disjoint infinite sets of *class names*, ranged over by A, B, C, $\cdots$, *field names*, ranged over by f, g, $\cdots$, *method names*, ranged over by m, n, $\cdots$, and *parameter names*, ranged over by x, y, $\cdots$, that contains the special name this. We write $\bar{C}$ as a shorthand for $C_1, \cdots, C_n$ and similarly for the other names. Sequences $C_1\ f_1, \cdots, C_n\ f_n$ are abbreviated as with $\bar{C}\ \bar{f}$.

The abstract syntax of *class declarations* CL, *method declarations* M, and *expressions* e of FJg is the following

$$
\begin{array}{rcl}
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \bar{\mathtt{M}}\} \\
\mathtt{M} & ::= & \mathtt{C\ m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return\ e\ ;}\ \} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{this.f} \mid \mathtt{this.f = e} \mid \mathtt{e!m}(\bar{\mathtt{e}}) \mid \mathtt{new\ C}(\bar{\mathtt{e}}) \mid \mathtt{e;\ e} \\
 & \mid & \mathtt{newg\ C}(\bar{\mathtt{e}}) \mid \mathtt{e.get} \mid \mathtt{e.await}
\end{array}
$$

Sequences of field declarations $\bar{C}\ \bar{f}$, method declarations $\bar{M}$, and parameter declarations $\bar{C}\ \bar{x}$ are assumed to contain no duplicate names.

A program is a pair (CT, e), where the *class table* CT is a finite mapping from class names to class declarations CL and e is an expression. In what follows we always assume a fixed class table CT. According to the syntax, every class has a superclass declared with extends. To avoid circularities, we assume a distinguished class name

Table 2. Lookup auxiliary functions ($\bullet$ is the empty sequence)

Field lookup:

$$fields(\texttt{Object}) = \bullet \qquad \frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad fields(\texttt{D}) = \bar{\texttt{C}}'\ \bar{\texttt{g}}}{fields(\texttt{C}) = \bar{\texttt{C}}\ \bar{\texttt{f}},\ \bar{\texttt{C}}'\ \bar{\texttt{g}}}$$

Method type lookup:

$$\frac{\begin{array}{c}\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\}\\ \texttt{C}'\ \texttt{m}\ (\bar{\texttt{C}}'\ \bar{\texttt{x}})\{\texttt{return e;}\ \} \in \bar{\texttt{M}}\end{array}}{mtype(\texttt{m}, \texttt{C}) = \bar{\texttt{C}}' \to \texttt{C}'} \qquad \frac{\begin{array}{c}\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\}\\ \texttt{m} \notin \bar{\texttt{M}}\end{array}}{mtype(\texttt{m}, \texttt{C}) = mtype(\texttt{m}, \texttt{D})}$$

Method body lookup:

$$\frac{\begin{array}{c}\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\}\\ \texttt{C}'\ \texttt{m}\ (\bar{\texttt{C}}'\ \bar{\texttt{x}})\{\texttt{return e;}\ \} \in \bar{\texttt{M}}\end{array}}{mbody(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}.\texttt{e}} \qquad \frac{\begin{array}{c}\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\}\\ \texttt{m} \notin \bar{\texttt{M}}\end{array}}{mbody(\texttt{m}, \texttt{C}) = mbody(\texttt{m}, \texttt{D})}$$

Heap lookup functions:

$$\frac{\mathsf{H}(\mathsf{o}) = (\mathsf{C}, \mathsf{G}, [\bar{\texttt{f}} : \bar{\mathsf{v}}])}{class(\mathsf{H}, \mathsf{o}) = \mathsf{C} \qquad group(\mathsf{H}, \mathsf{o}) = \mathsf{G} \qquad field(\mathsf{H}, \mathsf{o}, \texttt{f}_i) = \mathsf{v}_i}$$

Object with no field and method declarations whose definition does not appear in the class table. As usual, $\texttt{class C}\ \{\cdots\}$ abbreviates $\texttt{class C extends Object}\ \{\cdots\}$.

Let *types* T be either class names C or *futures* Fut(C). Let also *fields*(C), *mtype*(m, C), and *mbody*(m, C) [9] be the standard FJ lookup functions that are reported in Table 2. The class table satisfies the following well-formed conditions:

(*i*) $\texttt{Object} \notin dom(\textsc{ct})$;
(*ii*) for every $\texttt{C} \in dom(\textsc{ct})$, $\textsc{ct}(\texttt{C}) = \texttt{class C} \cdots$;
(*iii*) every class name occurring in $\textsc{ct}$ belongs to $dom(\textsc{ct})$;
(*iv*) the least relation <: , called *subtyping relation*, over types T, closed by reflexivity and transitivity and containing

$$\frac{\texttt{C}_1 <: \texttt{C}_2}{\texttt{Fut}(\texttt{C}_1) <: \texttt{Fut}(\texttt{C}_2)} \qquad \frac{\textsc{ct}(\texttt{C}_1) = \texttt{class}\ \texttt{C}_1\ \texttt{extends}\ \texttt{C}_2\ \{\cdots\}}{\texttt{C}_1 <: \texttt{C}_2}$$

is antisymmetric;
(*v*) if $\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\cdots\}$ and $mtype(\texttt{m}, \texttt{C}) = \bar{\texttt{C}}' \to \texttt{C}'$ and $mtype(\texttt{m}, \texttt{D}) = \bar{\texttt{D}}' \to \texttt{D}'$ then $\texttt{C}' <: \texttt{D}'$ and $\bar{\texttt{D}}' <: \bar{\texttt{C}}'$.

It is worth to remark that future types never appear in FJg programs, where types of fields and of methods are always classes. This restriction excludes either to store future values in fields or to invoke methods with future values (that later on may be

2.2 Semantics

Below we use an infinite set of *object names*, ranged over by $\mathsf{o}, \mathsf{o}', \cdots$, an infinite set of *group names*, ranged over by $\mathsf{G}, \mathsf{G}', \cdots$, and an infinite set of *task names*, ranged over

by $\mathtt{t}$, $\mathtt{t}'$, $\cdots$. We assume that the set of group names has a distinguished element $\circlearrowleft$, associated to the expressions irrelevant for the deadlock analysis, such as `this.f`.

`FJg` has an operational semantics that is defined in terms of a transition relation $\longrightarrow$ between *configurations* $\mathsf{H} \Vdash \mathsf{S}$, where H is the *heap* and S is a set of *tasks*. The heap maps (i) objects names $\mathtt{o}$ to tuples $(\mathsf{C}, \mathsf{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}])$ that record their class, their group, and their fields' values; (ii) group names G to either $\bot$ or $\top$ that specify whether the group is unlocked or locked, respectively. We use the standard update operations on heaps $\mathsf{H}[\mathtt{o} \mapsto (\mathsf{C}, \mathsf{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}])]$ and $\mathsf{H}[\mathsf{G} \mapsto \mathtt{l}]$ and on fields $[\bar{\mathtt{f}} : \bar{\mathtt{v}}][\mathtt{f} : \mathtt{v}]$ with the usual meanings. Tasks are tuples $\mathtt{t} :_{\mathtt{o}}^{\mathtt{l}} \mathtt{e}$, where $\mathtt{t}$ is the task name, $\mathtt{o}$ is the object of the task, $\mathtt{l}$ is either $\top$ (if the task owns the group lock) or $\bot$ (if not), and $\mathtt{e}$ is the expression to evaluate. The superscript $\mathtt{l}$ is omitted when it is not relevant. In the semantic clauses, by abuse of the notation, the syntactic category $\mathtt{e}$ also addresses *values*, ranged over by $\mathtt{v}$, which are either object or task names. The set of object and task names in $\mathtt{e}$ is returned by the function *names*($\mathtt{e}$). The same function, when applied to a set of tasks S returns the object, group, and task names in S. The operational semantics also uses

- the *heap lookup functions class*($\mathsf{H}, \mathtt{o}$), *group*($\mathsf{H}, \mathtt{o}$), and *field*($\mathsf{H}, \mathtt{o}, \mathtt{f}_i$) that respectively return the class, the group and the values of i-th field of $\mathtt{o}$ in H (see Table 2);
- *evaluation contexts* E whose syntax is:

$$\begin{array}{rl} \mathsf{E} ::= & [\,] \mid \mathsf{E}!\mathtt{m}(\bar{\mathtt{e}}) \mid \mathtt{this.f} = \mathsf{E} \mid \mathtt{o}!\mathtt{m}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \mid \mathtt{new}\ \mathsf{C}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \\ \mid & \mathtt{newg}\ \mathsf{C}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \mid \mathsf{E}.\mathtt{get} \mid \mathsf{E}.\mathtt{await} \mid \mathsf{E}\,;\ \mathtt{e} \end{array}$$

The preliminary notions are all in place for the definition of the transition relation that is given in Table 3. It is worth to notice that, in every rule, a task moves if it owns the lock of its group. Apart this detail, the operations of update, object creation, and sequence are standard. We therefore focus on the operations about groups and futures. Rule (Invk) defines asynchronous method invocation, therefore the evaluation produces a future reference $\mathtt{t}'$ to the returned value, which may be retrieved by a `get` operation if needed. Rule (Newg) defines `newg` $\mathsf{C}(\bar{\mathtt{v}})$, which creates a new group G' and a new object $\mathtt{o}'$ of that group with fields initialized to $\bar{\mathtt{v}}$. This object is returned and the group G' is unlocked – no task of its is running. It will be locked as soon as a method of $\mathtt{o}'$ is invoked – see (Lock). Rule (Release) models method termination that amounts to store the returned value in the configuration and releasing the group lock. Rule (Get) allows one to retrieve the value returned by a method. Rules (AwaitT) and (AwaitF) model the `await` operation: if the task $\mathtt{t}'$ is terminated – it is paired to a value in the configuration – then `await` is unblocking; otherwise the group lock is released and the task $\mathtt{t}$ is blocked. Rule (Config) has standard premises for avoiding unwanted name matches when lifting local reductions to complete configurations.

The initial configuration of a program (ct, $\mathtt{e}$) is $\mathsf{H} \Vdash \mathtt{t} :_{\mathtt{o}}^{\top} \mathtt{e}[^{\mathtt{o}}/_{\mathtt{this}}]$ where $\mathsf{H} = \mathtt{o} \mapsto (\mathtt{Object}, \mathsf{G}, [\,]), \mathsf{G} \mapsto \top$ (following our previous agreement, the class table is implicit).

Example 1. As an example, we detail the evaluation of the expression `(newg D())!n(newg D()).get`, where the class `D` is defined in Table 1.

$$\begin{array}{lll} \mathsf{H} \Vdash \mathtt{t} :_{\mathtt{o}}^{\top} \mathtt{(newg\ D())!n(newg\ D()).get} & & \\ \quad \longrightarrow \mathsf{H}_1 \Vdash \mathtt{t} :_{\mathtt{o}}^{\top} \mathtt{o1!n(newg\ D()).get} & & (1) \\ \quad \longrightarrow \mathsf{H}_2 \Vdash \mathtt{t} :_{\mathtt{o}}^{\top} \mathtt{o1!n(o2).get} & & (2) \\ \quad \longrightarrow \mathsf{H}_2 \Vdash \mathtt{t} :_{\mathtt{o}}^{\top} \mathtt{t1.get}, \mathtt{t1} :_{\mathtt{o1}}^{\bot} \mathtt{o2!m().get} & & (3) \end{array}$$

Table 3. The transition relation of FJg

(Update)
$$\frac{\mathsf{H}(\mathsf{o}') = (\mathsf{C}, \mathsf{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}])}{\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{o}'.\mathtt{f} = \mathtt{v}] \longrightarrow \mathsf{H}[\mathtt{o}' \mapsto (\mathsf{C}, \mathsf{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}][\mathtt{f} : \mathtt{v}])] \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{v}]}$$

(Invk)
$$\frac{class(\mathsf{H}, \mathtt{o}') = \mathsf{C} \quad mbody(\mathtt{m}, \mathsf{C}) = \bar{\mathtt{x}}.\mathtt{e} \quad \mathtt{t}' \neq \mathtt{t}}{\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{o}'!\mathtt{m}(\bar{\mathtt{v}})] \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{t}'],\ \mathtt{t}' :^{\bot}_{\mathtt{o}'} \mathtt{e}[^{\mathtt{o}'}/_{\mathtt{this}}][^{\bar{\mathtt{v}}}/_{\bar{\mathtt{x}}}]}$$

(New)
$$\frac{group(\mathsf{H}, \mathtt{o}) = \mathsf{G} \quad fields(\mathsf{C}) = \bar{\mathsf{T}}\ \bar{\mathtt{f}} \qquad \mathtt{o}' \notin \mathrm{dom}(\mathsf{H}) \qquad \mathsf{H}' = \mathsf{H}[\mathtt{o}' \mapsto (\mathsf{C}, \mathsf{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}])]}{\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{new}\ \mathsf{C}(\bar{\mathtt{v}})] \longrightarrow \mathsf{H}' \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{o}']}$$

(Newg)
$$\frac{fields(\mathsf{C}) = \bar{\mathsf{T}}\ \bar{\mathtt{f}} \quad \mathtt{o}', \mathsf{G}' \notin \mathrm{dom}(\mathsf{H}) \qquad \mathsf{H}' = \mathsf{H}[\mathtt{o}' \mapsto (\mathsf{C}, \mathsf{G}', [\bar{\mathtt{f}} : \bar{\mathtt{v}}])][\mathsf{G}' \mapsto \bot]}{\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{newg}\ \mathsf{C}(\bar{\mathtt{v}})] \longrightarrow \mathsf{H}' \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{o}']}$$

(Get)
$$\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{t}'.\mathtt{get}],\ \mathtt{t}' :_{\mathtt{o}'} \mathtt{v} \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{v}],\ \mathtt{t}' :_{\mathtt{o}'} \mathtt{v}$$

(AwaitT)
$$\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{t}'.\mathtt{await}],\ \mathtt{t}' :_{\mathtt{o}'} \mathtt{v} \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{t}'],\ \mathtt{t}' :_{\mathtt{o}'} \mathtt{v}$$

(AwaitF)
$$\frac{group(\mathsf{H}, \mathtt{o}) = \mathsf{G}}{\mathsf{H}[\mathsf{G} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{t}'.\mathtt{await}] \longrightarrow \mathsf{H}[\mathsf{G} \mapsto \bot] \Vdash \mathtt{t} :^{\bot}_{\mathtt{o}} \mathsf{E}[\mathtt{t}'.\mathtt{await}]}$$

(Lock)
$$\frac{group(\mathsf{H}, \mathtt{o}) = \mathsf{G} \quad \mathtt{e} \neq \mathtt{v}}{\mathsf{H}[\mathsf{G} \mapsto \bot] \Vdash \mathtt{t} :^{\bot}_{\mathtt{o}} \mathtt{e} \longrightarrow \mathsf{H}[\mathsf{G} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{e}}$$

(Release)
$$\frac{group(\mathsf{H}, \mathtt{o}) = \mathsf{G}}{\mathsf{H}[\mathsf{G} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{v} \longrightarrow \mathsf{H}[\mathsf{G} \mapsto \bot] \Vdash \mathtt{t} :^{\bot}_{\mathtt{o}} \mathtt{v}}$$

(Seq)
$$\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{v}; \mathtt{e} \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{e}$$

(Config)
$$\frac{\mathsf{H} \Vdash \mathsf{S} \longrightarrow \mathsf{H}' \Vdash \mathsf{S}' \qquad (names(\mathsf{S}') \setminus names(\mathsf{S})) \cap names(\mathsf{S}'') = \emptyset}{\mathsf{H} \Vdash \mathsf{S}, \mathsf{S}'' \longrightarrow \mathsf{H}' \Vdash \mathsf{S}', \mathsf{S}''}$$

$$\longrightarrow \mathsf{H}_2[\mathtt{G1} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{o2!m().get} \tag{4}$$
$$\longrightarrow \mathsf{H}_2[\mathtt{G1} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{t2.get}, \mathtt{t2} :^{\bot}_{\mathtt{o2}} \mathtt{newg\ C()} \tag{5}$$
$$\longrightarrow \mathsf{H}_3 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{t2.get}\ , \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{newg\ C()} \tag{6}$$
$$\longrightarrow \mathsf{H}_4 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{t2.get}\ , \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{o3} \tag{7}$$
$$\longrightarrow \mathsf{H}_4 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{o3}\ , \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{o3} \tag{8}$$

where $\mathsf{H} = \mathtt{o} \mapsto (\mathtt{Object}, \mathsf{G}, [\,]), \mathsf{G} \mapsto \top$ $\quad \mathsf{H}_1 = \mathsf{H}[\mathtt{o1} \mapsto (\mathtt{D}, \mathtt{G1}, [\,]), \mathtt{G1} \mapsto \bot]$
$\mathsf{H}_2 = \mathsf{H}_1[\mathtt{o2} \mapsto (\mathtt{D}, \mathtt{G2}, [\,]), \mathtt{G2} \mapsto \bot]$ $\quad \mathsf{H}_3 = \mathsf{H}_2[\mathtt{G1} \mapsto \top, \mathtt{G2} \mapsto \top]$
$\mathsf{H}_4 = \mathsf{H}_3[\mathtt{o3} \mapsto (\mathtt{C}, \mathtt{G3}, [\,]), \mathtt{G3} \mapsto \bot]$

The reader may notice that, in the final configuration, the tasks `t1` and `t2` will terminate one after the other by releasing all the group locks.

3 Deadlocks

We introduce our formal developments about deadlock analysis by discussing a couple of expressions that manifest deadlocks. Let `D` be the class of Table 1 and consider the expression `(newg D())!n(new D()).get`. This expression differs from the one of Example 1 for the argument of the method (now it is `new D()`, before it was `newg`

D()). The computation of (newg D())!n(new D()).get is the same of the one in Example 1 till step (5), replacing the value of H_2 with $H_1[o2 \mapsto (D, G1, [\ ])]$ (o1 and o2 belong to the same group G1). At step (5), the task t2 will indefinitely wait for getting the lock of G1 since t1 will never release it.

Deadlocks may be difficult to discover when they are caused by schedulers' choices. For example, let D′ be the following extension of the class C in Table 1:

```
class D′ extends C {   D′ n(D′ b, D′ c){ return b!p(c);c!p(b);this ;}
                       C p(D′ c){ return (c!m()).get ;}  }
```

and consider the expression (newg D′())!n(newg D′(),new D′()).get. The evaluation of this expression yields the tasks $\mathtt{t} :^{\perp}_{\mathtt{o}} \mathtt{o1}, \mathtt{t1} :^{\perp}_{\mathtt{o1}} \mathtt{o1}, \mathtt{t2} :^{\perp}_{\mathtt{o2}} \mathtt{o3!m().get}, \mathtt{t3} :^{\perp}_{\mathtt{o3}} \mathtt{o2!m().get}$ with $\mathtt{o} \in \mathtt{G}$, $\mathtt{o1}, \mathtt{o3} \in \mathtt{G1}$ and $\mathtt{o2} \in \mathtt{G2}$. If t2 is completed before t3 grabs the lock (or conversely) then no deadlock will be manifested. On the contrary, the above tasks may evolve into $\mathtt{t} :^{\perp}_{\mathtt{o}} \mathtt{o1}, \mathtt{t1} :^{\perp}_{\mathtt{o1}} \mathtt{o1}, \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{t4.get}, \mathtt{t3} :^{\top}_{\mathtt{o3}} \mathtt{t5.get}, \mathtt{t4} :^{\perp}_{\mathtt{o3}} \mathtt{newg\ C()}, \mathtt{t5} :^{\perp}_{\mathtt{o2}} \mathtt{newg\ C()}$ that is a deadlock because neither t4 nor t5 will have any chance to progress.

4 Contracts in FJg

In the following we will consider *plain* FJg programs where methods never return fields nor it is possible to invoke methods with fields in the subject part or in the object part. For example, the expressions (this.f)!m() and x!n(this.f) are not admitted, as well as a method declaration like C p(){ return this.f ; }. (The contract system in Table 4 and 5 will ban not-plain programs.) This restriction simplifies the foregoing formal developments about deadlock analysis; the impact of the restriction on the analysis of deadlocks is discussed in Section 7.

The analysis of deadlocks in FJg uses abstract descriptions of behaviours, called *contracts*, and an inference system for associating contracts to expressions (and methods). (The algorithm taking contracts and returning details about deadlocks is postponed to the next section.) Formally, *contracts* $\gamma, \gamma', \cdots$ are terms defined by the rules:

$$\gamma ::= \varepsilon \ \mid\ \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}});\gamma \ \mid\ \mathtt{C.m}^{\mathtt{g}} : \mathtt{G}(\bar{\mathtt{G}});\gamma \ \mid\ \mathtt{C.m}^{\mathtt{a}} : \mathtt{G}(\bar{\mathtt{G}});\gamma$$

As usual, $\gamma;\varepsilon = \gamma = \varepsilon;\gamma$. When $\bar{\gamma}$ is a tuple of contracts $(\gamma_1, \cdots, \gamma_n)$, $\mathtt{seq}(\bar{\gamma})$ is a shortening for the sequential composition $\gamma_1; \cdots ;\gamma_n$. The sequence γ collects the method invocations inside expressions. In particular, the items of the sequence may be empty, noted ε; or $\mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})$, specifying that the method m of class C is going to be invoked on an object of group G and with arguments of group $\bar{\mathtt{G}}$; or $\mathtt{C.m}^{\mathtt{g}} : \mathtt{G}(\bar{\mathtt{G}})$, a method invocation followed by a get operation; or $\mathtt{C.m}^{\mathtt{a}} : \mathtt{G}(\bar{\mathtt{G}})$, a method invocation followed by an await operation. For example, the contract $\mathtt{C.m{:}G();\ D.n{:}G'()}$ defines two method invocations on groups G and G′, respectively (methods carry no arguments). The contract $\mathtt{C.m{:}G();\ D.n^{g}{:}G'();\ E.p^{a}{:}G''()}$ defines three method invocations on different groups; the second invocation is followed by a get and the third one by an await.

Method contracts, ranged over by $\mathcal{G}, \mathcal{G}', \cdots$, are $\mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$, where $\mathtt{G}, \bar{\mathtt{G}}$ are pairwise different group names – $\mathtt{G}(\bar{\mathtt{G}})$ is the *header* –, G′ is the *returned group*, and γ is a *contract*.

A contract $\mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$ binds the group of the object `this` and the group of the arguments of the method invocation in the sequence γ. The returned group $\mathtt{G}'$ may belong to $\mathtt{G}, \bar{\mathtt{G}}$ or not, that is it may be a new group created by the method. For example, let $\gamma = \mathtt{C.m:G();\ D.n^g:G'();\ E.p^a:G''()}$ in (i) $\mathtt{G}(\mathtt{G}', \mathtt{G}'')\{\gamma\}\ \mathtt{G}''$ and (ii) $\mathtt{G}(\mathtt{G}')\{\gamma\}\ \mathtt{G}''$. In case (i) every group name in γ is *bound* by names in the header. This means that method invocations are bound to either the group name of the caller or to group names of the arguments. This is not the case for (ii), where the third method invocation in its body and the returned group address a group name that is unbound by the header. This means that the method with contract (ii) is creating an object of class E belonging to a new group – called $\mathtt{G}''$ in the body – and is performing the third invocation to a method of this object.

Method contracts are quotiented by the least equivalence $=^\alpha$ identifying two contracts that are equivalent after an injective renaming of (free and bound) group names. For example $\mathtt{G}(\mathtt{G}')\{\mathtt{C.m:G();\ D.n^g:G'();\ E.p^a:G''()}\}\ \mathtt{G}' =^\alpha \mathtt{G_1}(\mathtt{G_2})\{\mathtt{C.m:G_1();\ D.n^g:G_2();\ E.p^a:G''()}\}\ \mathtt{G_2}$. Additionally, since the occurrence of $\mathtt{G}''$ represents an unbound group, writing $\mathtt{G}''$ or any other free group name is the same. That is $\mathtt{G}(\mathtt{G}')\{\mathtt{C.m:G();\ D.n^g:G'();\ E.p^a:G''()}\}\ \mathtt{G}' =^\alpha \mathtt{G_1}(\mathtt{G_2})\{\mathtt{C.m:G_1();\ D.n^g:G_2();\ E.p^a:G_3()}\}\ \mathtt{G_2}$.

Let Γ, called *environment*, be a map from either names to pairs $(\mathtt{T}, \mathtt{G})$ or class and method names, *i.e.* C.m, to terms $\mathtt{G}(\bar{\mathtt{G}}) \to \mathtt{G}'$, called *group types*, where $\mathtt{G}, \bar{\mathtt{G}}, \mathtt{G}'$ are all different from $\eth$. The contract judgement for expressions has the following form and meaning: $\Gamma \vdash \mathtt{e} : (\mathtt{T}, \mathtt{G}), \gamma$ means that the expression e has type T and group G and has contract γ in the environment Γ.

Contract rules for expressions are presented in Tables 4 where,

– in rule (T-Invk) we use the operator *fresh*$(\bar{\mathtt{G}}, \mathtt{G})$ that returns G if $\mathtt{G} \in \bar{\mathtt{G}}$ or a fresh group name otherwise;
– in rules (T-Get) and (T-Await), we use the operator $\between$ defined as follows:

$$\begin{array}{rlrl}
\varepsilon \between \mathtt{await} & = \varepsilon & & \\
(\gamma; \mathtt{C.m}\ \mathtt{G}(\bar{\mathtt{G}})) \between \mathtt{await} & = \gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}}) & (\gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}})) \between \mathtt{await} & = \gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}}) \\
(\gamma; \mathtt{C.m}\ \mathtt{G}(\bar{\mathtt{G}})) \between \mathtt{get} & = \gamma; \mathtt{C.m^g}\ \mathtt{G}(\bar{\mathtt{G}}) & (\gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}})) \between \mathtt{get} & = \gamma; \mathtt{C.m^a}\ \mathtt{G}(\bar{\mathtt{G}})
\end{array}$$

(the other combinations of `get` and `await` are forbidden by the contract system).

The rule (T-Field) associates the group $\eth$ to the expression `this.f`, provided the field f exists. This judgment, together with the premises of (T-Invk) and the assumption that $\eth$ does not appear in $\Gamma(\mathtt{C.m})$, imply that subjects and objects of method invocations cannot be expressions as `this.f`. Apart these constraint, the contract of $\mathtt{e!m}(\bar{\mathtt{e}})$ is as expected, *i.e.* the sequence of the contract of e, plus the one of $\bar{\mathtt{e}}$, with a tailing item $\mathtt{C.m}\ \mathtt{G}(\bar{\mathtt{G}})$. Rules (T-New) and (T-NewG) are almost the same, except the fact that the latter one returns a fresh group name while the former one return the group of `this`. The other rules are standard.

Let $\mathtt{G}(\bar{\mathtt{G}}) \to \mathtt{G}' =^\alpha \mathtt{H}(\bar{\mathtt{H}}) \to \mathtt{H}'$ if and only if $\mathtt{G}(\bar{\mathtt{G}})\{\varepsilon\}\mathtt{G}' =^\alpha \mathtt{H}(\bar{\mathtt{H}})\{\varepsilon\}\mathtt{H}'$. The contract judgements for method declarations, class declarations and programs have the following forms and meanings:

– $\Gamma; \mathtt{C} \vdash \mathtt{D'\ m\ (\bar{D}\ \bar{x})\{return\ e;\}} : \mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$ means that the method $\mathtt{D'\ m\ (\bar{D}\ \bar{x})\{return\ e;\}}$ has method contract $\mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$ in the class C and in the environment Γ;

Table 4. Contract rules of `FJg` expressions

(T-Var)
$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}), \varepsilon$$

(T-Field)
$$\frac{\Gamma \vdash \mathtt{this} : (\mathtt{C},\mathtt{G}), \varepsilon \quad \mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C})}{\Gamma \vdash \mathtt{this.f} : (\mathtt{D}, \eth), \varepsilon}$$

(T-Invk)
$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : (\mathtt{C},\mathtt{G}), \gamma \quad \Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{D}},\bar{\mathtt{G}}), \bar{\gamma} \quad \eth \notin \mathtt{G}\bar{\mathtt{G}} \\ \mathit{mtype}(\mathtt{m},\mathtt{C}) = \bar{\mathtt{C}} \to \mathtt{C}' \quad \bar{\mathtt{D}} <: \bar{\mathtt{C}} \\ \Gamma(\mathtt{C.m}) = \mathtt{G}'(\bar{\mathtt{G}}') \to \mathtt{G}'' \quad \mathtt{G}''' = \mathit{fresh}(\mathtt{G}\bar{\mathtt{G}}, \mathtt{G}''[^{\mathtt{G}\bar{\mathtt{G}}}/_{\mathtt{G}'\bar{\mathtt{G}}'}])\end{array}}{\Gamma \vdash \mathtt{e!m}(\bar{\mathtt{e}}) : (\mathtt{Fut}(\mathtt{C}'), \mathtt{G}'''), \gamma; (\mathtt{seq}(\bar{\gamma})); \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})}$$

(T-New)
$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{this} : (\mathtt{C},\mathtt{G}), \varepsilon \quad \Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{C}},\bar{\mathtt{G}}), \bar{\gamma} \\ \mathit{fields}(\mathtt{C}') = \bar{\mathtt{C}}'\ \bar{\mathtt{f}}' \quad \bar{\mathtt{C}} <: \bar{\mathtt{C}}'\end{array}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}'(\bar{\mathtt{e}}) : (\mathtt{C}', \mathtt{G}), (\mathtt{seq}(\bar{\gamma}))}$$

(T-NewG)
$$\frac{\begin{array}{c}\Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{C}},\bar{\mathtt{G}}), \bar{\gamma} \quad \mathtt{G}\ \mathit{fresh} \\ \mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}'\ \bar{\mathtt{f}}' \quad \bar{\mathtt{C}} <: \bar{\mathtt{C}}'\end{array}}{\Gamma \vdash \mathtt{newg}\ \mathtt{C}(\bar{\mathtt{e}}) : (\mathtt{C}, \mathtt{G}), (\mathtt{seq}(\bar{\gamma}))}$$

(T-Get)
$$\frac{\Gamma \vdash \mathtt{e} : (\mathtt{Fut}(\mathtt{C}),\mathtt{G}), \gamma}{\Gamma \vdash \mathtt{e.get} : (\mathtt{C},\mathtt{G}), \gamma \between \mathtt{get}}$$

(T-Await)
$$\frac{\Gamma \vdash \mathtt{e} : (\mathtt{Fut}(\mathtt{C}),\mathtt{G}), \gamma}{\Gamma \vdash \mathtt{e.await} : (\mathtt{Fut}(\mathtt{C}),\mathtt{G}), \gamma \between \mathtt{await}}$$

(T-Update)
$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{this} : (\mathtt{C},\mathtt{G}), \varepsilon \quad \mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C}) \\ \Gamma \vdash \mathtt{e} : (\mathtt{D}',\mathtt{G}'), \gamma \quad \mathtt{D}' <: \mathtt{D}\end{array}}{\Gamma \vdash \mathtt{this.f} = \mathtt{e} : (\mathtt{D}',\mathtt{G}'), \gamma}$$

(T-Seq)
$$\frac{\Gamma \vdash \mathtt{e} : (\mathtt{T},\mathtt{G}), \gamma \quad \Gamma \vdash \mathtt{e}' : (\mathtt{T}',\mathtt{G}'), \gamma'}{\Gamma \vdash \mathtt{e}\,;\,\mathtt{e}' : (\mathtt{T}',\mathtt{G}'), \gamma;\gamma'}$$

- $\Gamma \vdash \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \bar{\mathtt{M}}\} : \{\bar{\mathtt{m}} \mapsto \bar{\mathcal{G}}\}$ means that the class declaration `C` has contract $\{\bar{\mathtt{m}} \mapsto \bar{\mathcal{G}}\}$ in the environment Γ;
- $\vdash (\textsc{ct}, \mathtt{e}) : \textsc{cct}, (\mathtt{T}, \mathtt{G}), \gamma$ means that the program $(\textsc{ct}, \mathtt{e})$ has *contract class table* CCT and type/group/contract $(\mathtt{T}, \mathtt{G}), \gamma$, where a contract class table maps class names to terms $\{\bar{\mathtt{m}} : \bar{\mathcal{G}}\}$.

Table 5 reports the typing judgments for method and class declarations and for programs. We use the auxiliary function $\mathit{mname}(\bar{\mathtt{M}})$ that returns the sequence of method names in $\bar{\mathtt{M}}$. We also write $m \in \textsc{ct}(\mathtt{C})$ if $\textsc{ct}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \bar{\mathtt{M}}\}$ and $m \in \mathit{mname}(\bar{\mathtt{M}})$. Rule (T-Program) requires that if a subclass overrides a method of a superclass then the two methods must have equal contract. This constraint is expressed by the predicate CCT *is* CT *consistent* defined as follows:

$$\begin{array}{l}\textit{for every}\ \textsc{ct}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\cdots\} : \\ \qquad\qquad m \in \textsc{ct}(\mathtt{C})\ \textit{and}\ m \in \textsc{ct}(\mathtt{D}) \quad \textit{implies} \quad \textsc{cct}(\mathtt{C})(\mathtt{m}) =^{\alpha} \textsc{cct}(\mathtt{D})(\mathtt{m})\end{array}$$

This consistency requirement may be definitely weakened: we defer to future works the issue of studying a sub-contract relation that is correct with respect to class inheritance.

The proof of correctness of the contract system in Tables 4 and 5 requires additional rules that define the contract correctness of (runtime) configurations. These rules are:

(T-Task)
$$\frac{\Gamma \vdash \mathtt{this} : (\mathtt{C},\mathtt{G}), \varepsilon \quad \Gamma \vdash \mathtt{t} : (\mathtt{Fut}(\mathtt{C}),\mathtt{G}'), \varepsilon}{\Gamma \vdash \mathtt{t.get} : (\mathtt{C},\mathtt{G}'), (\mathtt{G},\mathtt{G}')}$$

(T-GetR)
$$\frac{\Gamma \vdash \mathtt{e} : (\mathtt{Fut}(\mathtt{C}),\mathtt{G}), \varepsilon \quad \mathtt{e} \neq \mathtt{t}}{\Gamma \vdash \mathtt{e.get} : (\mathtt{C},\mathtt{G}), \varepsilon}$$

Table 5. Contract rules for method declarations and class declarations

Contractually correct method declaration and class declaration:

(T-METHOD)

$$\frac{\begin{array}{c} mtype(\mathtt{m},\mathtt{C}) = \bar{\mathtt{C}}' \to \mathtt{C}' \qquad mbody(\mathtt{m},\mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e} \\ \bar{\mathtt{G}},\mathtt{G}\ \text{fresh} \quad \Gamma + \bar{\mathtt{x}} : (\bar{\mathtt{C}}',\bar{\mathtt{G}}) + \mathtt{this} : (\mathtt{C},\mathtt{G}) \vdash \mathtt{e} : (\mathtt{T}',\mathtt{G}'),\gamma \quad \mathtt{T}' <: \mathtt{C}' \\ \Gamma(\mathtt{C.m}) =^{\alpha} \mathtt{G}(\bar{\mathtt{G}}) \to \mathtt{G}' \end{array}}{\Gamma;\mathtt{C} \vdash \ \mathtt{C}'\ \mathtt{m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} : \mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'}$$

Contractually correct class and program:

(T-CLASS)

$$\frac{\Gamma;\mathtt{C} \vdash \bar{\mathtt{M}} : \bar{\mathcal{G}}}{\Gamma \vdash \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \ \bar{\mathtt{M}}\} : \{mname(\bar{\mathtt{M}}) \mapsto \bar{\mathcal{G}}\}}$$

(T-PROGRAM)

$$\frac{\begin{array}{c} \mathtt{C} \in dom(\textsc{ct})\ \ \text{implies}\ \ \Gamma \vdash \textsc{ct}(\mathtt{C}) : \textsc{cct}(\mathtt{C}) \\ \textsc{cct}\ \textit{is}\ \textsc{ct}\ \textit{consistent} \\ \mathtt{G}\ \textit{fresh} \quad \Gamma + \mathtt{this} : (\mathtt{Object},\mathtt{G}) \vdash \mathtt{e} : (\mathtt{T},\mathtt{G}),\gamma \end{array}}{\vdash (\textsc{ct},\mathtt{e}) : \textsc{cct}, (\mathtt{T},\mathtt{G}),\gamma}$$

(T-CONFIGURATION)

$$\frac{\begin{array}{c} \mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}} \\ \mathsf{H}(\mathsf{o}) = (\mathtt{C},\mathtt{G},[\bar{\mathtt{f}} : \bar{\mathtt{v}}]) \quad \text{implies} \quad \Gamma \vdash \mathtt{o} : (\mathtt{C},\mathtt{G}),\varepsilon\ \text{and}\ \Gamma \vdash \bar{\mathtt{v}} : \bar{\mathtt{C}}'\ \text{and}\ \bar{\mathtt{C}}' <: \bar{\mathtt{C}} \\ \mathtt{t} :_{\mathsf{o}} \mathtt{e} \in \mathsf{S} \quad \text{implies} \quad \Gamma \vdash \mathtt{t} : (\mathtt{Fut}(\mathtt{D}),\mathtt{G}'),\varepsilon\ \text{and}\ \Gamma \vdash \mathtt{e} : (\mathtt{D},\mathtt{G}'),\gamma\ \text{and}\ \mathsf{o} \in dom(\mathsf{H}) \end{array}}{\Gamma \vdash (\mathsf{H} \Vdash \mathsf{S})}$$

Rule (T-TASK) define contract correctness of runtime expressions as `t.get`. The rule uses contracts extended with terms $(\mathtt{G},\mathtt{G}').\gamma$. While rule (T-GETR) deals with the expression `t.await.get`. It is worth to notice the absence of rules for the runtime expression `t.await`. In fact, the judgment of this expression follows by (T-AWAIT) and the definition of $\varepsilon \between$ `await`.

Theorem 1 (Subject reduction)

1. *If* $\vdash (\textsc{ct},\mathtt{e}) : (\textsc{cct},\gamma)$ *then the initial configuration of* $(\textsc{ct},\mathtt{e})$ *is contractually correct. Namely, there is* Γ *such that* $\Gamma \vdash (\mathsf{H} \Vdash \mathtt{t} :_{\mathsf{o}}^{\top} \mathtt{e}[^{\mathsf{o}}/_{\mathtt{this}}])$*, where* $\Gamma = \mathsf{o} \mapsto (\mathtt{Object},\mathtt{G}), \mathtt{t} \mapsto (\mathtt{Fut}(\mathtt{C}),\mathtt{G})$ *and* $\mathsf{H} = \mathsf{o} \mapsto (\mathtt{Object},\mathtt{G},[\ ]), \mathtt{G} \mapsto \top$.
2. *Let* $\Gamma \vdash (\mathsf{H} \Vdash \mathsf{S})$ *and* $\mathsf{H} \Vdash \mathsf{S} \longrightarrow \mathsf{H}' \Vdash \mathsf{S}'$*. Then there is* Γ' *such that* $\Gamma' \vdash (\mathsf{H}' \Vdash \mathsf{S}')$.

5 Deadlock Analysis in FJg

The contract system in Tables 4 and 5 does not convey any information about deadlocks: it only associates contracts to expressions (and methods). The point is that contracts retain the necessary informations about deadlocks and the analysis may be safely reduced to them, overlooking all the other details. We begin with the formal definition of a deadlock.

Definition 1. *A configuration* $\mathsf{H} \Vdash \mathsf{S}$ *is* deadlocked *if there are* $\mathtt{t}_i$, $\mathtt{o}_i$, E_i, *and* $\mathtt{e}_i$, *with* $1 \leq i \leq n+k$, *such that* $n \geq 1$ *and*

- *every* $1 \leq i \leq n$ *is* $\mathtt{t}_i :^{\top}_{\mathtt{o}_i} \mathsf{E}_i[\mathtt{t}_{\ell_i}.\mathtt{get}]$ *with* $\ell_i \in 1..n+k$ *and*
- *every* $n+1 \leq j \leq n+k$ *is* $\mathtt{t}_j :^{\bot}_{\mathtt{o}_j} \mathtt{e}_j$ *with* $group(\mathsf{H}, \mathtt{o}_j) \in \{group(\mathsf{H}, \mathtt{o}_1), \cdots, group(\mathsf{H}, \mathtt{o}_n)\}$.

A configuration $\mathsf{H} \Vdash \mathsf{S}$ *is* deadlock-free *if, for every* $\mathsf{H} \Vdash \mathsf{S} \longrightarrow^* \mathsf{H}' \Vdash \mathsf{S}'$, $\mathsf{H}' \Vdash \mathsf{S}'$ *is not deadlocked. A program* $(\textsc{ct}, \mathtt{e})$ *is deadlock-free if its initial configuration is deadlock-free.*

It is easy to verify that the programs discussed in Section 3 are not deadlock-free. We observe that a configuration may have a blocked task without retaining any deadlock. This is the case of the process $\mathtt{C.m^g}\ \mathtt{G}()$, where $\mathtt{C.m} : \mathtt{G}()\{\mathtt{C.m^g}(\mathtt{G}')\}\ \mathtt{G}$, that produces an infinite chain of tasks $\mathtt{t}_i :^{\top}_{\mathtt{o}_i} \mathtt{t}_{i+1}.\mathtt{get}$. (The following `dla` algorithm will reject this contract.)

We say that a configuration $\mathsf{H} \Vdash \mathsf{S}$ has a *group-dependency* $(\mathtt{G}, \mathtt{G}')$ if S contains either the tasks $\mathtt{t} :^{\top}_{\mathtt{o}} \mathsf{E}[\mathtt{t}'.\mathtt{get}], \mathtt{t}' :_{\mathtt{o}'} \mathtt{e}$, with $\mathtt{t}'$ retaining or not its group lock, or the tasks $\mathtt{t} :^{\bot}_{\mathtt{o}} \mathtt{e}, \mathtt{t}' :^{\top}_{\mathtt{o}'} \mathsf{E}[\mathtt{t}''.\mathtt{get}]$ (in both cases $\mathtt{e}$ is not a value) and $\mathtt{G} = group(\mathsf{H}, \mathtt{o})$ and $\mathtt{G}' = group(\mathsf{H}, \mathtt{o}')$. A configuration contains a *group-circularity* if the transitive closure of its group-dependencies has a pair $(\mathtt{G}, \mathtt{G})$. The following statement asserts that a group-circularity signals the presence of a sequence of tasks mutually waiting for the release of the group lock of the other.

Proposition 1. *A configuration is deadlocked if and only if it has a* group-circularity.

In the following, sets of dependencies will be noted $\boldsymbol{G}, \boldsymbol{G}', \cdots$. Sequences $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$ are also used and shortened into $\overline{\boldsymbol{G}}$. Let $\boldsymbol{G} \cup (\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n)$ be $\boldsymbol{G} \cup \boldsymbol{G}_1; \cdots; \boldsymbol{G} \cup \boldsymbol{G}_n$. A set $\boldsymbol{G}$ is *not circular*, written $\boldsymbol{G}$: `not-circular`, if the transitive closure of $\boldsymbol{G}$ does not contain any pair $(\mathtt{G}, \mathtt{G})$. The definition of being not circular is extended to sequences $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$, written $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$: `not-circular`, by constraining every $\boldsymbol{G}_i$ to be not circular.

Dependencies between group names are extracted from contracts by the algorithm `dla` defined in Table 6. This algorithm takes an *abstract class contract table* Δ_{CCT}, a group name $\mathtt{G}$ and a contract γ and returns a sequence $\overline{\boldsymbol{G}}$. The abstract class contract table Δ_{CCT} takes a pair class name $\mathtt{C}$/method name $\mathtt{m}$, written $\mathtt{C.m}$, and returns an abstract method contract $(\mathtt{G}, \bar{\mathtt{G}})\boldsymbol{G}$. The map Δ_{CCT} is *the least one* such that

$$\Delta_{\mathrm{CCT}}(\mathtt{C.m}) = (\mathtt{G}, \bar{\mathtt{G}}) \bigcup_{i \in 1..n} \boldsymbol{G}_i \quad \text{if and only if} \quad \begin{array}{l} \textsc{cct}(\mathtt{C})(\mathtt{m}) = \mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}' \\ \text{and } \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma) = \boldsymbol{G}_1; \cdots; \boldsymbol{G}_n \end{array}$$

We notice that Δ_{CCT} is well-defined because: (i) group names in $\textsc{cct}$ are finitely many; (ii) `dla` never introduces new group names; (iii) for every $\mathtt{C.m}$, the element $\Delta_{\mathrm{CCT}}(\mathtt{C.m})$ is a finite lattice where elements have shape $(\mathtt{G}, \bar{\mathtt{G}})\boldsymbol{G}$ and where the greatest set $\boldsymbol{G}$ is the cartesian product of group names in $\textsc{cct}$. Additionally, in order to augment the precision of Δ_{CCT}, we assume that $\textsc{cct}$ satisfies the constraint that, for every $\mathtt{C.m}$ and $\mathtt{D.n}$ such that $\mathtt{C.m} \neq \mathtt{D.n}$, $\textsc{cct}(\mathtt{C})(\mathtt{m})$ and $\textsc{cct}(\mathtt{D})(\mathtt{n})$ have no group name in common (both bound and free). (When this is not the case, sets in the codomain of Δ_{CCT} are smaller, thus manifesting more circularities.)

Table 6. The algorithm dla

$$\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \varepsilon) = \emptyset \qquad \frac{\Delta_{\mathrm{CCT}}(\mathtt{C.m}) = (\mathtt{G}'';\bar{\mathtt{G}}'')\boldsymbol{G} \quad \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma') = \overline{\boldsymbol{G}}}{\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \mathtt{C.m}\ \mathtt{G}'(\bar{\mathtt{G}}'); \gamma') = \boldsymbol{G}[^{\mathtt{G}';\bar{\mathtt{G}}'}/_{\mathtt{G}'';\bar{\mathtt{G}}''}] \cup \overline{\boldsymbol{G}}}$$

$$\frac{\Delta_{\mathrm{CCT}}(\mathtt{C.m}) = (\mathtt{G}'';\bar{\mathtt{G}}'')\boldsymbol{G} \quad \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma') = \overline{\boldsymbol{G}}}{\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \mathtt{C.m}^{\mathtt{g}}\ \mathtt{G}'(\bar{\mathtt{G}}'); \gamma') = (\{(\mathtt{G}, \mathtt{G}')\} \cup \boldsymbol{G}[^{\mathtt{G}';\bar{\mathtt{G}}'}/_{\mathtt{G}'';\bar{\mathtt{G}}''}]); \overline{\boldsymbol{G}}}$$

$$\frac{\Delta_{\mathrm{CCT}}(\mathtt{C.m}) = (\mathtt{G}'';\bar{\mathtt{G}}'')\boldsymbol{G} \quad \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma') = \overline{\boldsymbol{G}}}{\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \mathtt{C.m}^{\mathtt{a}}\ \mathtt{G}'(\bar{\mathtt{G}}'); \gamma') = \boldsymbol{G}[^{\mathtt{G}';\bar{\mathtt{G}}'}/_{\mathtt{G}'';\bar{\mathtt{G}}''}]; \overline{\boldsymbol{G}}}$$

Let us comment the rules of Table 6. The second rule of `dla` accounts for method invocations $\mathtt{C.m}\ \mathtt{G}'(\bar{\mathtt{G}}');\gamma'$. Since the code of `C.m` will run asynchronously with respect to the continuation γ', *i.e.* it may be executed at any stage of γ', the rule adds the pairs of `C.m` (stored in $\Delta_{\mathrm{CCT}}(\mathtt{C.m})$) to every set of the sequence corresponding to γ'. The third rule of `dla` accounts for method invocations followed by `get` $\mathtt{C.m}^{\mathtt{g}}\ \mathtt{G}'(\bar{\mathtt{G}}');\gamma'$. Since the code of `C.m` will run *before* the continuation γ', the rule prefixes the sequence corresponding to γ' with the pairs of `C.m` extended with $(\mathtt{G}, \mathtt{G}')$, where `G` is the group of the caller and $\mathtt{G}'$ is the group of the called method. The rule for method invocations followed by `await` is similar to the previous one, except that no pair is added because the `await` operation releases the group lock of the caller.

A program (CT, `e`) is deadlock free if $\vdash (\mathrm{CT}, \mathtt{e}) : \mathrm{CCT}, (\mathtt{T}, \mathtt{G}), \gamma$ and $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma)$: `not-circular`, where `G` is a fresh group name, that is `G` does not clash with group names in CCT (group names in γ are either `G` or fresh as well – see Table 4).

Example 2. Let `C` and `D` be the classes of Table 1 and $\mathtt{D}'$ be the class in Section 3. We derive the following contract class table CCT and abstract contract class table Δ_{CCT}:

```
C.m  ↦ G(){ ε } G'                                C.m  ↦ (G)∅
D.n  ↦ E(E'){D.m^g:E'()} E''                      D.n  ↦ (E,E'){ (E,E') }
D.m  ↦ F(){ ε } F'                                D.m  ↦ (F)∅
D'.n ↦ H(H',H''){D'.p:H'(H''); D'.p:H''(H')} H    D'.n ↦ (H,H',H''){(H',H''),(H'',H')}
D'.p ↦ I(I'){ D'.m^g: I'() } I''                  D'.p ↦ (I,I'){ (I,I') }
D'.m ↦ L() { ε } L'                               D'.m ↦ (L)∅
```

Now consider the expressions `(newg D())!n(newg D()).get` and `(newg D())!n(new D()).get` of Section 2, which have contracts $\mathtt{D.n}^{\mathtt{g}}\mathtt{:L}_2(\mathtt{L}_3)$ and $\mathtt{D.n}^{\mathtt{g}}\mathtt{:L}_2(\mathtt{L}_1)$, respectively, with $\mathtt{L}_1$ being the group of `this`. We obtain $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{L}_1, \mathtt{D.n}^{\mathtt{g}} : \mathtt{L}_2(\mathtt{L}_3)) = \{(\mathtt{L}_2, \mathtt{L}_3), (\mathtt{L}_1, \mathtt{L}_2)\}$ and $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{L}_1, \mathtt{D.n}^{\mathtt{g}} : \mathtt{L}_2(\mathtt{L}')) = \{(\mathtt{L}_2, \mathtt{L}_1), (\mathtt{L}_1, \mathtt{L}_2)\}$ where the first set of dependencies has no group-circularity – therefore `(newg D())!n(newg D()).get` is deadlock-free – while the second has a group-circularity – `(newg D())!n(new D()).get` – may manifest a deadlock, and indeed it does.

Next consider the expression `(newg D'())!n(newg D'(),new D'()).get` of Section 3, which has contract $\mathtt{D'.n}^{\mathtt{g}}\mathtt{:L}''(\mathtt{L}''',\mathtt{L}')$, being $\mathtt{L}'$ the group of `this`. We obtain

$$\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{L}', \mathtt{(newg\ D'())!n(newg\ D'(),new\ D'()).get}) = \{(\mathtt{L}''', \mathtt{L}'), (\mathtt{L}', \mathtt{L}'''), (\mathtt{L}', \mathtt{L}'')\}$$

where the set of dependencies manifests circularities. In fact, in Section 3, we observed that the above expression may manifest a deadlock.

The `dla` algorithm is correct, that is, if its result contains a group-circularity, then the evaluation of the analyzed expression may manifest a deadlock (*vice versa*, if there is no group-circularity then no deadlock will be ever manifested).

Theorem 2. *If* $\vdash$ (CT, e) : (CCT, γ) *and* $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma)$ *is not circular then* (CT, e) *is deadlock-free.*

The algorithm `dla` may be strengthened in several ways. Let us discuss this issue with a couple of examples. Let C′ be the class

```
class C′ { C′ m(C′ b, C′ c){ return b!n(c).get ; c!n(b).get ;}
           C′ n(D′ c){ return (c!p).get ;}
           C′ p() { return new C′() ;}  }
```

and let CCT be its contract class table:

$$\begin{array}{l} \mathtt{C'.m} \mapsto \mathtt{G(G',G'')\{C'.n^g:G'(G'')\ ;\ C'.n^g:G''(G')\}\ G'} \\ \mathtt{C'.n} \mapsto \mathtt{F(F')\{C'.p^g:F'()\}\ F'} \\ \mathtt{C'.p} \mapsto \mathtt{E()\{\epsilon\}\ E} \end{array}$$

The reader may verify that the expression `(new C′())!m1(newg C′(), newg C′())` never deadlocks. However, since $\Delta_{\mathrm{CCT}}(\mathtt{C'.m}) = (\mathtt{G}, \mathtt{G'}, \mathtt{G''})\{(\mathtt{G}, \mathtt{G'}), (\mathtt{G'}, \mathtt{G''}), (\mathtt{G}, \mathtt{G''}), (\mathtt{G''}, \mathtt{G'})\}$, the algorithm `dla` wrongly returns a circular set of dependencies. This problem is due to the fact that Δ_{CCT} melds the group dependencies of different time points into a single set. Had we preserved the temporal separation, that is $\{(\mathtt{G}, \mathtt{G'}), (\mathtt{G'}, \mathtt{G''})\}; \{(\mathtt{G}, \mathtt{G''}), (\mathtt{G''}, \mathtt{G'})\}$, no group-circularity should have been manifested.

The second problem is due to the fact that free group names in method contracts should be renamed each time the method is invoked (with fresh group names) because two invocations of a same method create different group names. On the contrary, the algorithm `dla` always uses the same (free) name. This oversimplification gives a wrong result in this case. Let C″ be `class C″ { C″ m(){ return (newg C″())!m().get ;} }` (with $\mathrm{CCT}(\mathtt{C''.m}) = \mathtt{G()\{C'.m^g : G'()\}G'}$) and consider the expression `(newg C″())!m().get`. The evaluation of this expression never manifests a deadlock, however its contract is $\mathtt{C''.m^g : F()}$ and the algorithm `dla` will return the set $\{(\mathtt{G}, \mathtt{F}), (\mathtt{F}, \mathtt{G'}), (\mathtt{G'}, \mathtt{G'}), \}$, which has a group-circularity. In the conclusions we will discuss the techniques for reducing these errors.

6 Related Works

The notion of grouping objects dates back at least to the mid 80'es with the works of Yonezawa on the language ABCL/1 [8,21]. Since then, several languages have a notion of group for structuring systems, such as Eiffel// [3], Hybrid [17], and ASP [4]. A library for object groups has also been defined for CORBA [7]. In these proposals, a single task is responsible for executing the code inside a group. Therefore it is difficult to model behaviours such as waiting for messages without blocking the group for other activities.

Our FJg calculus is inspired to the language Creol that proposes object groups, called *JCoBoxes*, with multiple cooperatively scheduled tasks [10]. In particular FJg is a subcalculus of JCoBox^c in [19], where the emphasis was the definition of the semantics and the type system of the calculus and the implementation in Java.

The proposals for static analyses of deadlocks are largely based on types (see for instance [11,20] and, for object-oriented programs [1]). In these papers, a type system is defined that computes a partial order of the locks in a program and a subject reduction theorem demonstrates that tasks follow this order. On the contrary, our technique does not computes any ordering of locks, thus being more flexible: a computation may acquire two locks in different order at different stages, thus being correct in our case, but incorrect with the other techniques. A further difference with the above works is that we use contracts that are terms in simple (= with finite states) process algebras [12]. The use of simple process algebras to describe (communication or synchronization) protocols is not new. This is the case of the exchange patterns in SSDL [18], which are based on CSP [2] and the pi-calculus [14], or of the behavioral types in [16] and [5], which use CCS [13]. We expect that finite state abstract descriptions of behaviors can support techniques that are more powerful than the one used in this contribution.

7 Conclusions

We have developed a technique for the deadlock analysis of object groups that is based on abstract descriptions of methods behaviours.

This study can be extended in several directions. One direction is the total coverage of the full language FJg. This is possible by using *group records* $\Theta, \Theta' = \mathtt{G}[\mathtt{f}_1 : \Theta_1, \cdots, \mathtt{f}_k : \Theta_k]$ instead of simple group names. Then contracts such as $\mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})$ become $\mathtt{C.m} : \Theta(\Theta_1, \cdots, \Theta_n)$ and the rule (T-FIELD) is refined into

(T-FIELD-REF)
$$\frac{\Gamma \vdash \mathtt{this} : (\mathtt{C}, \mathtt{G}[\bar{\mathtt{f}} : \overline{\Theta}]), \varepsilon \quad \mathtt{D}\,\mathtt{f} \in \mathit{fields}(\mathtt{C}) \quad \mathtt{f} : \Theta' \in \bar{\mathtt{f}} : \overline{\Theta}}{\Gamma \vdash \mathtt{this.f} : (\mathtt{D}, \Theta'), \varepsilon}$$

The overall effect of this extension is to hinder the notation used in the paper, without conveying any interesting difficulty (for this reason we have restricted our analysis to a sublanguage). We claim that every statement for plain FJg in this paper also hold for full FJg.

A different direction of research is the study of techniques for augmenting the accuracy of the algorithm dla, which is imprecise at the moment. The intent is to use finite state automata with name creation, such as those in [15], and modeling method contracts in terms of finite automata and study deadlocks in sets of these automata.

Other directions address extensions of the language FJg. One of these extensions is the removal of the constraint that future types cannot be used by programmers in FJg. Future types augment the expressivity of the language. For example it is possible to synchroniza several tasks and define *livelocks*:

```
class C { f: Fut(C) ; C m() { return this.f = this!n() ; new C() ;}
                      C n() { return this.f.get ; new C() ;}   }
```

Another extension is about re-entrant method invocations (usually used for tail recursions), which are synchronous invocations. Such extension requires revisions of semantics rules, of the contract rules in Table 4, and of the `dla` algorithm.

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for java. ACM Trans. Program. Lang. Syst. 28 (2006)
2. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM 31, 560–599 (1984)
3. Caromel, D.: Towards a method of object-oriented concurrent programming. Commun. ACM 36(9), 90–102 (1993)
4. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: Proc. POPL 2004, pp. 123–134. ACM Press, New York (2004)
5. Chaki, S., Rajamani, S.K., Rehof, J.: Types as models: model checking message-passing programs. SIGPLAN Not. 37(1), 45–57 (2002)
6. de Boer, F., Clarke, D., Johnsen, E.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
7. Felber, P., Guerraoui, R.: Programming with object groups in corba. IEEE Concurrency 8, 48–58 (2000)
8. Honda, Y., Yonezawa, A.: Debugging concurrent systems based on object groups. In: Gjessing, S., Chepoi, V. (eds.) ECOOP 1988. LNCS, vol. 322, pp. 267–282. Springer, Heidelberg (1988)
9. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23, 396–450 (2001)
10. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and System Modeling 6(1), 39–58 (2007)
11. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)
12. Laneve, C., Padovani, L.: The *must* preorder revisited. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)
13. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1982)
14. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, ii. Inf. and Comput. 100, 41–77 (1992)
15. Montanari, U., Pistore, M.: History-dependent automata: An introduction. In: Bernardo, M., Bogliolo, A. (eds.) SFM-Moby 2005. LNCS, vol. 3465, pp. 1–28. Springer, Heidelberg (2005)
16. Nielson, H.R., Nielson, F.: Higher-order concurrent programs with finite communication topology. In: Proc. POPL 1994, pp. 84–97. ACM Press, New York (1994)
17. Nierstrasz, O.: Active objects in Hybrid. In: Proc. OOPSLA 1987, pp. 243–253 (1987)
18. Parastatidis, S., Webber, J.: MEP SSDL Protocol Framework (April 2005), `http://ssdl.org`
19. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
20. Vasconcelos, V.T., Martins, F., Cogumbreiro, T.: Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In: Proc. PLACES 2009. EPTCS, vol. 17, pp. 95–109 (2009)
21. Yonezawa, A., Briot, J.-P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Proc. OOPSLA 1986, pp. 258–268 (1986)

Monitoring Distributed Systems Using Knowledge

Susanne Graf[1], Doron Peled[2], and Sophie Quinton[3]

[1] VERIMAG, Centre Equation, Avenue de Vignate, 38610 Gières, France
[2] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
[3] Institute of Computer and Network Engineering, 38106 Braunschweig, Germany

Abstract. In this paper, we use knowledge-based control theory to monitor global properties in a distributed system. We control the system to enforce that if a given global property is violated, at least one process knows this fact, and therefore may report it. Our approach uses knowledge properties that are precalculated based on model checking. As local knowledge is not always sufficient to monitor a global property in a concurrent system, we allow adding temporary synchronizations between two or more processes to achieve sufficient knowledge. Since synchronizations are expensive, we aim at minimizing their number using the knowledge analysis.

1 Introduction

The goal of this paper is to transform a distributed system such that it can detect and report violations of invariants. Such properties can be described by a predicate ψ on global states in distributed systems. This may express for example that the overall power in the system is below a certain threshold. When a violation is detected, some activity to adjust the situation may be triggered. There is no global observer that can decide whether a global state violating the given property ψ has been reached. On the other hand, the processes may not have locally enough information to decide this and thus need sometimes to communicate with each other to obtain more information.

Our solution for controlling a system to detect global failure is based on precalculating *knowledge* properties of the distributed system [5,12]. We first calculate in which local states a process has enough information to identify that ψ is violated: in each reachable global state in which ψ becomes false, at least one process must detect this situation. This process may then react, e.g. by informing the other processes or by launching some repair action. Furthermore, we do not want false alarms. Due to the distribute nature of the system, there can be states where firing a transition t would lead to a state in which no process knows (alone) whether ψ has been violated. In that case, additional knowledge is necessary to fire t. We achieve this by adding temporary synchronizations that allow combining the knowledge of a set of processes. To realize at runtime the temporary synchronizations needed to achieve such combined knowledge, as precalculated using the knowledge analysis at compile time, a synchronization

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 183–197, 2011.
© IFIP International Federation for Information Processing 2011

algorithm (such as α-core [14]) is used. To reduce the communication overhead, it is desirable to minimize both the number of additional synchronizations and the number of participants in each synchronization.

This work is related to the knowledge based control method of [2,7]. There, knowledge obtained by model checking is used to control the system in order to enforce some property, which may be a state invariant ψ. Here, we want to control the system to enforce that there is always at least one process with knowledge to detect a violation of such a property as soon as it happens. Controlling the system to *avoid* property violation is a different task from controlling the system to *detect* it. In some cases, controlling for avoidance may lead to restricting the system behavior much more severely than controlling for detection. Note also that for an application such as runtime verification, prevention is not needed while detection is required (e.g., it is acceptable that the temperature raises above its maximal expected level, but whenever this happens, some specific intervention is required).

Monitoring is a simpler task than controlling as it is *nonblocking.* Attempting to enforce a property ψ may result in being blocked in a state where any continuation will violate ψ. This may require strengthening ψ in order not to reach such states, through an expensive global state search, which may consequently imply a severe reduction in nondeterministic choice. This situation does not happen in monitoring; at worst, this may lead to a synchronization between processes.

As an alternative to monitoring one may use *snapshot algorithms* such as those of Chandy and Lamport [4] or Apt and Francez [1]. However, snapshot algorithms only report about some sampled global states. If the property ψ is not stable, that is, if $\psi \Rightarrow \Box\psi$ is not guaranteed, then the fact that ψ has been true at some global state may go undetected.

2 Preliminaries

We represent distributed systems as Petri nets, but the method and algorithms developed here can equally apply to other models, e.g., communicating automata or transition systems.

Definition 1. *A (safe)* Petri net N *is a tuple* (P, T, E, s_0) *where:*

- P *is a finite set of* places. *The set of* states *(markings) is defined as* $S = 2^P$.
- T *is a finite set of* transitions.
- $E \subseteq (P \times T) \cup (T \times P)$ *is a bipartite relation between the places and the transitions.*
- $s_0 \subseteq 2^P$ *is the* initial state *(initial marking).*

Definition 2. *For a transition* $t \in T$*, we define the set of* input places ${}^\bullet t$ *as* $\{p \in P | (p, t) \in E\}$*, and the set of* output places $t^\bullet$ *as* $\{p \in P | (t, p) \in E\}$.

Definition 3. *A transition* t *is* enabled *in a state* s *if* ${}^\bullet t \subseteq s$ *and* $(t^\bullet \setminus {}^\bullet t) \cap s = \emptyset$. *We denote the fact that* t *is enabled from* s *by* $s[t\rangle$.

A state s is in *deadlock* if there is no enabled transition from it.

Definition 4. *The* execution *(*firing*) of a transition t leads from state s to state s', which is denoted by $s[t\rangle s'$, when t is enabled in s and $s' = (s \backslash {}^{\bullet}t) \cup t^{\bullet}$.*

We use the Petri net of Figure 1 as a running example. As usual, transitions are represented as blocks, places as circles, and the relation E as arrows from transitions to places and from places to transitions. The Petri net of Figure 1 has places named $p_1, p_2, \ldots, p_8$ and transitions $a, b, \ldots, e$. We represent a state s by putting *tokens* inside the places of s. In the example of Figure 1, the depicted initial state s_0 is $\{p_1, p_4\}$. The transitions enabled in s_0 are a and b. Firing a from s_0 means removing the token from p_1 and adding one to p_3.

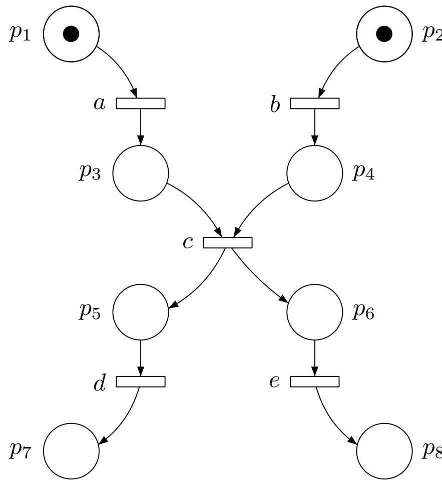


Fig. 1. A Petri net with initial state $\{p_1, p_2\}$

Definition 5. *An* execution *is a maximal (i.e., it cannot be extended) alternating sequence of states and transitions $s_0 t_1 s_1 t_2 s_2 \ldots$ with s_0 the initial state of the Petri net, such that for each state s_i in the sequence with $i > 0$, it holds that $s_{i-1}[t_i\rangle s_i$.*

We denote the set of executions of a Petri net N by $exec(N)$. The set of prefixes of the executions in a set X is denoted by $pref(X)$. A state is *reachable* in N if it appears in at least one execution of N. We denote the set of reachable states of N by $reach(N)$.

A Petri net can be seen as a distributed system, consisting of a set of concurrently executing and temporarily synchronizing processes. There are several options for defining the notion of process in Petri nets: we choose to consider transition sets as processes.

Definition 6. *A* process π *of a Petri net* N *is a subset of the transitions of* N*, i.e.,* $\pi \subseteq T$*.*

We assume a given set of processes Π_N that covers all the transitions of N, i.e., $\bigcup_{\pi \in \Pi_N} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes. The set of processes to which t belongs is denoted $proc(t)$.

In this section, all the notions and notations related to processes extend naturally to sets of processes. Thus, we usually provide definitions directly for sets of processes. Then, when a formula refers to a set of processes Π, we will often replace writing the singleton process set $\{\pi\}$ by writing π instead. The neighborhood of a process π describes the places of the system whose state π can observe.

Definition 7. *The* neighborhood $ngb(\pi)$ *of a process* π *is the set of places* $\bigcup_{t \in \pi} ({}^\bullet t \cup t^\bullet)$*. For a set of processes* Π*,* $ngb(\Pi) = \bigcup_{\pi \in \Pi} ngb(\pi)$*.*

Definition 8. *The* local state *of a set of processes* Π *in a (global) state* $s \in S$ *is defined as* $s|_\Pi = s \cap ngb(\Pi)$*. A local state* s_Π *of* Π *is* part of *a global state* $s \in S$ *if and only if* $s|_\Pi = s_\Pi$*.*

That is, the local state of a process π in a global state s consists of the restriction of s to the neighborhood of π. It describes what π can see of s based on its limited view. In particular, according to this definition, any process π can see whether one of its transitions is enabled. The local state of a set of processes Π containing more than one process is called a *joint* local state.

Definition 9. *Define an equivalence on states* $\equiv_\Pi \subseteq S \times S$ *such that* $s \equiv_\Pi s'$ *when* $s|_\Pi = s'|_\Pi$*.*

Thus, if $t \in \bigcup_{\pi \in \Pi} \pi$ and $s \equiv_\Pi s'$ then $s[t\rangle$ if and only if $s'[t\rangle$.

Figure 2 represents one possible distribution of our running example. We represent processes by drawing dashed lines between them. Here, the left process π_l consists of transitions a, c and d while the right process π_r consists of transitions b, c and e. The neighborhood of π_l contains all the places of the Petri net except p_2 and p_8. The local state $s_0|_{\pi_l}$, part of the initial state $s_0 = \{p_1, p_2\}$ is $\{p_1\}$. Note that the local state $s|_{\pi_l}$, part of $s = \{p_1, p_4\}$ is also $\{p_1\}$, hence $s_0 \equiv_{\pi_l} s$.

We identify properties with the sets of states in which they hold. Formally, a *state property* is a Boolean formula in which places in P are used as atomic predicates. Then, given a state $s \in S$ and a place $p_i \in P$, we have $s \models p_i$ if and only if $p_i \in s$. For a state s, we denote by φ_s the conjunction of the places that are in s and the negated places that are not in s. Thus, φ_s is satisfied by state s and by no other state. A set of states $Q \subseteq S$ can be characterized by a property $\varphi_Q = \bigvee_{s \in Q} \varphi_s$ or any equivalent Boolean formula. For the Petri net of Figure 2, the initial state s is characterized by $\varphi_s = p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge \neg p_7 \wedge \neg p_8$.

Our approach for achieving a local or semi-local decision on which transitions may be fired, while preserving observability, is based on the *knowledge* of processes [5] or sets of processes.

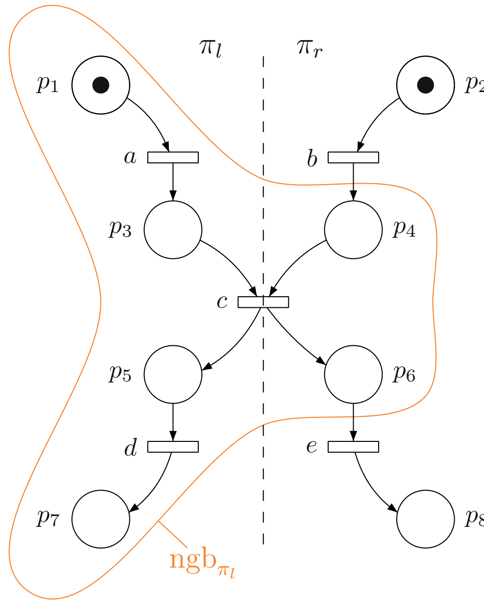


Fig. 2. A distributed Petri net with two processes π_l and π_r

Definition 10. *Given a set of processes* Π *and a property* φ*, we define the property* $K_\Pi\varphi$ *as the set of global states* s *such that for each* reachable s' *with* $s \equiv_\Pi s'$, $s' \models \varphi$. *Whenever* $s \models K_\Pi\varphi$ *for some state* s*, we say that* Π *(jointly)* knows φ in s.

We easily obtain that if $s \models K_\Pi\varphi$ and $s \equiv_\Pi s'$, then $s' \models K_\Pi\varphi$. Hence we can write $s|_\Pi \models K_\Pi\varphi$ rather than $s \models K_\Pi\varphi$ to emphasize that this knowledge property is calculated based on the local state of Π. Given a Petri net and a property φ, one can perform model checking in order to decide whether $s \models K_\Pi\varphi$ for some state s. If Π contains more than one process, we call it *joint* knowledge.

Note that when a process π needs to know and distinguish whether η or μ holds, we write $K_\pi\eta \vee K_\pi\mu$. When we do not need to distinguish between these cases but only need to know whether at least one of them holds, we use the weaker $K_\pi(\eta \vee \mu)$.

3 Knowledge Properties for Monitoring

Our goal is to control the system, i.e., restrict its possible choice of firing transitions, in order to enforce that if a given property ψ becomes false, at least one process knows it. This should interfere minimally with the execution of the system in order to monitor when ψ is violated. To detect the occurrence of a failure, we need the following notion of a "weakest precondition":

Definition 11. *For a given property φ, $wp_t(\varphi)$ is the property such that for any state s, $s \models wp_t(\varphi)$ if and only if $s[t\rangle$ and $s' \models \varphi$ where $s[t\rangle s'$.*

Remember that in a Petri net, there is exactly one state s' such that $s[t\rangle s'$ for a given state s and transition t.

We take into account, with an increasing degree of complication:

- Whether it is allowed to report the same violation of ψ multiple times.
- Whether there exists one or several types of property violation. That is, $\neg\psi$ may be of the form $\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$, where each φ_i represents a certain kind of failure to satisfy ψ. Then, whenever ψ is violated, we may need to identify and report which failure occurred.
- Whether a single transition may cause several failures to occur at the same time (and each one of them needs to be identified).

First, we assume that ψ consists of only one type of failure, and furthermore, that there is no harm in reporting it several times; it is the responsibility of the recovery algorithm to take care of resolving the situation and ignore duplicate reports. For a transition $t \in T$ and a set of processes $\Pi \subseteq proc(t)$, we define a property $\delta(\Pi, t)$ as follows: if t is fired and ψ is falsified by the execution of t, Π will jointly know it. Formally:

$$\delta_1(\Pi, t) = K_\Pi wp_t(\neg\psi) \vee K_\Pi(\neg\psi \vee wp_t(\psi))$$

In other words, $\delta_1(\Pi, t)$ holds if and only if the processes in Π either jointly know that after firing t property ψ will be violated; or they know that firing t cannot make ψ *become* false: either it is already false or it will be true after firing t. Note that the knowledge operators separate the case where $\neg\psi$ will hold in the next state from the other two cases. There is no need to distinguish between the latter two cases, hence we could use the weaker requirement, where both of them appear inside a single knowledge operator $K_\Pi(\neg\psi \vee wp_t(\psi))$.

Now, suppose that we should not report that ψ is violated again, when it was already violated before the execution of t, and therefore has already been or will be reported. This requires strengthening the knowledge:

$$\delta_2(\Pi, t) = K_\Pi(\psi \wedge wp_t(\neg\psi)) \vee K_\Pi(\neg\psi \vee wp_t(\psi))$$

For the case where failure of ψ means one out of several failures $\varphi_1, \ldots, \varphi_n$, but firing one transition cannot cause more than only one particular failure, we need to consider stronger knowledge:

$$\delta_3(\Pi, t) = \bigvee_{i \leq n} K_\Pi(\neg\varphi_i \wedge wp_t(\varphi_i)) \vee \bigwedge_{i \leq n} K_\Pi(\varphi_i \vee wp_t(\neg\varphi_i))$$

Finally, if one single transition may cause multiple failures, we need even stronger knowledge:

$$\delta_4(\Pi, t) = \bigwedge_{i \leq n} (K_\Pi(\neg\varphi_i \wedge wp_t(\varphi_i)) \vee K_\Pi(\varphi_i \vee wp_t(\neg\varphi_i)))$$

Note that in $\delta_j(\Pi, t)$, for $1 \leq j \leq 4$, the left disjunct, when holding, is responsible for identifying the occurrence of the failure, and also for $j \in \{3, 4\}$, identifying its type. In the following, $\delta(\Pi, t)$ stands for $\delta_j(\Pi, t)$, where $1 \leq j \leq 4$.

Definition 12. *A* knowledgeable step *for a set of processes* $\Pi \subseteq \Pi_N$ *is a pair* $(s|_\Pi, t)$ *such that* $s|_\Pi \models \delta(\Pi, t)$ *and there is at least one process* $\pi \in \Pi$ *with* $t \in \pi$.

Note that if all processes synchronize at every step, a violation of Ψ can always be detected as soon as it happens. Of course, the additional synchronizations required to achieve joint knowledge induce some communication overhead, which we have to minimize. We explain how we do this in the next section.

4 Building the Knowledge Table

We use model checking to identify knowledgeable steps following a method similar to [7]. The basic principle of our monitoring policy is the following: a transition t may be fired in a state s if and only if, in addition to its original enabledness condition, $(s|_\pi, t)$ is a knowledgeable step for at least one process π containing t. However, there may be some state in which no individual process has enough knowledge to take a knowledgeable step. In that case, we consider knowledgeable steps for pairs of processes, then triples etc. until we can prove that no deadlock is introduced by the monitoring policy.

The monitoring policy is based on a *knowledge table* Δ which indicates, for a process or a set of processes Π in a given reachable (joint) local state $s|_\Pi$, whether there exists a knowledgeable step $(s|_\Pi, t)$, and then which transition t may thus be safely fired. When building such a table, two issues must be considered: first, the monitoring policy should not introduce deadlocks with respect to the original Petri net N. This means that we have to check that for every reachable non-deadlock global state of N, there is at least one corresponding knowledgeable step in Δ. Second, achieving joint knowledge requires additional synchronization, which induces some communication overhead, as will be explained in the next section. Therefore we must add as few knowledgeable steps involving several processes as possible.

Definition 13. *For a given Petri net* N*, a* knowledge table Δ *is a set of knowledgeable steps.*

To avoid introduce new deadlocks, we require that the table Δ contains enough joint local states to cover all reachable global states. This is done by requiring the following invariant.

Definition 14. *A knowledge table* Δ *is an* invariant *if for each reachable non-deadlock state* s *of* N*, there is at least one (joint) local state in* Δ *that is part of* s*.*

Given a Petri net N and a property ψ, the corresponding knowledge table Δ is built iteratively as follows:

The first iteration includes in Δ, for every process $\pi \in \Pi_N$, all knowledgeable steps $(s|_\pi, t)$ where $s|_\pi$ is a *reachable* local state of π, i.e., it is part of some reachable global state of N. If Δ is an invariant after the first iteration, then taking only knowledgeable steps appearing in Δ does not introduce deadlocks. If Δ is not an invariant, we proceed to a second iteration. Let U be the set of reachable non-deadlock global states s of N for which there is no (joint) local state in Δ that is part of s.

In a second iteration, we add to Δ knowledgeable steps $(s|_{\{\pi,\rho\}}, t)$ such that $s|_{\{\pi,\rho\}}$ is part of some global state in U. For a given local state $s|_{\{\pi,\rho\}}$, all corresponding knowledgeable steps are added together to the knowledge table. The second iteration terminates as soon as Δ becomes an invariant or if all knowledgeable steps for pairs of processes not strictly including knowledgeable steps consisting of single processes have been added to the table.

If Δ is still not an invariant, then we perform further iterations where we consider knowledgeable steps for triples of processes, and so forth. Eventually, Δ becomes an invariant, in the worst case by adding knowledgeable steps involving all processes.

5 Monitoring Using a Knowledge Table

As in [7], we use the knowledge table Δ to control (restrict) the executions of N so as to allow only knowledgeable steps. Formally, this can be represented as an extended Petri net [6,8] N^Δ where processes may have local variables, and transitions have an enabling condition and a data transformation.

Definition 15. *An* extended *Petri net N' consists of (1) a Petri net N (2) a finite set of variables V with given initial values and (3) for each transition $t \in T$, an enabling condition en_t and a transformation predicate f_t on variables in V. In order to fire t, en_t must hold in addition to the usual Petri net enabling condition on the input and output places of t. When t is executed, in addition to the usual changes to the tokens, the variables in V are updated according to f_t.*

A Petri net N' *extends* N if N' is an extended Petri net obtained from N according to Definition 15. The comparison between the original Petri net N and N' extending it is based only on places and transitions. That is, we project out the additional variables.

Lemma 1. *For a given Petri net N and an extension N' of N, we have: $exec(N') \subseteq pref(exec(N))$.*

Proof. The extended Petri net N' strengthens the enabling conditions, thus it can only restrict the executions. However, these restrictions may result in new deadlocks. □

Furthermore, we have the following monotonicity property.

Theorem 1. *Let N be a Petri net and N' an extension of N according to Definition 15 and φ a state predicate for N. If $s \models K_\pi \varphi$ in N, then $s \models K_\pi \varphi$ also in N'.*

Proof. The extended Petri net N' restricts the set of executions, and possibly the set of reachable states, of N. Each local state $s|_\pi$ is part of fewer global states, and thus the knowledge in $s|_\pi$ can only increase. □

The latter lemma and theorem show that the additional variables used to extend a Petri net N define a controller for N.

Definition 16. *Given a Petri net N and a property Ψ, from which a knowledge table Δ has been precalculated, the extended Petri net N^Δ is obtained as follows:*

- *Encode in a set of Boolean variables en_t^Π for $\Pi \subseteq \Pi_N$ and $t \in T$ the knowledge properties calculated in Δ such that en_t^Π is true if and only if $(s|_\Pi, t)$ is a knowledgeable step, where $s|_\Pi$ is the current local state of Π.*
- *Encode in each f_t the update of variables as t is fired and local states modified.*
- *Define each en_t as $\bigvee_{\Pi \subseteq \Pi_N} en_t^\Pi$. That is, t can be fired if at least one set of processes knows that it is part of a knowledgeable step[1].*

In practice, we obtain joint knowledge by adding synchronizations amongst the processes involved. Such synchronizations are achieved by using an algorithm like α-core [14], which allows processes to notify, using asynchronous message passing, a set of coordinators about their wish to be involved in a joint action. This is encoded into the extended Petri net. Once a coordinator has been notified by all the participants in the synchronization it is in charge of, it checks whether some conflicting synchronization is already under way (a process may have notified several coordinators but may not be part of several synchronizations at the same time). If this is not the case, the synchronization takes place. The correctness of the algorithm guarantees the atomic-like behavior of the coordination process, allowing us to reason at a higher level of abstraction where we treat the synchronizations provided by α-core (or any similar algorithm) as transitions that are joint between several participating processes.

Specifically, each process π of N is equipped with a local table Δ_π containing the knowledgeable steps $(s|_\pi, t)$ that appear in the knowledge table Δ and the knowledgeable steps $(s|_\Pi, t)$ such that $\pi \in \Pi$. Before firing a transition in a given local state $s|_\pi$, process π consults its local table Δ_π. It Δ_π contains a knowledgeable step $(s|_\pi, t)$, then π notifies α-core about its wish to *initiate* t, so that the coordinator algorithm will handle potential conflicts with other knowledgeable steps. If Δ_π contains a knowledgeable step $(s|_\Pi, t)$ such that $s|_\pi$ is part of $s|_\Pi$, then π notifies α-core about its wish to achieve joint knowledge through synchronization with the other processes in Π. If the synchronization takes place, then any process in Π and containing t may initiate it. The processes in Π remain synchronized until t has been fired or disabled by some other knowledgeable step.

[1] Note that this condition comes in conjunction with the usual Petri net firing rule based on the places in the neighborhood of t, as in Definition 3.

6 Implementation and Experimental Results

In this section we apply our approach to a concrete example that was implemented in a modified version of the prototype presented in [7]. We have implemented properties δ_1 to δ_3 as defined in Section 3. Property δ_4 is not relevant here because a single transition may never cause multiple failures. The prototype implementation computes the knowledge table Δ based on the local knowledge of processes, as described in Section 4.

The example presented here is a Petri net representing the following scenario: trains enter and exit a train station such as the one represented in Figure 3 (trains that are outside the train station are not represented), evolving between track segments (numbered from 1 to 12). A track segment can accept at most one train at a time, therefore there must be some mechanism to detect and resolve conflicts amongst trains trying to access the same track segment. Trains enter and exit the station on the left, i.e. entry segments are tracks 1 to 4. After entering the station, a train moves from left to right until it reaches one of the platforms (tracks 9 to 12); then it starts moving from right to left until it exits the station on one of the entry segment. A train leaving the station on a given track segment may reenter the station only on this segment.

We monitor a property Ψ which we call absence of *partial gridlock*. A partial gridlock is a situation where some trains are blocking each other at a given intersection. These trains cannot follow their normal schedule and must inform a supervisor process that initiates some specific repair action, e.g. requesting some trains to backtrack. For each intersection where n track segments meet, a partial gridlock is reached when there is one train on each of these segments that is moving toward the intersection. A global state satisfies Ψ if and only if it does not contain any partial gridlock.

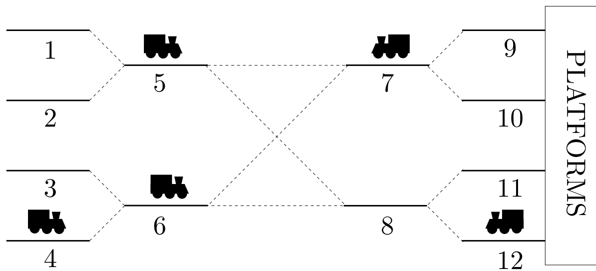


Fig. 3. Train station TS_1

Transitions describe how trains can enter, exit and move within the station. Processes correspond to track segments. That is, the process associated with a segment σ, denoted π_σ, consists of all the transitions involving σ (a train arriving

on σ or leaving it). In particular, this means that transitions corresponding to a train moving from one segment σ_1 to another segment σ_2 belong to π_{σ_1} and π_{σ_2} while transitions representing a train entering or exiting the train station belong to exactly one process namely the entry segment on which the train is entering or leaving. Furthermore, according to the definition of neighborhood, a process π_σ knows if there is a train on segment σ and also on the *neighbors* of σ, i.e., the track segments from which a train may reach σ.

Example 1. Let us first focus on train station TS_1 of Figure 3. A train entering on segment 1 can progress to segment 5 and then segment 7 or 8. From there, it can reach either platform 9 or 10, or platform 11 or 12, respectively. Then, it moves from right to left, until it exits the train station through one of the segments 1 to 4.

Two patterns of partial gridlocks are represented in Figure 4. All possible partial gridlocks of train station TS_1 can be represented by a set of similar patterns. If we consider 6 trains, there are 820,368 reachable global states in this example, of which 11,830 contain a partial gridlock and 48 are global deadlock states.

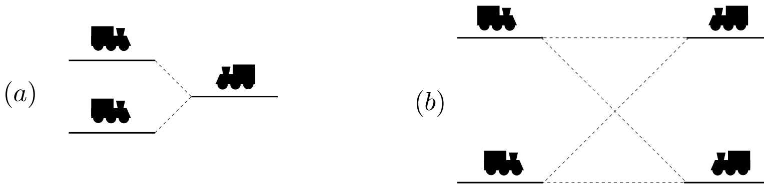


Fig. 4. Two possible partial gridlocks, i.e., violations of Ψ

Interestingly, no additional synchronization is needed to enforce that only knowledgeable steps are taken in this example, independently of the choice of the knowledge property δ_i. The reason for this is twofold. First, partial gridlock (a) of Figure 4 is always detected by the track segment on the right, which knows that there is a train on all three segments. Second, partial gridlock (b) of Figure 4 is not reachable, as we further explain. No additional synchronization is needed. Intuitively, partial gridlock (b) is not reachable for the following reason: whenever train station TS_1 reaches a global state similar to that represented in Figure 3, segment 8 cannot be entered by the train on segment 12. Indeed, this move is a knowledgeable step neither for process π_{12} nor for process π_8, since none of them knows whether this move would introduce a partial gridlock or not. Remember that π_8 does not know whether there is a train or not on segment 7. However, moves from the trains on segments 5 and 6 to 8 are knowledgeable steps for π_8, as π_8 knows they do not introduce any partial gridlock.

Table 1 presents some results about the influence of our monitoring policy on the behavior of the system. The notation NR indicates that some information is irrelevant for the uncontrolled system. A first observation is that the behavior

Table 1. Results for 1000 executions of 10,000 steps

system controlled according to	δ_1	δ_2	δ_3	uncontrolled
states actually reachable	820,096	820,026	820,096	820,368
transitions inhibited	46	99	46	NR (none)
transitions supported	6,856	7,656	6,913	NR (all)
steps to first partial gridlock	117	454	121	56

of the system is not dramatically affected by the monitoring policy: whatever the knowledge property δ_j used to define the monitoring policy, very few global states become unreachable compared to the uncontrolled system. Also, the ratio of supported transitions to inhibited transitions is around 150:1 for δ_1 and δ_3, and 75:1 for δ_2. Finally, the fact that a partial gridlock is reached on average after 56 steps in the uncontrolled system shows that monitoring the system does not drive it, in this example, into states in which the property under study Ψ is violated.

Furthermore, note that δ_1 and δ_3 yield similar results in contrast with δ_2. This is due to the fact that every process knows exactly whether it is creating a given partial gridlock, but it does not always know whether it is creating the first partial gridlock in the system. As a result, the ratio of transitions *inhibited*, that is, transitions enabled but not part of a knowledgeable step, is higher when the system is monitored according to δ_2 than when it is monitored according to δ_1 and δ_3.

Example 2. Figure 5 shows another train station TS_2 and a global reachable non-deadlock state in which there is no knowledgeable step for a single process. As a result, an additional synchronization must be added.

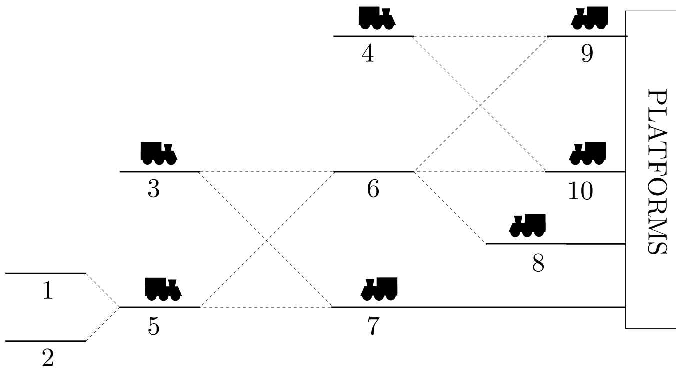


Fig. 5. Train station TS_2

Our experiments on train station TS_2 consider 7 trains. There are 1,173,822 reachable global states in the corresponding Petri net, among which 9,302 contain a partial gridlock. Besides, there are 27 global deadlock states. However, there is only one global reachable non-deadlock state for which there is no corresponding knowledgeable step for a process alone. This state, which we denote s_{dl}, is represented in Figure 5. A synchronization between processes π_3 and π_6 is sufficient to ensure that no deadlock is introduced by the monitoring policy, as there are three transitions t such that (s_{dl}, t) is a knowledgeable step for $\{\pi_3, \pi_6\}$.

We have also performed some experiments to evaluate the number of synchronizations added by the monitoring policy as well as the number of transitions inhibited at runtime. All δ_j yield similar results, so we present them together. Interestingly, the number of synchronizations due to the monitoring policy is very low and although the only progress property that we preserve is deadlock-freedom, few transitions are inhibited in this example. Besides, only 1,972 global states are not actually reachable in the controlled system. Thus, in this example, controlling the system in order to preserve the knowledge about absence of partial gridlock hardly induces any communication overhead and does not alter significantly the behavior of the global system.

Table 2. Results for 100 executions of 10,000 steps

system controlled according to	δ_j for $j \in \{1, 2, 3\}$
states actually reachable	1,171,850
synchronizations	0.01
transitions inhibited	62
transitions supported	18,456

Note that it is sufficient here to add one temporary synchronization between two processes in order to detect that a partial gridlock occurred, whereas knowledge about *absence* of a partial gridlock would require an almost global synchronization. Besides, controlling the system in order to enforce absence of partial gridlocks (instead of monitoring it) would require that processes avoid states in which every possible move leads (inevitably) to a partial gridlock. That is, it requires look-ahead.

7 Conclusion

In this paper, we have proposed an alternative approach to distributed runtime monitoring that guarantees by a combination of monitoring and control (property enforcement) that the fact that some property ψ becomes false is always detected instantaneously when the corresponding transition is fired. Furthermore, there are no "false alarms", that is whenever ψ is detected, it does hold

at least in the state reached at that instant. In other words, we use control as introduced in [2,7,3] to enforce a strong form of local monitorability of ψ, rather than to enforce ψ itself.

We use synchronizations amongst a set of processes, which are realized by a coordinator algorithm such as α-core [14], in order to reliably detect that: either ψ will be false after the transition or the transition does not change the status of ψ. We use model checking to calculate whether (joint) local states have the required knowledge to fire a given transition. We control the system by allowing only such *knowledgeable* steps and we add as many synchronizations as necessary to enforce absence of global deadlocks which do not already appear in the original system.

We have applied this approach to a nontrivial example, showing the interest of enforcing some knowledge about the property instead of the property itself.

References

1. Apt, K., Francez, N.: Modeling the Distributed Termination Convention of CSP. ACM Trans. Program. Lang. Syst. 6(3), 370–379 (1984)
2. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority Scheduling of Distributed Systems based on Model Checking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 79–93. Springer, Heidelberg (2009)
3. Bensalem, S., Bozga, M., Graf, S., Peled, D., Quinton, S.: Methods for Knowledge Based Controlling of Distributed Systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 52–66. Springer, Heidelberg (2010)
4. Chandy, K., Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Trans. Comput. Syst. 3(1), 63–75 (1985)
5. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
6. Genrich, H.J., Lautenbauch, K.: System Modeling with High-level Petri Nets. Theoretical Computer Science 13, 109–135 (1981)
7. Graf, S., Peled, D., Quinton, S.: Achieving Distributed Control through Model Checking. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 396–409. Springer, Heidelberg (2010)
8. Keller, R.M.: Formal Verification of Parallel Programs. Communications of the ACM 19, 371–384 (1976)
9. Havelund, K., Rosu, G.: Monitoring Java Programs with Java PathExplorer. Electr. Notes Theor. Comput. Sci. 55(2) (2001)
10. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. STTT 6(2), 158–173 (2004)
11. Katz, G., Peled, D.: Code Mutation in Verification and Automatic Code Generation. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 435–450. Springer, Heidelberg (2010)
12. van der Meyden, R.: Common Knowledge and Update in Finite Environment. Information and Computation 140(2), 115–157 (1998)
13. Orlin, J.: Contentment in Graph Theory: Covering Graphs with Cliques. Indagationes Mathematicae 80(5), 406–424 (1977)

14. Pérez, J., Corchuelo, R., Toro, M.: An Order-based Algorithm for Implementing Multiparty Synchronization. Concurrency - Practice and Experience 16(12), 1173–1206 (2004)
15. Thistle, J.G.: Undecidability in Decentralized Supervision. System and Control Letters 54, 503–509 (2005)
16. Thomas, W.: On the Synthesis of Strategies in Infinite Games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)
17. Tripakis, S.: Undecidable Problems of Decentralized Observation and Control on Regular Languages. Information Processing Letters 90(1), 21–28 (2004)

Global State Estimates for Distributed Systems

Gabriel Kalyon[1], Tristan Le Gall[2], Hervé Marchand[3], and Thierry Massart[1,⋆]

[1] Université Libre de Bruxelles (U.L.B.), Campus de la Plaine, Bruxelles, Belgique
[2] CEA LIST, LMeASI, boîte 94, 91141 Gif-sur-Yvette, France
[3] INRIA, Rennes - Bretagne Atlantique, France

Abstract. We consider distributed systems modeled as *communicating finite state machines* with reliable unbounded FIFO channels. As an essential subroutine for control, monitoring and diagnosis applications, we provide an algorithm that computes, during the execution of the system, an estimate of the current global state of the distributed system for each local subsystem. This algorithm does not change the behavior of the system; each subsystem only computes and records a symbolic representation of the state estimates, and piggybacks some extra information to the messages sent to the other subsystems in order to refine their estimates. Our algorithm relies on the computation of reachable states. Since the reachability problem is undecidable in our model, we use abstract interpretation techniques to obtain regular overapproximations of the possible FIFO channel contents, and hence of the possible current global states. An implementation of this algorithm provides an empirical evaluation of our method.

1 Introduction

During the execution of a computer system, the knowledge of its global state may be crucial information, for instance to control which action can or must be done, to monitor its behavior or perform some diagnostic. Distributed systems, are generally divided into two classes, depending on whether the communication between subsystems is *synchronous* or not. When the synchrony hypothesis [1] can be made, each local subsystem can easily know, at each step of the execution, the global state of the system (assuming that there is no internal action). When considering *asynchronous* distributed systems, this knowledge is in general impossible, since the communication delays between the components of the system must be taken into account. Therefore, each local subsystem can *a priori* not immediately know either the local state of the other subsystems or the messages that are currently in transfer. In this paper, we consider the asynchronous framework where a system is composed of n subsystems that asynchronously communicate through reliable *unbounded* FIFO channels and modeled by *communicating finite state machines* (CFSM) [3]. This model appears to be essential for concurrent systems in which components cooperate via asynchronous message passing through unbounded buffers (they are e.g. widely used to model communication protocols). We thus assume that the distributed system is already built and the architecture of communication between the different subsystems is fixed. Our aim is to provide an algorithm that allows

⋆ This work has been done in the MoVES project (P6/39), part of the IAP-Phase VI Interuniversity Attraction Poles Programme funded by the Belgian State, Belgian Science Policy.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 198–212, 2011.
© IFIP International Federation for Information Processing 2011

us to compute, in each subsystem of a distributed system $\mathcal{T}$, an estimate of the current state of $\mathcal{T}$. More precisely, each subsystem or a local associated *estimator* computes a set of possible global states, including the contents of the channels, in which the system $\mathcal{T}$ can be; it can be seen as a particular case of monitoring with partial observation. We assume that the subsystems (or associated estimators) can record their own state estimate and that some extra information can be piggybacked to the messages normally exchanged by the subsystems. Without this additional information, since a local subsystem cannot observe the other subsystems nor the FIFO channel contents, the computed state estimates might be too rough. Our computation is based on the use of the reachability operator, which cannot always be done in the CFSM model for undecidability reasons. Therefore, we rely on the abstract interpretation techniques we presented previously in [13]. They ensure the termination of the computations by overapproximating in a symbolic way the possible FIFO channel contents (and hence the state estimates) by *regular languages*. Computing state estimates is useful in many applications. For example, this information can be used to control the system in order to prevent it from reaching some given forbidden global states [4], or to perform some diagnosis to detect some faults in the system [7,19]. For these two potential applications, a more precise state estimate allows the controller or the diagnoser to take better decisions.

This problem differs from the synthesis problem (see e.g. [15,8,5]) which consists in synthesizing a distributed system (together with its architecture of communication) equivalent to a given specification. It also differs from the methodology described in [9] where the problem is to infer from a distributed observation of a distributed system (modeled by a High Level Message Sequence Chart) the set of sequences that explains this observation. It is also different from *model checking* techniques [2,10] that proceed to a symbolic exploration of all the possible states of the system, without running it. We however use the same symbolic representation of queue contents as in [2,10]. In [21], Kumar and Xu propose a distributed algorithm which computes an estimate of the current state of a system. Local estimators maintain and update local state estimates from their own observation of the system and information received from the other estimators. In their framework, the local estimators communicate between them through reliable FIFO channels with delays, whereas the system is monolithic and therefore in their case, a global state is simpler than for our distributed systems composed of several subsystems together with communicating FIFO channels. In [20], Tripakis studies the decidability of the existence of controllers such that a set of responsiveness properties is satisfied in a decentralized framework with communication delays between the controllers. This problem is undecidable when there is no communication or when the communication delays are unbounded. He conjectures that the problem is decidable when the communication delays are bounded. See [18,14] for other works dealing with communication (with or without delay) between agents.

Below, in section 2, we define the formalism of *communicating finite state machines*, that we use. We formally define, in section 3, the state estimate mechanisms and the notion of state estimators. In section 4, we provide an algorithm to compute an estimate of the current state of a distributed system and prove its correctness. We explain, in section 5, how the termination of this algorithm is ensured by using abstract interpretation techniques. Section 6 gives some experimental results. Proofs can be found in [11].

2 Communicating Finite State Machines as a Model of the System

We model a distributed system by *communicating finite state machines* [3] which use reliable unbounded FIFO channels (also called *queues*) to communicate. A *global state* in this model is given by the local state of each subsystem together with the content of each FIFO queue. As no bound is given either in the transmission delay, or on the length of the queues, the state space of the system is *a priori* infinite.

Definition 1 (Communicating Finite State Machines). *A* communicating finite state machine *(*CFSM*) $\mathcal{T}$ is defined as a 6-tuple $\langle L, \ell_0, Q, M, \Sigma, \Delta \rangle$, where (i) L is a finite set of locations, (ii) $\ell_0 \in L$ is the initial location, (iii) Q is a set of queues that $\mathcal{T}$ can use, (iv) M is a finite set of messages, (v) $\Sigma \subseteq Q \times \{!, ?\} \times M$ is a finite set of actions, that are either an* output $a!m$ *to specify that the message $m \in M$ is written on the queue $a \in Q$ or an* input $a?m$ *to specify that the message $m \in M$ is read on the queue $a \in Q$, (vi) $\Delta \subseteq L \times \Sigma \times L$ is a finite set of transitions.*

A transition $\langle \ell, i!m, \ell' \rangle$ indicates that when the system moves from the ℓ to ℓ', a message m is added at the end of the queue i. $\langle \ell, i?m, \ell' \rangle$ indicates that, when the system moves from ℓ to ℓ', a message m must be present at the beginning of the queue i and is removed from this queue. Moreover, throughout this paper, we assume that $\mathcal{T}$ is deterministic, meaning that for all $\ell \in L$ and $\sigma \in \Sigma$, there exists at most one location $\ell' \in L$ such that $\langle \ell, \sigma, \ell' \rangle \in \Delta$. For $\sigma \in \Sigma$, $\mathsf{Trans}(\sigma)$ denotes the set of transitions of $\mathcal{T}$ labeled by σ. The occurrence of a transition will be called an *event* and given an event e, δ_e denotes the corresponding transition. The semantics of a CFSM is defined as follows:

Definition 2. *The semantics of a CFSM $\mathcal{T} = \langle L, \ell_0, Q, M, \Sigma, \Delta \rangle$ is given by an infinite Labeled Transition System (LTS) $[\![\mathcal{T}]\!] = \langle X, \boldsymbol{x}_0, \Sigma, \rightarrow \rangle$, where (i) $X \stackrel{\mathbf{def}}{=} L \times (M^*)^{|Q|}$ is the set of states, (ii) $\boldsymbol{x}_0 \stackrel{\mathbf{def}}{=} \langle \ell_0, \epsilon, \ldots, \epsilon \rangle$ is the initial state, (iii) Σ is the set of actions, and (iv) $\rightarrow \stackrel{\mathbf{def}}{=} \bigcup_{\delta \in \Delta} \xrightarrow{\delta} \subseteq X \times \Sigma \times X$ is the transition relation where $\xrightarrow{\delta}$ is defined by:*

$$\frac{\delta = \langle \ell, i!m, \ell' \rangle \in \Delta \quad w_i' = w_i \cdot m}{\langle \ell, w_1, \ldots, w_i, \ldots, w_{|Q|} \rangle \xrightarrow{\delta} \langle \ell', w_1, \ldots, w_i', \ldots, w_{|Q|} \rangle}$$

$$\frac{\delta = \langle \ell, i?m, \ell' \rangle \in \Delta \quad w_i = m \cdot w_i'}{\langle \ell, w_1, \ldots, w_i, \ldots, w_{|Q|} \rangle \xrightarrow{\delta} \langle \ell', w_1, \ldots, w_i', \ldots, w_{|Q|} \rangle}$$

A global state of a CFSM $\mathcal{T}$ is thus a tuple $\langle \ell, w_1, ..., w_{|Q|} \rangle \in X = L \times (M^*)^{|Q|}$ where ℓ is the current location of $\mathcal{T}$ and $w_1, ..., w_{|Q|}$ are finite words on M^* which give the content of the queues in Q. At the beginning, all queues are empty, so the initial state is $\boldsymbol{x}_0 = \langle \ell_0, \varepsilon, \cdots, \varepsilon \rangle$. Given a CFSM $\mathcal{T}$, two states $\boldsymbol{x}, \boldsymbol{x}' \in X$ and an event e, to simplify the notations we sometimes denote $\boldsymbol{x} \xrightarrow{\delta_e} \boldsymbol{x}'$ by $\boldsymbol{x} \xrightarrow{e} \boldsymbol{x}'$. An *execution* of $\mathcal{T}$ is a sequence $\boldsymbol{x}_0 \xrightarrow{e_1} \boldsymbol{x}_1 \xrightarrow{e_2} \ldots \xrightarrow{e_m} \boldsymbol{x}_m$ where $\boldsymbol{x}_i \xrightarrow{e_{i+1}} \boldsymbol{x}_{i+1} \in \rightarrow \forall i \in [0, m-1]$. Given a set of states $Y \subseteq X$, $\mathsf{Reach}_{\Delta'}^{\mathcal{T}}(Y)$ corresponds to the set of states that are reachable in $[\![\mathcal{T}]\!]$ from Y only firing transitions of $\Delta' \subseteq \Delta$ in $\mathcal{T}$. It is defined by $\mathsf{Reach}_{\Delta'}^{\mathcal{T}}(Y) \stackrel{\mathbf{def}}{=} \bigcup_{n \geq 0} (\mathsf{Post}_{\Delta'}^{\mathcal{T}}(Y))^n$ where $(\mathsf{Post}_{\Delta'}^{\mathcal{T}}(Y))^n$ is the nth functional power of $\mathsf{Post}_{\Delta'}^{\mathcal{T}}(Y)$,

defined by: $\mathsf{Post}^{\mathcal{T}}_{\Delta'}(Y) \stackrel{\text{def}}{=} \{\boldsymbol{x}' \in X | \exists \boldsymbol{x} \in Y, \exists \delta \in \Delta' : \boldsymbol{x} \xrightarrow{\delta} \boldsymbol{x}'\}$. Although there is no general algorithm that can exactly compute the reachability set in our setting [3], there exist some techniques that allow us to compute an overapproximation of this set (see section 5). Given a sequence of actions $\overline{\sigma} = \sigma_1 \cdots \sigma_m \in \Sigma^*$ and two states $x, x' \in X$, $x \xrightarrow{\overline{\sigma}} x'$ denotes that the state x' is reachable from x by executing $\overline{\sigma}$.

Asynchronous Product. A distributed system $\mathcal{T}$ is generally composed of several subsystems $\mathcal{T}_i$ ($\forall i \in [1, n]$) acting in parallel. In fact, $\mathcal{T}$ is defined by a CFSM resulting from the asynchronous (interleaved) product of the n subsystems $\mathcal{T}_i$, also modeled by CFSMs. This can be defined through the asynchronous product of two subsystems.

Definition 3. *Given 2 CFSMs $\mathcal{T}_i = \langle L_i, \ell_{0,i}, Q_i, M_i, \Sigma_i, \Delta_i\rangle$ ($i = 1, 2$), their asynchronous product, denoted by $\mathcal{T}_1||\mathcal{T}_2$, is defined by a CFSM $\mathcal{T} = \langle L, \ell_0, Q, M, \Sigma, \Delta\rangle$, where $L \stackrel{\text{def}}{=} L_1 \times L_2$, $\ell_0 \stackrel{\text{def}}{=} \ell_{0,1} \times \ell_{0,2}$, $Q \stackrel{\text{def}}{=} Q_1 \cup Q_2$, $M \stackrel{\text{def}}{=} M_1 \cup M_2$, $\Sigma \stackrel{\text{def}}{=} \Sigma_1 \cup \Sigma_2$, and $\Delta \stackrel{\text{def}}{=} \{\langle\langle \ell_1, \ell_2\rangle, \sigma_1, \langle \ell'_1, \ell_2\rangle\rangle | (\langle \ell_1, \sigma_1, \ell'_1\rangle \in \Delta_1) \wedge (\ell_2 \in L_2)\} \cup \{\langle\langle \ell_1, \ell_2\rangle, \sigma_2, \langle \ell_1, \ell'_2\rangle\rangle | (\langle \ell_2, \sigma_2, \ell'_2\rangle \in \Delta_2) \wedge (\ell_1 \in L_1)\}$.*

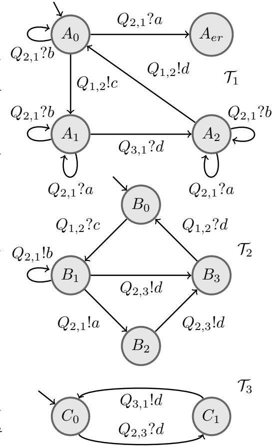
Note that in the previous definition, Q_1 and Q_2 are not necessarily disjoint; this allows the subsystems to communicate between them via common queues. Composing the various subsystems $\mathcal{T}_i$ ($\forall i \in [1, n]$) two-by-two in any order gives the global distributed system $\mathcal{T}$ whose semantics (up to state isomorphism) does not depend on the order.

Definition 4 (Distributed system). *A distributed system $\mathcal{T} = \langle L, \ell_0, Q, M, \Sigma, \Delta\rangle$ is defined by the asynchronous product of n CFSMs $\mathcal{T}_i = \langle L_i, \ell_{0,i}, Q_i, M, \Sigma_i, \Delta_i\rangle$ ($\forall i \in [1, n]$) acting in parallel and exchanging information through FIFO channels.*

Note that a distributed system is also modeled by a CFSM, since the asynchronous product of several CFSMs is a CFSM. In the sequel, a CFSM $\mathcal{T}_i$ always denotes the model of a single process, and a distributed system $\mathcal{T} = \langle L, \ell_0, Q, M, \Sigma, \Delta\rangle$ always denotes the model of the global system, as in Definition 4. Below, unless stated explicitly, $\mathcal{T} = \mathcal{T}_1|| \ldots ||\mathcal{T}_n$ is the considered distributed system.

Communication Architecture of the System. We consider an architecture for the system $\mathcal{T} = \mathcal{T}_1|| \ldots ||\mathcal{T}_n$ defined in Definition 4 with *point-to-point* communication i.e., any subsystem $\mathcal{T}_i$ can send messages to any other subsystem $\mathcal{T}_j$ through a queue $Q_{i,j}$. Thus, only $\mathcal{T}_i$ can write a message m on $Q_{i,j}$ (this is denoted by $Q_{i,j}!m$) and only $\mathcal{T}_j$ can read a message m on this queue (this is denoted by $Q_{i,j}?m$). Moreover, we suppose that the queues are unbounded, that the message transfers between the subsystems are reliable and may suffer from arbitrary non-zero delays, and that no *global clock* or *perfectly synchronized local clocks* are available. With this architecture, the set Q_i of $\mathcal{T}_i$ ($\forall i \in [1, n]$) can be rewritten as $Q_i = \{Q_{i,j}, Q_{j,i} \,|\, (1 \leq j \leq n) \wedge (j \neq i)\}$ and $\forall j \neq i \in [1, n]$, $\Sigma_i \cap \Sigma_j = \emptyset$. Let $\delta_i = \langle \ell_i, \sigma_i, \ell'_i\rangle \in \Delta_i$ be a transition of $\mathcal{T}_i$, $\mathsf{global}(\delta_i) \stackrel{\text{def}}{=} \{\langle\langle \ell_1, \ldots, \ell_{i-1}, \ell_i, \ell_{i+1}, \ldots, \ell_n\rangle, \sigma_i, \langle \ell_1, \ldots, \ell_{i-1}, \ell'_i, \ell_{i+1}, \ldots, \ell_n\rangle\rangle \in \Delta \,|\forall j \neq i \in [1, n] : \ell_j \in L_j\}$ is the set of transitions of Δ that can be built from δ_i in $\mathcal{T}$. We extend this definition to sets of transitions $D \subseteq \Delta_i$ of the subsystem $\mathcal{T}_i$: $\mathsf{global}(D) \stackrel{\text{def}}{=} \bigcup_{\delta_i \in D} \mathsf{global}(\delta_i)$. We abuse notation and write $\Delta \setminus \Delta_i$ instead of $\Delta \setminus \mathsf{global}(\Delta_i)$ to denote the set of transitions of Δ that are not built from Δ_i. Given the set Σ_i of $\mathcal{T}_i$ ($\forall i \in [1, n]$) and the set Σ of $\mathcal{T}$, the projection P_i of Σ onto

Σ_i is standard: $P_i(\varepsilon) = \varepsilon$ and $\forall w \in \Sigma^*$, $\forall a \in \Sigma$, $P_i(wa) = P_i(w)a$ if $a \in \Sigma_i$, and $P_i(w)$ otherwise. The inverse projection P_i^{-1} is defined, for each $L \subseteq \Sigma_i^*$, by $P_i^{-1}(L) = \{w \in \Sigma^* \mid P_i(w) \in L\}$.



Example 1. Let us illustrate the concepts of distributed system and CFSM with our running example depicted on the right hand side. It models a factory composed of three components $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$. The subsystem $\mathcal{T}_2$ produces two kinds of items, a and b, and sends these items to $\mathcal{T}_1$ to finish the job. At reception, $\mathcal{T}_1$ must immediately terminate the process of each received item. $\mathcal{T}_1$ can receive and process b items at any time, but must be in a *turbo mode* to receive and process a items. The subsystem $\mathcal{T}_1$ can therefore be in *normal* mode modeled by the location A_0 or in *turbo* mode (locations A_1 and A_2). In normal mode, if $\mathcal{T}_1$ receives an item a, an error occurs (transition in location A_{er}). Since $\mathcal{T}_1$ cannot always be in turbo mode, a protocol between $\mathcal{T}_1$ and $\mathcal{T}_2$ is imagined. At the beginning, $\mathcal{T}_1$ informs (*connect* action, modeled by $\stackrel{Q_{1,2}!c}{\rightarrow}$) $\mathcal{T}_2$ that it goes in a turbo mode, then $\mathcal{T}_2$ sends a and b items. At the end of a working session, $\mathcal{T}_2$ informs $\mathcal{T}_1$ (*disconnect* action, modeled by $\stackrel{Q_{2,3}!d}{\rightarrow}$) that it has completed its session, so that $\mathcal{T}_1$ can go back in normal mode. This information has to transit through $\mathcal{T}_3$ via queues $Q_{2,3}$ and $Q_{3,1}$, as $\mathcal{T}_3$ must also record this end of session. Since d can be transmitted faster than some items a and b, one can easily find a scenario where $\mathcal{T}_1$ decides to go back to A_0 and ends up in the A_{er} location by reading the message a. Indeed, as $\mathcal{T}_1$ cannot observe the content of the queues, it does not know whether there is a message a in queue $Q_{2,1}$ when it arrives in A_0. This motivates the interest of computing good state estimates of the current state of the system. If each subsystem maintains good estimates of the current state of the system, then $\mathcal{T}_1$ can know whether there is a message a in $Q_{2,1}$, and reach A_0 only if it is not the case.

3 State Estimates of Distributed Systems

We introduce here the framework and the problem we are interested in.

Local View of the Global System. A global state of $\mathcal{T} = \mathcal{T}_1 || \ldots || \mathcal{T}_n$ is given by a tuple of locations (one for each subsystem) and the content of all the FIFO queues. Informally our problem consists in defining one local estimator per subsystem, knowing that each of them can only observe the occurrences of actions of its own local subsystem, such that these estimators compute *online* (i.e., during the execution of the system) estimates of the global state of $\mathcal{T}$. We assume that each local *estimator* $\mathcal{E}_i$ has a precise observation of subsystem $\mathcal{T}_i$, and that the model of the global system is known by all the estimators (i.e., the structure of each subsystem and the architecture of the queues between them). Each estimator $\mathcal{E}_i$ must determine online the *smallest possible set* of global states E_i that contains the actual current global state. Note that if $\mathcal{E}_i$ observes that the location of $\mathcal{T}_i$ is ℓ_i, a very rough state estimate is $L_1 \times \ldots \times \{\ell_i\} \times \cdots \times L_n \times (M^*)^{|Q|}$. In other words all the global states of the system such that location of $\mathcal{T}_i$ is ℓ_i; however, this rough estimate does not provide a very useful information.

Online State Estimates. The estimators must compute the state estimates online. Since each estimator $\mathcal{E}_i$ is local to its subsystem, we suppose that $\mathcal{E}_i$ synchronously observes the actions fired by its subsystem; hence since each subsystem is deterministic, each time an event occurs in the local subsystem, it can immediately infer the new location of $\mathcal{T}_i$ and use this information to define its new state estimate. In order to have better state estimates, we also assume that the estimators can communicate with each other by adding some information (some timestamps and their state estimates) to the messages exchanged by the subsystems. Notice that, due to the communication delay, the estimators cannot communicate synchronously, and therefore the state estimate attached to a message might be out-of-date. A classical way to reduce this uncertainty is to timestamp the messages, e.g., by means of vector clocks (see section 4.1).

Estimates Based on Reachability Sets. Each local estimator maintains a symbolic representation of all global states of the distributed system that are compatible with its observation and with the information it received previously from the other estimators. In section 4.2, we detail the algorithms which update these symbolic representations whenever an event occurs. But first, let us explain the intuition behind the computation of an estimate. We consider the simplest case: the initial state estimate before the system begins its execution. Each FIFO channel is empty, and each subsystem $\mathcal{T}_i$ is in its initial location $\ell_{i,0}$. So the initial global state is known by every estimator $\mathcal{E}_i$. A subsystem $\mathcal{T}_j$ may however start its execution, while $\mathcal{T}_i$ is still in its initial location, and therefore $\mathcal{E}_i$ must thus take into account all the global states that are reachable by taking the transitions of the other subsystems $\mathcal{T}_j$. The initial estimate E_i is this set of reachable global states. This computation of reachable global states also occurs in the update algorithms which take into account any new local event occurred or message received (see section 4.2). The reachability problem is however undecidable for distributed FIFO systems. In section 5, we explain how we overcome this obstacle by using abstract interpretation techniques.

Properties of the Estimators. Estimators may have two important properties: soundness and completeness. Completeness refers to the fact that the current state of the global system is always included in the state estimates computed by each state estimator. Soundness refers to the fact that all states included in the state estimate of $\mathcal{E}_i$ ($\forall i \in [1, n]$) can be reached by one of the sequences of actions that are compatible with the observation of $\mathcal{T}_i$ performed by $\mathcal{E}_i$.

Definition 5 (Completeness and Soundness). *The estimators* $(\mathcal{E}_i)_{i \leq n}$ *are* (i) complete *if and only if, for any execution* $\boldsymbol{x}_0 \xrightarrow{e_1} \boldsymbol{x}_1 \xrightarrow{e_2} \ldots \xrightarrow{e_m} \boldsymbol{x}_m$ *of* $\mathcal{T}$, $\boldsymbol{x}_m \in \bigcap_{i=1}^n E_i$, *and* (ii) sound *if and only if, for any execution* $\boldsymbol{x}_0 \xrightarrow{e_1} \boldsymbol{x}_1 \xrightarrow{e_2} \ldots \xrightarrow{e_m} \boldsymbol{x}_m$ *of* $\mathcal{T}$, $E_i \subseteq \{x' \in X | \exists \overline{\sigma} \in P_i^{-1}(P_i(\sigma_{e_1}.\sigma_{e_2} \ldots \sigma_{e_m})) : \boldsymbol{x}_0 \xrightarrow{\overline{\sigma}} x'\}$ *(*$\forall i \leq n$*) where* σ_{e_k} *(*$\forall k \in [1, m]$*) is the action that labels the transition corresponding to* e_k.

4 Algorithm to Compute the State Estimates

We now present our algorithm that computes estimates of the current state of a distributed system. But first, we recall the notion of *vector clocks* [12], a standard concept that we shall use to compute a more precise state estimates.

4.1 Vector Clocks

To allow the estimators to have a better understanding of the concurrent execution of the distributed system, it is important to determine the causal and temporal relationship between the events that occur in its execution. In a distributed system, events emitted by the same process are ordered, while events emitted by different processes are generally not. When the concurrent processes communicate, additional ordering information can however be obtained. In this case, the communication scheme can be used to obtain a partial order on the events of the system. In practice, vectors of logical clocks, called *vector clocks* [12], can be used to time-stamp the events of the distributed system. The order of two events can then be determined by comparing the value of their respective vector clocks. When these vector clocks are incomparable, the exact order in which the events occur cannot be determined. Vector clocks are formally defined as follows:

Definition 6 (Vector Clocks). *Let* $\langle D, \sqsubseteq \rangle$ *be a partially ordered set, a* vector clock mapping *of width* n *is a function* $V : D \rightarrow \mathbb{N}^n$ *such that* $\forall d_1, d_2 \in D : (d_1 \sqsubseteq d_2) \Leftrightarrow (V(d_1) \leq V(d_2))$.

In general, for a distributed system composed of n subsystems, the partial order on events is represented by a vector clock mapping of width n. The method for computing this vector clock mapping depends on the communication scheme of the distributed system. For CFSMs, this vector clock mapping can be computed by the Mattern's algorithm [16], which is based on the causal and thus temporal relationship between the sending and reception of any message transferred through any FIFO channel. This information is then used to determine a partial order, called *causality (or happened-before) relation* $\prec_c$, on the events of the distributed system. This relation is the smallest transitive relation satisfying the following conditions: (i) if the events $e_i \neq e_j$ occur in the same subsystem $\mathcal{T}_i$ and if e_i comes before e_j in the execution, then $e_i \prec_c e_j$, and (ii) if e_i is an output event occurring in $\mathcal{T}_i$ and if e_j is the corresponding input event occurring in $\mathcal{T}_j$, then $e_i \prec_c e_j$. In Mattern's algorithm [16], each process $\mathcal{T}_i$ ($\forall i \in [1, n]$) has a vector clock $V_i \in \mathbb{N}^n$ of width n and each element $V_i[j]$ ($\forall j \in [1, n]$) is a counter which represents the knowledge of $\mathcal{T}_i$ regarding $\mathcal{T}_j$ and which means that $\mathcal{T}_i$ knows that $\mathcal{T}_j$ has executed at least $V_i[j]$ events. Each time an event occurs in a subsystem $\mathcal{T}_i$, the vector clock V_i is updated to take into account the occurrence of this event (see [16] for details). When $\mathcal{T}_i$ sends a message to some subsystem $\mathcal{T}_j$, this vector clock is piggybacked and allows $\mathcal{T}_j$, after reception, to update its own vector clock. Our state estimate algorithm uses vector clocks and follows Mattern's algorithm, which ensures the correctness of the vector clocks that we use (see section 4.2).

4.2 Computation of State Estimates

Our state estimate algorithm computes, for each estimator $\mathcal{E}_i$ and for each event occurring in the subsystem $\mathcal{T}_i$, a vector clock V_i and a state estimate E_i that contains the current state of $\mathcal{T}$ and any future state that can be reached from this current state by firing actions that do not belong to $\mathcal{T}_i$. This computation obviously depends on the information that $\mathcal{E}_i$ receives. As a reminder, $\mathcal{E}_i$ observes the last action fired by $\mathcal{T}_i$ and can infer the fired transition. $\mathcal{T}_i$ also receives from the other estimators $\mathcal{E}_j$ their state estimate E_j and their vector clock V_j. Our state estimate algorithm proceeds as follows:

Algorithm 1. initialization($\mathcal{T}$)

input : $\mathcal{T} = \mathcal{T}_1 || \dots || \mathcal{T}_n$.
output: The initial state estimate E_i of the estimator $\mathcal{E}_i$ ($\forall i \in [1, n]$).
1 **begin**
2 **for** $i \leftarrow 1$ ***to*** n **do** **for** $j \leftarrow 1$ ***to*** n **do** $V_i[j] \leftarrow 0$
3 **for** $i \leftarrow 1$ ***to*** n **do** $E_i \leftarrow \mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\langle \ell_{0,1}, \dots, \ell_{0,n}, \epsilon, \dots, \epsilon \rangle)$
4 **end**

- When the subsystem $\mathcal{T}_i$ sends a message m to $\mathcal{T}_j$, $\mathcal{T}_i$ attaches the vector clock V_i and the state estimate E_i of $\mathcal{E}_i$ to this message. Next, $\mathcal{E}_i$ receives the action fired by $\mathcal{T}_i$, and infers the fired transition. It then uses this information to update its state estimate E_i.
- When the subsystem $\mathcal{T}_i$ receives a message m from $\mathcal{T}_j$, $\mathcal{E}_i$ receives the action fired by $\mathcal{T}_i$ and the information sent by $\mathcal{T}_j$ i.e., the state estimate E_j and the vector clock V_j of $\mathcal{E}_j$. It computes its new state estimate from these elements.

In both cases, the computation of the new state estimate E_i depends on the computation of reachable states. In this section, we assume that we have an operator that can compute an *approximation* of the reachable states (which is *undecidable* is the CFSM model). We will explain in section 5 how such an operator can be computed effectively.

State Estimate Algorithm. Our algorithm, called *SE-algorithm*, computes estimates of the current state of a distributed system. It is composed of three sub-algorithms: (i) the initialization algorithm, which is only used when the system starts its execution, computes, for each estimator, its initial state estimate (ii) the outputTransition algorithm computes online the new state estimate of $\mathcal{E}_i$ after an output of $\mathcal{T}_i$, and (iii) the inputTransition algorithm computes online the new state estimate of $\mathcal{E}_i$ after an input of $\mathcal{T}_i$.

INITIALIZATION *Algorithm:* According to the Mattern's algorithm [16], each component of the vector V_i is set to 0. To take into account that, before the execution of the first action of $\mathcal{T}_i$, the other subsystems $\mathcal{T}_j$ ($\forall j \neq i \in [1, n]$) could perform inputs and outputs, the initial state estimate of $\mathcal{E}_i$ is given by $E_i = \mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\langle \ell_{0,1}, \dots, \ell_{0,n}, \epsilon, \dots, \epsilon \rangle)$.

Algorithm 2. outputTransition($\mathcal{T}, V_i, E_i, \delta$)

input : $\mathcal{T} = \mathcal{T}_1 || \dots || \mathcal{T}_n$, the vector clock V_i of $\mathcal{E}_i$, the current state estimate E_i of $\mathcal{E}_i$, and a transition $\delta = \langle \ell_1, Q_{i,j}!m, \ell_2 \rangle \in \Delta_i$.
output: The state estimate E_i after the output transition δ.
1 **begin**
2 $V_i[i] \leftarrow V_i[i] + 1$
3 $\mathcal{T}_i$ tags message m with $\langle E_i, V_i, \delta \rangle$ and it writes this tagged message on $Q_{i,j}$
4 $E_i \leftarrow \mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\mathsf{Post}^{\mathcal{T}}_{\delta}(E_i))$
5 **end**

OUTPUT *Algorithm:* Let E_i be the current state estimate of $\mathcal{E}_i$. When $\mathcal{T}_i$ wants to execute a transition $\delta = \langle \ell_1, Q_{i,j}!m, \ell_2 \rangle \in \Delta_i$ corresponding to an output on the queue $Q_{i,j}$, the following instructions are computed to update the state estimate E_i:

- according to the Mattern's algorithm [16], $V_i[i]$ is incremented (i.e., $V_i[i] \leftarrow V_i[i]+1$) to indicate that a new event has occurred in $\mathcal{T}_i$.
- $\mathcal{T}_i$ tags message m with $\langle E_i, V_i, \delta \rangle$ and writes this information on $Q_{i,j}$. The estimate E_i tagging m contains the set of states in which $\mathcal{T}$ can be *before* the execution of δ. The additional information $\langle E_i, V_i, \delta \rangle$ will be used by $\mathcal{T}_j$ to refine its state estimate.
- E_i is updated as follows, to contain the current state of $\mathcal{T}$ and any future state that can be reached from this current state by firing actions that do not belong to $\mathcal{T}_i$: $E_i \leftarrow \mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\mathsf{Post}^{\mathcal{T}}_{\delta}(E_i))$. More precisely, $\mathsf{Post}^{\mathcal{T}}_{\delta}(E_i)$ gives the set of states in which $\mathcal{T}$ can be after the execution of δ. But, after the execution of this transition, $\mathcal{T}_j$ ($\forall j \neq i \in [1, n]$) could read and write on their queues. Therefore, we define the estimate E_i by $\mathsf{Reach}^{\mathcal{T}}_{\Delta \setminus \Delta_i}(\mathsf{Post}^{\mathcal{T}}_{\delta}(E_i))$.

Algorithm 3. $\mathsf{inputTransition}(\mathcal{T}, V_i, E_i, \delta)$

input : $\mathcal{T} = \mathcal{T}_1 || \ldots || \mathcal{T}_n$, the vector clock V_i of $\mathcal{E}_i$, the current state estimate E_i of $\mathcal{E}_i$ and a transition $\delta = \langle \ell_1, Q_{j,i}?m, \ell_2 \rangle \in \Delta_i$. Message m is tagged with the triple $\langle E_j, V_j, \delta' \rangle$ where (i) E_j is the state estimate of $\mathcal{E}_j$ before the execution of δ' by $\mathcal{T}_j$, (ii) V_j is the vector clock of $\mathcal{E}_j$ after the execution of δ' by $\mathcal{T}_j$, and (iii) $\delta' = \langle \ell'_1, Q_{j,i}!m, \ell'_2 \rangle \in \Delta_j$ is the output corresponding to δ.

output: The state estimate E_i after the input transition δ.

```
1  begin
2     \\ We consider three cases to update E_j
3     if V_j[i] = V_i[i] then Temp_1 ← Post^T_δ(Reach^T_{Δ\Δ_i}(Post^T_δ'(E_j)))
4     else if V_j[j] > V_i[j] then
      Temp_1 ← Post^T_δ(Reach^T_{Δ\Δ_i}(Reach^T_{Δ\Δ_j}(Post^T_δ'(E_j))))
5     else Temp_1 ← Post^T_δ(Reach^T_Δ(Post^T_δ'(E_j)))
6     E_i ← Post^T_δ(E_i) \\ We update E_i
7     E_i ← E_i ∩ Temp_1 \\ E_i = update of E_i ∩ update of E_j (i.e., Temp_1)
8     V_i[i] ← V_i[i] + 1
9     for k ← 1 to n do V_i[k] ← max(V_i[k], V_j[k])
10 end
```

INPUT *Algorithm:* Let E_i be the current state estimate of $\mathcal{E}_i$. When $\mathcal{T}_i$ executes a transition $\delta = \langle \ell_1, Q_{j,i}?m, \ell_2 \rangle \in \Delta_i$, corresponding to an input on the queue $Q_{j,i}$, it also reads the information $\langle E_j, V_j, \delta' \rangle$ (where E_j is the state estimate of $\mathcal{E}_j$ before the execution of δ' by $\mathcal{T}_j$, V_j is the vector clock of $\mathcal{E}_j$ after the execution of δ' by $\mathcal{T}_j$, and $\delta' = \langle \ell'_1, Q_{j,i}!m, \ell'_2 \rangle \in \Delta_j$ is the output corresponding to δ) tagging m, and the following operations are performed to update E_i:

- we update the state estimate E_j of $\mathcal{E}_j$ (this update is denoted by $Temp_1$) by using the vector clocks to guess the possible behaviors of $\mathcal{T}$ between the execution of the transition δ' and the execution of δ. We consider three cases:

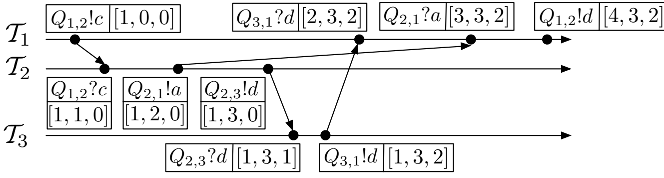


Fig. 1. An execution of the running example

- if $V_j[i] = V_i[i]$: $Temp_1 \leftarrow \mathsf{Post}_{\delta}^{\mathcal{T}}(\mathsf{Reach}_{\Delta\setminus\Delta_i}^{\mathcal{T}}(\mathsf{Post}_{\delta'}^{\mathcal{T}}(E_j)))$. In this case, thanks to the vector clocks, we know that $\mathcal{T}_i$ has executed no transition between the execution of δ' by $\mathcal{T}_j$ and the execution of δ by $\mathcal{T}_i$. Thus, only transitions in $\Delta \setminus \Delta_i$ could have occurred during this period. We then update E_j as follows. We compute (i) $\mathsf{Post}_{\delta'}^{\mathcal{T}}(E_j)$ to take into account the execution of δ' by $\mathcal{T}_j$, (ii) $\mathsf{Reach}_{\Delta\setminus\Delta_i}^{\mathcal{T}}(\mathsf{Post}_{\delta'}^{\mathcal{T}}(E_j))$ to take into account the transitions that could occur between the execution of δ' and the execution of δ, and (iii) $\mathsf{Post}_{\delta}^{\mathcal{T}}(\mathsf{Reach}_{\Delta\setminus\Delta_i}^{\mathcal{T}}(\mathsf{Post}_{\delta'}^{\mathcal{T}}(E_j)))$ to take into account the execution of δ.
- else if $V_j[j] > V_i[j]$: $Temp_1 \leftarrow \mathsf{Post}_{\delta}^{\mathcal{T}}(\mathsf{Reach}_{\Delta\setminus\Delta_i}^{\mathcal{T}}(\mathsf{Reach}_{\Delta\setminus\Delta_j}^{\mathcal{T}}(\mathsf{Post}_{\delta'}^{\mathcal{T}}(E_j))))$. Indeed, in this case, we can prove (see Theorem 1) that if we reorder the transitions executed between the occurrence of δ' and the occurrence of δ in order to execute the transitions of Δ_i before the ones of Δ_j, we obtain a correct update of E_i. Intuitively, this reordering is possible, because there is no causal relation between the events of $\mathcal{T}_i$ and the events of $\mathcal{T}_j$, that have occurred between δ' and δ. So, in this reordered sequence, we know that, after the execution of δ, only transitions in $\Delta \setminus \Delta_j$ could occur followed by transitions in $\Delta \setminus \Delta_i$.
- else $Temp_1 \leftarrow \mathsf{Post}_{\delta}^{\mathcal{T}}(\mathsf{Reach}_{\Delta}^{\mathcal{T}}(\mathsf{Post}_{\delta'}^{\mathcal{T}}(E_j)))$. Indeed, in this case, the vector clocks do not allow us to deduce information regarding the behavior of $\mathcal{T}$ between the execution of δ' and the execution of δ. Therefore, to have a correct state estimate, we update E_j by taking into account all the possible behaviors of $\mathcal{T}$ between the execution of δ' and the execution of δ.

- we update the estimate E_i to take into account the execution of δ: $E_i \leftarrow \mathsf{Post}_{\delta}^{\mathcal{T}}(E_i)$.
- we intersect $Temp_1$ and E_i to obtain a better state estimate: $E_i \leftarrow E_i \cap Temp_1$.
- according to the Mattern's algorithm [16], the vector clock V_i is incremented to take into account the execution of δ and subsequently is set to the component-wise maximum of V_i and V_j. This last operation allows us to take into account the fact that any event that precedes the sending of m should also precede the occurrence of δ.

Example 2. We illustrate SE-algorithm with a sequence of actions of our running example depicted in Figure 1 (the vector clocks are given in the figure). A state of the global system is denoted by $\langle \ell_1, \ell_2, \ell_3, w_{1,2}, w_{2,1}, w_{2,3}, w_{3,1}\rangle$ where ℓ_i is the location of $\mathcal{T}_i$ (for $i = 1, 2, 3$) and $w_{1,2}$, $w_{2,1}$, $w_{2,3}$ and $w_{3,1}$ denote the content of the queues $Q_{1,2}$, $Q_{2,1}$, $Q_{2,3}$ and $Q_{3,1}$. At the beginning of the execution, the state estimates of the three subsystems are (i) $E_1 = \{\langle A_0, B_0, C_0, \epsilon, \epsilon, \epsilon, \epsilon\rangle\}$, (ii) $E_2 = \{\langle A_0, B_0, C_0, \epsilon, \epsilon, \epsilon, \epsilon\rangle, \langle A_1, B_0, C_0, c, \epsilon, \epsilon, \epsilon\rangle\}$, and (iii) $E_3 = \{\langle A_0, B_0, C_0, \epsilon, \epsilon, \epsilon, \epsilon\rangle, \langle A_1, B_0, C_0, c, \epsilon, \epsilon, \epsilon\rangle, \langle A_1, B_1, C_0, \epsilon, b^*, \epsilon, \epsilon\rangle, \langle A_1, B_2, C_0, \epsilon, b^*(a+\epsilon), \epsilon, \epsilon\rangle, \langle A_1, B_3, C_0, \epsilon, b^*(a+\epsilon), d, \epsilon\rangle\}$.

After the first transition $\langle A_0, Q_{1,2}!c, A_1\rangle$, the state estimate of $\mathcal{E}_1$ is not really precise, because a lot of events may have happened without the estimator $\mathcal{E}_1$ being informed: $E_1 = \{\langle A_1, B_0, C_0, c, \epsilon, \epsilon, \epsilon\rangle, \langle A_1, B_1, C_0, \epsilon, b^*, \epsilon, \epsilon\rangle, \langle A_1, B_2, C_0, \epsilon, b^*a, \epsilon, \epsilon\rangle, \langle A_1, B_3, C_0, \epsilon, b^*(a+\epsilon), d, \epsilon\rangle, \langle A_1, B_3, C_1, \epsilon, b^*(a+\epsilon), \epsilon, \epsilon\rangle, \langle A_1, B_3, C_0, \epsilon, b^*(a+\epsilon), \epsilon, d\rangle\}$. But, after the second transition $\langle B_0, Q_{1,2}?c, B_1\rangle$, $\mathcal{E}_2$ has an accurate state estimate: $E_2 = \{\langle A_1, B_1, C_0, \epsilon, \epsilon, \epsilon, \epsilon\rangle\}$. We skip a few steps and consider the state estimates before the sixth transition $\langle C_1, Q_{3,1}!d, C_0\rangle$: E_1 is still the same, because $\mathcal{T}_1$ did not perform any action, $E_3 = \{\langle A_1, B_3, C_1, \epsilon, b^*(a+\epsilon), \epsilon, \epsilon\rangle\}$, and we do not indicate E_2, because $\mathcal{T}_2$ is no longer involved. When $\mathcal{T}_3$ sends the message d to $\mathcal{T}_1$ (the transition $\langle C_1, Q_{3,1}!d, C_0\rangle$), it attaches E_3 to this message. When $\mathcal{T}_1$ reads this message, it computes $E_1 = \{\langle A_2, B_3, C_0, \epsilon, b^*(a+\epsilon), \epsilon, \epsilon\rangle\}$ and when it reads the message a, it updates E_1: $E_1 = \{\langle A_2, B_3, C_0, \epsilon, b^*, \epsilon, \epsilon\rangle\}$. Thus, $\mathcal{E}_1$ knows, after this action, that there is no a in $Q_{2,1}$, and that after writing d on $Q_{1,2}$, it cannot reach A_{er} from A_0. This example shows the importance of knowing the content of the queues as without this knowledge, $\mathcal{E}_1$ may think that there is an a in $Q_{2,1}$, so an error might occur if $\langle A_2, Q_{1,2}!d, A_0\rangle$ is enabled. $\diamond$

Properties. As explained above, we assume that we can compute an approximation of the reachable states. In this part, we present the properties of our state estimate algorithm w.r.t. the kind of approximations that we use.

Theorem 1. *SE-algorithm is complete, if the* Reach *operator computes an overapproximation of the reachable states.*

Theorem 2. *SE-algorithm is sound, if the* Reach *operator computes an underapproximation of the reachable states.*

The proofs of these theorems are given in [11]. If we compute an underapproximation of the reachable states, our state estimate algorithm is not complete. If we compute an overapproximation of the reachable states, our state estimate algorithm is not sound. So, depending on the approximations, our algorithm is either complete or sound. Completeness is a more important property, because it ensures that the computed state estimates always contain the current global state. Therefore, in section 5, we define an effective algorithm for the state estimate problem by computing overapproximations of the reachable states. Finally, note that our method proposes that we only add information to existing transmitted messages. We can show that increasing the information exchanged between the estimators (for example, each time an estimator computes a new state estimate, this estimate is sent to all the other estimators) improves their state estimate. This can be done only if the channels and the subsystems can handle this extra load.

5 Effective Computation of State Estimates by Means of Abstract Interpretation

The algorithm described in the previous section requires the computation of reachability operators, which cannot always be computed exactly in general. In this section, we overcome this obstacle by using abstract interpretation techniques (see e.g. [6,13]) to compute, in a finite number of steps, an overapproximation of the reachability operators and thus of the state estimates E_i.

Computation of Reachability Sets by the Means of Abstract Interpretation. For a given set of global states $X' \subseteq X$ and a given set of transitions $\Delta' \subseteq \Delta$, the states reachable from X' can be characterized by the least fixpoint: $\mathsf{Reach}^{\mathcal{T}}_{\Delta'}(X') = \mu Y.X' \cup \mathsf{Post}^{\mathcal{T}}_{\Delta'}(Y)$. Abstract interpretation provides a theoretical framework to compute efficient overapproximation of such fixpoints. The concrete domain (i.e., the sets of states 2^X), is substituted by a simpler abstract domain Λ, linked by a *Galois Connection* $2^X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \Lambda$ [6], where α (resp. γ) is the abstraction (resp. concretization) function. The fixpoint equation is transposed into the abstract domain: $\lambda = F^{\sharp}_{\Delta'}(\lambda)$, with $\lambda \in \Lambda$ and $F^{\sharp}_{\Delta'} \sqsupseteq \alpha \circ F_{\Delta'} \circ \gamma$. In this setting, a standard way to ensures that the fixpoint computation converges after a finite number of steps to some overapproximation λ_∞, is to use a *widening operator* ∇. The concretization $c_\infty = \gamma(\lambda_\infty)$ is an overapproximation of the least fixpoint of the function $F_{\Delta'}$.

Choice of the Abstract Domain. In abstract interpretation based techniques, the quality of the approximation we obtain depends on the choice of the abstract domain Λ. In our case, the main issue is to abstract the content of the FIFO channels. As discussed in [13], a good abstract domain is the class of regular languages, which can be represented by finite automata. Let us recall the main ideas of this abstraction.

Finite Automata as an Abstract Domain. We first assume that there is only one queue in the distributed system $\mathcal{T}$; we explain later how to handle a distributed system with several queues. With one queue, the concrete domain of the system $\mathcal{T}$ is defined by $X = 2^{L \times M^*}$. A set of states $Y \in 2^{L \times M^*}$ can be viewed as a map $Y : L \mapsto 2^{M^*}$ that associates a language $Y(\ell)$ with each location $\ell \in L$; $Y(\ell)$ therefore represents the possible contents of the queue in the location ℓ. To simplify the computation, we substitute the concrete domain $\langle L \mapsto 2^{M^*}, \subseteq, \cup, \cap, L \times M^*, \emptyset\rangle$ by the abstract domain $\langle L \mapsto \mathrm{Reg}(M), \subseteq, \cup, \cap, L \times M^*, \emptyset\rangle$, where $\mathrm{Reg}(M)$ is the set of *regular languages* over the alphabet M. Since regular languages have a canonical representation given by finite automata, each operation (union, intersection, left concatenation,...) in the abstract domain can be performed on finite automata.

Widening Operator. The widening operator is also performed on a finite automaton, and consists in quotienting the nodes[1] of the automaton by the *k-bounded bisimulation equivalence relation* $\equiv_k$; $k \in \mathbb{N}$ is a parameter which allows us to tune the precision, since increasing k improves the quality of the abstractions in general. Two nodes are equivalent w.r.t. $\equiv_k$ if they have the same outgoing path (sequence of labeled transitions) up to length k. While we merge the equivalent nodes, we keep all transitions and we obtain an automaton recognizing a larger language. Note that for a fixed k, the class of automata which results from such a quotient operation from any original automaton, is finite and its cardinality is bounded by a number which is only function of k. So, when we apply this widening operator regularly, the fixpoint computation terminates (see [13] for more details).

Example 3. We consider the automaton $\mathcal{A}$ depicted in Figure 2, whose recognized language is $a + ba + bba + bbba$. We consider the 1-bounded bisimulation relation i.e., two

[1] The states of an automaton representing the queue contents are called nodes to avoid the confusion with the states of a CFSM.

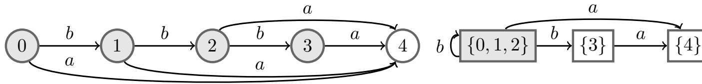


Fig. 2. Illustration of the 1-bounded bisimulation relation $\equiv_1$ for $\mathcal{A}$

nodes of the automaton are equivalent if they have the same outgoing transitions. So, nodes $0, 1, 2$ are equivalent, since they all have two transitions labeled by a and b. Nodes 3 and 4 are equivalent to no other node since 4 has no outgoing transition whereas only a is enabled in node 3. When we quotient $\mathcal{A}$ by this equivalent relation, we obtain the automaton on the right side of Figure 2, whose recognized language is b^*a. ◇

When the system contains several queues $Q = \{Q_1, \ldots, Q_r\}$, their content can be represented by a concatenated word $w_1\sharp\ldots\sharp w_r$ with one w_i for each queue Q_i and $\sharp$, a delimiter. With this encoding, we represent a set of queue contents by a finite automaton of a special kind, namely a QDD [2]. Since QDDs are finite automata, classical operations (union, intersection, left concatenation,...) in the abstract domain are performed as was done previously. We must only use a slightly different widening operator not to merge the different queue contents [13].

Effective SE-algorithm. The Reach operator is computed using those abstract interpretation techniques: we proceed to an iterative computation in the abstract domain of regular languages and the widening operator ensures that this computation terminates after a finite number of steps [6]. So the Reach operator always gives an overapproximation of the reachable states regardless the distributed system. The efficiency of these approximations is measured in the experiments of section 6. Because of Theorem 1, our SE-algorithm is complete.

6 Experiments

We have implemented the SE-algorithm as a new feature of the McScM tool [17], a model checker for distributed systems modeled by CFSM. Since it represents queue contents by QDDs, this software provides most of the functionalities needed by our algorithm, like effective computation of reachable states. We have also added a mechanism to manage vector clocks, and an interactive simulator. This simulator first computes and displays the initial state estimates. At each step, it asks the user to choose a possible transition.

We proceeded to an evaluation of our algorithm measuring the size of the state estimates. Note that this size is not the number of global states of the state estimate (which may be infinite) but the number of nodes of its QDD representation. We generated random sequences of transitions for our running example and some other examples of [10]. Table 1 shows the average execution time for a random sequence of 100 transitions, the memory required (heap size), the average and maximal size of the state estimates. Default value of the widening parameter is $k = 1$. Experiments were done on a standard MacBook Pro with a 2.4 GHz Intel core 2 duo CPU. These results show that the computation of state estimates takes about 50ms per transition and that the symbolic representation of state estimates we add to messages are automata with a few dozen nodes. A sensitive element in the method is the size of the computed and transmitted

Table 1. Experiments

example	# subsystems	# channels	time [s]	memory [MB]	maximal size	average size
running example	3	4	7.13	5.09	143	73.0
c/d protocol	2	2	5.32	8.00	183	83.2
non-regular protocol	2	1	0.99	2.19	172	47.4
ABP	2	3	1.19	2.19	49	24.8
sliding window	2	2	3.26	4.12	21	10.1
POP3	2	2	3.08	4.12	22	8.5

information. It can be improved by the use of compression techniques to reduce the size of this information. A more evolved technique would consist in the offline computation of the set of possible estimates. Estimates are indexed in a table, available at execution time to each local estimator. If we want to keep an online algorithm, we can use the memoization technique. When a state estimate is computed for the first time, it is associated with an index that is transmitted to the subsystem which records both values. If the same estimate must be transmitted, only its index can be transmitted and the receiver can find from its table the corresponding estimate. We also highlight that our method works better on the real-life communication protocols we have tested (alternating bit protocol, sliding window, POP3) than on the examples we introduced to test our tool.

7 Conclusion and Future Work

We have proposed an effective algorithm to compute online, locally to each subsystem, an estimate of the global state of a running distributed system, modeled as *communicating finite state machines* with reliable unbounded FIFO queues. With such a system, a global state is composed of the current location of each subsystem together with the channel contents. The principle is to add a local estimator to each subsystem such that most of the system is preserved; each local estimator is only able to compute information and in particular symbolic representations of state estimates and to piggyback some of this computed information to the transmitted messages. Since these estimates may be infinite, a crucial point of our work has been to propose and evaluate the use of regular languages to abstract sets of FIFO queues. In practice, we have used k-bisimilarity relations, which allows us to represent each (possibly infinite) set of queue contents by the minimal and canonical k-bisimilar finite automaton which gives an overapproximation of this set. Our algorithm transmits state estimates and vector clocks between subsystems to allow them to refine and preserve consistent state estimates. More elaborate examples must be taken to analyze the precision of our algorithm and see, in practice, if the estimates are sufficient to solve diagnosis or control problems. Anyway, it appears important to study the possibility of reducing the size of the added communication while preserving or even increasing the precision in the transmitted state estimates.

References

1. Berry, G., Gonthier, G.: The esterel synchronous programming language: Design, semantics, implementation. Sci. Comput. Program. 19(2), 87–152 (1992)
2. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The power of QDDs. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 172–186. Springer, Heidelberg (1997)

3. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30(2), 323–342 (1983)
4. Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems. Kluwer Academic Publishers, Dordrecht (1999)
5. Chatain, T., Gastin, P., Sznajder, N.: Natural specifications yield decidability for distributed synthesis of asynchronous systems. In: Nielsen, M., Kučera, A., Miltersen, P.B., Palamidessi, C., Tůma, P., Valencia, F. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 141–152. Springer, Heidelberg (2009)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252 (1977)
7. Debouk, R., Lafortune, S., Teneketzis, D.: Coordinated decentralized protocols for failure diagnosis of discrete event systems. Discrete Event Dynamical Systems: Theory and Applications 10, 33–79 (2000)
8. Genest, B.: On implementation of global concurrent systems with local asynchronous controllers. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 443–457. Springer, Heidelberg (2005)
9. Hélouet, L., Gazagnaire, T., Genest, B.: Diagnosis from scenarios. In: Proc. of the 8th Int. Workshop on Discrete Events Systems, WODES 2006, pp. 307–312 (2006)
10. Heußner, A., Le Gall, T., Sutre, G.: Extrapolation-based path invariants for abstraction refinement of fifo systems. In: Păsăreanu, C.S. (ed.) Model Checking Software. LNCS, vol. 5578, pp. 107–124. Springer, Heidelberg (2009)
11. Kalyon, G., Le Gall, T., Marchand, H., Massart, T.: Global state estimates for distributed systems (version with proofs). In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 194–203. Springer, Heidelberg (2011), `http://hal.inria.fr/inria-00581259/`
12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
13. Le Gall, T., Jeannet, B., Jéron, T.: Verification of communication protocols using abstract interpretation of FIFO queues. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 204–219. Springer, Heidelberg (2006)
14. Lin, F., Rudie, K., Lafortune, S.: Minimal communication for essential transitions in a distributed discrete-event system. Trans. on Automatic Control 52(8), 1495–1502 (2007)
15. Massart, T.: A calculus to define correct tranformations of lotos specifications. In: FORTE 1991. IFIP Transactions, vol. C-2, pp. 281–296 (1991)
16. Mattern, F.: Virtual time and global states of distributed systems. In: Proceedings of the Workshop on Parallel and Distributed Algorithms, pp. 215–226 (1989)
17. The McScM library (2009), `http://altarica.labri.fr/forge/projects/mcscm/wiki/`
18. Ricker, L., Caillaud, B.: Mind the gap: Expanding communication options in decentralized discrete-event control. In: Conference on Decision and Control (2007)
19. Sampath, M., Sengupta, R., Lafortune, S., Sinaamohideen, K., Teneketzis, D.: Failure diagnosis using discrete event models. IEEE Transactions on Control Systems Technology 4(2), 105–124 (1996)
20. Tripakis, S.: Decentralized control of discrete event systems with bounded or unbounded delay communication. IEEE Trans. on Automatic Control 49(9), 1489–1501 (2004)
21. Xu, S., Kumar, R.: Distributed state estimation in discrete event systems. In: ACC 2009: Proc. of the 2009 conference on American Control Conference, pp. 4735–4740 (2009)

A Process Calculus for Dynamic Networks

Dimitrios Kouzapas[1] and Anna Philippou[2]

[1] Imperial College, London, UK
dk208@doc.ic.ac.uk
[2] University of Cyprus, Cyprus
annap@cs.ucy.ac.cy

Abstract. In this paper we propose a process calculus framework for dynamic networks in which the network topology may change as computation proceeds. The proposed calculus allows one to abstract away from neighborhood-discovery computations and it contains features for broadcasting at multiple transmission ranges and for viewing networks at different levels of abstraction. We develop a theory of confluence for the calculus and we use the machinery developed towards the verification of a leader-election algorithm for mobile ad hoc networks.

1 Introduction

Distributed and wireless systems present today one of the most challenging areas of research in computer science. Their high complexity, dynamic nature and features such as broadcasting communication, mobility and fault tolerance, render their construction, description and analysis a challenging task. The development of formal frameworks for describing and associated methodologies for reasoning about distributed systems has been an active area of research for the last few decades. Process calculi, e.g. [7,8], are one such formalism. Since their inception, they have been extensively studied and they have been extended for modeling and reasoning about a variety of aspects of process behavior including mobility, distribution and broadcasting.

Our goal in this paper is to propose a process calculus in which to be able to reason about mobile ad hoc networks (MANETs) and their protocols. Our proposal, the Calculus for Systems with Dynamic Topology, CSDT, is inspired by works which have previously appeared in the literature [2,5,4,12,13] on issues such as broadcasting, movement and separating between a node's control and topology information. However, our calculus extends these works by considering two additional MANET features. The first concerns the ability of MANETs to broadcast messages at different transmission levels. This is a standard feature of mobile ad hoc networks; a node may choose to broadcast a message at a high transmission level in order to communicate with a wider set of nodes, as required by a protocol or application, or it may choose to use a low transmission level in order to conserve its power. The second feature concerns that of neighbor discovery. As computation proceeds and nodes move in and out of each other's transmission range, each node attempts to remain aware of its connection topology in order to successfully complete its tasks (e.g. routing). To achieve this, neighbor discovery protocols are implemented and run which typically involve periodically emitting "hello" messages and acknowledging such messages received by one's neighbors. Although it

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 213–227, 2011.
© IFIP International Federation for Information Processing 2011

is possible that at various points in time a node does not have the precise information regarding its neighbors, these protocols aim to ensure that the updated topology is discovered and that the node adjusts its behavior accordingly so that correct behavior is achieved. Thus, when one is called to model and verify a MANET protocol, it is important to take into account that the neighbor-information available to a node may not be correct at all times. To handle this one might model an actual neighbor-discovery protocol in parallel to the algorithm under study and verify the composition of the two. Although such a study would be beneficial towards obtaining a better understanding of the behavior of both protocols, it would turn an already laborious task into a more laborious one and it would impose further requirements on behalf of the modeling language (e.g. to capture timed behaviors).

CSDT allows for reasoning about both of these aspects of behavior. To begin with, it allows nodes to broadcast information at two different transmission levels. The transmission level of a broadcast is encoded as a parameter of broadcasted messages. Regarding the issue of neighborhood discovery, the semantics of CSDT contain rules that mimic the behavior of a neighborhood-discovery protocol: CSDT equips nodes with knowledge of their believed (and not necessarily actual) sets of neighbors which is continuously updated, similarly to the way a neighborhood-discovery algorithm operates and gradually discerns changes in the connectivity of a node. Furthermore, these neighbor sets are accessible from the control part of a node and may affect the flow of the node's execution. A final novelty of CSDT is the introduction of a hiding construct that allows us to observe networks at different levels of abstraction. Since messages in our network descriptions are *broadcasted* over the medium, the notion of channel restriction (or name hiding) becomes irrelevant. Nonetheless, the effect of hiding behaviors remains useful in our setting, especially for analysis purposes. To achieve this, we associate every message with a "type" and we implement hiding by restricting the set of message types which should be observable at the highest level of a process. A similar encapsulation construct has also recently been proposed in [1].

The operational semantics of our calculus is given in terms of a labelled transition system on which we propose a notion of weak bisimulation. Subsequently, we develop a theory of confluence. The notion of confluence was first studied in the context of concurrent systems by Milner in CCS [7] and subsequently in various other settings [14,10,11]. Its essence, is that "of any two possible actions, the occurrence of one will never preclude the other". This paper, is the first to consider the theory of confluence in a setting of broadcasting communication. We establish a variety of results including a theorem that allows for compositional reasoning of confluent behavior under the assumption of a stationary topology of a network. We illustrate the utility of these techniques as well as the formalism via the analysis of a leader-election algorithm.

Related Work. Several process calculi have recently been proposed for dynamic networks such as CBS# [9], CMAN [2], CMN [5], CNT [1], CWS [4], TCSW [6], and the ω-calculus [13]. Most of these calculi, support broadcasting communication, message loss and mobility of nodes, with the exception of [4], and explicit location information, with the exception of the ω-calculus, where neighborhood information is captured via groups which can be dynamically created or updated thus modeling dynamic network topologies, and CNT where topology changes are modeled in the semantics as opposed

to the syntax of the language. Perhaps closest to CSDT is the CMN process calculus. Differences that exist between the two calculi include (1) the treatment of the notion of a location (while in CMN locations can be viewed as values in a coordinate system and neighborhood is computed via a metric distance function, in CSDT locations and their interconnections are captured as a graph), (2) the fact that CSDT allows point-to-point communication in addition to broadcasting and (3) CMN caters for lossy communication whereas CSDT does not. Furthermore, as it has already been discussed, CSDT extends all of these frameworks by allowing to broadcast at different transmission levels and by associating nodes with their believed sets of neighbors which are updated by semantic rules that mimic the behavior of a neighbor discovery protocol.

Contribution. The contribution of our work is summarized as follows:

1. We define a new process calculus for reasoning about dynamic networks. This calculus introduces a number of new ideas which have been selected in view of facilitating the modeling and the analysis of mobile ad hoc network protocols.
2. We develop the theory of confluence in the context of our calculus. This is the first such theory for process calculi featuring broadcast communication.
3. We provide a correctness proof of a non-trivial leader-election protocol proposed in [15] for mobile ad hoc networks.

2 The Process Calculus

In our Calculus for Systems with Dynamic Topology, CSDT, we consider a system as a set of nodes operating in space. Each node possesses a physical location and a unique identifier. Movement is modeled as the change in the location of a node, with the restriction that the originating and the destination locations are neighboring locations.

Nodes in CSDT can communicate with each other by broadcasting messages. Broadcasting may take place at different transmission levels as required by an underlying protocol and/or for power-saving purposes. Specifically, in CSDT we consider two transmission levels, a normal level and a high level.

Neighbor discovery, that is, determining which nodes fall within the transmission range of a node, is a building block of network protocols and applications. To facilitate the reasoning about such protocols we embed in the semantics of our calculus rules that mimic the behavior of a neighbor discovery algorithm, which observes changes in the network's topology and, specifically, the departure and arrival of nodes within the normal and high transmission ranges of a node. To achieve this, each node is associated with two sets of nodes, N and H, which are the sets of nodes believed to be within the normal and high transmission ranges of a node, respectively. Thus, we write $P{:}[\![id, \ell, N, H]\!]$, for describing a node running code P with unique identifer id, located at physical location ℓ, believed normal-range and high-range neighbors N, and H.

2.1 The Syntax

We now continue to formalize the above intuitions into the syntax of CSDT. We begin by describing the basic entities of CSDT. We assume a set of node identifiers $\mathcal{I}$ ranged

over by id, i, j, and a set of physical locations $\mathcal{L}$ ranged over by ℓ, ℓ' and we say that two locations ℓ, ℓ' are neighboring locations if $(\ell, \ell') \in \mathit{Nb}$, where $\mathit{Nb} \subseteq \mathcal{L} \times \mathcal{L}$. Furthermore, we assume a set of transmission levels $\{\mathrm{n}, \mathrm{h}\}$ and associated with these levels the functions $\mathrm{range}_{\mathrm{n}} : \mathcal{L} \rightarrow 2^{\mathcal{L}}$ and $\mathrm{range}_{\mathrm{h}} : \mathcal{L} \rightarrow 2^{\mathcal{L}}$ which, given a location, return the locations that fall within its normal and high transmission levels, respectively. These functions, need not take a symmetric view on locations and may be defined so as to yield unidirectional links. Furthermore, we assume a set of special labels $\mathcal{T}$. Elements of $\mathcal{T}$ are mere keywords appended to messages indicating the message type.

In addition, we assume a set of *terms*, ranged over by e, built over (1) a set of *constants*, ranged over by u, v, (2) a set of *variables* ranged over by x, y, and (3) function applications of the form $f(e_1, \ldots e_n)$ where f is a function from a set of functions (e.g. logical connectives, set operators and arithmetic operators), and the e_i are terms. We say that a term is *closed* if it contains no variables. The evaluation relation $\twoheadrightarrow$ for closed terms is defined in the expected manner. We write $\tilde{r}$ for a tuple of syntactic entities $r_1, \ldots, r_n$. Finally, we assume a set of process constants $\mathcal{C}$, denoted by C, each with an associated definition of the form $C\langle\tilde{x}\rangle \stackrel{\mathrm{def}}{=} P$, where P may contain occurrences of C, as well as other constants. Based on these basic entities, the syntax of of CSDT is given in Table 1, where $T \subseteq \mathcal{T}$.

Table 1. The Syntax

Actions: η	$::=$	$\overline{b}(w, t, \tilde{v}, tl)$	broadcast
	$\mid$	$r(t, \tilde{x})$	input
Processes: P	$::=$	$\mathbf{0}$	Inactive Process
	$\mid$	$\eta.P$	Action Prefix
	$\mid$	$P_1 + P_2$	Nondeterministic Choice
	$\mid$	$\mathsf{cond}\ (e_1 \rhd P_1, \ldots, e_n \rhd P_n)$	Conditional
	$\mid$	$C\langle\tilde{v}\rangle$	Process Constant
Networks: M	$::=$	$\mathbf{0}$	
	$\mid$	$P{:}\sigma$	Located Node
	$\mid$	$M_1 \mid M_2$	Parallel Composition
	$\mid$	$M\backslash T$	Restriction
Interfaces: σ	$::=$	$[\![id, \ell, N, H]\!]$	

There are two types of actions in CSDT. A broadcast action $\overline{b}(w, t, \tilde{v}, tl)$ is a transmission at transition level $tl \in \{\mathrm{n}, \mathrm{h}\}$, of type $t \in \mathcal{T}$ of the tuple $\tilde{v}$ with intended recipients w, where $'-'$ denotes that the message is intended for all neighbors of the transmitting node and $j \in \mathcal{I}$ denotes that it is intended solely for node j. An input action $r(t, \tilde{x})$ represents a receipt of a message $\tilde{x}$ of type t. In turn, a process can then be inactive, an action-prefixed process, the nondeterministic choice between two processes, a process constant or a conditional process. The conditional process $\mathsf{cond}\ (e_1 \rhd P_1, \ldots, e_n \rhd P_n)$ presents the conditional choice between a set of processes: it behaves as P_i, where i is the smallest integer for which e_i evaluates to $\mathtt{true}$.

On the other hand, networks are built on the basis of located processes, $P{:}\sigma$, where σ is the node's topology information which we call its *interface*. An interface σ contains the node identifier id, its location ℓ as well as its normal-range and high-range neighbors N and H, according to its current knowledge. We allow the control part of a located process $P{:}\sigma$, namely P, to access information mentioned in the interface σ, by using the special labels id, l, N and H, thus allowing the control part of a process to express dependencies on the node's neighborhood information. For example, an expression "$4 \in \mathrm{N}$" occurring within $P : [\![1, \ell, \{2\}, \{2, 3\}]\!]$ is evaluated as "$4 \in \{2\}$".

Thus, a network can be an inactive network **0**, a located node $P{:}\sigma$, a parallel composition of networks $M_1 \mid M_2$, or a restricted network $M \backslash T$. In $M \backslash T$ the scope of messages of all types in T is restricted to network M: components of M may exchange messages of these types to interact with one another but not with M's environment. To avoid name collisions on types, we consider α-conversion on processes as the renaming of types that are bound by some restriction and we say that P and Q are α-equivalent, $P \equiv_\alpha Q$, if P and Q differ by a renaming of bound types.

In what follows, given an interface σ, we write $\mathrm{l}(\sigma)$ and $\mathrm{id}(\sigma)$ for the location and the identifier mentioned in σ, respectively. Moreover, we use $\mathrm{types}(M)$ to denote the set of all types occurring in activities of M.

2.2 The Semantics

The semantics of CSDT is defined in terms of a structural congruence relation $\equiv$, found in Table 2, and a structural operational semantics, given in Tables 3 and 4.

Table 2. Structural congruence relation

(N1) $M \equiv M \vert 0$	(N5) $M \backslash T - \{t\} \equiv M \backslash T$ if $t \notin \mathrm{types}(M)$
(N2) $M_1 \vert M_2 \equiv M_2 \vert M_1$	(N6) $M \backslash T \equiv (M \backslash T - \{t\}) \backslash \{t\}$
(N3) $(M_1 \vert M_2) \vert M_3 \equiv M_1 \vert (M_2 \vert M_3)$	(N7) $M_1 \equiv M_2$ if $M_1 \equiv_\alpha M_2$
(N4) $M_1 \backslash \{t\} \vert M_2 \equiv (M_1 \vert M_2) \backslash \{t\}$ if $t \notin \mathrm{types}(M_2)$	

The rules of Table 3 describe the behavior of located processes in isolation whereas the rules in Table 4 the behavior of networks. A transition of P (or M) has the form $P \xrightarrow{\alpha} P'$, specifying that P can perform action α and evolve into P' where α can have one of the following forms:

– $\overline{b}(w, t, \tilde{v}, tl, \ell)$ denotes a broadcast to recipients w of a message $\tilde{v}$ of type t at transmission level tl, taking place at location ℓ.
– $r(id, t, \tilde{v}, \ell)$ denotes a receipt by node id of a message $\tilde{v}$ of type t, taking place at location ℓ.

– $r?(id, t, \tilde{v}, \ell)$ denotes an advertisement by node id that it is willing to receive a message $\tilde{v}$ of type t, at location ℓ.
– τ and μ denote two types of unobservable actions in the calculus. Action τ is associated with the effect of restriction (rule (Hide2), Table 4) and μ is associated with the movement of nodes (rule (L6), Table 3) as well as with updates of neighborhood information (rules (InSN), (InSH), (OutSN) and (OutSH), Table 4).

We let *Act* denote the set of all actions and let α and β range over *Act* and we write $\mathrm{type}(\alpha)$ for the type of an $\alpha \in Act - \{\tau, \mu\}$.

Table 3. Transition rules for located nodes

(L1) $(\overline{b}(w, t, \tilde{v}, tl).P){:}\sigma \stackrel{\overline{b}(w,t,\tilde{v},tl,\mathrm{l}(\sigma))}{\longrightarrow} P{:}\sigma$ (L2) $(r(t, \tilde{x}).P){:}\sigma \stackrel{r(\mathrm{id}(\sigma),t,\tilde{v},\mathrm{l}(\sigma))}{\longrightarrow} P\{\tilde{v}/\tilde{x}\}{:}\sigma$

(L3) $\dfrac{[P\{\tilde{v}/\tilde{x}\}]{:}\sigma \stackrel{\alpha}{\longrightarrow} P'{:}\sigma}{[C\langle\tilde{v}\rangle]{:}\sigma \stackrel{\alpha}{\longrightarrow} P'{:}\sigma}$ $C\langle\tilde{x}\rangle \stackrel{\mathrm{def}}{=} P$ (L4) $\dfrac{P_i{:}\sigma \stackrel{\alpha}{\longrightarrow} P_i'{:}\sigma}{(P_1 + P_2){:}\sigma \stackrel{\alpha}{\longrightarrow} P_i'{:}\sigma}, \quad i \in \{1, 2\}$

(L5) $\dfrac{P_i{:}\sigma \stackrel{\alpha}{\longrightarrow} P_i'{:}\sigma}{(\mathsf{cond}\ (e_1 \triangleright P_1, \ldots, e_n \triangleright P_n)){:}\sigma \stackrel{\alpha}{\longrightarrow} P_i'{:}\sigma}$ $e_i \twoheadrightarrow \mathsf{true}, \forall j < i, e_j \twoheadrightarrow \mathsf{false}$

(L6) $\dfrac{(\ell, \ell') \in Nb}{P{:}[\![i, \ell, N, H]\!] \stackrel{\mu}{\longrightarrow} P{:}[\![i, \ell', N, H]\!]}$

Moving our attention to the rules of Table 4, we point out that the first four rules implement the underlying neighborhood discovery protocol. Nodes which are within the normal and high transmission ranges of a located process are included in the appropriate sets in its interface and, similarly, nodes which have exited these transmission ranges are removed. In all four cases a μ internal action takes place.

The two (BrC) rules which follow give semantics to broadcasting within the language. They employ the compatibility function comp, where $\mathrm{comp}(w, i)$ is evaluated to true only if identifier i is compatible with intended recipient w: $\mathrm{comp}(w, i) = (w =' -' \vee w = i)$. Axiom (BrC1) specifies that, if a broadcast is available and there exists a compatible recipient within the range of the broadcast, the message is received and the broadcast message is propagated. If there is no appropriate receiver then, again, the broadcast is propagated (BrC2). Note that rule (BrC2) (as well as (Rec)) is defined in terms of the inability of executing an action via the use of relation $\stackrel{\alpha}{\not\rightarrow}$, where $P \stackrel{\alpha}{\not\rightarrow}$, if $\neg(P \stackrel{\alpha}{\longrightarrow} P')$ for any P'.

Moving on to rule (Rec), we observe that a network can advertise the fact that it may receive an input. This is necessary, otherwise an inactive network and a network such as $M \stackrel{\mathrm{def}}{=} [r(t, \tilde{x}).P]{:}\sigma_1 \mid [r(t, \tilde{y}).Q]{:}\sigma_2$ would have exactly the same transition systems when clearly they would yield distinct behaviors when placed in parallel with a network such as $M' = [\overline{b}(-, t, \tilde{v}, \mathrm{n}).S]{:}\sigma$ (affecting compositionality in the calculus). Now, if rule (Rec) was enunciated via a normal receive action instead of $r?$, we would have: $M \stackrel{r(\mathrm{id}(\sigma_1),t,\tilde{v},\mathrm{l}(\sigma_1))}{\longrightarrow} [P\{\tilde{v}/\tilde{x}\}]{:}\sigma_1 \mid [r(t, \tilde{y}).Q]{:}\sigma_2$ and subsequently

Table 4. Transition rules for networks

(InSN) $$\frac{\mathrm{l}(\sigma) \in \mathrm{range_n}(\ell),\ \mathrm{id}(\sigma) \notin N}{P{:}[\![id, \ell, N, H]\!] \mid Q{:}\sigma \xrightarrow{\mu} P{:}[\![id, \ell, N \cup \{\mathrm{id}(\sigma)\}, H]\!] \mid Q{:}\sigma}$$

(OutSN) $$\frac{\mathrm{l}(\sigma) \notin \mathrm{range_n}(\ell),\ \mathrm{id}(\sigma) \in N}{P{:}[\![id, \ell, N, H]\!] \mid Q{:}\sigma \xrightarrow{\mu} P{:}[\![id, \ell, N - \{\mathrm{id}(\sigma)\}, H]\!] \mid Q{:}\sigma}$$

(InSH) $$\frac{\mathrm{l}(\sigma) \in \mathrm{range_h}(\ell),\ \mathrm{id}(\sigma) \notin H}{P{:}[\![id, \ell, N, H]\!] \mid Q{:}\sigma \xrightarrow{\mu} P{:}[\![id, \ell, N, H \cup \{\mathrm{id}(\sigma)\}]\!] \mid Q{:}\sigma}$$

(OutSH) $$\frac{\mathrm{l}(\sigma) \notin \mathrm{range_h}(\ell),\ \mathrm{id}(\sigma) \in H}{P{:}[\![id, \ell, N, H]\!] \mid Q{:}\sigma \xrightarrow{\mu} P{:}[\![id, \ell, N, H - \{\mathrm{id}(\sigma)\}]\!] \mid Q{:}\sigma}$$

(BrC1) $$\frac{M_1 \xrightarrow{\overline{b}(w,t,\tilde{v},tl,\ell)} M_1', M_2 \xrightarrow{r(id,t,\tilde{v},\ell')} M_2', \mathrm{comp}(w, id), \ell' \in \mathrm{range}_{tl}(\ell)}{M_1 \mid M_2 \xrightarrow{\overline{b}(w,t,\tilde{v},tl,\ell)} M_1' \mid M_2'}$$

(BrC2) $$\frac{M_1 \xrightarrow{\overline{b}(w,t,\tilde{v},tl,\ell)} M_1' \text{ and } M_2 \not\xrightarrow{r(id,t,\tilde{v},\ell')} \forall id, \ell' \cdot \mathrm{comp}(w, id), \ell' \in \mathrm{range}_{tl}(\ell)}{M_1 \mid M_2 \xrightarrow{\overline{b}(w,\ell,l,t,\tilde{v})} M_1' \mid M_2}$$

(Rec) $$\frac{M_1 \xrightarrow{r(id,\ell,t,\tilde{v})} M_1' \text{ and } M_2 \not\xrightarrow{\overline{b}(w,t,\tilde{v},tl,\ell')} \forall w, \ell', tl \cdot \mathrm{comp}(w, id), \ell \in \mathrm{range}_{tl}(\ell')}{M_1 \mid M_2 \xrightarrow{r?(id,\ell,t,\tilde{v})} M_1' \mid M_2}$$

(Hide1) $$\frac{M \xrightarrow{\alpha} M' \text{ and type}(\alpha) \notin T}{M \backslash T \xrightarrow{\alpha} M' \backslash T}$$

(Hide2) $$\frac{M \xrightarrow{\alpha} M' \text{ and type}(\alpha) \in T}{M \backslash T \xrightarrow{\tau} M' \backslash T}$$

(Par) $$\frac{M_1 \xrightarrow{\alpha} M_1', \quad \alpha \in \{\tau, \mu\}}{M_1 \mid M_2 \xrightarrow{\alpha} M_1' \mid M_2}$$

(Struct) $$\frac{M_1 \equiv M_2, M_2 \xrightarrow{\alpha} M_2', M_2' \equiv M_1'}{M_1 \xrightarrow{\alpha} M_1'}$$

$M \mid M' \xrightarrow{\overline{b}(-,t,\tilde{v},\mathrm{n},\mathrm{l}(\sigma))} ([P\{\tilde{v}/\tilde{x}\}]{:}\sigma_1 \mid [r(t,\tilde{y}).Q]{:}\sigma_2) \mid S{:}\sigma$, In this way only one of the two components of M ends up receiving the broadcasted message of M' which is not what one would expect of a broadcasting communication. To achieve the correct interpretation, only located nodes can emit an $r(\ldots)$ action (see (Rec), Table 3) and to obtain the transition of $M \mid M'$ we would employ structural congruence and the facts that (1) $M \mid M' \equiv M_1 = [r(t,\tilde{x}).P]{:}\sigma_1 \mid ([r(t,\tilde{y}).Q]{:}\sigma_2 \mid [\overline{b}(-,t,\tilde{v},\mathrm{n}).S]{:}\sigma)$, (2) $M_1 \xrightarrow{\overline{b}(-,t,\tilde{v},\mathrm{n},\mathrm{l}(\sigma))} [P\{\tilde{v}/\tilde{x}\}]{:}\sigma_1 \mid ([Q\{\tilde{v}/\tilde{y}\}]{:}\sigma_2 \mid S{:}\sigma)$, to obtain, by rule (Struct), that $M \mid M' \xrightarrow{\overline{b}(-,t,\tilde{v},\mathrm{n},\mathrm{l}(\sigma))} ([P\{\tilde{v}/\tilde{x}\}]{:}\sigma_1 \mid [Q\{\tilde{v}/\tilde{y}\}]{:}\sigma_2) \mid S{:}\sigma$.

Finally, rules (Hide1) and (Hide2) specify that the effect of restricting a set of types T within a process is to transform all actions of these types into internal actions.

2.3 Bisimulation and Confluence

In the next section we build some machinery for reasoning about broadcasting networks. Due to lack of space, all proofs from this section are omitted. The complete exposition can be found in [3]. First, let us recall that M' is a *derivative* of M, if there exist actions $\alpha_1, \ldots, \alpha_n, n \geq 0$, such that $M \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} M'$. Moreover, we define weak transitions as follows: given an action α we write $M \Longrightarrow M'$ for $M(\xrightarrow{\tau,\mu})^* M'$, $M \stackrel{\alpha}{\Longrightarrow} M'$ for $M \Longrightarrow\xrightarrow{\alpha}\Longrightarrow M'$, and $M \stackrel{\hat{\alpha}}{\Longrightarrow} M'$ for $M \Longrightarrow M'$ if $\alpha \in \{\tau, \mu\}$, $M \stackrel{r(id,\ell,t,\tilde{v})}{\Longrightarrow} M'$ if $\alpha = r?(id, \ell, t, \tilde{v})$, and $M \stackrel{\alpha}{\Longrightarrow} M'$ otherwise.

The first useful tool which accompanies CSDT is a notion of observational equivalence:

Definition 1. *Bisimilarity* is the largest symmetric relation, denoted by $\approx$, such that, if $M_1 \approx M_2$ and $M_1 \xrightarrow{\alpha} M_1'$, there exists M_2' such that $M_2 \stackrel{\hat{\alpha}}{\Longrightarrow} M_2'$ and $M_1' \approx M_2'$.

We may prove that bisimilarity is a congruence relation. It is also worth pointing out that we may establish the following property of mobile nodes pertaining to their ubiquitous nature, namely:

Lemma 1. For any process P, $id \in \mathcal{I}$, $\ell, \ell' \in \mathcal{L}$ and $N, H \subseteq \mathcal{L}$, if $(\ell, \ell') \in Nb^+$, where Nb^+ is the transitive closure of relation *Nb*, then $P[\![id, \ell, N, H]\!] \approx P[\![id, \ell', N, H]\!]$.

We now turn to consider the notions of determinacy and confluence in our calculus. These make use of the following notation: given actions α and α' we write $\alpha \bowtie \alpha'$ if α and α' differ only in their specified location, i.e. $\alpha' = \alpha[\ell'/\ell]$, for some ℓ and ℓ'.

Definition 2. M is *determinate* if, for every derivative M' of M and for all actions α, α', $\alpha \bowtie \alpha'$, whenever $M' \xrightarrow{\alpha} M_1$ and $M' \stackrel{\hat{\alpha'}}{\Longrightarrow} M_2$ then $M_1 \approx M_2$.

This definition makes precise the requirement that, when an experiment is conducted on a network, it should always lead to the same state up to bisimulation. This is irrespective of the location at which the action is taking place which explains the use of the $\bowtie$ operator. As an example, consider networks

$$M_1 \stackrel{\mathrm{def}}{=} \overline{b}(-, t, \tilde{v}, \mathrm{n}).\mathbf{0}{:}[\![1, \ell, \{2\}, \{2\}]\!] \qquad M_1' \stackrel{\mathrm{def}}{=} \overline{b}(-, t, \tilde{v}, \mathrm{n}).\mathbf{0}{:}[\![1, \ell', \{2\}, \{2\}]\!]$$

$$M_2 \stackrel{\mathrm{def}}{=} (r(t, \tilde{x}).\overline{b}(-, s, \tilde{x}, \mathrm{h}).\mathbf{0},){:}[\![2, \ell, \{1\}, \{1\}]\!]$$

We observe that M_1 and M_2 are determinate, whereas $M_1 \mid M_2$ is not: Assuming that $\ell \notin \mathrm{range}_n(\ell')$, $M_1 \mid M_2 \xrightarrow{\overline{b}(-,t,\tilde{v},\mathrm{n},\ell)} M \equiv \overline{b}(-, s, \tilde{x}, \mathrm{h}).\mathbf{0}{:}[\![2, \ell, \ldots]\!]$ and $M_1 \mid M_2 \xrightarrow{\mu} M_1' \mid M_2 \xrightarrow{\overline{b}(-,t,\tilde{v},\mathrm{n},\ell')} M' \equiv M_2$, where $M \not\approx M'$.

A conclusion that we may draw from this example is that determinacy is not preserved by parallel composition. We may in fact show that determinacy is preserved by prefix and conditional statements. In order to strengthen determinacy into a notion preserved by a wider range of operators, and in particular parallel composition, Milner [7] introduced the notion of confluence. According to the definition of [7], a CCS process P is *confluent* if it is determinate and, for each of its derivatives Q and distinct actions α,

β, given the transitions to $Q \xrightarrow{\alpha} Q_1$ and $Q \stackrel{\beta}{\Longrightarrow} Q_2$, the diamond property is satisfied, that is, there exist transitions $Q_1 \stackrel{\beta}{\Longrightarrow} Q_1'$ and $Q_2 \stackrel{\alpha}{\Longrightarrow} Q_2'$ such that $Q_1' \approx Q_2'$.

In the context of our work, we observe that the nature of broadcasting communication allows to enunciate confluence through a weaker treatment of input actions. In particular, given the fact that the emission of messages by network nodes is continuously enabled and is not blocked by the availability of a receiver, a network that can receive two distinct inputs need not satisfy the diamond property with respects to these inputs; what matters is that the network can reach equivalent states after each of the two inputs, either by executing the other input or not.

Definition 3. M is *confluent* if it is determinate and, for each of its derivatives M' and distinct actions α, β, where $\neg(\alpha \bowtie \beta)$, whenever $M' \xrightarrow{\alpha} M_1$ and $M' \stackrel{\beta}{\Longrightarrow} M_2$ then,

1. if $\alpha = r(id, t, \tilde{u}, \ell)$ and $\beta = r(id, t, \tilde{v}, \ell')$, then, either (1) $M_1 \approx M_2$, or (2) $M_1 \stackrel{\beta}{\Longrightarrow} M_1' \approx M_2$, or (3) $M_2 \stackrel{\alpha}{\Longrightarrow} M_2' \approx M_1$, or (4) $M_1 \stackrel{\beta}{\Longrightarrow} M_1'$, $M_2 \stackrel{\alpha}{\Longrightarrow} M_2'$, and $M_1' \approx M_2'$,
2. otherwise, there are M_1' and M_2' such that $M_2 \stackrel{\hat{\alpha}}{\Longrightarrow} M_2'$, $M_1 \stackrel{\hat{\beta}}{\Longrightarrow} M_1'$ and $M_1' \approx M_2'$.

We note that the first clause of the definition captures the four possibilities for bringing together two input actions by receiving further input after the first input, or not.

We may see that bisimilarity preserves confluence. Furthermore, confluent networks possess an interesting property regarding internal actions. We define a network M to be τ*-inert* if, for each derivative M_1 of M, if $M_1 \xrightarrow{\tau} M_2$ or $M_1 \xrightarrow{\mu} M_2$, then $M_1 \approx M_2$. By a generalization of the proof in CCS, we obtain:

Lemma 2. If M is confluent then M is τ-inert.

Although confluence is preserved by more operators than determinacy, namely the restriction operator, it is still not preserved by parallel composition. The counter-example provided in the case of determinacy is still valid: while M_1 and M_2 are confluent, $M_1 \mid M_2$ is not.

The main obstacle in establishing the compositionality of confluence with respect to parallel composition, as well as other operators, is that of node mobility. In what follows we extend our study of confluence in the context of stationary CSDT systems, that is, systems in which there is no movement of nodes (i.e. no μ action). The benefits of this study are twofold. On one hand, we develop a theory of confluence for broadcasting systems which turns out to be both simple as well as compositional. On the other hand, this theory remains useful in the context of mobility since, during the verification of ad hoc network systems, correctness criteria focus on the behavior of systems once their topology remains stable for a sufficient amount of time.

We begin by defining a new notion of weak transition that permits τ actions and excludes μ actions. Specifically, we write $M \Longrightarrow_s M'$, $M \stackrel{\alpha}{\Longrightarrow}_s M'$ and $M \stackrel{\hat{\alpha}}{\Longrightarrow}_s M'$ for the subsets of relations $M \Longrightarrow M'$, $M \stackrel{\alpha}{\Longrightarrow} M'$ and $M \stackrel{\hat{\alpha}}{\Longrightarrow} M'$ where no $\xrightarrow{\mu}$ are present. In a similar vein, the new notion of bisimilarity, S-bisimilarity matches the behavior of equivalent systems excluding μ-actions:

Definition 4. *S-Bisimilarity* is the largest symmetric relation, denoted by $\approx_s$, such that, if $M_1 \approx_s M_2$ and $M_1 \xrightarrow{\alpha} M_1'$, $\alpha \in \mathit{Act} - \{\mu\}$, there exists M_2' such that $M_2 \stackrel{\hat{\alpha}}{\Longrightarrow}_s M_2'$ and $M_1' \approx_s M_2'$.

It is easy to see that $\approx \subset \approx_s$ and that S-bisimilarity is a congruence relation. Based on the notion of S-bisimulation, we may define the notions of S-determinacy and S-confluence via variations of Definitions 2 and 3 which replace $\stackrel{a}{\Longrightarrow}$, $\approx$, and *Act* by $\stackrel{\alpha}{\Longrightarrow}_s$, $\approx_s$, and $\mathit{Act} - \{\mu\}$. We may see that S-bisimilarity preserves S-confluence and that S-confluent networks possess the property that their observable behavior remains unaffected by τ actions. In particular, we define a network M to be *τ_s-inert* if, for each derivative M_1 of M, if $M_1 \xrightarrow{\tau} M_2$, then $M_1 \approx_s M_2$ and we may prove:

Lemma 3. If M is S-confluent then M is τ_s-inert.

We prove that S-confluence is preserved by most CSDT operators including that of parallel composition.

Lemma 4

1. If $P{:}\sigma$ is S-confluent then so are $(\tau.P){:}\sigma$ and $(\bar{b}(w, t, \tilde{x}, tl).P){:}\sigma$.
2. If $P_i{:}\sigma$, $1 \leq i \leq n$, are S-confluent, then so is $\mathsf{cond}\ (e_1 \rhd P_1, \ldots, e_n \rhd P_n){:}\sigma$.
3. If M is S-confluent, then so is $M \backslash T$.
4. If M_1 and M_2 are S-confluent then so is $M_1 \mid M_2$.

3 Specification and Verification of a Leader-Election Algorithm

3.1 The Algorithm

The algorithm we consider for our case study is the distributed leader-election algorithm presented in [15]. It operates on an arbitrary topology of nodes with distinct identifiers and it elects as the leader of the network the node with the maximum identifier.

We first describe the static version of the algorithm which assumes that no topology changes are possible. We then proceed to extend this description to the mobile setting. In brief, the static algorithm operates as follows. In its initial state, a network node may initiate a leader-election computation (note that more than one node may do this) or accept leader-election requests from its neighbors. Once a node initiates a computation, it triggers communication between the network nodes which results into the creation of a spanning tree of the graph: each node picks as its father the node from which it received the first request and forwards the request to its remaining neighbors. Each node awaits to receive from each of its children the maximum identifier of the subtree at which they are rooted and, then, forward it to its father. Naturally, the root will receive the maximum identifier of the entire computation tree which is the elected leader. The leader is then flooded to the entire network.

Note that if more than one node initiates a leader-election computation then only one computation survives. This is established by associating each computation with a source identifier. Whenever a node already in a computation receives a request for a computation with a greater source, it abandons its original computation and it restarts a computation with this new identifier.

In the mobile setting, it is easy to observe that with node mobility, crashes and failures as well as network partitions and merges, the above algorithm is inadequate. To begin with, let us note that once a leader is elected it emits so-called *heartbeat* messages to the environment, which are essentially messages sent at a high transmission level. The absence of a heartbeat message from its leader triggers a node to initiate a new computation of a leader. Note that such an absence may just indicate that the node is outside of the high transmission range of the leader even though they belong to the same connected component of the network. However, this does not affect the correctness of the algorithm. Based on this extension, computation proceeds as described by the static algorithm with the exception of the following fine points:

- **Losing a child node.** If a node loses a child then it removes the child from the set of nodes from which it expects an acknowledgement and continues its computation.
- **Losing a parent node.** If a node loses its father then it assigns itself the role of the root of its subtree and continues its computation.
- **Partition Merges.** If two components of the system meet each other, they each proceed with their computation (if they are still computing a leader) ignoring any invitations to join the other's computation. Once they elect their individual leaders and start flooding their results, each adopts the leader with the largest identifier. An exception to this is the case when the two nodes that have come into contact have the same previous-leader field (a piece of information maintained in the mobile version of the algorithm), in which case they proceed as with the static case with the highest valued-computation being the winner.

3.2 Specification of the Protocol

In this subsection we model the algorithm in CSDT. We assume that messages exchanged by the network nodes must be accompanied by one of the following types:

- $\mathbf{election}$: used when a node invites another to join its computation.
- $\mathbf{ack1}$: used to notify the recipient that the sender has agreed to enter its computation and commits to forward the maximum identifier among its downward nodes.
- $\mathbf{ack0}$: used to notify the recipient that the sender has not agreed to be one of its children.
- $\mathbf{leader}$: used to announce the elected leader of a computation during the flooding process.
- $\mathbf{reply}$: used to announce the computation in which a node participates. Such messages are important for the following reason: If a node x departs from its location, enters a new computation and returns to its previous location before its initial neighbors notice its departure, these reply messages will notify initial neighbors that x is no longer part of their computation and thus they will no longer expect x to reply.
- $\mathbf{hbeat}$: a message of this type is emitted by a leader node.

In its initial state the algorithm is modeled by the process S consisting of a set of nodes who possess a leader (although this leader may be outside of their range).

$$\mathrm{S} \stackrel{\mathrm{def}}{=} (\prod_{k \in K} \mathrm{Elected}\langle b_k, s_k, lead_k \rangle{:}\sigma_k)\backslash T,$$

where $T = \{\mathbf{election}, \mathbf{ack0}, \mathbf{ack1}, \mathbf{leader}, \mathbf{reply}\}$. Thus, we restrict all but **hbeat** messages emitted by leader nodes which are the messages we wish to observe. Based on these messages we will subsequently express the correctness criterion of the algorithm.

The description of the behavior of nodes can be found in Figure 1. To begin with, a node in $\mathrm{Elected}$ possesses a leader $lead$, a source s which records the computation in which it has participated to elect its leader, and a status b which records whether or not it needs to broadcast its leader. Note that a leader node (i.e. a node with $id = lead$) regularly sends a heartbeat to notify its heartbeat neighbors of its availability. We may see in Figure 1 that if a process $\mathrm{Elected}$ receives an election message from one of its neighbors or, if it observes the absence of its leader (see condition $lead \notin \mathrm{H}$), it enters InComp mode, wherein it begins its quest for a new leader.

The InComp process has a number of parameters: c records whether the node should broadcast a leader election invitation to its neighbors and f is the node's father to which eventually an **ack1** message needs to be returned (unless the node itself is the root of the tree). src and $lead$ are the computation's source and previous leader, whereas max is the maximum identifier observed by the node. Sets R and A record the neighbors of the node that are expected to reply and the neighbors to which an **ack0** should be returned, respectively. Note that these sets are regularly updated according to the node's interface: we write $R' = R \cap \mathrm{N}$, $A' = A \cap \mathrm{N}$ and $f' = f$, if $f \in \mathrm{N}$, and *NULL*, otherwise. It is worth noting that at these points (as well as others, e.g. "$f = \mathrm{id}$", "$lead \in \mathrm{H}$"), the ability of referring to the node's interface plays a crucial role for the specification of the protocol. Furthermore, the fact that sets N and H are the *believed* sets of neighbors and may contain errors is realistic and it allows us to explore the correctness of the protocol in its full generality.

Finally, node $\mathrm{Leader}\langle f, src, max, lead\rangle$ awaits to be notified of the component's leader by its father f. It recalls its computation characterized by its source and previous leader (src and $lead$) as well as the maximum node it has observed from its downstream nodes, max. In case of the loss of its father, the node elects this maximum node as the leader of the component.

3.3 Verification of the Protocol

The aim of our analysis is to establish that after a finite number of topology changes, every connected component of the network, where connectedness is defined in terms of the normal transmission range, will elect the node with the maximum identifier as its leader. We consider an arbitrary derivative of S, namely S_1, where we assume that for all nodes, sets N and H are consistent with the network's state, and we show:

Theorem 1. $S_1 \approx_s (\prod_{k\in K} \mathrm{Elected}\langle 1, s, max_k\rangle{:}\sigma_k)\backslash T$ *where* max_k *is the maximum node identifier in the connected component of node* k.

In words, our correctness criterion states that, eventually, assuming nodes stop moving, all nodes will learn a leader which is the node with the maximum identifier within their connected component. We proceed to sketch the proof of the result which can be found in its entirety in [3].

$$\mathrm{Elected}\langle 0, src, lead\rangle \stackrel{\text{def}}{=} \overline{b}(-, \mathbf{leader}, \langle lead\rangle, \mathrm{n}).\, \mathrm{Elected}\langle 1, src, lead\rangle$$

$$\begin{array}{l}
\mathrm{Elected}\langle 1, src, lead\rangle \stackrel{\text{def}}{=} \\
\quad \mathsf{cond}\ (lead = \mathrm{id} \rhd \overline{b}(-, \mathbf{hbeat}, \langle lead\rangle, \mathrm{h}).\mathrm{Elected}\langle 1, src, lead\rangle, \\
\qquad\quad lead \notin \mathrm{H} \rhd \mathrm{InComp}\langle 0, \mathrm{id}, \mathrm{id}, \mathrm{N}, \emptyset, \mathrm{id}, lead\rangle) \\
\quad + r(\mathbf{leader}, \langle lead'\rangle).\, \mathsf{cond}\ (lead < lead' \rhd \mathrm{Elected}\langle 0, src, lead'\rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad true \rhd \mathrm{Elected}\langle 1, src, lead\rangle) \\
\quad + \overline{b}(-, \mathbf{reply}, \langle \mathrm{id}, s\rangle, \mathrm{n}).\, \mathrm{Elected}\langle 1, src, lead\rangle \\
\quad + r(\mathbf{election}, \langle j, l, s\rangle).\, \mathsf{cond}\ (l = lead \rhd \mathrm{InComp}\langle 0, j, s, \mathrm{N} - \{j\}, \emptyset, \mathrm{id}, lead\rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad true \rhd \mathrm{Elected}\langle 1, src, lead\rangle)
\end{array}$$

$$\mathrm{InComp}\langle c, NULL, src, R, A, max, lead\rangle \stackrel{\text{def}}{=} \mathrm{InComp}\langle c, \mathrm{id}, src, R', A', max, lead\rangle$$

$$\begin{array}{l}
\mathrm{InComp}\langle 1, f, src, \emptyset, \emptyset, max, lead\rangle \stackrel{\text{def}}{=} \\
\quad \mathsf{cond}\ (f = \mathrm{id} \rhd \mathrm{Elected}\langle 0, src, max\rangle, \\
\qquad\quad true \rhd \overline{b}(f, \mathbf{ack1}, \langle \mathrm{id}, scr, max\rangle, \mathrm{n}).\, \mathrm{Leader}\langle f, src, max, lead\rangle)
\end{array}$$

$$\begin{array}{l}
\mathrm{InComp}\langle c, f, src, R, A, max, lead\rangle \stackrel{\text{def}}{=} \\
\quad \mathsf{cond}\ (c = 0 \rhd \overline{b}(-, \mathbf{election}, \langle \mathrm{id}, lead, scr\rangle, \mathrm{n}).\, \mathrm{InComp}\langle 1, f', scr, R', A', max, lead\rangle) \\
\quad + \sum_{j \in A} \overline{b}(j, \mathbf{ack0}, \langle \mathrm{id}\rangle, \mathrm{n}).\, \mathrm{InComp}\langle 1, f', scr, R', A' - \{j\}, max, lead\rangle \\
\quad + r(\mathbf{election}, \langle j, l, s\rangle). \\
\qquad\quad \mathsf{cond}\ (l = lead \wedge s > src \rhd \mathrm{InComp}\langle 0, j, s, \mathrm{N} - \{j\}, \emptyset, max, lead\rangle, \\
\qquad\qquad\quad l = lead \wedge s = scr \rhd \mathrm{InComp}\langle c, f', scr, R', A' \cup \{j\}, max, lead\rangle, \\
\qquad\qquad\quad true \rhd \mathrm{InComp}\langle c, f, scr, R', A', max, lead\rangle) \\
\quad + r(\mathbf{ack0}, \langle j\rangle).\, \mathrm{InComp}\langle 1, f', src, R' - \{j\}, A', max, lead\rangle \\
\quad + r(\mathbf{ack1}, \langle j, s, max'\rangle). \\
\qquad\quad \mathsf{cond}\ (s = src \wedge max' > max \rhd \mathrm{InComp}\langle c, f', src, R' - \{j\}, A', max', lead\rangle, \\
\qquad\qquad\quad s = src \wedge max' \leq max \rhd \mathrm{InComp}\langle c, f', src, R' - \{j\}, A', max, lead\rangle, \\
\qquad\qquad\quad true \rhd \mathrm{InComp}\langle c, f', src, R' - \{j\}, A', max, lead\rangle) \\
\quad + r(\mathbf{leader}, \langle l\rangle).\, \mathrm{InComp}\langle c, f', src, R', A', max, lead\rangle \\
\quad + r(\mathbf{reply}, \langle j, s\rangle). \\
\qquad \mathsf{cond}\ (src \neq s \rhd \mathrm{InComp}\langle c, f', src, R' - \{j\}, A' - \{j\}, max, lead\rangle, \\
\qquad\qquad true \rhd \mathrm{InComp}\langle c, f', src, R', A', max, lead\rangle) \\
\quad + \overline{b}(-, \mathbf{reply}, \langle \mathrm{id}, src\rangle, \mathrm{n}).\, \mathrm{InComp}\langle c, f', src, R', A', max, lead\rangle
\end{array}$$

$$\mathrm{Leader}\langle NULL, src, max, lead\rangle \stackrel{\text{def}}{=} \mathrm{Elected}\langle 0, src, max_i\rangle$$

$$\begin{array}{l}
\mathrm{Leader}\langle f, src, max, lead\rangle \stackrel{\text{def}}{=} \\
\quad r(\mathbf{election}, \langle j, l, s\rangle). \\
\qquad \mathsf{cond}\ (l = lead \wedge s > src \rhd \mathrm{InComp}\langle 0, j, s, \mathrm{N} - \{j\}, \emptyset, max, lead\rangle, \\
\qquad\qquad true \rhd \mathrm{Leader}\langle f', src, max, lead\rangle) \\
\quad + r(\mathbf{leader}, \langle l\rangle).\, \mathrm{Elected}\langle 0, src, l\rangle \\
\quad + \overline{b}(-, \mathbf{reply}, \langle \mathrm{id}, src\rangle, \mathrm{n}).\, \mathrm{Leader}\langle f', scr, max, lead\rangle
\end{array}$$

Fig. 1. The description of a node

Sketch of Proof of Theorem 3

Let us consider an arbitrary derivative of S. This may be viewed as the composition of a set of connected components CC_g where independent leader-election computations are taking place within some subsets of nodes N_g. We prove the following:

Lemma 5. *$CC_g \approx_s Spec_1$, where $Spec_1 \stackrel{\text{def}}{=} (\prod_{i \in N_g} \text{Elected}\langle 1, s_i, max_g\rangle{:}\sigma_i)\backslash T$ and $max_g = max\{max_i | i \in N_g\}$.*

The proof takes place in two steps. In the first step, we consider a simplification of CC_g, F_g, where each node x in F_g is associated with a specific node being the unique node that x can accept as its father. We show that, by construction and Lemma 4, F_g is an S-confluent process and we exhibit an execution $F_g \Longrightarrow_s Spec_1$. Here we may observe the power of confluence: it is sufficient to study only a single execution of F_g. Then, by τ_s-inertness, S-confluence guarantees that $F_g \approx_s Spec_1$. For the second step of the proof we show that whenever $CC_g \Longrightarrow_s D$ where exists an F_g (i.e. an assignment of fathers to nodes) that is similar to D. Since this is true for any D we conclude that CC_g cannot diverge from the behavior of the F_g's and that it is in fact destined to produce only their behavior. Hence, we deduce that $CC_g \approx_s Spec_1$ as required.

Bearing in mind that network S_1 is a composition of the components CC_g, each concerning a distinct communication neighborhood of the network, S-confluence arguments along with considerations regarding the availability of believed maximum nodes allow us to deduce the correctness of the theorem. □

4 Conclusions

We have presented CSDT, a process calculus for reasoning about systems with broadcasting communication and dynamic interconnections. A salient aspect of CSDT is its ability to simulate a neighbor discovery protocol through its semantics and to associate nodes with their (believed) set of neighbors. Furthermore, we have allowed nodes to broadcast messages at two different transmission levels. As illustrated via our case study, the ability to do so is not only useful with respect to saving power but can also play an important role for protocol correctness. We point out that this could easily be extended to a wider range of transmission levels, by considering a set of transmission levels Tl and replacing sets N and H in a node's interface by a relation $\mathcal{I} \times Tl$. Finally, we have introduced the notion of a message type and a hiding operator based on types which allows an observer to focus on a subset of the message exchange. These message types are reminiscent of tags by which various applications prefix different messages according to their roles. As illustrated via our case study, this can be especially useful for expressing appropriate correctness criteria via bisimulations.

We have also studied the notion of confluence for CSDT in the presence and absence of node mobility. It turns out that in the case of stationary systems confluence yields a compositional theory which is remarkably simple compared to similar value-passing theories: the absence of channels for communication has removed a number of considerations relating to confluence preservation. We believe that the results can be extended to other calculi featuring a broadcasting style of communication. As in our case study, these results allow one to conclude the confluence of the analyzed systems merely by construction and then to deduce the desired bisimilarity via examining a single execution path and appealing to τ-inertness.

We have illustrated the applicability of the new formalism via the analysis of a leader-election MANET protocol. In [15], the authors also give a proof of their algorithm using

temporal logic: in particular they show a "weak form of stabilization of the algorithm", namely, that after a finite number of topological changes, the algorithm converges to a desired stable state in a finite amount of time. As we do in our proof, they operate under the assumption of no message loss. The same algorithm has also been considered in [13] where its correctness was analyzed for a number of tree and ring-structured initial topologies for networks with 5 to 8 nodes. It was shown automatically that it is possible that eventually a node with the maximum id in a connected component will be elected as the leader of the component and that every node connected to it via one or more hops will learn about its election. The reachability nature of this result is due to the lossy communication implemented in the ω-calculus.

In conclusion, we believe that CSDT can be applied for specifying and verifying a wide range of dynamic-topology protocols and that the theory of confluence can play an important role in facilitating the construction of their proofs. This belief is supported by our current work on specifying and verifying a MANET routing algorithm. In future work we plan to extend our framework in the presence of message loss.

References

1. Ghassemi, F., Fokkink, W., Movaghar, A.: Equational reasoning on mobile ad hoc networks. Fundamenta Informaticae 105(4), 375–415 (2010)
2. Godskesen, J.C.: A calculus for mobile ad-hoc networks with static location binding. Electronic Notes in Theoretical Computer Science 242(1), 161–183 (2009)
3. Kouzapas, D., Philippou, A.: A process calculus for systems with dynamic topology. Technical Report TR-10-01, University of Cyprus (2010)
4. Lanese, I., Sangiorgi, D.: An operational semantics for a calculus for wireless systems. Theoretical Computer Science 158(19), 1928–1948 (2010)
5. Merro, M.: An observational theory for mobile ad hoc networks. Information and Computation 207(2), 194–208 (2009)
6. Merro, M., Sibilio, E.: A timed calculus for wireless systems. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 228–243. Springer, Heidelberg (2010)
7. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
8. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts 1 and 2. Information and Computation 100, 1–77 (1992)
9. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theoretical Computer Science 367(1-2), 203–227 (2006)
10. Nestmann, U.: On Determinacy and Non-determinacy in Concurrent Programming. PhD thesis, University of Erlangen (1996)
11. Philippou, A., Walker, D.: On confluence in the π-calculus. In: ICALP 1997. LNCS, vol. 1256, pp. 314–324 (1997)
12. Prasad, K.V.S.: A calculus of broadcasting systems. Science of Computer Programming 25(2-3), 285–327 (1995)
13. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. In: Wang, A.H., Tennenholtz, M. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 296–314. Springer, Heidelberg (2008)
14. Tofts, C.: Proof Methods and Pragmatics for Parallel Programming. PhD thesis, University of Edinburgh (1990)
15. Vasudevan, S., Kurose, J., Towsley, D.: Design and analysis of a leader election algorithm for mobile ad hoc networks. In: Proceedings of ICNP 2004, pp. 350–360. IEEE Computer Society Press, Los Alamitos (2004)

On Asynchronous Session Semantics

Dimitrios Kouzapas[1], Nobuko Yoshida[1], and Kohei Honda[2]

[1] Imperial College London
[2] Queen Mary, University of London

Abstract. This paper studies a behavioural theory of the π-calculus with session types under the fundamental principles of the practice of distributed computing — asynchronous communication which is order-preserving inside each connection (session), augmented with asynchronous inspection of events (message arrivals). A new theory of bisimulations is introduced, distinct from either standard asynchronous or synchronous bisimilarity, accurately capturing the semantic nature of session-based asynchronously communicating processes augmented with event primitives. The bisimilarity coincides with the reduction-closed barbed congruence. We examine its properties and compare them with existing semantics. Using the behavioural theory, we verify that the program transformation of multithreaded into event-driven session based processes, using Lauer-Needham duality, is type and semantic preserving.

1 Introduction

Modern transports such as TCP in distributed networks provide reliable, ordered delivery of messages from a program on one computer to another, once a connection is established. In practical communications programming, two parties start a conversation by establishing a connection over such a transport and exchange semantically meaningful, formatted messages through this connection. The distinction between possibly non order-preserving communications outside of connection and order-preserving ones inside each connection is a key feature of this practice: order preservation allows proper handling of a sequence of messages following an agreed-upon conversation structure, while unordered deliveries across connections enhance asynchronous, efficient bandwidth usage. Further, asynchronous event processing [18] using locally *buffered* messages enables the receiver to asynchronously *inspect* and *consume* events/messages.

This paper investigates semantic foundations of asynchronously communicating processes, capturing these key elements of modern communications programming – distinction between non order-preserving communications outside connections and the order-preserving ones inside each connection, as well as the incorporation of asynchronous inspection of message arrivals. We use the π-calculus augmented with session primitives, buffers and a simple event inspection primitive. Typed sessions capture structured conversations in connections with type safety; while a buffer represents an intermediary between a process and its environment, capturing non-blocking nature of communications, and enabling asynchronous event processing. The formalism is intended to be an idealised but expressive core communications programming language, offering a basis for a tractable semantic study. Our study shows that the combination

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 228–243, 2011.
© IFIP International Federation for Information Processing 2011

of these basic elements for modern communications programming leads to a rich behavioural theory which differs from both the standard synchronous communications semantics and the fully asynchronous one [8], captured through novel equational laws for asynchrony. These laws can then be used as a semantic justification of a well-known program transformation based on Lauer and Needham's duality principle [15], which translates multithreaded programs to their equivalent single-threaded, asynchronous, event-based programs. This transformation is regularly used in practice, albeit in an ad-hoc manner, playing a key role in e.g. high-performance servers. Our translation is given formally, is type-preserving and is backed up by a rigorous semantic justification. While we do not detail in the main sections, the transform is implemented in the session-extension of Java [14,13], resulting in competitive performance in comparison with more ad-hoc transformations.

Let us outline some of the key technical ideas of the present work informally. In the present theory, the asynchronous order-preserving communications over a connection are modelled as *asynchronous session communication*, extending the synchronous session calculus [24,9] with *message queues* [5,12,6]. A message queue, written $s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$, encapsulates input buffer ($\mathtt{i}$) with elements $\vec{h}$ and output buffer ($\mathtt{o}$) with $\vec{h}'$. Figure below represents the two end points of a session. A message v is first enqueued by a sender $s!\langle v\rangle;P$ at its output queue at s, which intuitively represents a communication pipe extending from the sender's locality to the receiver's. The message will eventually reach the receiver's locality, formalised as its transfer from the sender's output buffer (at s) to the receiver's input buffer (at $\overline{s}$). For a receiver, only when this transfer takes place, a visible (and asynchronous) message reception takes place, since only then the receiver can *inspect* and *consume* the message (as shown in **Remote** below).Note that dequeuing and enqueing actions inside a location are local to each process and is therefore invisible (τ-actions) (**Local** below).

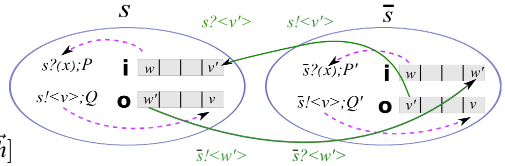
Local (the dashed arrows)

$$s!\langle v\rangle;Q \mid s[\mathtt{o}:\vec{h}] \xrightarrow{\tau} Q \mid s[\mathtt{o}:\vec{h}\cdot v]$$

$$s?(x).P \mid s[\mathtt{i}:w\cdot\vec{h}] \xrightarrow{\tau} P\{w/x\} \mid s[\mathtt{i}:\vec{h}]$$

Remote (the solid arrows)

$$s[\mathtt{i}:\vec{h}] \xrightarrow{s?\langle v\rangle} s[\mathtt{i}:\vec{h}\cdot v] \quad s[\mathtt{o}:v\cdot\vec{h}] \xrightarrow{s!\langle v\rangle} s[\mathtt{o}:\vec{h}]$$



The induced semantics captures the nature of asynchronous observables not studied before. For example, in weak asynchronous bisimilarity in the asynchronous π-calculus ($\approx_a$ in [8,10]), the message order is not observable ($s!\langle v_1\rangle \mid s!\langle v_2\rangle \approx_a s!\langle v_2\rangle \mid s!\langle v_1\rangle$) but in our semantics, messages for the same destination do *not* commute ($s!\langle v_1\rangle;s!\langle v_2\rangle \not\approx s!\langle v_2\rangle;s!\langle v_1\rangle$) as in the synchronous semantics [20] ($\approx_s$ in [8,10]); whereas two inputs for different targets commute ($s_1?(x);s_2?(y);P \approx s_2?(x);s_1?(y);P$) since the dequeue action is not observable, differing from the synchronous semantics, $s_1?(x);s_2?(y);P \not\approx_s s_2?(x);s_1?(y);P$.

Asynchronous event-handling [13] introduces further subtleties in observational laws. Asynchronous event-based programming is characterised by reactive flows driven by the detection of *events*, that is *message arrivals* at local buffers. In our formalism, this facility is distilled as an `arrived` predicate: e.g., $Q = \mathtt{if\ arrived}\ s\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ reduces to P_1 if the s input buffer contains one or more message; otherwise Q reduces to P_2. By `arrived`, we can observe the *movement of messages between two locations*.

For example, $Q \mid s[\mathtt{i}:\emptyset] \mid \overline{s}[\mathtt{o}:v]$ is not equivalent with $Q \mid s[\mathtt{i}:v] \mid \overline{s}[\mathtt{o}:\emptyset]$ because the former can reduce to P_2 (since v has not arrived at the local buffer at s yet) while the latter cannot.

Online appendix [23] lists the full definition of Lauer-Needham transformation, the detailed definitions, full proofs and the benchmark results in Session-based Java which demonstrate the potential of the session-type based translation as semantically transparent optimisation techniques.

2 Asynchronous Network Communications in Sessions

2.1 Syntax and Operational Semantics

We use a sub-calculus of the eventful session π-calculus [13], defined below.

$$\text{(Identifier) } u ::= a,b \mid x,y \quad k ::= s,\overline{s} \mid x,y \quad n ::= a,b \mid s,\overline{s} \quad \text{(Value) } v ::= \mathtt{tt},\mathtt{ff} \mid a,b \mid s,\overline{s}$$

$$\text{(Expression) } e ::= v \mid x,y,z \mid \mathtt{arrived}\, u \mid \mathtt{arrived}\, k \mid \mathtt{arrived}\, k\, h$$

$$\begin{aligned}\text{(Process) } P,Q ::= &\ u(x).P \mid \overline{u}(x);P \mid k!\langle e\rangle;P \mid k?(x).P \mid k \triangleleft l;P \mid k \triangleright \{l_i : P_i\}_{i\in I} \\ \mid &\ \mathtt{if}\, e\, \mathtt{then}\, P\, \mathtt{else}\, Q \mid (\nu\, a)P \mid P \mid Q \mid \mathbf{0} \mid \mu X.P \mid X \\ \mid &\ a[\vec{s}] \mid \overline{a}\langle s\rangle \mid (\nu\, s)P \mid s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}'] \qquad \text{(Message) } h ::= v \mid l\end{aligned}$$

Values $v, v', \ldots$ include *constants* $(\mathtt{tt},\mathtt{ff})$, *shared channels* a,b,c and *session channels* s,s'. A session channel denotes one endpoint of a session: s and $\overline{s}$ denote two ends of a single session, with $\overline{\overline{s}} = s$. Labels for branching and selection range over $l, l', \ldots$, variables over x,y,z, and process variables over X,Y,Z. Shared channel identifiers u,u' denote shared channels/variables; session identifiers k,k' are session endpoints and variables. n denotes either a or s. Expressions e are values, variables and the message arrival predicates ($\mathtt{arrived}\, u$, $\mathtt{arrived}\, k$ and $\mathtt{arrived}\, k\, h$: the last one checks for the arrival of the specific message h at k). $\vec{s}$ and $\vec{h}$ stand for vectors of session channels and messages respectively. ε denotes the empty vector.

We distinguish two kinds of asynchronous communications, *asynchronous session initiation* and *asynchronous session communication* (over an established session). The former involves the *unordered* delivery of a *session request message* $\overline{a}\langle s\rangle$, where $\overline{a}\langle s\rangle$ represents an asynchronous message in transit towards an acceptor at a, carrying a fresh session channel s. As in actual network, a request message will first move through the network and eventually get buffered at a receiver's end. Only then a message arrival can be detected. This aspect is formalised by the introduction of a *shared channel input queue* $a[\vec{s}]$, often called *shared input queue* for brevity, which denotes an acceptor's local buffer at a with pending session requests for $\vec{s}$. The intuitive meaning of the endpoint configuration $s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$ is explained in Introduction.

Requester $\overline{u}(x);P$ requests a session initiation, while acceptor $u(x).P$ accepts one. Through an established session, output $k!\langle e\rangle;P$ sends e through channel k asynchronously, input $k?(x).P$ receives through k, selection $k \triangleleft l;P$ chooses the branch with label l, and branching $k \triangleright \{l_i : P_i\}_{i\in I}$ offers branches. The $(\nu\, a)P$ binds a channel a, while $(\nu\, s)P$ binds the two endpoints, s and $\overline{s}$, making them private within P. The conditional, parallel composition, recursions and inaction are standard. $\mathbf{0}$ is often omitted. For brevity, one or more components may be omitted from a configuration when they are

irrelevant, writing e.g. $s[\mathtt{i}:\vec{h}]$ which denotes the input part of $s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$. The notions of free variables and channels are standard [21]; $\mathrm{fn}(P)$ denotes the set of free channels in P. $\overline{a}\langle s\rangle$, $(\nu s)P$ and $s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$ only appear at runtime. A process without free variables is called *closed* and a closed process without runtime syntax is called *program*.

Rule	Reduction
[Request1]	$\overline{a}(x);P \longrightarrow (\nu s)(P\{\overline{s}/x\} \mid \overline{s}[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid \overline{a}\langle s\rangle)\quad (s \notin \mathrm{fn}(P))$
[Request2]	$a[\vec{s}] \mid \overline{a}\langle s\rangle \longrightarrow a[\vec{s}\cdot s]$
[Accept]	$a(x).P \mid a[s\cdot\vec{s}] \longrightarrow P\{s/x\} \mid s[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid a[\vec{s}]$
[Send,Recv]	$s!\langle v\rangle;P \mid s[\mathtt{o}:\vec{h}] \longrightarrow P \mid s[\mathtt{o}:\vec{h}\cdot v]\qquad s?(x).P \mid s[\mathtt{i}:v\cdot\vec{h}] \longrightarrow P\{v/x\} \mid s[\mathtt{i}:\vec{h}]$
[Sel,Bra]	$s\lhd l_i;P \mid s[\mathtt{o}:\vec{h}] \longrightarrow P \mid s[\mathtt{o}:\vec{h}\cdot l_i]\quad s\rhd\{l_j:P_j\}_{j\in J} \mid s[\mathtt{i}:l_i\cdot\vec{h}] \longrightarrow P_i \mid s[\mathtt{i}:\vec{h}]\quad (i\in J)$
[Comm]	$s[\mathtt{o}:v\cdot\vec{h}] \mid \overline{s}[\mathtt{i}:\vec{h}'] \longrightarrow s[\mathtt{o}:\vec{h}] \mid \overline{s}[\mathtt{i}:\vec{h}'\cdot v]$
[Areq]	$E[\mathtt{arrived}\,a] \mid a[\vec{s}] \longrightarrow E[\mathtt{b}] \mid a[\vec{s}]\qquad (\lvert\vec{s}\rvert \geq 1)\searrow \mathtt{b}$
[Ases]	$E[\mathtt{arrived}\,s] \mid s[\mathtt{i}:\vec{h}] \longrightarrow E[\mathtt{b}] \mid s[\mathtt{i}:\vec{h}]\qquad (\lvert\vec{h}\rvert \geq 1)\searrow \mathtt{b}$
[Amsg]	$E[\mathtt{arrived}\,s\,h] \mid s[\mathtt{i}:\vec{h}] \longrightarrow E[\mathtt{b}] \mid s[\mathtt{i}:\vec{h}]\qquad (\vec{h} = h\cdot\vec{h}')\searrow \mathtt{b}$

The above table defines the reduction relation over closed terms. The key rules are given in Figure above. We use the standard evaluation contexts $E[_]$ defined as $E ::= - \mid s!\langle E\rangle;P \mid \mathtt{if}\ E\ \mathtt{then}\ P\ \mathtt{else}\ Q$. The structural congruence $\equiv$ and the rest of the reduction rules are standard. We set $\twoheadrightarrow = (\longrightarrow \cup \equiv)^*$.

The first three rules define the initialisation. In [Request1], a client requests a server for a fresh session via shared channel a. A fresh session channel, with two ends s (server-side) and $\overline{s}$ (client-side) as well as the empty configuration at the client side, are generated and the session request message $\overline{a}\langle s\rangle$ is dispatched. Rule [Request2] enqueues the request in the shared input queue at a. A server accepts a session request from the queue using [Accept], instantiating its variable with s in the request message; the new session is now established. Asynchronous order-preserving session communications are modelled by the next four rules. Rule [Send] enqueues a value in the o-buffer at the *local* configuration; rule [Receive] dequeues the first value from the i-buffer at the local configuration; rules [Sel] and [Bra] similarly enqueue and dequeue a label. The arrival of a message at a remote site is embodied by [Comm], which removes the first message from the o-buffer of the sender configuration and enqueues it in the i-buffer at the receiving configuration.

Output actions are always non-blocking. An input action can block if no message is available at the corresponding local input buffer. The use of the message arrivals can avoid this blocking: [Areq] evaluates $\mathtt{arrived}\,a$ to tt iff the queue is non-empty ($e \searrow \mathtt{b}$ means e evaluates to the boolean b); similarly for $\mathtt{arrived}\,k$ in [Areq]. [Amsg] evaluates $\mathtt{arrived}\,s\,h$ to tt iff the buffer is nonempty and its next message matches h.

2.2 Types and Typing

The type syntax follows the standard session types from [9].

(Shared) $U ::= \mathsf{bool} \mid \mathtt{i}\langle S\rangle \mid \mathtt{o}\langle S\rangle \mid \mathsf{X} \mid \mu\mathsf{X}.U$ (Value) $T ::= U \mid S$

(Session) $S ::= !(T);S \mid ?(T);S \mid \oplus\{l_i:S_i\}_{i\in I} \mid \&\{l_i:S_i\}_{i\in I} \mid \mu\mathsf{X}.S \mid \mathsf{X} \mid \mathtt{end}$

The shared types U include booleans bool (and, in examples, naturals nat); shared channel types $\mathtt{i}\langle S\rangle$ (input) and $\mathtt{o}\langle S\rangle$ (output) for shared channels through which a session

of type S is established; type variables $(\mathsf{X},\mathsf{Y},\mathsf{Z},..)$; and recursive types. The IO-types (often called server/client types) ensure a unique server and many clients [11]. In the present work they are used for controlling locality (queues are placed only at the server sides) and associated typed transitions, playing a central role in our behavioural theory. In session types, output type $!(T);S$ represents outputting values of type T, then performing as S. Dually for input type $?(T);S$. Selection type $\oplus\{l_i : S_i\}_{i\in I}$ describes a selection of one of the labels say l_i then behaves as T_i. Branching type $\&\{l_i : S_i\}_{i\in I}$ waits with I options, and behaves as type T_i if i-th label is chosen. End type $\mathtt{end}$ represents the session completion and is often omitted. In recursive type $\mu\mathsf{X}.S$, type variables are guarded in the standard sense.

The judgements of processes and expressions are $\Gamma \vdash P \triangleright \Delta$ and $\Gamma,\Delta \vdash e : T$, with $\Gamma ::= \emptyset \mid \Gamma\cdot u{:}U \mid \Gamma\cdot X{:}\Delta$ and $\Delta ::= \emptyset \mid \Delta\cdot a \mid \Delta\cdot k{:}\mathsf{T} \mid \Delta\cdot s$ where session type is extended to $\mathsf{T} ::= M;S \mid M$ with $M ::= \emptyset \mid \oplus l \mid \& l \mid !(T) \mid ?(T) \mid M;M$ which represents types for values stored in queues (note $\emptyset;S = S$). Γ is called *shared environment*, which maps shared channels and process variables to, respectively, constant types and value types; Δ is called *linear environment* maps session channels to session types and recording shared channels for acceptor's input queues and session channels for end-point queues. The judgement is read: program P is typed under shared environment Γ, uses channels as linear environment Δ. In the expression judgement, expression e has type T under Γ, and uses channels as linear environment Δ. We often omit Δ if it is clear from the context. The typing system is similar with [13,2], and can be found in online Appendix [23]. We say that Δ *well configured* if $s : S \in \Delta$, then $\overline{s}{:}\overline{S} \in \Delta$. We define: $\{s{:}!(T);S\cdot\overline{s}{:}?(T);S'\} \longrightarrow \{s{:}S\cdot\overline{s}{:}S'\}$, $\{s{:}\oplus\{l_i : S_i\}_{i\in I}\cdot\overline{s}{:}\&\{l_i : S'_i\}_{i\in I}\} \longrightarrow \{s{:}S_k\cdot\overline{s}{:}S'_k\}$ $(k \in I)$, and $\Delta\cup\Delta'' \longrightarrow \Delta'\cup\Delta''$ if $\Delta \longrightarrow \Delta'$.[1]

Proposition 2.1 (Subject Reduction). *if $\Gamma \vdash P \triangleright \Delta$ and $P \twoheadrightarrow Q$ and Δ is well-configured, then we have $\Gamma \vdash Q \triangleright \Delta'$ such that $\Delta \longrightarrow^* \Delta'$ and Δ' is well-configured.*

3 Asynchronous Session Bisimulations and Its Properties

3.1 Labelled Transitions and Bisimilarity

Untyped and Typed LTS. This section studies the basic properties of behavioural equivalences. We use the following labels $(\ell,\ell',...)$:

$$\ell ::= a\langle s\rangle \mid \overline{a}\langle s\rangle \mid \overline{a}(s) \mid s?\langle v\rangle \mid s!\langle v\rangle \mid s!(a) \mid s\& l \mid s\oplus l \mid \tau$$

where the labels denote the session accept, request, bound request, input, output, bound output, branching, selection and the τ-action. $\mathsf{sbj}(\ell)$ denotes the set of free subjects in ℓ; and $\mathsf{fn}(\ell)$ (resp. $\mathsf{bn}(\ell)$) denotes the set of free (resp. bound) names in ℓ. The symmetric operator $\ell \asymp \ell'$ on labels that denotes that ℓ is a dual of ℓ', is defined as: $a\langle s\rangle \asymp \overline{a}\langle s\rangle$, $a\langle s\rangle \asymp \overline{a}(s)$, $s?\langle v\rangle \asymp \overline{s}!\langle v\rangle$, $s?\langle a\rangle \asymp \overline{s}!(a)$, and $s\& l \asymp \overline{s}\oplus l$.

[1] In the following sections, we study semantic properties of typed processes: however these developments can be understood without knowing the details of the typing rules. This is because the properties of the typing system are captured by the typed LTS defined in section 3.1 later.

$\langle\mathsf{Acc}\rangle \quad a[\vec{s}] \xrightarrow{a\langle s\rangle} a[\vec{s}\cdot s] \qquad \langle\mathsf{Req}\rangle \quad \overline{a}\langle s\rangle \xrightarrow{\overline{a}\langle s\rangle} \mathbf{0} \qquad \langle\mathsf{In}\rangle \quad s[\mathtt{i}:\vec{h}] \xrightarrow{s?\langle v\rangle} s[\mathtt{i}:\vec{h}\cdot v]$

$\langle\mathsf{Out}\rangle \quad s[\mathtt{o}:v\cdot\vec{h}] \xrightarrow{s!\langle v\rangle} s[\mathtt{o}:\vec{h}] \qquad \langle\mathsf{Bra}\rangle \quad s[\mathtt{i}:\vec{h}] \xrightarrow{s\& l} s[\mathtt{i}:\vec{h}\cdot l] \qquad \langle\mathsf{Sel}\rangle \quad s[\mathtt{o}:l\cdot\vec{h}] \xrightarrow{s\oplus l} s[\mathtt{o}:h]$

$\langle\mathsf{Local}\rangle \dfrac{P \longrightarrow Q}{P \xrightarrow{\tau} Q} \quad \langle\mathsf{Par}\rangle \dfrac{P \xrightarrow{\ell} P' \quad \mathsf{bn}(\ell)\cap\mathsf{fn}(Q)=\emptyset}{P|Q \xrightarrow{\ell} P'|Q} \quad \langle\mathsf{Tau}\rangle \dfrac{P \xrightarrow{\ell} P' \quad Q \xrightarrow{\ell'} Q' \quad \ell\asymp\ell'}{P|Q \xrightarrow{\tau} (\nu\,\mathsf{bn}(\ell,\ell'))(P'|Q')}$

$\langle\mathsf{Res}\rangle \dfrac{P \xrightarrow{\ell} P' \quad n\notin\mathsf{fn}(\ell)}{(\nu n)P \xrightarrow{\ell} (\nu n)P'} \quad \langle\mathsf{OpS}\rangle \dfrac{P \xrightarrow{\overline{a}\langle s\rangle} P'}{(\nu s)P \xrightarrow{\overline{a}(s)} P'} \quad \langle\mathsf{OpN}\rangle \dfrac{P \xrightarrow{s!\langle a\rangle} P'}{(\nu a)P \xrightarrow{s!(a)} P'} \quad \langle\mathsf{Alpha}\rangle \dfrac{P \equiv_\alpha P' \quad P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q}$

Rule $\langle\mathsf{Local}\rangle$ is defined from the reductions by [Request1, Accept, Send, Recv, Bra, Sel, Areq, Ases, Amsg] as well as [Comm] when the communication object is a session (delegation).

In the untyped labelled transition system (LTS) defined above, $\langle\mathsf{Acc}\rangle/\langle\mathsf{Req}\rangle$ are for the session initialisation. The next four rules $\langle\mathsf{In}\rangle/\langle\mathsf{Out}\rangle/\langle\mathsf{Bra}\rangle/\langle\mathsf{Sel}\rangle$ say the action is observable when it moves from its local queue to its remote queue. When the process accesses its local queue, the action is *invisible* from the outside, as formalised by $\langle\mathsf{Local}\rangle$. In contrast, $\langle\mathsf{Com}\rangle$ expresses an interaction between two local configutations. This distinction is useful in our later proofs. Other compositional rules are standard. Based on the LTS, we use the standard notations [19] such as $P \stackrel{\ell}{\Longrightarrow} Q$, $P \stackrel{\vec{\ell}}{\Longrightarrow} Q$ and $P \stackrel{\hat{\ell}}{\Longrightarrow} Q$.

We define the typed LTS on the basis of the untyped one, using the type information to control the enabling of actions. This is realised by introducing the *environment transition*, defined below. A transition $(\Gamma,\Delta) \xrightarrow{\ell} (\Gamma',\Delta')$ means that an environment (Γ,Δ) allows an action ℓ to take place, and the resulting environment is (Γ',Δ'), constraining process transitions through the linear and shared environments. This constraint is at the heart of our typed LTS, accurately capturing interactions in the presence of sessions and local buffers. We write $\Gamma_1 \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma_2 \vdash P_2 \triangleright \Delta_2$ if $P_1 \xrightarrow{\ell} P_2$ and $(\Gamma_1,\Delta_1) \xrightarrow{\ell} (\Gamma_2,\Delta_2)$ with $\Gamma_i \vdash P_i \triangleright \Delta_i$. Similarly for other transition relations.

$$
\begin{array}{rrl}
\Gamma(a)=\mathtt{i}\langle S\rangle, a\in\Delta, s \text{ fresh} \Rightarrow & (\Gamma,\Delta) & \xrightarrow{a\langle s\rangle} (\Gamma,\Delta\cdot s:\overline{S}) \\
\Gamma(a)=\mathtt{o}\langle S\rangle, a\notin\Delta \Rightarrow & (\Gamma,\Delta) & \xrightarrow{\overline{a}\langle s\rangle} (\Gamma,\Delta) \\
\Gamma(a)=\mathtt{o}\langle S\rangle, a\notin\Delta, s \text{ fresh} \Rightarrow & (\Gamma,\Delta) & \xrightarrow{\overline{a}(s)} (\Gamma,\Delta\cdot s:S) \\
\Gamma\vdash v:U \text{ and } U\neq\mathtt{i}\langle S'\rangle \text{ and } \overline{s}\notin\mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:!(U);S) & \xrightarrow{s!\langle v\rangle} (\Gamma,\Delta\cdot s:S) \\
\overline{s}\notin\mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:!(\mathtt{o}\langle S'\rangle);S) & \xrightarrow{s!(a)} (\Gamma\cdot a:\mathtt{o}\langle S'\rangle,\Delta\cdot s:S) \\
\Gamma\vdash v:U \text{ and } U\neq\mathtt{i}\langle S'\rangle \text{ and } \overline{s}\notin\mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:?(U);S) & \xrightarrow{s?\langle v\rangle} (\Gamma,\Delta\cdot s:S) \\
\overline{s}\notin\mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:\oplus\{l_i:S_i\}_{i\in I}) & \xrightarrow{s\oplus l_k} (\Gamma,\Delta\cdot s:S_k) \\
\overline{s}\notin\mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:\&\{l_i:S_i\}_{i\in I}) & \xrightarrow{s\& l_k} (\Gamma,\Delta\cdot s:S_k) \\
\Delta\longrightarrow\Delta' \Rightarrow & (\Gamma,\Delta) & \xrightarrow{\tau} (\Gamma,\Delta')
\end{array}
$$

The first rule says that reception of a message via a is possible only when a is input-typed (i-mode) and its queue is present ($a\in\Delta$). The second is dual, saying that an output at a is possible only when a has o-mode and no queue exists. Similarly for a bound output action. The two session output rules ($\ell = s!\langle v\rangle$ and $s!(a)$) are the standard value output and a scope opening rule. The next is for value input. Label input and output are defined similarly. Note that we send and receive only a shared channel which

has o-mode. This is because a new accept should not be created without its queue in the same location. The final rule ($\ell = \tau$) follows the reduction rules defined before Proposition 2.1. The LTS omits delegations since it is not necessary in the bisimulation we consider (due to the notion of localisation, see the next paragraph).

Write $\bowtie$ for the symmetric and transitive closure of $\longrightarrow$ over linear environments. We say a relation on typed processes is a *typed relation* if, whenever it relates two typed processes, we have $\Gamma \vdash P_1 \triangleright \Delta_1$ and $\Gamma \vdash P_2 \triangleright \Delta_2$ such that $\Delta_1 \bowtie \Delta_2$. We write $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ if $(\Gamma \vdash P_1 \triangleright \Delta_1, \Gamma \vdash P_2 \triangleright \Delta_2)$ are in a typed relation $\mathscr{R}$. Further we often leave the environments implicit, writing simply $P_1 \mathscr{R} P_2$.

Localisation and Bisimulation. Our bisimulation is a typed relation over those processes which are *localised*, in the sense that they are equipped with all necessary local queues. We say an environment Δ is *delegation-free* if it contains types which are generated by deleting S from value type T in the syntax of types defined in § 2.2 (i.e. either $!(S);S'$ or $?(S);S'$ does not appear in Δ). Similarly for Γ. Now let P be closed and $\Gamma \vdash P \triangleright \Delta$ where Γ and Δ are delegation-free (note that P can perform delegations at hidden channels by $\langle \mathsf{Local} \rangle$). Then we say P is *localised* w.r.t. Γ, Δ if (1) For each $s : S \in \mathsf{dom}(\Delta)$, $s \in \Delta$; and (2) if $\Gamma(a) = \mathtt{i}\langle S \rangle$, then $a \in \Delta$. We say P *is localised* if it is so for a suitable pair of environments. For example, $s?(x);s!\langle x+1 \rangle;\mathbf{0}$ is not localised, but $s?(x);s!\langle x+1 \rangle;\mathbf{0} \mid s[\mathtt{i}:\vec{h}_1, \mathtt{o}:\vec{h}_2]$ is. Similarly, $a(x).P$ is not localised, but $a(x).P \mid a[\vec{s}]$ is. By composing buffers at appropriate channels, any typable closed process can become localised. If P is localised w.r.t. (Γ, Δ) then $\Gamma \vdash P \triangleright \Delta \stackrel{l}{\longrightarrow} \Gamma' \vdash P' \triangleright \Delta'$ implies P' is localised w.r.t. (Γ', Δ') (in the case of τ-transition, note queues always stay). We can now introduce the reduction congruence and the asynchronous bisimilarity.

Definition 3.1 (Reduction Congruence). We write $P \downarrow a$ if $P \equiv (\nu\, \vec{n})(\overline{a}\langle s \rangle \mid R)$ with $a \notin \vec{n}$. Similarly we write $P \downarrow s$ if $P \equiv (\nu\, \vec{n})(s[\mathtt{o}: h \cdot \vec{h}] \mid R)$ with $s \notin \vec{n}$. $P \Downarrow n$ means $\exists P'.P \longrightarrow\!\!\!\!\!\rightarrow P' \downarrow n$. A typed relation $\mathscr{R}$ is *reduction congruence* if it is a congruence and satisfies the following conditions for each $P_1 \mathscr{R} P_2$ whenever they are localised w.r.t. their given environments.

1. $P_1 \Downarrow n$ iff $P_2 \Downarrow n$.
2. Whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ holds, $P_1 \longrightarrow\!\!\!\!\!\rightarrow P_1'$ implies $P_2 \longrightarrow\!\!\!\!\!\rightarrow P_2'$ such that $\Gamma \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$ holds with $\Delta_1' \bowtie \Delta_2'$ and the symmetric case.

The maximum reduction congruence which is not a universal relation exists [10] which we call *reduction congruency*, denoted by $\cong$.

Definition 3.2 (Asynchronous Session Bisimulation). A typed relation $\mathscr{R}$ over localised processes is a *weak asynchronous session bisimulation* or often a *bisimulation* for brevity, if, whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$, the following two conditions holds: (1) $\Gamma \vdash P_1 \triangleright \Delta_1 \stackrel{\ell}{\longrightarrow} \Gamma' \vdash P_1' \triangleright \Delta_1'$ implies $\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\hat{\ell}}{\Longrightarrow} \Gamma' \vdash P_2' \triangleright \Delta_2'$ such that $\Gamma' \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$ with $\Delta_1' \bowtie \Delta_2'$ holds and (2) the symmetric case of (1). The maximum bisimulation exists which we call *bisimilarity*, denoted by $\approx$. We sometimes leave environments implicit, writing e.g. $P \approx Q$.

We extend $\approx$ to possibly non-localised closed terms by relating them when their minimal localisations are related by $\approx$ (given $\Gamma \vdash P \triangleright \Delta$, its *minimal localisation* adds empty

queues to P for the input shared channels in Γ and session channels in Δ that are missing their queues). Further $\approx$ is extended to open terms in the standard way [10].

3.2 Properties of Asynchronous Session Bisimilarity

Characterisation of Reduction Congruence. This subsection studies central properties of asynchronous session semantics. We first show that the bisimilarity coincides with the naturally defined reduction-closed congruence [10], given below.

Theorem 3.3 (Soundness and Completeness). $\approx = \cong$.

The soundness ($\approx \subset \cong$) is by showing $\approx$ is congruent. The most difficult case is a closure under parallel composition, which requires to check the side condition $\Delta_1' \bowtie \Delta_2'$ for each case. The completeness ($\cong \subset \approx$) follows [7, § 2.6] where we prove that every external action is definable by a testing process, see [23].

Asynchrony and Session Determinacy. Let us call ℓ an *output action* if ℓ is one of $\overline{a}\langle s\rangle, \overline{a}(s), s!\langle v\rangle, s!(a), s\oplus l$; and an *input action* if ℓ is one of $a\langle s\rangle, s?\langle v\rangle, s\&l$. In the following, the first property says that we can delay an output arbitrarily, while the second says that we can always immediately perform a (well-typed) input.

Lemma 3.4 (Input and Output Asynchrony). *Suppose* $\Gamma \vdash P \triangleright \Delta \stackrel{\ell}{\Longrightarrow} P' \triangleright \Delta'$.

– (output delay) *If* ℓ *is an output action, then* $\Gamma \vdash P \triangleright \Delta \Longrightarrow\stackrel{\ell}{\longrightarrow} P' \triangleright \Delta'$.
– (input advance) *If* ℓ *is an input action, then* $\Gamma \vdash P \triangleright \Delta \stackrel{\ell}{\longrightarrow}\Longrightarrow P' \triangleright \Delta'$.

The asynchronous interaction on the session buffers enables inputs to happen before multi-internal steps and outputs to happen after multi-internal steps.

Following [22], we define determinacy and confluence. Below and henceforth we often omit the environments in typed transitions.

Definition 3.5 (Determinacy). We say $\Gamma' \vdash Q \triangleright \Delta'$ is *derivative* of $\Gamma \vdash P \triangleright \Delta$ if there exists $\vec{\ell}$ such that $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow} \Gamma' \vdash Q \triangleright \Delta'$. We say $\Gamma \vdash P \triangleright \Delta$ is *determinate* if for each derivative Q of P and action ℓ, if $Q \stackrel{\ell}{\longrightarrow} Q'$ and $Q \stackrel{\hat{\ell}}{\Longrightarrow} Q''$ then $Q' \approx Q''$.

We then extend the above notions to session communications.

Definition 3.6 (Session Determinacy). Let us write $P \stackrel{\ell}{\longrightarrow}_s Q$ if $P \stackrel{\ell}{\longrightarrow} Q$ where if $\ell = \tau$ then it is generated without using [Request1], [Request2], [Accept], [Areq] nor [Amsg] from reduction rules (i.e. a communication is performed without arrival predicates or accept actions). We extend the definition to $\stackrel{\vec{\ell}}{\Longrightarrow}_s$ and $\stackrel{\hat{\ell}}{\Longrightarrow}_s$ etc. We say P is *session determinate* if P is typable, is localised and if $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow} Q \triangleright \Delta'$ then $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow}_s Q \triangleright \Delta'$. We call such Q a *session derivative* of P.

We define $\ell_1 \lfloor \ell_2$ ("residual of ℓ_1 after ℓ_2") as (1) $\overline{a}\langle s\rangle$ if $\ell_1 = \overline{a}(s')$ and $s' \in \mathsf{bn}(\ell_2)$; (2) $s!\langle s'\rangle$ if $\ell_1 = s!(s')$ and $s' \in \mathsf{bn}(\ell_2)$; (3) $s!\langle a\rangle$ if $\ell_1 = s!(a)$ and $a \in \mathsf{bn}(\ell_2)$; and otherwise ℓ_1. We write $l_1 \bowtie l_2$ when $l_1 \neq l_2$ and if l_1, l_2 are input actions, $\mathsf{sbj}(l_1) \neq \mathsf{sbj}(l_2)$.

Definition 3.7 (Confluence). $\Gamma \vdash P \triangleright \Delta$ is *confluent* if for each derivative Q of P and ℓ_1, ℓ_2 such that $\ell_1 \bowtie \ell_2$, (i) if $Q \stackrel{\ell}{\longrightarrow} Q_1$ and $Q \stackrel{\ell}{\Longrightarrow} Q_2$, then $Q_1 \Longrightarrow Q_1'$ and $Q_2 \Longrightarrow Q_2' \approx Q_1'$; and (ii) if $Q \stackrel{\ell_1}{\longrightarrow} Q_1$ and $Q \stackrel{\ell_2}{\Longrightarrow} Q_2$, then $Q_1 \stackrel{\widehat{\ell_2 \lfloor \ell_1}}{\Longrightarrow} Q_1'$ and $Q_2 \stackrel{\widehat{\ell_1 \lfloor \ell_2}}{\Longrightarrow} Q_2' \approx Q_1'$.

Lemma 3.8. *Let P be session determinate and $\Gamma \vdash P \Longrightarrow Q \triangleright \Delta$. Then $P \approx Q$.*

Theorem 3.9 (Session Determinacy). *Let P be session determinate. Then P is determinate and confluent.*

The following relation is used to prove the event-based optimisation. The proof of the following lemma is by showing $\Longrightarrow \mathscr{R} \Longleftarrow$ with $\Longrightarrow$ determinate is a bisimulation.

Definition 3.10 (Determinate Upto-expansion Relation). Let $\mathscr{R}$ be a symmetric, typed relation such that if $\Gamma \vdash P \triangleright \Delta \ \mathscr{R} \ Q \triangleright \Delta$ and (1) P, Q are determinate; (2) If $\Gamma \vdash P \triangleright \Delta \stackrel{l}{\longrightarrow} \Gamma' \vdash P'' \triangleright \Delta''$ then $\Gamma \vdash Q \triangleright \Delta \stackrel{l}{\Longrightarrow} \Gamma' \vdash Q' \triangleright \Delta'$ and $\Gamma' \vdash P'' \triangleright \Delta'' \Longrightarrow \Gamma' \vdash P' \triangleright \Delta'$ with $\Gamma' \vdash P' \triangleright \Delta' \mathscr{R} \ Q' \triangleright \Delta'$; and (3) the symmetric case. Then we call $\mathscr{R}$ a *determinate upto-expansion relation*, or often simply *upto-expansion relation*.

Lemma 3.11. *Let $\mathscr{R}$ be an upto-expansion relation. Then $\mathscr{R} \subset \approx$.*

4 Lauer-Needham Transform

In an early work [15], Lauer and Needham observed that a concurrent program may be written equivalently either in a thread-based programming style (with shared memory primitives) or in an event-based style (with a single-threaded event loop processing messages sequentially with non-blocking handlers). Following this framework and using high-level asynchronous event primitives such as *selectors* [18] for the event-based style, many studies compare these two programming styles, often focusing on performance of server architectures (see [13, § 6] for recent studies on event programming). These implementations implicitly or explicitly assume a *transformation* from a program written in the thread-based style, especially those which generate a new thread for each service request (as in thread-based web servers), to its *equivalent* event-based program, which treats concurrent services using a single threaded event-loop (as in event-based web servers). However the precise semantic effects of such a transformation nor the exact meaning of the associated "equivalence" has not been clarified.

We study the semantic effects of such a transformation using the asynchronous session bisimulation. We first specify both event and thread based programming models and introduce a formal mapping from a thread-based process to their event-based one, following [15]. As a threaded system we assume a server process whose code creates fresh threads at each service invocation. The key idea is to decompose this whole code into distinct smaller code segments, each handling the part of the original code starting from a blocking action. Such a blocking action is represented as reception of a message (input or branching). Then a single global event-loop can treat each message arrival by processing the corresponding code segment combined with an environment, returning to inspect the content of event/message buffers. We first stipulate a class of processes which we consider for our translation. Below $*a(x);P$ denotes an *input replication* abbreviating $\mu X.a(x).(P|X)$.

Definition 4.1 (Server). A *simple server at a* is a closed process $*a(x).P$ with a typing of form $a:\mathtt{i}\langle S\rangle, b_1:\mathtt{o}\langle S_1\rangle,..,b_n:\mathtt{o}\langle S_n\rangle$ where P is sequential (i.e. contains no parallel composition $|$) and is determinate and under any localisation. A simple server is often considered with its localisation with an empty queue $a[\varepsilon]$.

A server spawns an unbounded number of threads as it receives session requests repeatedly. Each thread may initiate other sessions with outside, and its interactions may involve delegations and name passing. Definition 4.1 assumes two conditions: (1) determinacy and (2) sequential processing of each event. A practical example of (1) is a web server which only serves static web pages. As will be discussed later, determinacy plays an essential role in our proofs while sequentiality is for simplicity of the mapping. Given a server $*a(w:S);P \mid a[\varepsilon]$, its translation, which we call *Lauer-Needham transform* or *LN-transform* for short, is written $\mathscr{LN}[\![*a(w:S);P \mid a[\varepsilon]]\!]$(the full mapping is non-trivial and given in [23]). The key elements of $\mathscr{LN}[\![*a(w:S);P]\!]$ follow:

1. A *selector* handles events on message arrival. Its function includes a *selector queue* $q\langle\varepsilon\rangle$ that stores sessions whose arrival is continuously inspected. Its initial element is $\langle a, c_0\rangle$. This data says: "if a message comes at a, jump to the code block (CPS procedure) whose subject is c_0".
2. A collection of *code blocks* $\mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle$, CPS procedures handling incoming messages. A code block originates from a threaded server *blocking subterm*, i.e. a subterm starting from an input or a branching.
3. $\mathsf{Loop}\langle o,q\rangle$ implements the *event-loop*. It passes execution from the selector to a code block in CPS style, after a successful arrive inspection from the selector.

We use the standard "select" primitive represented as a process, called *selector* [13]. It stores *a collection of session channels*, with each channel associated with an environment, binding variables to values. It then picks up one of them at which a message arrives, receives that message via that channel and has it be processed by the corresponding code block. Finally it stores the session and the associated environment back in the collection, and moves to the next iteration. Since a selector should handle channels of different types, it uses the `typecase` construct from [13]. $\mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i:T_i)\,P_i\}_{i\in I}$ takes a session endpoint k and a list of cases $(x_i:T_i)$, each binding the free variable x_i of type pattern T_i in P_i. Its reduction is defined as:

$$\mathtt{typecase}\ s\ \mathtt{of}\ \{(x_i:T_i)\,P_i\}_{i\in I} \mid s\,[S,\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}'] \longrightarrow P_j\{s/x_j\} \mid s\,[S,\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$$

where $j\in I$ such that $(\forall i<j.T_i \not\leq S \wedge T_j \leq S)$ where $\leq$ denotes a subtyping relation. The `typecase` construct finds a match of the session type of the tested channel among the session types in its list, and proceeds with the corresponding process. For the matching to take place, session endpoint configuration syntax is extended with the runtime session typing [13]. The selectors are defined by the following reduction relations:

$$\mathtt{new\ selector}\ r\ \mathtt{in}\ P \longrightarrow (\nu\, r)(P \mid \mathsf{sel}\langle r,\varepsilon\rangle) \quad \mathtt{register}\langle s',r\rangle;P \mid \mathsf{sel}\langle r,\vec{s}\rangle \longrightarrow P \mid \mathsf{sel}\langle r,\vec{s}\cdot s'\rangle$$

$$\begin{aligned}
&\mathtt{let}\ x=\mathtt{select}(r)\ \mathtt{in\ typecase}\ x\ \mathtt{of}\ \{(x_i:T_i):P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[S,\mathtt{i}:\vec{h}] \\
&\qquad\longrightarrow\ P_i\{s'/x_i\} \mid \mathsf{sel}\langle r,\vec{s}\rangle \mid s'\,[S,\mathtt{i}:\vec{h}] \quad (\vec{h}\neq\varepsilon) \\
&\mathtt{let}\ x=\mathtt{select}(r)\ \mathtt{in\ typecase}\ x\ \mathtt{of}\ \{(x_i:T_i):P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[\mathtt{i}:\varepsilon] \\
&\qquad\longrightarrow\ \mathtt{let}\ x=\mathtt{select}(r)\ \mathtt{in\ typecase}\ x\ \mathtt{of}\ \{(x_i:T_i):P_i\}_{i\in I} \mid \mathsf{sel}\langle r,\vec{s}\cdot s'\rangle \mid s'\,[\mathtt{i}:\varepsilon]
\end{aligned}$$

where in the third line S and T_i satisfy the condition for `typecase` in the reduction rule. The last two rules integrate reductions for `typecase` to highlight their combined usage (which is in effect the only way the selection is meaningfully performed). Operator `new selector` r `in` P (binding r in P) creates a new selector $\mathsf{sel}\langle r, \varepsilon\rangle$, named r and with the empty queue ε. Operator $\mathtt{register}\langle s', r\rangle; P$ registers a session channel s to r, adding s' to the original queue $\vec{s}$. The next `let` retrieves a registered session and checks the availability to test if an event has been triggered. If so, find the match of the type of s' among $\{T_i\}$ and select P_i; if not, the next session is tested. As proved in [13], these primitives are encodable in the original calculus augmented with `typecase`. The bisimulations and their properties (such as congruency of $\approx$) remain unchanged.

Example 4.1 (Lauer-Needham Transform). As an example of a server, consider:

$$P = *a(x);x?(y).x!\langle y+1\rangle;x?(z).x!\langle y+z\rangle;\mathbf{0} \mid a[\varepsilon]$$

This process has the session type $?(\mathsf{nat});!(\mathsf{nat})?(\mathsf{nat});!(\mathsf{nat})$ at a, and can be read: *a process should first expect to receive (?) a message of type* nat *and send (!) it, then to receive (? again) a* nat*, and finish by sending (!) a result.* We extract the blocking subterms from this process as follows.

Blocking Process	Type at Blocking Prefix
$a(x).x?(y).x!\langle y+1\rangle x?(z).x!\langle y+z\rangle;\mathbf{0}$	$\mathtt{i}\langle ?(\mathsf{nat});!(\mathsf{nat});?(\mathsf{nat});!(\mathsf{nat})\rangle$
$x?(y).x!\langle y+1\rangle x?(z).x!\langle y+z\rangle;\mathbf{0}$	$?(\mathsf{nat});!(\mathsf{nat});?(\mathsf{nat});!(\mathsf{nat})$
$x?(z).x!\langle y+z\rangle;\mathbf{0}$	$?(\mathsf{nat});!(\mathsf{nat})$

These blocking processes are translated into *code blocks* (CodeBlocks) given as:

$$\begin{array}{l} *c_0(y);a(x).\mathtt{update}(y,x,x);\ \mathtt{register}\ \langle sel,x,y,c_1\rangle;\overline{o} \mid \\ *c_1(x,y);x?(z);\mathtt{update}(y,z,z);\ x!\langle [\![z]\!]_y+1\rangle;\mathtt{register}\ \langle sel,x,y,c_2\rangle;\overline{o} \mid \\ *c_2(x,y);x?(z');\mathtt{update}(y,z',z');x!\langle [\![z]\!]_y+[\![z']\!]_y\rangle;\overline{o} \end{array}$$

which processes each message, using environments to record threads' states. The operation $\mathtt{update}(y,x,x);$ updates an environment, while `register` stores the blocking session channel, the associated continuation c_i and the current environment y in the selector queue sel.

Finally, using these code blocks, the main event-loop denoted Loop, is given as:

$$\begin{array}{lll} \mathsf{Loop} = *o.\mathtt{let}\ (x,y,z) = \mathtt{select\ from}\ sel\ \mathtt{in\ typecase}\ x\ \mathtt{of}\ \{ & & \\ \quad \mathtt{i}\langle ?(\mathsf{nat});!(\mathsf{nat});?(\mathsf{nat});!(\mathsf{nat})\rangle & : & \mathtt{new}\ y:\mathtt{env\ in}\ \overline{z}(y) \\ \quad ?(\mathsf{nat});!(\mathsf{nat});?(\mathsf{nat});!(\mathsf{nat}) & : & \overline{z}(x,y) \\ \quad ?(\mathsf{nat});!(\mathsf{nat}) & : & \overline{z}(x,y)\} \end{array}$$

Above `select from` sel `in` selects a message from the selector queue sel, and treats it in P. The `new` construct creates a new environment y. The `typecase` construct then branches into different processes depending on the session of the received message, and dispatch the task to each code block.

The determinate property allows us to conclude that:

Lemma 4.2. $*a(w:S);R \mid a[\varepsilon]$ *is confluent.*

We can now establish the correctness of the transform. The proofs of the following results are found in [23], which use a determinate upto-expansion relation (Definition 3.10) through Lemmas 3.11. First we give a basic equation for the standard event loop, handling events by non-blocking handlers. We use recursive equations of agents for legibility, which can be easily encoded into recursions.

$$P_1 = \texttt{if arrived}\, s_1 \,\texttt{then}\, (s_1?(x).R_1);P_2 \,\texttt{elseif arrived}\, s_2 \,\texttt{then}\, (s_2?(x).R_2);P_1 \,\texttt{else}\, P_1$$

$$P_2 = \texttt{if arrived}\, s_2 \,\texttt{then}\, (s_2?(x).R_2);P_1 \,\texttt{elseif arrived}\, s_1 \,\texttt{then}\, (s_1?(x).R_1);P_2 \,\texttt{else}\, P_2$$

where we assume well-typedness and each of $R_{1,2}$, under any closing substitution and localisation, is determinate and reaches $\mathbf{0}$ after a series of outputs and τ-actions. The sequencing $(s_1?(x).R_1);P_2$ denotes the process obtained by replacing each $\mathbf{0}$ in R_1 with P_2. We can then show $P_1 \approx P_2$ by using the up-to-expansion relation. The following lemma proves its generalisation, elucidating the selectors behaviour in the stateless environment. It says that we can permute the session channels in a selector queue while keeping the same behaviour because of the determinacy of the server's code.

Lemma 4.3. *Let* $P \stackrel{def}{=} \mu X.\texttt{let}\ x = \texttt{select}(r)\ \texttt{in}\ \texttt{typecase}\ x\ \texttt{of}\ \{(x_i : T_i) : R_i;X\}_{i \in I}$ *where each* R_i *is determinate and reaches* $\mathbf{0}$ *after a sequence of non-blocking actions (outputs and* τ*-actions as well as a single input/branching action at* x_i*). The sequencing* $R_i;X$ *is defined as above. Then, assuming typability, we have* $P \mid \mathsf{sel}\langle r, \vec{s}_1 \cdot s_1' \cdot s_2' \cdot \vec{s}_2\rangle \approx P \mid \mathsf{sel}\langle r, \vec{s}_1 \cdot s_2' \cdot s_1' \cdot \vec{s}_2\rangle$.

Thus the selector's behaviour can be matched with the original threaded behaviour step by step in the sense that there is no difference between which event (resp. thread) is selected to be executed first. We conclude:

Theorem 4.4 (Semantic Preservation). *Let* $*a(w:S);R \mid a\,[\varepsilon]$ *be a simple server. Then* $*a(w:S);P \mid a\,[\varepsilon] \approx \mathscr{LN}[\![a(w:S);P \mid a\,[\varepsilon]]\!]$.

5 Discussions

Comparisons with Asynchronous/Synchronous Calculi. We give comprehensive comparisons with other calculi, clarifying the relationship between (1) the session-typed asynchronous π-calculus [8] without queues ($\approx_a$, the asynchronous version of the labelled transition relation for the asynchronous π-calculus), (2) the session-typed synchronous π-calculus [24,9] without queues ($\approx_s$), (3) the asynchronous session π-calculus with two end-point queues without IO queues [6,5,21] ($\approx_2$), and (4) the asynchronous session π-calculus with two end-point IO-queues ($\approx$), i.e. the one developed in this paper. The semantics of (2) is called *non-local* since the output process directly puts the value into the input queue. The transition relation for non-local semantics (2) is defined by replacing the output and selection rules in the LTS relation to:

$$\langle \mathsf{Out}_n \rangle \;\; s!\langle v\rangle;P \xrightarrow{s!\langle v\rangle} P \qquad\qquad \langle \mathsf{Sel}_n \rangle \;\; s \oplus l;P \xrightarrow{s\oplus l} P$$

See [23] for the full definitions and proofs. The following figure summarises distinguishing examples. Non-Blocking Input/Output means inputs/outputs on different channels, while the Input/Output Order-Preserving means that the messages will be received/ delivered preserving the order. The final table explains whether Lemma 3.4 (1) (input

advance) or (2) (output delay) is satisfied or not. If not, we place a counterexample (in (4), $\Pi_{1,2}[s_i,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$ means $[s_1,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid [s_2,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$).

	Non-Blocking Input	Non-Blocking Output
(1)	$s_1?(x);s_2?(y);P \approx_a s_2?(y);s_1?(x);P$	$\overline{s_1}\langle v\rangle \mid \overline{s_2}\langle w\rangle \mid P \approx_a \overline{s_1}\langle w\rangle \mid \overline{s_2}\langle v\rangle \mid P$
(2)	$s_1?(x);s_2?(y);P \not\approx_s s_2?(y);s_1?(x);P$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \not\approx_s s_2!\langle w\rangle;s_1!\langle v\rangle;P$
(3)	$s_1?(x);s_2?(y);P \mid s_1[\varepsilon] \mid s_2[\varepsilon] \approx_2 s_2?(y);s_1?(x);P \mid s_1[\varepsilon] \mid s_2[\varepsilon]$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid s_1[\varepsilon] \mid s_2[\varepsilon] \not\approx_2 s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid s_1[\varepsilon] \mid s_2[\varepsilon]$
(4)	$s_1?(x);s_2?(y);P \mid \Pi_{1,2}[s_i,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \approx s_2?(y);s_1?(x);P \mid \Pi_{1,2}[s_i,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid \Pi_{1,2}[s_i,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \approx s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid \Pi_{1,2}[s_i,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$

	Input Order-Preserving	Output Order-Preserving
(1)	$s?(x);s?(y);P \approx_a s?(y);s?(x);P$	$\overline{s}\langle v\rangle \mid \overline{s}\langle w\rangle \mid P \approx_a \overline{s}\langle w\rangle \mid \overline{s}\langle v\rangle \mid P$
(2)	$s?(x);s?(y);P \not\approx_s s?(y);s?(x);P$	$s!\langle v\rangle;s!\langle w\rangle;P \not\approx_s s!\langle w\rangle;s!\langle v\rangle;P$
(3)	$s?(x);s?(y);P \mid s[\varepsilon] \not\approx_2 s?(x);s?(y);P \mid s[\varepsilon]$	$s!\langle v\rangle;s!\langle w\rangle;P \mid s[\varepsilon] \not\approx_2 s!\langle w\rangle;s!\langle v\rangle;P \mid s[\varepsilon]$
(4)	$s?(x);s?(y);P \mid \Pi_{1,2}[s_i,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \not\approx s?(x);s?(y);P \mid \Pi_{1,2}[s_i,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$	$s!\langle v\rangle;s!\langle w\rangle;P \mid \Pi_{1,2}[s_i,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \not\approx s!\langle w\rangle;s!\langle v\rangle;P \mid \Pi_{1,2}[s_i,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$

	Lemma 3.4 (1)	Lemma 3.4 (2)
(1)	yes	yes
(2)	$(\nu\ s)(s!\langle v\rangle;s'?(x);\mathbf{0} \mid s?(x);\mathbf{0})$	$(\nu\ s)(s!\langle v\rangle;s'!\langle v'\rangle;\mathbf{0} \mid s'?(x);\mathbf{0})$
(3)	yes	$s!\langle v\rangle;\ s'?(x);\mathbf{0} \mid s'[v']$
(4)	yes	yes

Another technical interest is the effects of the `arrived` predicate on these combinations. We define the synchronous and asynchronous π-calculi augmented with the `arrived` predicate and local buffers. For the asynchronous π-calculus, we add $a[\vec{h}]$ and `arrived` a in the syntax, and define the following rules for input and outputs.

$$\overline{a}\langle v\rangle \xrightarrow{\overline{a}\langle v\rangle} \mathbf{0} \qquad a[\vec{h}] \xrightarrow{a\langle h\rangle} a[\vec{h}\cdot h] \qquad \texttt{if arrived}\ a\ \texttt{then}\ P\ \texttt{else}\ Q | a[\varepsilon] \xrightarrow{\tau} Q \mid a[\varepsilon]$$

$$a?(x).P \mid a[\vec{h_1}\cdot h_i\cdot\vec{h_2}] \longrightarrow P\{h_i/x\} \mid a[\vec{h_1}\cdot\vec{h_2}] \qquad \texttt{if arrived}\ a\ \texttt{then}\ P\ \texttt{else}\ Q | a[\vec{h}] \xrightarrow{\tau} P \mid a[\vec{h}]$$

where, in the last rule, $|\vec{h}| \geq 1$. The above definition precludes the order preservation as the property of transport, but still keeps the non-blocking property as in the asynchronous π-calculus. The synchronous version is similarly defined by setting the buffer size to be one. The non-local version is defined just by adding `arrived` predicate.

Let $Q = \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$ with $P_1 \not\approx P_2$. If the syntax does not include arrival predicates, we have $Q \mid s[\mathtt{i}:\emptyset] \mid \overline{s}[\mathtt{o}:v] \approx Q \mid s[\mathtt{i}:v] \mid \overline{s}[\mathtt{o}:\emptyset]$. In the presence of the arrival predicate, we have $Q \mid s[\mathtt{i}:\emptyset] \mid \overline{s}[\mathtt{o}:v] \not\approx Q \mid s[\mathtt{i}:v] \mid \overline{s}[\mathtt{o}:\emptyset]$ with $e = \texttt{arrived}\ s$. Interestingly in all of the calculi (1–4), the same example as the above, which separate semantics with/without the `arrived`, are effective.

The IO queues provide non-blocking inputs and outputs, while preserving the input/output ordering, which distinguishes the present framework from other known semantics. As a whole, we observe that the present semantic framework is closer to the asynchronous bisimulation (1) $\approx_a$, augmented with order-preserving nature per session. Its key properties arise from local, buffered session semantics and typing. We have also seen the semantic significance of the `arrived` predicates, which enables processes to observe the effects of fine-grained synchronisations.

Related Work. Some of the key proof methods of our work draw their ideas from [22], which study an extension of the confluence theory on the π-calculus. Our work differs in that we investigate the effect of asynchronous IO queues and its relationship to confluence. The work [1] examines expressiveness of various messaging mediums by adding message bags (no ordering), stacks (LIFO policy) and message queues (FIFO policy) in the asynchronous π-calculus [8]. They show that the calculus with the message bags is encodable into the asynchronous π-calculus, but it is impossible to encode the message queues and stacks. Neither the effects of locality, queues, typed transitions, and event-based programming are studied.

Programming constructs that can test the presence of actions or events are studied in the context of the Linda language [3] and CSP [16,17]. The work [3] measures expressive powers between three variants of asynchronous Linda-like calculi, with a construct for inspecting the output in the tuple space, which is reminiscent of the *inp* predicate of Linda. The first calculus (called *instantaneous*) corresponds to (1) [8], the second one (called *ordered*) formalises emissions of messages to the tuple spaces, and the third one (called *unordered*) models unordered outputs in the tuple space by decomposing one messaging into two stages — emission from an output process and rendering from the tuple space. It shows that the instantaneous and ordered calculi are Turing powerful, while the unordered is not. The work [16] studies CSP with a construct that checks if a parallel process is able to perform an output action on a given channel and a subsequent work [17] investigates the expressiveness of its variants focusing on the full abstraction theorem of the trace equivalence. Our calculi (1,2,3,4) are Turing powerful and we aim to examine properties and applications of the typed bisimilarity characterised by buffered sessions: on the other hand, the focus of [3] is a tuple space where our input/output order preserving examples (which treat different objects with the same session channel) cannot be naturally (and efficiently) defined. The same point applies to [16,17]. As another difference, the nature of localities has not been considered either in [3,16,17] since no notion of a local or remote tuple or environment is defined. Further, none of the above work [3,16,17,22,1] treats large applications which include these constructs (§ 4) or the performance analysis of the proposed primitives.

Using eventful session types, we have demonstrated that our bisimulation theory is applicable, through the verification of the correctness of the Lauer-Needham transform. The asynchronous nature realised through IO message queues provides a precise analysis of local and eventful behaviours, found in major distributed transports such as TCP. The benchmark results from high-performance clusters in [23] show that the throughput for the thread-eliminated Server implementations in Session Java [13] exhibits higher throughput than the multithreaded Server implementations, justifying the effect of the type and semantic preserving LN-transformation.

As the future work, we plan to investigate bisimulation theories under multiparty session types [12] and a relationship with a linear logic interpretation of sessions, which connects a behavioural theory and permutation laws under locality assumption [4].

Acknowledgements. We thank Raymond Hu for collaborations and the reviewers for their useful comments. The work is partially supported by EP/F003757/1, EP/G015635/1, EP/G015481/1 and EP/F002114/1.

References

1. Beauxis, R., Palamidessi, C., Valencia, F.: On the asynchronous nature of the asynchronous π-calculus. In: Degano, P., De Nicola, R., Bevilacqua, V. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 473–492. Springer, Heidelberg (2008)
2. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
3. Busi, N., Gorrieri, R., Zavattaro, G.: Comparing three semantics for Linda-like languages. Theor. Comput. Sci. 240(1), 49–90 (2000)
4. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010)
5. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N.: Asynchronous Session Types and Progress for Object Oriented Languages. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 1–31. Springer, Heidelberg (2007)
6. Gay, S., Vasconcelos, V.T.: Linear type theory for asynchronous session types. JFP (2009)
7. Hennessy, M.: A Distributed Pi-Calculus. CUP (2007)
8. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
9. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
10. Honda, K., Yoshida, N.: On reduction-based process semantics. TCS 151(2), 437–486 (1995)
11. Honda, K., Yoshida, N.: A uniform type structure for secure information flow. TOPLAS 29(6) (2007)
12. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM Press, New York (2008)
13. Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K.: Type-safe eventful sessions in java. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 329–353. Springer, Heidelberg (2010)
14. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 516–541. Springer, Heidelberg (2008)
15. Lauer, H.C., Needham, R.M.: On the duality of operating system structures. SIGOPS Oper. Syst. Rev. 13(2), 3–19 (1979)
16. Lowe, G.: Extending csp with tests for availability. In: Procedings of Communicating Process Architectures, CPA 2009 (2009)
17. Lowe, G.: Models for csp with availability information. In: EXPRESS 2010. EPTCS, vol. 41, pp. 91–105 (2010)
18. S. Microsystems Inc. New IO APIs, `http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html`
19. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
20. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. Info. & Comp. 100(1) (1992)

21. Mostrous, D., Yoshida, N.: Session-based communication optimisation for higher-order mobile processes. In: Curien, P.-L. (ed.) TLCA 2009. LNCS, vol. 5608, pp. 203–218. Springer, Heidelberg (2009)
22. Philippou, A., Walker, D.: On confluence in the picalculus. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 314–324. Springer, Heidelberg (1997)
23. Online Appendix of this paper, `http://www.doc.ic.ac.uk/~dk208/semantics.html`
24. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)

Towards Verification of the Pastry Protocol Using TLA$^+$

Tianxiang Lu[1,2], Stephan Merz[2], and Christoph Weidenbach[1]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
{tianlu,weidenbach}@mpi-inf.mpg.de
[2] INRIA Nancy & LORIA, Nancy, France
Stephan.Merz@loria.fr

Abstract. Pastry is an algorithm that provides a scalable distributed hash table over an underlying P2P network. Several implementations of Pastry are available and have been applied in practice, but no attempt has so far been made to formally describe the algorithm or to verify its properties. Since Pastry combines rather complex data structures, asynchronous communication, concurrency, resilience to *churn* and fault tolerance, it makes an interesting target for verification. We have modeled Pastry's core routing algorithms and communication protocol in the specification language TLA$^+$. In order to validate the model and to search for bugs we employed the TLA$^+$ model checker TLC to analyze several qualitative properties. We obtained non-trivial insights in the behavior of Pastry through the model checking analysis. Furthermore, we started to verify Pastry using the very same model and the interactive theorem prover TLAPS for TLA$^+$. A first result is the reduction of global Pastry correctness properties to invariants of the underlying data structures.

Keywords: formal specification, model checking, verification methods, network protocols.

1 Introduction

Pastry [9,3,5] is an overlay network protocol that implements a distributed hash table. The network nodes are assigned logical identifiers from an Id space of naturals in the interval $[0, 2^M - 1]$ for some M. The Id space is considered as a ring, i.e., $2^M - 1$ is the neighbor of 0. The Ids serve two purposes. First, they are the logical network addresses of nodes. Second, they are the keys of the hash table. An active node is in particular responsible for keys that are numerically close to its network Id, i.e., it provides the primary storage for the hash table entries associated with these keys. Key responsibility is divided equally according to the distance between two neighbor nodes. If a node is responsible for a key we say it *covers* the key.

The most important sub-protocols of Pastry are *join* and *lookup*. The join protocol eventually adds a new node with an unused network Id to the ring.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 244–258, 2011.
© IFIP International Federation for Information Processing 2011

The lookup protocol delivers the hash table entry for a given key. An important correctness property of Pastry is *Correct Key Delivery*, requiring that there is always at most one node responsible for a given key. This property is non-trivial to obtain in the presence of spontaneous arrival and departure of nodes. Nodes may simply drop off, and Pastry is meant to be robust against such changes, i.e., *churn*. For this reason, every node holds two *leaf sets* of size l containing its closest neighbors to either side (l nodes to the left and l to the right). A node also holds the hash table content of its leaf set neighbors. If a node detects, e.g. by a ping, that one of its direct neighbor nodes dropped off, the node takes actions to recover from this state. So the value of l is relevant for the amount of "drop off" and fault tolerance of the protocol.

A lookup request must be routed to the node responsible for the key. Routing using the leaf sets of nodes is possible in principle, but results in a linear number of steps before the responsible node receives the message. Therefore, on top of the leaf sets of a node a routing table is implemented that enables routing in a logarithmic number of steps in the size of the ring.

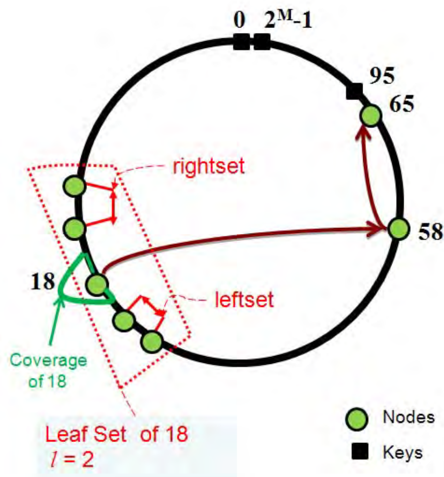


Fig. 1. Pastry Routing Example

Pastry routes a message by forwarding it to nodes that match progressively longer prefixes with the destination key. In the example of Fig. 1, node 18 received a lookup message for key 95. The key is outside node 18's coverage and furthermore, it doesn't lie between the leftmost node and the rightmost node of its leaf sets. Querying its routing table, node 18 finds node 58, whose identifier matches the longest prefix with the destination key and then forwards the message to that node. Node 58 repeats the process and finally, the lookup message is answered by node 65, which is the closest node to the key 95, i.e., it covers

key 95. In this case, we say that node 65 *delivers* the lookup request for key 95 (see also Fig. 4).

The first challenge in modeling Pastry was to determine an appropriate level of abstraction. As a guiding principle, we focused the model towards supporting detailed proofs of the correctness properties. We abstracted from an explicit notion of time because it does not contribute to the verification of correctness properties. For example, time-triggered periodic maintenance messages exchanged between neighbors are modelled by non-deterministic sending of such messages. In contrast, we developed a detailed model for the address ring, the routing tables, the leaf sets, as well as the messages and actions of the protocol because these parts are central to the correctness of Pastry.

The second challenge was to fill in needed details for the formal model that are not contained in the published descriptions of Pastry. Model checking was very helpful for justifying our decisions. For instance, it was not explicitly stated what it means for a leaf set to be "complete", i.e., when a node starts taking over coverage and becoming an active member on the ring. It was not stated whether an overlap between the right and left leaf set is permitted or whether the sets should always be disjoint. We made explicit assumptions on how such corner cases should be handled, sometimes based on an exploration of the source code of the FreePastry implementation [8]. Thus, we implemented an overlap in our model only if there are at most $2l$ nodes present on the entire network, where l is the size of each leaf set. A complete leaf set only contains less than l nodes, if there are less than l nodes on the overall ring.

A further challenge was to formulate the correctness property; in fact, it is not stated explicitly in the literature [9,3,5]. The main property that we are interested in is that the lookup message for a particular key is answered by at most one "ready" node covering the key. We introduced a more fine-grained status notion for nodes, where only "ready" nodes answer lookup and join requests. The additional status of a node being "ok" was added in the refined model described in Section 4 to support several nodes joining simultaneously between two consecutive "ready" nodes.

The paper is organized as follows. In Section 2 we explain the basic mechanisms behind the join protocol of Pastry. This protocol is the most important part of Pastry for correctness. Key aspects of our formal model are introduced in Section 3. To the best of our knowledge, we present the first formal model covering the full Pastry algorithm. A number of important properties are model checked, subsections 3.3–3.4 and subsections 4.2–4.3, and the results are used to refine our model in case the model checker found undesired behavior. In addition to model checking our model, we have also been able to prove an important reduction property of Pastry. Basically, the correctness of the protocol can be reduced to the consistency of leaf sets, as we show in Section 5, Theorem 6. The paper ends with a summary of our results, related work, and future directions of research in Section 6. Further details and all proofs can be found in a technical report [7].

2 The Join Protocol

The most sophisticated part of Pastry is the protocol for a node to join the ring. In its simplest form, a single node joins between two "ready" nodes on the ring. The new node receives its leaf sets from the "ready" nodes, negotiates with both the new leaf sets and then goes to status "ready".

The join protocol is complicated because any node may drop off at any time, in particular while it handles a join request. Moreover, several nodes may join the ring concurrently between two adjacent "ready" nodes. Still, all "ready" nodes must agree on key coverage.

Figure 2 presents a first version of the join protocol in more detail, according to our understanding from [9] and [3]. We will refine this protocol in Sect. 4 according to the description in [5].

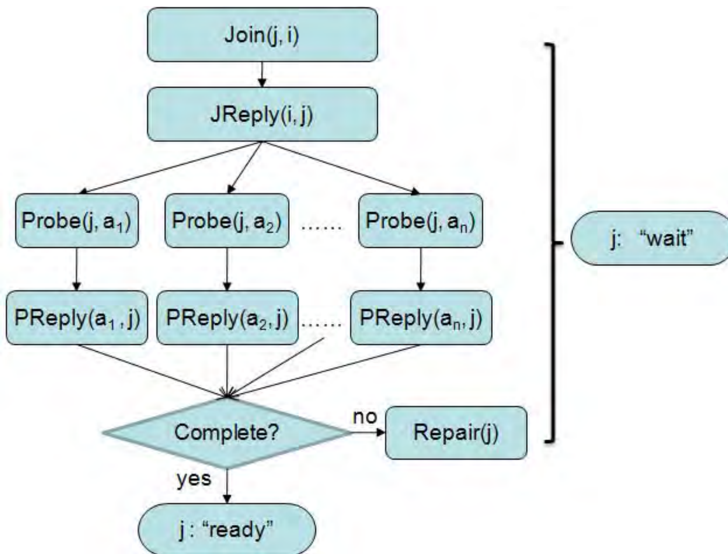


Fig. 2. Overview of the join protocol

Node j announces its interest in joining the ring by performing a *Join* action. At this point, its status is "wait". Its join request will be routed to the closest "ready" node i just like routing a lookup message, treating j as the key. Node i replies to j by performing a join reply, *JReply* action, transmitting its current leaf sets to enable node j to construct its own leaf sets. Then the node j *probes* all nodes in its leaf sets in order to confirm their presence on the ring. A probe reply, action *PReply*, signals j that the respective leaf set node received the probe message from j and updated its local leaf set with j. The reply contains the updated leaf set. Each time the node j receives a probe reply message, it updates the local information based on the received message and checks if there

are outstanding probes. If no outstanding probe exists anymore or a timeout occurs, it checks whether its leaf set is *complete*. If it is, it finishes the join phase and goes to status "ready". Otherwise, any fault case is summarized in Fig. 2 by *Repair*. For example, if a probe eventually fails, the probed node needs to be removed from the leaf set. Then the node j probes the most distant nodes (leftmost and rightmost) in its leaf sets to get more nodes, retrying to complete its leaf set.

3 A First Formal Model of Pastry

We modeled Pastry as a (potentially infinite-state) transition system in TLA$^+$ [6]. Although there are of course alternative logics and respective theorem provers for modelling Pastry, TLA$^+$ fits protocol verification quite nicely, because its concept of actions matches the rule/message based definition of protocols. Our model is available on the Web[1]. We explain those parts of the model that are used later on for model checking and for proving the reduction theorem.

3.1 Static Model

Several parameters define the size of the ring and of the fundamental data structures. In particular, $M \in \mathbb{N}$ defines the space $I = [0, 2^M - 1]$ of node and key identifiers, and $l \in \mathbb{N}$ indicates the size of each leaf set. The following definition introduces different notions of distances between nodes or keys that will be used in the model.

Definition 1 (Distances). *Given* x, $y \in I$*:*

$$
\begin{aligned}
\mathit{Dist}(x,y) &\triangleq \begin{cases} x - y + 2^{M-1} & \mathit{if}\ \ x - y < -2^{M-1} \\ x - y - 2^{M-1} & \mathit{if}\ \ x - y > 2^{M-1} \\ x - y, & \mathit{else} \end{cases} \\
\mathit{AbsDist}(x,y) &\triangleq |\mathit{Dist}(x,y)| \\
\mathit{CwDist}(x,y) &\triangleq \begin{cases} \mathit{AbsDist}(x,y) & \mathit{if}\ \ \mathit{Dist}(x,y) < 0 \\ 2^M - \mathit{AbsDist}(x,y) & \mathit{else} \end{cases}
\end{aligned}
$$

The sign of $\mathit{Dist}(x, y)$ is positive if there are fewer identifiers on the counter-clockwise path from x to y than on the clockwise path; it is negative otherwise. The absolute value $\mathit{AbsDist}(x, y)$ gives the length of the shortest path along the ring from x to y. Finally, the clockwise distance $\mathit{CwDist}(x, y)$ returns the length of the clockwise path from x to y.

The leaf set data structure *ls* of a node is modeled as a record with three components *ls.node*, *ls.left* and *ls.right*. The first component contains the identifier of the node maintaining the leaf set, the other two components are the two leaf sets to either side of the node. The following operations access leaf sets.

[1] `http://www.mpi-inf.mpg.de/~tianlu/software/PastryModelChecking.zip`

Definition 2 (Operations on Leaf Sets)

$$GetLSetContent(ls) \triangleq ls.left \cup ls.right \cup \{ls.node\}$$

$$LeftNeighbor(ls) \triangleq \begin{cases} ls.node & \text{if } ls.left = \{\} \\ n \in ls.left : \forall p \in ls.left : \\ \quad CwDist(p, ls.node) \\ \quad \geq CwDist(n, ls.node) & \text{else} \end{cases}$$

$$RightNeighbor(ls) \triangleq \begin{cases} ls.node & \text{if } ls.right = \{\} \\ n \in ls.right : \forall q \in ls.right : \\ \quad CwDist(ls.node, q) \\ \quad \geq CwDist(ls.node, n) & \text{else} \end{cases}$$

$$LeftCover(ls) \triangleq (ls.node + CwDist(LeftNeighbor(ls), ls.node) \div 2)\%2^M$$

$$RightCover(ls) \triangleq (RightNeighbor(ls) + CwDist(ls.node, RightNeighbor(ls)) \div 2 + 1)\%2^M$$

$$Covers(ls, k) \triangleq CwDist(LeftCover(ls), k) \leq CwDist(LeftCover(ls), RightCover(ls))$$

In these definitions, $\div$ and $\%$ stand for division and modulo on the natural numbers, respectively. Note that they are in fact only applied in the above definitions to natural numbers. We also define the operation $AddToLSet(A, ls)$ that updates the leaf set data structure with a set A of nodes. More precisely, both leaf sets in the resulting data structure ls' contain the l nodes closest to $ls.node$ among those contained in ls and the nodes in A, according to the clockwise or counter-clockwise distance.

3.2 Dynamic Model

Fig. 3 shows the high-level outline of the transition model specification in TLA$^+$. The overall system specification *Spec* is defined as $Init \wedge \Box[Next]_{vars}$, which is the standard form of TLA$^+$ system specifications. It requires that all runs start with a state that satisfies the initial condition *Init*, and that every transition either does not change *vars* (defined as the tuple of all state variables) or corresponds to a system transition as defined by formula *Next*. This form of system specification is sufficient for proving safety properties. If we were interested in proving liveness properties of our model, we should add fairness hypotheses asserting that certain actions eventually occur.

The variable *receivedMsgs* holds the set of messages in transit. Our model assumes that messages are never modified. However, message loss is implicitly covered because no action is ever required to execute. The other variables hold

$$
\begin{array}{lll}
\mathit{vars} & \triangleq & \langle \mathit{receivedMsgs}, \mathit{status}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{rtable} \rangle \\
\mathit{Init} & \triangleq & \land\ \mathit{receivedMsgs} = \{\} \\
& & \land\ \mathit{status} = [i \in I \mapsto \text{IF } i \in A \text{ THEN } \text{"ready" ELSE "dead"}] \\
& & \land\ \mathit{lset} = [i \in I \mapsto \text{IF } i \in A \\
& & \qquad\qquad \text{THEN } \mathit{AddToLSet}(A, [\mathit{node} \mapsto i, \mathit{left} \mapsto \{\}, \mathit{right} \mapsto \{\}]) \\
& & \qquad\qquad \text{ELSE } [\mathit{node} \mapsto i, \mathit{left} \mapsto \{\}, \mathit{right} \mapsto \{\}]] \\
& & \land\ \mathit{probing} = [i \in I \mapsto \{\}] \\
& & \land\ \mathit{failed} = [i \in I \mapsto \{\}] \\
& & \land\ \mathit{rtable} = \ldots \\
\mathit{Next} & \triangleq & \exists i, j \in I : \lor\ \mathit{Deliver}(i, j) \\
& & \qquad\qquad \lor\ \mathit{Join}(i, j) \\
& & \qquad\qquad \lor\ \mathit{JReply}(i, j) \\
& & \qquad\qquad \lor\ \mathit{Probe}(i, j) \\
& & \qquad\qquad \lor\ \mathit{PReply}(i, j) \\
& & \qquad\qquad \lor\ \ldots \\
\mathit{Spec} & \triangleq & \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}
\end{array}
$$

Fig. 3. Overall Structure of the TLA$^+$ Specification of Pastry

arrays that assign to every node $i \in I$ its status, leaf set, the set of nodes it is currently probing, the set of nodes it has determined to have dropped off the ring, and its routing table. The predicate *Init* is defined as a conjunction that initializes all variables; in particular, the model takes a parameter A indicating the set of nodes that are initially "ready".

The next-state relation *Next* is a disjunction of all possible system actions, for all pairs of identifiers $i, j \in I$. Each action is defined as a TLA$^+$ action formula, which is a first-order formula containing unprimed as well as primed occurrences of the state variables, which refer respectively to the values of these variables at the states before and after the action. As an example, Fig. 4 shows the definition of action $\mathit{Deliver}(i, k)$ in TLA$^+$. The action is executable if the node i is "ready", if there exists an unhandled message of type "lookup" addressed to i, and if k, the ID of the requested key, falls within the coverage of node i (cf. Definition 2). Its effect is here simply defined as removing the message m from

$$
\begin{array}{l}
\mathit{Deliver}(i, k) \triangleq \\
\quad \land\ \mathit{status}[i] = \text{"ready"} \\
\quad \land\ \exists m \in \mathit{receivedMsgs} : \land\ m.\mathit{mreq.type} = \text{"lookup"} \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ m.\mathit{destination} = i \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ m.\mathit{mreq.node} = k \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \mathit{Covers}(\mathit{lset}[i], k) \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \mathit{receivedMsgs}' = \mathit{receivedMsgs} \setminus \{m\} \\
\quad \land\ \text{UNCHANGED } \langle \mathit{status}, \mathit{rtable}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{lease} \rangle
\end{array}
$$

Fig. 4. TLA$^+$ specification of action *Deliver*

the network, because we are only interested in the execution of the action, not in the answer message that it generates. The other variables are unchanged (in TLA$^+$, UNCHANGED e is a shorthand for the formula $e' = e$).

3.3 Validation by Model Checking

We used TLC [11], the TLA$^+$ model checker, to validate and debug our model. It is all too easy to introduce errors into a model that prevent the system from ever performing any useful transition, so we want to make sure that nodes can successfully perform *Deliver* actions or execute the join protocol described in Section 2. We used the model checker by asserting their impossibility, using the following formulas.

Property 3 (NeverDeliver and NeverJoin)

$$\begin{array}{lcl} \mathit{NeverDeliver} & \triangleq & \forall i, j \in I : \Box[\neg \mathit{Deliver}(i,j)]_{vars} \\ \mathit{NeverJoin} & \triangleq & \forall j \in I \setminus A : \Box(\mathit{status}[j] \neq \text{“ready”}) \end{array}$$

The first formula asserts that the *Deliver* action can never be executed, for any $i, j \in I$. Similarly, the second formula asserts that the only nodes that may ever become “ready” are those in the set A of nodes initialized to be “ready”. Running the model checker on our model, it quickly produced counter-examples to these claims, which we examined to ensure that the runs look as expected. (Section 4.3 summarizes the results for the model checking runs that we performed.)

We validated the model by checking several similar properties. For example, we defined formulas *ConcurrentJoin* and *CcJoinDeliver* whose violation yielded counter-examples that show how two nodes may join concurrently in close proximity to the same existing node, and how they may subsequently execute *Deliver* actions for keys for which they acquired responsibility.

3.4 Correct Key Delivery

As the main correctness property of Pastry, we want to show that at any time there can be only one node responsible for any key. This is formally expressed as follows.

Property 4 (Correct Key Delivery)

$$\begin{array}{l} \mathit{CorrectDeliver} \triangleq \forall i, k \in I : \\ \quad \text{ENABLED } \mathit{Deliver}(i,k) \\ \quad \Rightarrow \land\ \forall n \in I : \mathit{status}[n] = \text{“ready”} \Rightarrow \mathit{AbsDist}(i,k) \leq \mathit{AbsDist}(n,k) \\ \qquad \land\ \forall j \in I \setminus \{i\} : \neg \text{ENABLED } \mathit{Deliver}(j,k) \end{array}$$

For an action formula A, the state formula ENABLED A is obtained by existential quantification over all primed state variables occurring in A; it is true at a state s

whenever there exists some successor state t such that A is true for the pair (s, t), that is, when A can execute at state s. Thus, *CorrectDeliver* asserts that whenever node i can execute the *Deliver* action for key k then (a) node i has minimal absolute distance from k among all the "ready" nodes and (b) i is the only node that may execute *Deliver* for key k.[2]

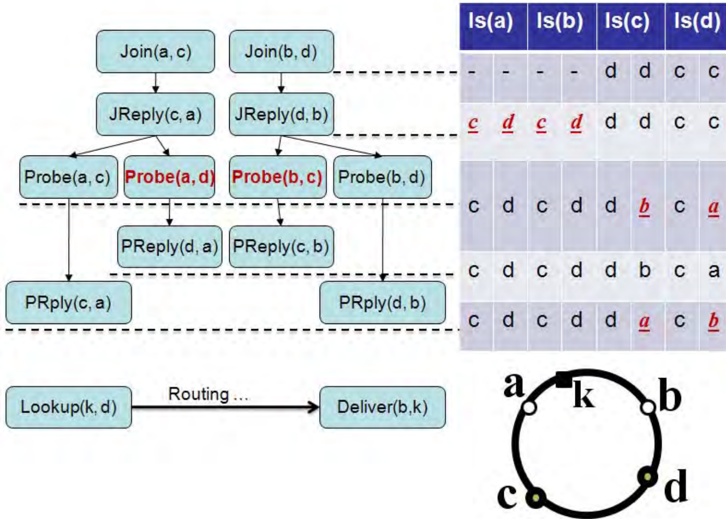


Fig. 5. Counter-example leading to a violation of *CorrectDeliver*

When we attempted to verify Property 4, the model checker produced a counter-example, which we illustrate in Fig. 5. The run starts in a state with just two "ready" nodes c and d that contain each other in their respective leaf sets (the actual size of the leaf sets being 1). Two nodes a and b concurrently join between nodes c and d. According to their location on the ring, a's join request is handled by node c, and b's request by d. Both nodes learn about the presence of c and d, and add them to their leaf sets, then send probe requests to both c and d in order to update the leaf sets. Now, suppose that node d is the first to handle a's probe message, and that node c first handles b's probe. Learning that a new node has joined, which is closer than the previous entry in the respective leaf set, c and d update their leaf sets with b and a, respectively (cf. Fig. 5), and send these updated leaf sets to b and a. Based on the reply from d, node a will not update its leaf set because its closest left-hand neighbor is still found to be c, while it learns no new information about the neighborhood to the right. Similarly, node b maintains its leaf sets containing c and d. Now,

[2] Observe that there can be two nodes with minimal distance from k, to either side of the key. The asymmetry in the definition of *LeftCover* and *RightCover* is designed to break the tie and ensure that only one node is allowed to deliver.

the other probe messages are handled. Consider node c receiving a's probe: it learns of the existence of a new node to its right closer to the one currently in its leaf set (b) and updates its leaf set accordingly, then replies to a. However, node a still does not learn about node b from this reply and maintains its leaf sets containing c and d. Symmetrically, node d updates its leaf set to contain b instead of a, but b does not learn about the presence of a. At the end, the leaf sets of the old nodes c and d are correct, but a and b do not know about each other and have incorrect leaf set entries.

Finally, a lookup message arrives for key k, which lies between a and b, but closer to a. This lookup message may be routed to node b, which incorrectly believes that it covers key k (since k is closer to b than to c, which b believes to be its left-hand neighbor), and delivers the key.

The counter-example shows that our model of the join protocol may lead to inconsistent views of "ready" nodes about their neighborhoods on the ring, and is therefore insufficient. Indeed, after the initial publication of Pastry, Haeberlen et al. [5] presented a refined description of Pastry's join protocol, without providing an explicit motivation. We believe that the above counter example explains the refinement of [5], which we model and analyze in the sequel.

4 Refining the Join Protocol

In contrast to the join protocol described in Section 2, the refined join protocol requires an explicit transfer of coverage from the "ready" neighbor nodes before a joining node can become "ready" and answer lookup requests. In the case of the counter-example shown in Fig. 5, node a would request grants from the nodes c and d, which it believes to be its neighbors. Node d would refuse this request and instead inform node a of the presence of node b, enabling it to rebuild its leaf sets. Similarly, node b would learn about the presence of node a. Finally, the two nodes grant each other a lease for the nodes they cover. We now describe the extended protocol as we have understood and modelled it, and our further verification efforts. In fact, our formal model is also inspired by the implementation in FreePastry [8], where nodes periodically exchange their leaf sets to spread information about nodes dropping off and arriving.

4.1 Lease Granting Protocol

Figure 6 depicts the extension to the join protocol as described in Section 2 (cf. Fig. 2). After node i has built complete leaf sets, it reaches status "ok". It sends messages to its neighbors ln and rn (the two closest nodes in its current leaf sets), requesting a lease for the keys it covers. A node receiving a lease request from a node that it considers to be its neighbor grants the lease, otherwise it returns its own leaf sets to the requesting node. The receiving node will update its own leaf sets accordingly and request a lease from the new neighbor(s). Only when both neighbors grant the lease will node i become "ready".

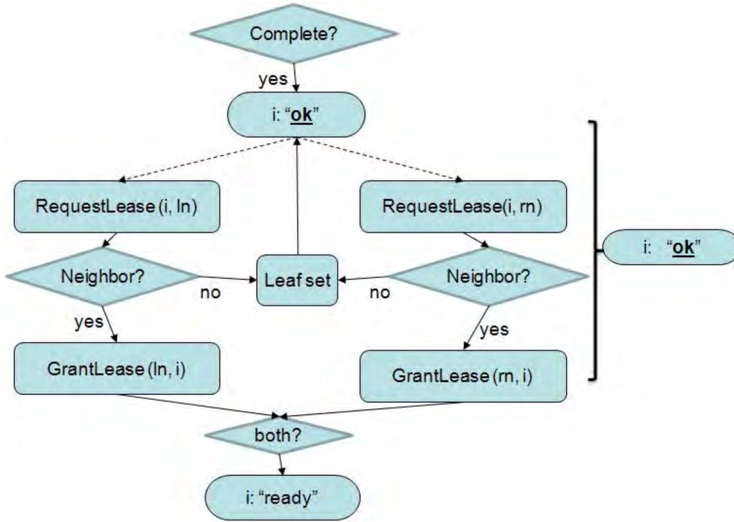


Fig. 6. Extending the Join Protocol by Lease Granting

Moreover, any node that is "ok" or "ready" may non-deterministically re-run the lease granting protocol at any time. In the actual implementation, this happens periodically, as well as when a node suspects its neighbor to have left the ring.

We amended our TLA$^+$ model to reflect this extended join protocol and reran TLC on the extended model. Whereas the results for the properties used to validate the model (cf. Section 3.3) were unchanged, the model checker no longer produced a counter-example to Property 4. However, we were unable to complete the model checking run and killed TLC after it had been running for more than a month.

4.2 Symmetry of Leaf Sets

Based on the counter-example shown in Section 3.4, one may be tempted to assert that leaf set membership of nodes should be symmetrical in the sense that for any two "ready" nodes i, j it holds that i appears in the leaf sets of j if and only if j appears in the leaf sets of i.

Property 5 (Symmetry of leaf set membership)

$$\begin{aligned}&\mathit{Symmetry} \triangleq\\&\quad \forall i, j \in I : \mathit{status}[i] = \text{“ready”} \land \mathit{status}[j] = \text{“ready”}\\&\qquad\qquad \Rightarrow (i \in \mathit{GetLSetContent}(\mathit{lset}[j]) \Leftrightarrow j \in \mathit{GetLSetContent}(\mathit{lset}[i]))\end{aligned}$$

However, the above property is violated during the execution of the join protocol and TLC yields the following counter-example: a node k joins between i and j.

Table 1. TLC result with four nodes, leaf set length $l = 1$

Examples	Time	Depth	# states	Counter Example
NeverDeliver	1”	5	101	yes
NeverJoin	1”	9	19	yes
ConcurrentJoin	3’53”	21	212719	yes
CcJoinLookup	23’16”	23	1141123	yes
Symmetry	17”	17	19828	yes
Neighbor	5’35”	16	278904	yes
NeighborProp	> 1 month	31	1331364126	no
CorrectDeliver	> 1 month	21	1952882411	no

It finishes its communication with i getting its coverage set from i, but its communication with j is not yet finished. Hence, i and j are “ready” whereas k is not. Furthermore, i may have removed j from its leaf set, so the symmetry is broken.

4.3 Validation

Table 1 summarizes the model checking experiments we have described so far, over the extended model. TLC was run with two worker threads (on two CPUs) on a 32 Bit Linux machine with Xeon(R) X5460 CPUs running at 3.16GHz with about 4 GB of memory per CPU. For each run, we report the running time, the number of states generated until TLC found a counter-example (or, in the case of Property 4, until we killed the process), and the largest depth of these states. Since the verification of Property 4 did not produce a counter-example, we ran the model checker in breadth-first search mode. We can therefore assert that if the model contains a counter-example to this property, it must be of depth at least 21.

All properties except *Neighbor* and *NeighborProp* were introduced in previous sections. The property *Neighbor* is inspired by the counter-example described in Section 3.4. It is actually the *NeighborClosest* property relaxed to “ok” and “ready” nodes. It asserts that whenever i, j are nodes that are “ok” or “ready”, then the left and right neighbors of node i according to its leaf set contents must be at least as close to i than is node j. This property does not hold, as the counter-example of Section 3.4 shows, but it does if node i is in fact “ready”, which corresponds the *NeighborClosest* property. The *NeighborProp* property is the conjunction $\mathit{HalfNeighbor} \land \mathit{NeighborClosest}$, see the next section.

5 Theorem Proving

Having gained confidence in our model, we now turn to formally proving the main correctness Property 4, using the interactive TLA$^+$ proof system (TLAPS) [4].

Our full proofs are available on the Web[3]. The intuition gained from the counter-example of Section 3.4 tells us that the key to establishing Property 4 is that the leaf sets of all nodes participating in the protocol contain the expected elements. We start by defining a number of auxiliary formulas.

$$
\begin{array}{l}
\mathit{Ready} \triangleq \{i \in I : \mathit{status}[i] = \text{“ready”}\} \\
\mathit{ReadyOK} \triangleq \{i \in I : \mathit{status}[i] \in \{\text{“ready”}, \text{“ok”}\}\} \\
\mathit{HalfNeighbor} \triangleq \\
\quad \lor\ \forall i \in \mathit{ReadyOK} : \mathit{RightNeighbor}(\mathit{lset}[i]) \neq i \land \mathit{LeftNeighbor}(\mathit{lset}[i]) \neq i \\
\quad \lor\ \land\ \mathit{Cardinality}(\mathit{ReadyOK}) \leq 1 \\
\quad\quad \land\ \forall i \in \mathit{ReadyOK} : \mathit{LeftNeighbor}(\mathit{lset}[i]) = i \land \mathit{RightNeighbor}(\mathit{lset}[i]) = i \\
\mathit{NeighborClosest} \triangleq \forall i, j \in \mathit{Ready} : \\
\quad i \neq j \Rightarrow \land\ \mathit{CwDist}(\mathit{LeftNeighbor}(\mathit{lset}[i]), i) \leq \mathit{CwDist}(j, i) \\
\qquad\qquad\quad \land\ \mathit{CwDist}(i, \mathit{RightNeighbor}(\mathit{lset}[i])) \leq \mathit{CwDist}(i, j)
\end{array}
$$

Sets *Ready* and *ReadyOK* contain the nodes that are “ready”, resp. “ready” or “ok”. Formula *HalfNeighbor* asserts that whenever there is more than one “ready” or “ok” node i, then the left and right neighbors of every such node i are different from i. In particular, it follows by Definition 2 that no leaf set of i can be empty. The formula *NeighborClosest* states that the left and right neighbors of any “ready” node i lie closer to i than any “ready” node j different from i.

We used TLC to verify $\mathit{NeighborProp} \triangleq \mathit{HalfNeighbor} \land \mathit{NeighborClosest}$. Running TLC for more than a month did not yield a counter-example. Using TLAPS, we have mechanically proved that *NeighborProp* implies Property 4, as asserted by the following theorem.

Theorem 6 (Reduction). $\mathit{HalfNeighbor} \land \mathit{NeighborClosest} \Rightarrow \mathit{CorrectDeliver}$.

We sketch our mechanically verified TLAPS proof of Theorem 6 by two lemmas. The first lemma shows that, assuming the hypotheses of Theorem 6, then for any two “ready” nodes i, n, with $i \neq n$ and key k, if node i covers k then i must be at least as close to k as is n.

Lemma 7 (Coverage Lemma)

$$
\begin{array}{l}
\mathit{HalfNeighbor} \land \mathit{NeighborClosest} \\
\Rightarrow\ \forall i, n \in \mathit{Ready} : \forall k \in I : i \neq n \land \mathit{Covers}(\mathit{lset}[i], k) \\
\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathit{AbsDist}(i, k) \leq \mathit{AbsDist}(n, k)
\end{array}
$$

The second lemma shows, under the same hypotheses, that if i covers k then n cannot cover k. Taking together Lemma 7 and Lemma 8, Theorem 6 follows easily by the definitions of the property *CorrectDeliver* (Property 4) and the action *Deliver* (see Fig. 4).

[3] `http://www.mpi-inf.mpg.de/~tianlu/software/PastryTheoremProving.zip`

Lemma 8 (Disjoint Covers)

$$\begin{aligned}&\mathit{HalfNeighbor} \land \mathit{NeighborClosest}\\ &\Rightarrow\ \forall i, n \in \mathit{Ready} : \forall k \in I : i \neq n \land \mathit{Covers}(\mathit{lset}[i], k)\\ &\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \neg \mathit{Covers}(\mathit{lset}[n], k)\end{aligned}$$

In order to complete the proof that our model of Pastry satisfies Property 4, it is enough by Theorem 6 to show that every reachable state satisfies properties *HalfNeighbor* and *NeighborClosest*. We have embarked on an invariant proof and have defined a predicate that strengthens these properties. We are currently in the process of showing that it is indeed preserved by all actions of our model.

6 Conclusion and Future Work

In this paper we have presented a formal model of the Pastry routing protocol, a fundamental building block of P2P overlay networks. To the best of our knowledge, this is the first formal model of Pastry, although the application of formal modeling and verification techniques to P2P protocols is not entirely new. For example, Velipalasar et al. [10] report on experiments of applying the Spin model checker to a model of a communication protocol used in a P2P multimedia system. More closely related to our topic, Borgström et al. [2] present initial work towards the verification of a distributed hash table in a P2P overlay network in a process calculus setting, but only considered fixed configurations with perfect routing information. As we have seen, the main challenge in verifying Pastry lies in the correct handling of nodes joining the system on the fly. Bakhshi and Gurov [1] model the Pure Join protocol of Chord in the π-calculus and show that the routing information along the ring eventually stabilizes in the presence of potentially concurrent joins. Numerous technical differences aside, they do not consider possible interferences between the join and lookup sub-protocols, as we do in our model.

Pastry is a reasonably complex algorithm that mixes complex data structures, dynamic network protocols, and timed behavior for periodic node updates. We decided to abstract from timing aspects, which are mainly important for performance, but otherwise model the algorithm as faithfully as possible. Our main difficulties were to fill in details that are not obvious from the published descriptions of the algorithm, and to formally state the correctness properties expected from Pastry. In this respect, the model checker helped us understand the need for the extension of the join protocol by lease granting, as described in [5]. It was also invaluable to improve our understanding of the protocol because it allowed us to state "what-if" questions and refute conjectures such as the symmetry of leaf set membership (Property 5). The building of the first overall model of Pastry in TLA$^+$ took us about 3 months. Almost two third of it was devoted to the formal development of the underlying data structures, such as the address ring, leaf sets or routing tables.

After having built up confidence in the correctness of our model, we started full formal verification using theorem proving. In particular, we have reduced the

correctness Property 4 to a predicate about leaf sets that the algorithm should maintain, and have defined a candidate for an inductive invariant. Future work will include full verification of the correctness properties. Afterwards, we will extend the model by considering liveness properties, which obviously require assumptions about the ring being sufficiently stable. We also intend to study which parts of the proof are amenable to automated theorem proving techniques, as the effort currently required by interactive proofs is too high to scale to more complete P2P protocols.

Acknowledgements. We would like to thank the reviewers for their valuable comments.

References

1. Bakhshi, R., Gurov, D.: Verification of peer-to-peer algorithms: A case study. Electr. Notes Theor. Comput. Sci. 181, 35–47 (2007)
2. Borgström, J., Nestmann, U., Onana, L., Gurov, D.: Verifying a structured peer-to-peer overlay network: The static case. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 250–265. Springer, Heidelberg (2005)
3. Castro, M., Costa, M., Rowstron, A.I.T.: Performance and dependability of structured peer-to-peer overlays. In: International Conference on Dependable Systems and Networks (DSN 2004), Florence, Italy, pp. 9–18. IEEE Computer Society Press, Los Alamitos (2004)
4. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the TLA$^+$ proof system. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 142–148. Springer, Heidelberg (2010)
5. Haeberlen, A., Hoye, J., Mislove, A., Druschel, P.: Consistent key mapping in structured overlays. Technical Report TR05-456, Rice University, Department of Computer Science (August 2005)
6. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Reading (2002)
7. Lu, T., Merz, S., Weidenbach, C.: Towards verification of the pastry protocol using TLA+. Technical Report MPI-I-2011-RG1-002, Max-Planck-Institute für Informatik (April 2011)
8. Rice University and Max-Planck Institute for Software Systems. Pastry: A substrate for peer-to-peer applications, `http://www.freepastry.org/`
9. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pp. 329–350 (November 2001)
10. Velipasalar, S., Lin, C.H., Schlessman, J., Wolf, W.: Design and verification of communication protocols for peer-to-peer multimedia systems. In: IEEE Intl. Conf. Multimedia and Expo (ICME 2006), Toronto, Canada, pp. 1421–1424. IEEE, Los Alamitos (2006)
11. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999)

Dynamic Soundness in Resource-Constrained Workflow Nets*

María Martos-Salgado and Fernando Rosa-Velardo

Sistemas Informáticos y Computación,
Universidad Complutense de Madrid
mrmartos@estumail.ucm.es, fernandorosa@sip.ucm.es

Abstract. Workflow Petri nets (wf-nets) are an important formalism for the modeling of business processes. For them we are typically interested in the soundness problem, that intuitively consists in deciding whether several concurrent executions can always terminate properly. Resource-Constrained Workflow Nets (rcfw-nets) are wf-nets enriched with static places, that model global resources. In this paper we prove the undecidability of soundness for rcwf-nets when there may be several static places and in which instances are allowed to terminate having created or consumed resources. In order to have a clearer presentation of the proof, we define an asynchronous version of a class of Petri nets with dynamic name creation. Then, we prove that reachability is undecidable for them, and reduce it to dynamic soundness in rcwf-nets. Finally, we prove that if we restrict our class of rcwf-nets, assuming in particular that a single instance is sound when it is given infinitely many global resources, then dynamic soundness is decidable by reducing it to the home space problem in P/T nets for a linear set of markings.

1 Introduction

Workflow Nets have been identified and widely used as a solid model of business processes [1], with a rich theory for their analysis and verification, sustained in more than 40 years of development of the theory of Petri Nets. Workflow nets (wf-nets) model business processes, that start in a given initial state and must eventually finish (under a strong fairness assumption) in a final state in which its task has been completed. One of the central problems in this area is that of soundness, that of checking whether a wf-net can always reach its final state properly [2].

Here we follow the works in [3,4]. In them, the authors study extensions of wf-nets in which processes must share some global resources. Resource-constrained workflow nets (rcwf-nets) are wf-nets in which some places are dynamic and some are static. Following a terminology from OOP, a rcwf-net can be seen as the definition of a class, with its local and static attributes, represented by

* Work supported by the MEC Spanish project DESAFIOS10 TIN2009-14599-C03-01, and Comunidad de Madrid program PROMETIDOS S2009/TIC-1465.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 259–273, 2011.
© IFIP International Federation for Information Processing 2011

dynamic and static places, respectively. Then, the rcwf-net can be instantiated several times, but every instance must share the tokens in static places.

Even if a singe instance of a rcwf-net is sound, several instances could deadlock because of static places. In [3] the authors define dynamic soundness, which essentially amounts to the condition stating that any number of instances running simultaneously can always reach the final state, that in which all the tasks have been completed.

In both works, the authors consider rcwf-nets that do not create or consume static resources, that is, rcwf-nets that always return a global resource after using it. In particular, the behavior of a single instance of a rcwf-net is such that the number of tokens in the static places in the initial and final markings coincide. Under this assumption, the number of tokens in the static places is bounded by the number of tokens in the initial marking. The authors prove in [3] that dynamic soundness is decidable whenever there is only a single static place, that is, whenever there is a single type of global resources. Recently, [4] further studies the problem of dynamic soundness, extending the previous result to rcwf-nets with any number of static places, but considering a fixed number of initial resources (unlike in [3], in which the existence of a minimal number of resources for which the rcwf-net is sound is part of the problem). Under these assumptions, it is enough for the authors to study the absence of deadlocks.

In this paper we continue the works in [3,4] by studying the problem of dynamic soundness for rcwf-nets with any number of static places, and without restricting their behavior so that instances can terminate their task having created new global resources or having consumed some.

We prove that dynamic soundness under these hypotheses is undecidable. It is to the best of our knowledge the first undecidability result regarding soundness in wf-nets (without special arcs like inhibitor or reset arcs [5]). The proof considers a class of colored Petri nets with dynamic fresh name creation that we have defined in previous works [6], called ν-PN. It is based on a non-trivial reduction of reachability in ν-PN, which is undecidable [7,8], to dynamic soundness in rcwf-nets. Moreover, we believe that the simulation of ν-PN by means of rcwf-nets, and the reduction of the reachability problem, are interesting by themselves, and could be used to obtain other decidability/undecidability results.

Finally, we consider the same problem for a subclass of rcwf-nets, arguably a sensible subclass of rcwf-nets. We will consider rcwf-nets which are sound for a single instance (that is, such that a single instance can always finish properly) whenever it is provided with infinitely many resources, and such that every transition may contribute to the completion of the task. We will prove that dynamic soundness in this case can be reduced to a home space problem in ordinary P/T nets, which is decidable [9,10].

The rest of the paper is organized as follows. Section 2 presents the basic concepts we will need, like P/T nets and ν-PN. Section 3 defines asynchronous ν-PN and proves undecidability of reachability for them. In Sect. 4 we present our rcwf-nets, in terms of asynchronous ν-PN. Section 5 proves that dynamic soundness is undecidable for rcwf-nets. In Sect. 6 we consider a restricted version

of the problem, and prove decidability in this case. Finally, Sect. 7 presents our conclusions and directions for further study.

2 Preliminaries

A (finite) multiset m over a set A is a mapping $m : A \to \mathbb{N}$. We denote by $A^{\oplus}$ the set of finite multisets over A. For two multisets m_1 and m_2 over A we define $m_1+m_2 \in A^{\oplus}$ by $(m_1+m_2)(a) = m_1(a)+m_2(a)$ and $m_1 \subseteq m_2$ if $m_1(a) \leq m_2(a)$ for every $a \in A$. When $m_1 \subseteq m_2$ we can define $m_2-m_1 \in A^{\oplus}$ by $(m_2-m_1)(a) = m_2(a) - m_1(a)$. We denote by $\emptyset$ the empty multiset, that is, $\emptyset(a) = 0$ for every $a \in A$. Finally, for $\lambda \in \mathbb{N}$ and $m \in A^{\oplus}$ we define $\lambda * m = m + \overset{\lambda}{\ldots} + m \in A^{\oplus}$.

Petri Nets. A Place/Transition Net [11] (P/T net for short) is a tuple $N = (P, T, F)$, where P is a finite set of places, T is a finite set of transitions (disjoint with P) and $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the flow function. P/T nets are depicted as usual: places are drawn by circles, transitions are drawn by boxes and F is represented by arrows labeled by a natural, that is, we draw an arrow from x to y labeled by $F(x, y)$ (or without any label if $F(x, y) = 1$). We do not show the arrow whenever $F(x, y) = 0$.

A marking of N is an element of $P^{\oplus}$. For a transition t we define ${}^{\bullet}t \in P^{\oplus}$ as ${}^{\bullet}t(p) = F(p, t)$. Analogously, we take $t^{\bullet}(p) = F(t, p)$. A marking m enables a transition $t \in T$ if ${}^{\bullet}t \subseteq m$. In that case t can be fired, reaching the marking $m' = (m - {}^{\bullet}t) + t^{\bullet}$, in which case we write $m \xrightarrow{t} m'$.

Given a P/T net $N = (P, T, F)$ with initial marking m_0, we say that a place $p \in P$ is *bounded* if there exists $b \in \mathbb{N}$ such that for each reachable marking m, $m(p) \leq b$. It is bounded if all its places are bounded. Boundedness is decidable for P/T nets. Moreover, the problem of deciding whether a place is bounded, is also decidable [10]. Given a P/T net N, a marking m_0 and a set $\mathcal{H}$ of markings of N, we say that $\mathcal{H}$ is a home space if for every reachable marking m, there is a marking $m' \in \mathcal{H}$ reachable from m.

The problem of deciding whether a linear set of markings is a home space is decidable too [10,9]. A linear set of markings of a P/T net N is a set of markings that can be obtained as linear combinations of markings of N. More precisely, a marking m_0 and a finite set of markings $\{m_1, ..., m_n\}$ define the linear set of markings $\mathcal{L} = \{m_0 + \sum_{i=1}^{n} \lambda_i * m_i \mid \lambda_i \in \mathbb{N}\}$.

Workflow Petri Nets. We will use the definition in [4]. A workflow Petri net (shortly a wf-net) is a P/T net $N = (P, T, F)$ such that:

- there are $in, out \in P$ with ${}^{\bullet}in = \emptyset$ and $out^{\bullet} = \emptyset$,
- for each $p \in P \setminus \{in, out\}$, ${}^{\bullet}p \neq \emptyset$ and $p^{\bullet} \neq \emptyset$.

The second condition intuitively states that all the places contribute to the completion of the task. In this paper, we can always force that condition to be satisfied, so that we will from now on ignore it.

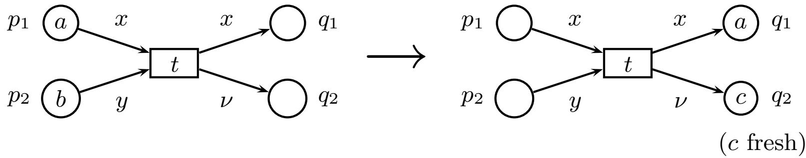


Fig. 1. Two simple ν-PN

Petri Nets with dynamic name creation. Now we briefly define ν-PN [6]. We consider an infinite set Id of names, a set Var of variables and a subset of special variables $\Upsilon \subset Var$ for name creation. A ν-PN is a tuple $N = (P, T, F)$, where P and T are finite disjoint sets, and $F : (P \times T) \cup (T \times P) \to Var^{\oplus}$.

A *marking* is a mapping $m : P \to Id^{\oplus}$. We denote by $\emptyset$ the empty marking, that which satisfies $\emptyset(p) = \emptyset$ for all $p \in P$. We write $Id(m)$ to denote the set of names that appear in m. We denote by $Var(t)$ the set of variables in arcs adjacent to t. Analogously, we will write $Var(p)$ for the set of variables adjacent to a place p. A *mode* is a mapping $\sigma : Var(t) \to Id$. A transition t can be fired with mode σ for a marking m if for all $p \in P$, $\sigma(F(p,t)) \subseteq m(p)$ and for every $\nu \in \Upsilon$, $\sigma(\nu) \notin m(p)$ for all p. In that case we have $m \xrightarrow{t} m'$, where $m'(p) = (m(p) - \sigma(F(p,t))) + \sigma(F(t,p))$ for all $p \in P$.

In order to keep usual notations in P/T nets, we will assume that there is a "distinguished" color $\bullet \in Id$. By "name" we mean any color different from $\bullet$. Moreover, we will use a distinguished variable ϵ that can only be instantiated to $\bullet$, which will be omitted in our figures. Thus, we can manage ordinary black tokens with the same notations as in P/T nets.

The reachability problem is undecidable for ν-PN [7,8], that is, given m_0 and m_f, markings of a ν-PN N, the problem of deciding whether $m_0 \to^* m_f$ is undecidable. By following a standard reduction that removes m_f, we can prove that the problem of deciding whether the empty marking is reachable is also undecidable. Moreover, without loss of generality we can assume that m_0 contains a single token.

3 Asynchronous ν-PN

Intuitively, one can see each name in a ν-PN as a process. Then, we can see a firing of a transition in which different names are involved as a synchronization between the corresponding processes.

Next, we prove that we can assume that actually each process can only synchronize with a global shared memory, so that a synchronization between two processes must be achieved via this shared memory. Technically, we will use ordinary black tokens to represent this global memory, and names to represent processes.

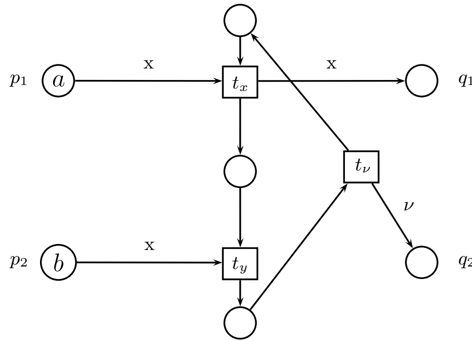


Fig. 2. Simulation of the ν-PN in the left of Fig. 1 by an asynchronous ν-PN

Definition 1. *An asynchronous ν-PN is a ν-PN (P, T, F) such that:*

- *for each $t \in T$, either $Var(t) \subseteq \{\nu, \epsilon\}$ or $Var(t) \subseteq \{x, \epsilon\}$,*
- *for each $p \in P$, either $Var(p) = \{x\}$ or $Var(p) = \{\epsilon\}$.*

We call static places those $p \in P$ with $Var(p) = \{\epsilon\}$, and dynamic places those $p \in P$ with $Var(p) = \{x\}$. We will write $P = P_S \cup P_D$, with P_S the set of static places and P_D the set of dynamic places. Thus, we will disallow a situation in which $x, y \in Var(t)$. Let us now see that asynchronous ν-PN can simulate ν-PN so that reachability is preserved.

Proposition 1. *Let N be a ν-PN, and m_0 a marking of N. There is an asynchronous ν-PN N' and a marking m'_0 of N' such that $m_0 \rightarrow^* \emptyset$ iff $m'_0 \rightarrow^* \emptyset$.*

Proof (sketch). We simulate each transition t by the sequential firing of several transitions satisfying the requirement above. Assume $Var(t) = \{x_1, ..., x_n\}$. We add transitions $t_1, ..., t_n$ so that t_i is used to remove and add the tokens to which x_i is instantiated (using each of them a single variable x for that purpose). In order to guarantee that they are fired sequentially, we add auxiliary places, controlled by arcs labeled by ϵ. One of these auxiliary places also guarantees that the simulation of a transition is done atomically, that is, whenever such a simulation is started, no other simulation can start until the former has finished. Notice that this simulation can introduce deadlocks (for instance, when we fire t_1 but we cannot continue with t_2 due to abscess of tokens), but it does preserve reachability. Fig. 2 illustrates the previous construction when $Var(t) = \{x, y\}$.

Corollary 1. *Reachability of $\emptyset$ is undecidable for asynchronous ν-PN.*

4 Resource-Constrained Workflow Nets

We propose here a presentation of rcwf-nets slightly different from the presentation of [3], though equivalent. We directly define rcwf-nets using (asynchronous) ν-PN, in order to shorten the gap between rcwf-nets and ν-PN.

Given a ν-PN $N = (P, T, F)$ and $x \in Var$ we define the P/T net $N_x = (P, T, F_x)$, where $F_x(n, m) = F(n, m)(x)$. Moreover, for $Q \subseteq P$, by $F|_Q$ we mean F restricted to $(Q \times T) \cup (T \times Q)$. We are now ready to define rcwf-nets.

Definition 2. *A* resource constrained wf-net *(or rcwf-net) is an asynchronous ν-PN $N = (P, T, F)$ such that:*

- *for all $t \in T$, $\nu \notin Var(t)$,*
- *$N_p = (P_D, T, F|_{P_D})_x$ is a wf-net.*

N_p is the P/T net obtained by removing static places, which we call *production net* of N. Then, a rcwf-net is an asynchronous ν-PN that does not create new tokens (because the variable ν does not label any arc) and such that its production net is a wf-net. In particular, it contains two special places *in* and *out* given by the definition of wf-nets. When there is no confusion we will simple refer to these places as *in* and *out*, respectively.

Definition 3. *Let $N = (P, T, F)$ be a rcwf-net and $m_0 \in P_S^{\oplus}$. For any $k \geq 0$, we define m_0^k, as the marking of N given by:*

- *$m_0^k(s)$ contains $m_0(s)$ black tokens, for each $s \in P_S$,*
- *$m_0^k(in)$ contains k pairwise different names,*
- *$m_0^k(d)$ is empty for every $d \in P_D \setminus \{in\}$.*

Moreover, for m_0^k we define the set of final markings $\mathcal{M}_{out}^k$ that contain the same k names in out, and empty in the rest of the dynamic places.

Notice that in the final markings we are not fixing the amount of tokens in static places, unlike in [3,4].

Definition 4. *Let $N = (P, T, F)$ be a rcwf-net and $m_0 \in P_S^{\oplus}$. We say N is dynamically sound for m_0 if for each $k \geq 0$ and for each m reachable from m_0^k, we can reach some marking in $\mathcal{M}_{out}^k$.*

5 Undecidability of Dynamic Soundness

In this section we prove undecidability of dynamic soundness for rcwf-nets by reducing reachability for asynchronous ν-PN, which is undecidable, to it.

For this purpose, given an asynchronous ν-PN N, an initial marking m_0 of N (which we can assume to contain a single token in a given place i), we are going to construct a rcwf-net N' which is dynamic sound if and only if the empty marking is not reachable from m_0. Intuitively, the runs of N' will be divided into four steps: In the first step, the net gets ready for the simulation; in the second step, the initial marking m_0 of N is set; the third step simulates N; and finally, the last step is intuitively used to force that $\emptyset$ is not reachable if and only if N' is dynamically sound.

Let us explain with detail the four steps. In order to control in which step we are in, we consider four static places *step*1, *step*2, *step*3 and *step*4, that will be marked in mutual exclusion. Initially, *step*1 is marked.

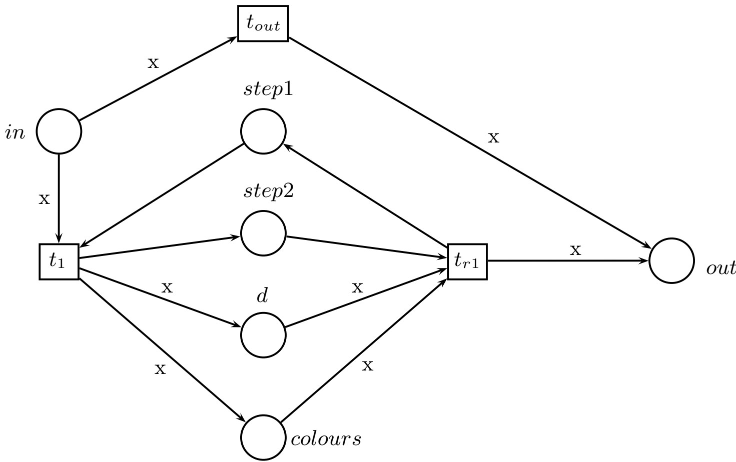


Fig. 3. Step 1

5.1 Step 1: Getting Ready

First of all, as we want to build a rcwf-net, we add two special places *in* and *out*. We add a transition t_{out} which can move a token from *in* to *out*. This transition does not have any other precondition, so that it can be fired in any of the steps.

We will also consider two dynamic places, d and *colours*. The purpose of d will be explained in the last step. The place *colours* will store all the colours that we will use in the simulation of N, so that each transition in the construction which takes a token from *in*, will add it to *colours*. We store all the colours in order to be able to add them to *out* even if N consume all the tokens of some color. We need the place *colours* because N could erase some names, but we cannot do this in N' without being dynamically unsound.

In this first step, a transition t_1 is fired, removing a token from *in* and adding it to the two dynamic places d and *colours*. The purpose of placing a token in d will be explained later, in the last step. It also moves the toke from *step*1 to *step*2, thus moving on to the next step.

Finally, we need the firing of t_1 to be "reversible" (for the case in which we have a single name in *in*). Therefore, we add a new transition t_{r1} which moves a token from *step*2 to *step*1, removes the tokens in *colours* and d, and adds a token of the same color to *out* (not to *in*, since it cannot have incoming arcs). Fig. 3 illustrates the first step.

5.2 Step 2: Setting the Initial Marking

In order to simulate the behavior of N, we consider in N' the set of places of N. In this step we set the initial marking, which consists only of a name in the place of N that we call i. Therefore, we take a token from *in* and put it both in i and in *colours*. Moreover, we move the token from *step*2 to *step*3.

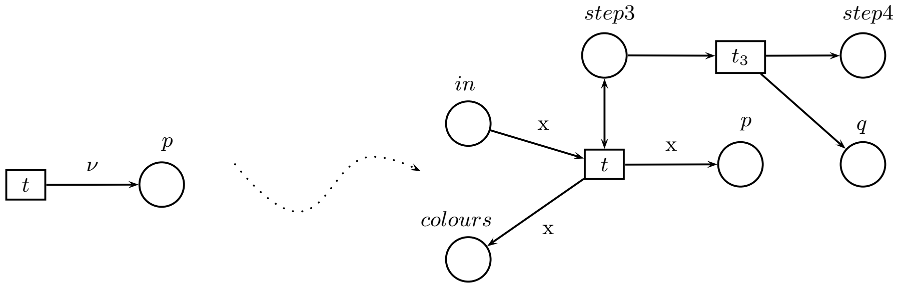


Fig. 4. Step 3

5.3 Step 3: Simulating *N*

In this step we simulate the behavior of N. Since N is an asynchronous ν-PN, it only uses variables x, ν and ϵ. Since N' is a rcwf-net, we have to simulate the creation of new names without using ν. We do it analogously as in the previous steps, by taking from *in* a name whenever one must be created, and placing it both in *colours* and whatever places pointed by arcs labeled by ν. Since all the names contained in the place *in* are different, this is a correct simulation of the creation of a fresh name.

It may be the case that at some point there are no more tokens in the place *in*, so that no more name creations can be simulated. Therefore, a run of N' with k different names in the place *in* simulates a run of N in which at most k names are used (actually, $k-1$ because of the name that is put in d). Notice that the dynamic soundness has to do with the behavior of a rcwf-net from any initial marking, so that all the behaviors of N will be considered.

In this step we add *step*3 both as precondition and postcondition of any transition in N, so that transitions in N can only be fired in this step. At any point, we can fire a transition t_3 that moves the token from *step*3 to *step*4, thus finishing the simulation of N. Moreover, it also puts a black token in a new static place q, whose purpose we will explain later. Figure 4 shows the simulation of a transition with a ν.

5.4 Step 4: Reducing Reachability to Dynamic Soundness

When the fourth step starts, there is a name in d, a black token in *step*4 (which will stay there until the end of the execution of N') and in q, the set of names that have been used along the execution of the rcwf-net is stored in *colours* and the places of N are marked with a marking which is reachable in N.

We add a transition t_f, which can move all the tokens from *colours* to *out*, and with *step*4 both as precondition and postcondition, so that it cannot be fired until this step starts.

We want to force N' to be dynamically unsound whenever $\emptyset$ is reachable. Since we can move names directly from *in* to *out*, we need to build a marking from which it is not possible to remove names from places different from *out*.

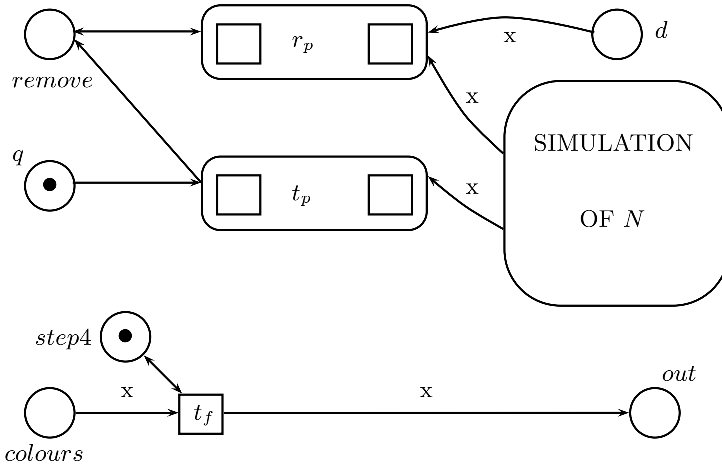


Fig. 5. Step4

We add to N' a transition t_p for each place p of N. When q is marked, there is a choice between all the transitions t_p, each of which removes a token from p, and puts a black token in a static place *remove*. Intuitively, we are only able to fire some t_p if the current marking of N is not $\emptyset$. Otherwise, if t_3 was fired exactly from $\emptyset$, then no transition t_p can be fired.

If we are able to fire some t_p then we have a token in *remove*. In that case, we can fire transitions r_p for each dynamic place p (different from *colours*, *in* and *out*), that removes a token from p, and puts the token back to *remove*. Therefore, if *remove* is marked, we can empty every dynamic place different from *colours*, *in* and *out*. In particular, the firing of r_d is the only way to remove the token in d. Figure 5 sketches how the fourth step is performed.

5.5 Undecidability

Now we are ready to prove that the previous construction reduces reachability for asynchronous ν-PN to dynamic soundness for rcwf-nets.

Proposition 2. *Given a ν-PN N with initial marking m_0, the rcwf-net N' built is dynamically sound if and only if $\emptyset$ is not reachable from m_0 in N.*

Proof. First, let us suppose that $\emptyset$ is reachable from m_0 in N. Let n be the number of different names created in some run that reaches $\emptyset$. If we consider the net N' with $n+1$ or more instances (that is, with at least $n+1$ different names in the place *in*), then we can reach a marking m' of N' in which the places of N are unmarked, the names that have been used in the computation are stored in *colours*, d is marked by a color and *step*4 and q are marked with black tokens. From this marking, we cannot fire any of the t_p transitions, and therefore, we cannot remove the token from q. Therefore, *remove* cannot be marked, which

is the only way in which the name in d can be removed. Summing up, from the initial marking with $n+1$ different names in *in* we have reached a marking from which we cannot reach a final marking of N' (that in which the only marked dynamic place is *out*), so that N' is not dynamically sound.

Conversely, let us suppose that $\emptyset$ is not reachable. We have to prove that for each $k \geq 0$ and for each m reachable from m_0^k, we can reach some marking in $\mathcal{M}_{out}^k$. Let us consider several cases, depending on which step the considered marking is in.

- If *step*1 is marked in m then all the names are either in the place *in* or in *out*. Therefore, we can fire t_{out} repeatedly, transferring all the tokens in *in* to *out*, and we are done.
- If *step*2 is marked in m we can fire t_{r1}, reaching a marking in which *step*1 is marked, so we can apply the previous case.
- If *step*3 is marked in m we can fire t_3, reaching a marking in which *step*4 is marked. We discuss this case next.
- If *step*4 is marked in m we can fire t_f repeatedly, putting all the names that have been used by the construction in *out*, thus emptying *colours*. Moreover, we can fire t_{out} repeatedly, moving all the tokens which remain in *in* to *out*. Therefore, all the tokens that initially were set in *in*, are set in *out*, so we only have to prove that we can empty the other dynamic places. If *step*4 is marked then there must be a token in q or *remove*. If the token is in q, since $\emptyset$ is not reachable, there is some name in some place p of N. Therefore, we can fire the transition t_p from m, reaching a marking in which *remove* is marked. Finally, if *remove* is marked in m, we can remove all the tokens from the dynamic places different from *colours*, *in* and *out*, reaching the desired marking.

The previous result proves that reachability of the empty marking in asynchronous ν-PN, which is undecidable, can be reduced to dynamic soundness for rcwf-nets. Therefore, we finally obtain the following result:

Corollary 2. *Dynamic soundness is undecidable for rcwf-nets.*

6 Decidability of Dynamic Soundness for a Subclass of rcwf-nets

We have proved that dynamic soundness is undecidable in general. However, if we consider more restrictive requirements for our rcwf-nets, dynamic soundness turns decidable. In the literature, several notions of rcwf-nets and soundness have been studied, most of them being more restrictive than our general definition. In particular, in [3] the authors consider wf-nets which satisfy the following condition, which we have not required: for each node n, there are paths from *in* to n and from n to *out*. We are going to consider a less restrictive requirement, namely that every transition has some dynamic postcondition. In that case, and considering some very reasonable requirements, dynamic soundness is decidable

even if shared resources can be consumed or created by instances. This reasonable requirement is the following: when a single instance is given arbitrarily many global resources, then it evolves properly. This is equivalent to just removing static places.

Let $N = (P, T, F)$ be a wf-net. We denote by m_{in} the marking of N given by $m_{in}(in) = 1$ and $m_{in}(p) = 0$ for $p \neq in$. Analogously, we define m_{out} as the marking of N given by $m_{out}(out) = 1$ and $m_{out}(p) = 0$ for $p \neq out$. A wf-net N is *sound* [2] if for every marking m reachable from m_{in}, m_{out} is reachable from m. We are now ready to define our subclass of rcwf-nets:

Definition 5. *We say that a rcwf-net $N = (P, T, F)$ is a* proper *rcwf-net if the two following conditions hold:*

- *for each $t \in T$, $t^\bullet \cap P_D \neq \emptyset$,*
- *the production net N_p of N is sound.*

Intuitively, the behavior of N_p represents the maximal behavior of each instance of N. In particular, if m is a reachable marking of a rcwf-net N, then the markings of N_p obtained by projecting m to each of the names in m are all reachable too.

In [3,4] other classes more restricted than proper rcwf-nets are defined.[1] However, the previous conditions are enough for our decidability result, and indeed our requirement can be deduced from the conditions required in [3,4].

Lemma 1. *The production net N_p of a proper rcwf-net N is bounded.*

Proof. Let us suppose that N_p is sound and unbounded (assuming the initial marking m_0^1). Then, there are markings of N_p, m_1, m_2, and m_1' such that $m_0^1 \rightarrow^* m_1 \rightarrow^* m_2 = m_1 + m_1'$ with m_1' non empty. Since N_p is sound, $m_1 \rightarrow out$, so that $m_2 = m_1 + m_1' \rightarrow^* out + m_1'$. Again, by soundness of N_p, it must be the case that $out + m_1' \rightarrow^* out$. Since $out^\bullet = \emptyset$, it must be the case that $m_1' \rightarrow^* \emptyset$, but this is not possible because N is proper (and, in particular, all the transitions of N_p have postconditions).

Actually, in the proof of decidability of dynamic soundness for proper rcwf-nets, we only need that the production net is bounded (and boundedness is decidable for P/T nets). By the previous result, we know that the production net of a proper rcwf-net is bounded, but even if our rcwf-net is not proper, we can still check whether its production net is bounded, in which case our proof still holds. We reduce dynamic soundness to a home space problem in P/T nets.

Let us explain intuitively how the construction works. It is similar to a construction used in [4]. Given a proper rcwf-net N, we know that N_p is bounded. Then, we can consider the state machine associated to the reachability graph of N_p. More precisely, if m is a reachable marking in N_p, then we will consider a place also denoted by m. A token in m stands for an instance of N in state m.

[1] E.g., by demanding that there are paths from *in* to every node, and from every node to *out*.

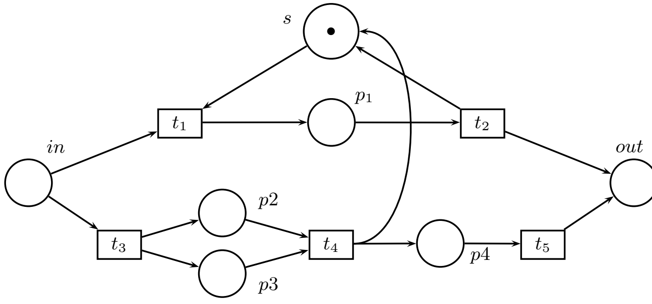


Fig. 6. A proper rcwf-net N

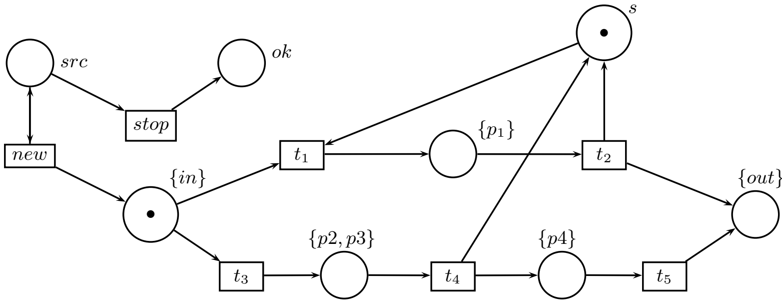


Fig. 7. N^{tr} obtained by applying Def. 6 to N in Fig. 6 (omitting the arcs from *ok*)

Notice that this is correct because all the markings reachable in N must be reachable in N_p (after projecting). So far, it is like in [4]. Moreover, the static places will be considered as places of the new net too, and will be pre/postconditions of the transitions we add, in the same way as they were in the original net.

Finally, we have to consider one more place *src* in order to set the initial number of instances that we are going to consider for the net. Let us denote by $\mathcal{R}(N)$ the set of markings reachable in a wf-net net N from m_{in}. Now we are ready to define the construction which will let us prove the decidability of dynamic soundness.

Definition 6. *Let $N = (P, T, F)$ be a proper rcwf-net and $m_0^s \in P_S^{\oplus}$. We define the P/T net $N^{tr} = (P^{tr}, T^{tr}, F^{tr})$ as follows:*

- $P^{tr} = P_S \cup \mathcal{R}(N_p) \cup \{src, ok\}$,
- $T^{tr} = \{(m_1, t, m_2) \in \mathcal{R}(N_p) \times T \times \mathcal{R}(N_p) \mid m_1 \xrightarrow{t} m_2 \text{ in } N_p\} \cup \{new, stop\}$,
- F^{tr} *is such that:*
  - $F^{tr}(m_1, (m_1, t, m_2)) = F^{tr}((m_1, t, m_2), m_2) = 1$,
  - $F^{tr}(src, stop) = F^{tr}(stop, ok) = 1$,
  - $F^{tr}(src, new) = F^{tr}(new, src) = F^{tr}(new, in) = 1$,
  - $F^{tr}(ok, (m_1, t, m_2)) = F((m_1, t, m_2), ok) = 1$,
  - *If* $s \in P_S$, $F^{tr}((m_1, t, m_2), s) = F(t, s)$ *and* $F^{tr}(s, (m_1, t, m_2)) = F(s, t)$,
  - $F^{tr}(x, y) = 0$, *otherwise.*

The initial marking of N^{tr} is m_0^{tr}, given by $m_0^{tr}(src) = 1$, $m_0^{tr}(m) = 0$ for $m \in \mathcal{R}(N_p)$ and $m_0^{tr}(s) = m_0^s(s)$ for $s \in P_S$.

Figure 7 shows the previous construction for the net in Fig. 6. Note that N^{tr} is finite because N_p is bounded, so that it can be effectively computed. Intuitively, N^{tr} creates by means of transition *new* several instances in its initial state, after which if fires *stop*, marking place *ok*, which is a precondition of the rest of the transitions, so that from then on they can be fired.[2] Each token in a place $m \in \mathcal{R}(N_p)$ of N^{tr} represents an instance of N, running concurrently with other instances and sharing the resources in the static places with them. Therefore, the net will simulate runs of as many instances of the original net as times the transition *new* has been fired. Let us define a correspondence between the markings of N and the markings of N^{tr}.

Definition 7. *Given a marking m of N, we define the marking m^{tr} of N^{tr} as follows: $m^{tr}(src) = 0$, $m^{tr}(ok) = 1$, $m^{tr}(s) = m(s)(\bullet)$ for $s \in P_S$, and $m^{tr}(m') = |\{a \in Id(m) \mid m(p)(a) = m'(p)\ \forall p \in P_D\}|$, that is, the number of instances in state m'.*

Notice that all the markings reachable in N^{tr} with *ok* marked are of the form m^{tr} for some marking m reachable in N. The following result is trivial by construction of N^{tr}.

Lemma 2. *$m_0^k \rightarrow^* m$ in N if and only if $m_0^{tr} \xrightarrow{new^k \cdot stop} (m_0^k)^{tr} \rightarrow^* m^{tr}$. Moreover, all the computations in N^{tr} start by firing new $k \geq 0$ times, possibly followed by stop, in which case $(m_0^k)^{tr}$ is reached.*

Finally, we are ready to prove that this construction reduces the dynamic soundness problem for proper rcwf-nets to the home space problem for P/T nets. We denote by e_p the marking given by $e_p(p) = 1$ and $e_p(q) = 0$ for $p \neq q$.

Proposition 3. *Let N be a proper rcwf-net. N is dynamically sound if and only if the linear set $\mathcal{L}$ generated by $\{out\} \cup \{e_s \mid s \in P_S\}$ is a home space for N^{tr}.*

Proof. We start by remarking that $\mathcal{L}$ contains markings with any number of tokens in *out* and in static places, and empty elsewhere. Notice also that each transition different from *new* and *stop* has exactly one precondition in $\mathcal{R}(N_p)$ and one postcondition in $\mathcal{R}(N_p)$. Therefore, after the firing of *stop*, the total number of tokens in places in $\mathcal{R}(N_p)$ remains constant. Therefore, if *new* is fired k times and a marking in $\mathcal{L}$ is reached, then necessarily this marking has k tokens in *out*. Finally, notice that $m \in \mathcal{M}_{out}^k$ iff $m^{tr} \in \mathcal{L}$ and it contains exactly k tokens in *out*.

Let us first suppose that N is not dynamically sound. Then, there is a $k > 0$ and a marking m reachable from m_0^k from which no marking in $\mathcal{M}_{out}^k$ is reachable. By Lemma 2, the marking m^{tr} is reachable after firing *new* k times. Then, from

[2] Actually, the construction still works without place *ok*, though it simplifies the forthcoming explanations.

m^{tr} no marking in $\mathcal{L}$ can be reached. Indeed, if some marking m'^{tr} in $\mathcal{L}$ is reached from m^{tr} it has necessarily k tokens in *out* and again by Lemma 2, $m' \in \mathcal{M}^k_{out}$ is reached in N, contradicting our first hypothesis. Then, $\mathcal{L}$ is not a home space and we conclude this implication.

Reciprocally, let us assume that $\mathcal{L}$ is not a home space. Then, there is a reachable marking of N^{tr} from which no marking of $\mathcal{L}$ can be reached. Let us suppose that this marking is of the form m^{tr} (otherwise, we consider the marking obtained after firing *stop*, and no marking of $\mathcal{L}$ can be reached from it). Let us suppose that there are k tokens in places in $\mathcal{R}(N_p)$ in m^{tr}. Then, by Lemma 2 and the previous remarks (analogously to the previous case) no marking in $\mathcal{M}^k_{out}$ can be reached from m, so that N is not dynamically sound.

Finally, as the home space problem is decidable for linear sets of markings of P/T nets [9], we obtain the following result:

Corollary 3. *Dynamic soundness for proper rcwf-nets is decidable.*

7 Conclusions and Future Work

In this paper we have continued the study of concurrent workflow processes that share some global resources, first studied in [3] and more recently in [4]. In particular, we consider resource-constrained workflow nets in which each instance is allowed to consume or create new resources.

We have first established the undecidability of dynamic soundness for rcwf-nets when the use of resources is unrestricted, so that each instance is allowed to terminate a run having consumed or created global resources. Such result is achieved by means of an alternative presentation of rcwf-nets which is closer to ν-PN. More precisely, we have defined a subclass of ν-PN in which processes can only interact asynchronously with each other via a global shared memory, thus bringing together ν-PN and rcwf-nets. We have then seen that the undecidability of the reachability problem for ν-PN can be transfered to its asynchronous subclass. Although we have focused on reachability, we claim that most undecidability results can also be transfered, so that both classes are essentially equivalent. Then we have reduced this reachability problem to dynamic soundness of rcwf-nets. This reduction is not at all trivial, even though the alternative presentation of rcwf-nets eases the simulation of ν-PN by means of them (third step of the reduction).

Then we have considered a subproblem of the latter. In the first place, we assume that each instance is sound when it is given infinitely many resources (which amounts to saying that its behavior is not restricted by global resources). Moreover, we assume a technical condition, which is weaker than the standard "path property", that intuitively meas that all transitions are significant in the completion of the task. Under these hypotheses, we prove that dynamic soundness is decidable, by reducing it to a home space problem for a linear set of home markings, which is decidable.

There are many ways in which this work must be continued. The most important one could be to bridge the gap between our undecidability and our decidability result. In other words, we must address the problem of dynamic soundness whenever the path property holds, without assuming that a single instance of the net is necessarily sound (and always without assuming that instances give back the resources they use). Notice that our undecidability proof is no longer valid if we assume that property (indeed, the step 4 of our construction has transitions without postconditions), and it does not seem possible to easily fix this issue.

A dynamically sound rcwf-nets in which some proper runs may consume global resources without returning them must necessarily have other runs in which there is no need to consume global resources. Intuitively, the first run may be more desirable than the second one in terms of some measure which lies outside of the model. In this sense, a *priced* extension of rcwf-nets [12] in which runs from *in* to *out* have an associated cost could be interesting to be studied.

References

1. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge (2002)
2. van der Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
3. van Hee, K.M., Serebrenik, A., Sidorova, N., Voorhoeve, M.: Soundness of resource-constrained workflow nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 250–267. Springer, Heidelberg (2005)
4. Juhás, G., Kazlov, I., Juhásová, A.: Instance deadlock: A mystery behind frozen programs. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 1–17. Springer, Heidelberg (2010)
5. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets with reset arcs. T. Petri Nets and Other Models of Concurrency 3, 50–70 (2009)
6. Rosa-Velardo, F., de Frutos-Escrig, D.: Name creation vs. replication in petri net systems. Fundam. Inform. 88, 329–356 (2008)
7. Rosa-Velardo, F., de Frutos-Escrig, D., Alonso, O.M.: On the expressiveness of mobile synchronizing petri nets. Electr. Notes Theor. Comput. Sci. 180, 77–94 (2007)
8. Rosa-Velardo, F., de Frutos-Escrig, D.: Decision problems for petri nets with names. CoRR abs/1011.3964 3964 (2010)
9. de Frutos-Escrig, D., Johnen, C.: Decidability of home space property. Technical Report LRI-503, Univ. de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique (1989)
10. Esparza, J., Nielsen, M.: Decidability issues for petri nets - a survey. Bulletin of the EATCS 52, 244–262 (1994)
11. Reisig, W.: Petri Nets: An Introduction. Monographs in Theoretical Computer Science. An EATCS Series, vol. 4. Springer, Heidelberg (1985)
12. Abdulla, P.A., Mayr, R.: Minimal cost reachability/coverability in priced timed petri nets. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 348–363. Springer, Heidelberg (2009)

SimGrid MC: Verification Support for a Multi-API Simulation Platform

Stephan Merz[1], Martin Quinson[2], and Cristian Rosa[3]

[1] INRIA Research Center Nancy, France
stephan.merz@loria.fr
[2] Université Henri Poincaré Nancy 1, Nancy, France
martin.quinson@loria.fr
[3] Université Henri Poincaré Nancy 1, Nancy, France
cristian.rosa@loria.fr

Abstract. SimGrid MC is a stateless model checker for distributed systems that is part of the SimGrid Simulation Framework. It verifies implementations of distributed algorithms, written in C and using any of several communication APIs provided by the simulator. Because the model checker is fully integrated in the simulator that programmers use to validate their implementations, they gain powerful verification capabilities without having to adapt their code. We describe the architecture of SimGrid MC, and show how it copes with the state space explosion problem. In particular, we argue that a generic Dynamic Partial Order Reductions algorithm is effective for handling the different communication APIs that are provided by SimGrid. As a case study, we verify an implementation of Chord, where SimGrid MC helped us discover an intricate bug in a matter of seconds.

1 Introduction

Distributed systems are in the mainstream of information technology. It has become standard to rely on multiple distributed units that collectively contribute to a business or scientific application. Designing and debugging such applications is particularly difficult: beyond issues common to any parallel (i.e., concurrent or distributed) program, such as race conditions, deadlocks or livelocks, distributed systems pose additional problems due to asynchronous communication between nodes and the impossibility for any node to observe a global system state.

The most common approach to validating distributed applications is to execute them over a given testbed. However, many different execution platforms exist, and it is difficult to assess the behavior of the application on another platform than the one that the programmer has access to. Simulation constitutes another approach, offering the ability to evaluate the code in more comprehensive (even if not exhaustive) test campaigns. It remains however difficult to determine whether the test campaign is sufficient to cover all situations that may occur in real settings. That is why distributed applications are usually only tested on a very limited set of conditions before being used in production.

In recent years the use of formal validation techniques has become more prominent for the evaluation of concurrent and distributed software. Due to their

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 274–288, 2011.
© IFIP International Federation for Information Processing 2011

simplicity and the high degree of automation, model checking techniques have been particularly successful. Relying on exhaustive state space exploration, they can be used to establish whether a system, or a formal model of it, meets a given specification.

Initially, verification techniques were developed for formal modeling languages such as process algebras [1], Petri nets [2], Promela [3] or TLA$^+$ [4], in which algorithms and protocols can be modeled at a high level of abstraction. However, many errors are introduced at the implementation phase, and these can obviously not be detected by formal verification of models. Several authors have considered the application of model checking techniques to source or binary code [5,6,7,8], and our work contributes to this line of research. It promises to catch subtle bugs in actual programs that escape standard testing or simulation and to give non-specialists access to powerful verification methods.

The main impediment to make the approach work in practice is the well-known state explosion problem: the state space of even small programs executed by a few processes is far too large to construct exhaustively. Powerful reduction techniques such as dynamic partial-order reduction (DPOR [9]) must be used to cut down the number of executions that must be explored. DPOR relies on the notion of independent transitions, which must be established for the semantics of real-world programs, which can be a daunting task [10].

The contributions of this article are the following:

- We present SimGrid MC, an extension of the SimGrid simulation framework [11] for the formal verification of properties of distributed applications that communicate by message passing. We believe that by integrating verification capabilities into an existing simulation environment, we are able to close the loop of the development process: developers can assess their implementations for both correctness and performance, using the same overall framework.
- We detail the changes that we made to the main simulation loop to implement the model checking functionality, and how this was eased by the similarities between both tasks.
- We explain how we could implement the DPOR algorithm to support the different communication APIs offered by SimGrid through an intermediate communication layer, for which independence of transitions is established.

This article is organized as follows: Section 2 introduces the SimGrid simulation framework. Section 3 presents the model checker SimGrid MC, its implementation within the SimGrid framework, and our implementation of DPOR for multiple communication APIs available in SimGrid. Section 4 evaluates the resulting tool through several experiments. Finally, Section 5 concludes the paper and discusses future work.

1.1 State of the Art

The idea of applying model checking to actual programs originated in the late 1990s [5,8]. One of the main problems is the representation and storage of

system states: C programs freely manipulate the stack and heap, and it becomes difficult to determine the part that should be saved, and costly to actually store and retrieve it. Godefroid [12] proposed the idea of stateless model checking, in which executions are re-run instead of saving system states. Flanagan and Godefroid also introduced the idea of DPOR [9], although they were not primarily interested in model checking distributed systems. The closest work to ours is probably ISP [13], which is a stateless model checker for MPI applications that also relies on DPOR for effective reductions. ISP is not implemented in a simulation framework, but intercepts the calls to the runtime to force the desired interleavings. MACE [6] is a set of C++ APIs for implementing distributed systems; it contains a model checker geared towards finding dead states. To our knowledge SimGrid MC is the only model checker for distributed applications that supports multiple communication APIs and is tightly integrated with a simulation platform.

2 The SimGrid Framework

2.1 SimGrid Architecture

The SimGrid framework [11] is a collection of tools for the simulation of distributed computer systems. The simulator requires the following inputs:

The application or protocol to test. It must use one of the communication APIs provided in SimGrid, and must be written in one of the supported languages, including C and Java.

Analytical models of the used hardware. These models are used by the simulator to compute the completion time of each application action, taking in account the hardware capacity and the resources shared between application elements and with the external load.

A description of the experimental setup. This includes the hardware platform (hosts, network topology and routing), the external workload experienced by the platform during the experiment, and a description of the test application deployment.

The simulator then executes the application processes in a controlled environment, in which certain events, and in particular communications, are intercepted to evaluate timing and the use of shared resources, according to the models and the description of the setup (links, hosts, etc).

Figure 1 shows the architecture of the SimGrid framework. The analytical models of the resources are provided by the SURF layer, which is the simulation core. On top of this, SIMIX constitutes the virtualization layer. It adds the notion of process, synchronization, communication primitives, and controls the execution of the user processes. Three different communication APIs (or user interfaces) are built on top of the abstractions provided by SIMIX; they are adapted to different usage contexts. MSG uses a communication model that is based on messages exchanged through mailboxes; messages are characterized

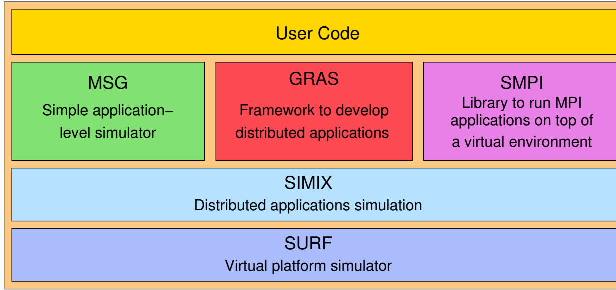


Fig. 1. The SimGrid Architecture

as tasks with computation and communication costs. GRAS is a socket-based event loop API designed to be executed in the simulator or deployed in real-life platforms. SMPI stands for Simulated MPI, and allows a user to simulate standard MPI programs without modifications, unlike the other two interfaces that are specific to SimGrid. It implements a subset of MPI that includes two-sided synchronous and asynchronous communication, as well as group operations.

The experimental setup is provided by the user through configuration files that instantiate the models, describing where the processes will be deployed.

Let us point out that SimGrid is a *simulator* and not an *emulator*. Its goal is to compute the timings of the events issued by the system as if it would execute on the virtual platform, irrespective of the speed of the platform on which the simulation is run. In particular, processes always execute at full speed.

2.2 The Simulation Loop

Before we present the model checker for SimGrid, we give some more details about the simulator's main loop, *i.e.* how the simulator controls the execution of the tested application depending on the platform models. This background is necessary in order to understand how SimGrid MC builds upon the infrastructure provided by the simulator.

SimGrid runs the entire simulation as a single process in the host machine by folding every simulated user process in a separate thread. The simulator itself runs in a special distinguished thread called *maestro*, which controls the scheduling.

Figure 2 depicts two simulation rounds starting at time t_{n-1} with two user threads T_1 and T_2 running the simulated processes and the maestro thread M. The round starts by calling SURF to compute and advance to the time of the next ending actions, in this case t_n. Next, it passes the list of finished actions to SIMIX that has a table associating them to the blocked threads. Using this table, SIMIX schedules all the unblocked threads. The user threads run without interruption until they block, waiting for a new simulation action, such as a communication. These actions are denoted in Fig. 2 by a and b for threads T_1 and T_2. The simulation round finishes once all the threads were executed until

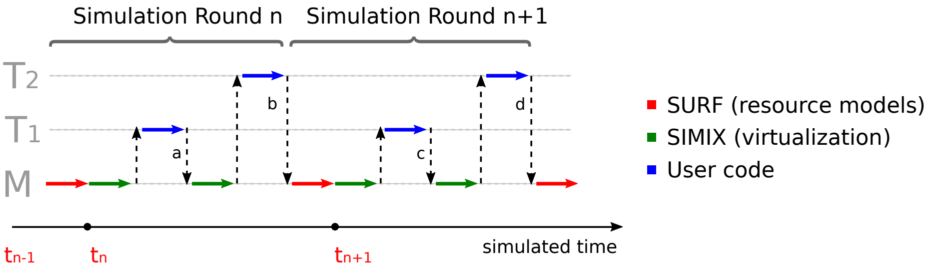


Fig. 2. Simulation Main Loop

they block. Note that the time advances only between scheduling rounds, thus from the simulator's point of view, all the actions performed by the user in a scheduling round happen at the same time.

3 Model Checking Distributed Programs in SimGrid

SimGrid provides us with the capability of simulating distributed programs in a controlled environment. In particular, it includes functionality for managing the control state, memory, and communication requests of simulated processes, which can be run selectively and interrupted at visible (communication) actions. We now outline how we used this framework to implement verification capabilities, and describe the techniques we employed to make them efficient. In this article, we focus only on the C interface.

3.1 SimGrid MC

Unlike simulation, which is mainly concerned with the use of available resources and the performance of a distributed application in a given scenario, verification attempts to exhaustively explore the state space of a system in order to detect corner cases that one would be unlikely to encounter in simulation runs. Typically, the number of process instances will be significantly smaller in verification than in simulation because of the well known problem of state space explosion. We designed SimGrid MC as a complement to the simulator functionality, allowing a user to verify instances of distributed systems, *without requiring any modifications to the program code*. In the schema presented in Fig. 1, SimGrid MC replaces the SURF module and a few submodules of SIMIX with a state exploration algorithm that exhaustively explores the executions arising from all possible non-deterministic choices of the application.

As we explained before, a distributed system in SimGrid consists of a set of processes that execute asynchronously in separate address spaces, and that interact by exchanging messages. In other words, there is no global clock for synchronization, nor shared memory accessed by different processes.

More precisely, the state of a process is determined by its CPU registers, the stack, and the allocated heap memory. The network's state is given by the

messages in transit, and it is the only shared state among processes. Finally, the global state consists of the state of every process plus the network state. The only way a process can modify the shared state (the network) is by issuing calls to the communication APIs, thus the model-checker considers these as the only visible transitions. A process transition as seen by the model checker therefore comprises the modification of the shared state, followed by all the internal computations of the process until the instruction before the next call to the communication API. The state space is then generated by the different interleavings of these transitions; it is generally infinite even for a bounded number of processes due to the unconstrained effects on the memory and the operations processes perform.

Because the global state contains unstructured heaps, and the transition relation is determined by the execution of C program code, it is impractical to represent the state space or the transition relation symbolically. Instead, SimGrid MC is an explicit-state model checker that explores the state space by systematically interleaving process executions in depth-first order, storing a stack that represents the schedule history. As the state space may be infinite, the exploration is cut off when a user-specified execution depth is reached. Of course, this means that error states beyond the search bound will be missed, but we consider SimGrid MC as a debugging tool that is most useful when it succeeds in finding an error. SimGrid MC ensures complete exploration of the state space up to the search bound.

When state exploration hits the search bound (or if the program terminates earlier), we need to backtrack to a suitable point in the search history and continue exploration from that global state. In a naïve implementation, this would mean check-pointing the global system state at every step, which is prohibitive due to the memory requirements and the performance hit incurred by copying all the heaps. Instead, we adopt the idea of stateless model checking [12] where backtracking is implemented by resetting the system to its initial state and re-executing the schedule stored in the search stack until the desired backtracking point. Because global states are not stored, SimGrid MC has no way of detecting cycles in the search history and may re-explore parts of the state space that it has already seen. Note that even if we decided to (perhaps occasionally) checkpoint the system state, dynamic memory allocation would require us to implement some form of heap canonicalization [8,14] in order to reliably detect cycles. In the context of bounded search that we use, the possible overhead of re-exploring states because of undetected loops is a minor concern for the verification of safety properties. It would, however, become necessary for checking liveness properties.

Figure 3 illustrates the exploration technique used by SimGrid MC on the example used in Sect. 2.2. The model checker first executes the code of all threads up to, but excluding, their first call to the communication API (actions a and b in this example). The resulting global state S_0 (indicated by a red dot in Fig. 3) is pushed on the exploration stack; it is also stored as the snapshot corresponding to the initial state, and the model checker records the enabled actions. It then chooses one action (say, a) for execution and schedules the associated thread, which performs the communication action and all following local program steps

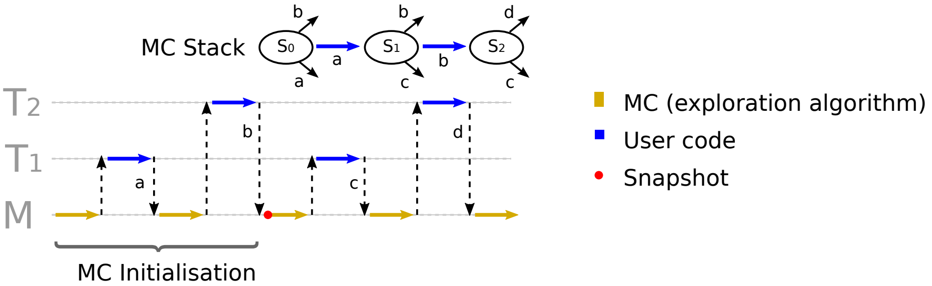


Fig. 3. State Exploration by SimGrid MC

up to, but excluding, the next API call (c). This execution corresponds to one transition as considered by the model checker, which records the actions enabled at this point and selects one of them (say, b), continuing in this way until the exploration reaches the depth bound or no more actions are enabled; depending on the process states, the latter situation corresponds either to a deadlock or to program termination.

At this point, the model checker has to backtrack. It does so by retrieving the global state S_0 stored at the beginning of the execution, restoring the process states (CPU registers, stack and heap) from this snapshot, and then replaying the previously considered execution until it reaches the global state from which it wishes to continue the exploration. (The choice of backtrack points will be explained in more detail in Sect. 3.2 below.) In this way, it achieves the illusion of rewinding the global application state until a previous point in history.

Figure 4 illustrates the architecture of SimGrid MC. Each solid box labeled P_i represents a thread executing the code of a process in the distributed system being verified. The exploration algorithm is executed by a particular thread labeled MC that intercepts the calls to the communication API (dashed box) and updates the state of the (simulated) communication network. The areas colored blue represent the system being explored, the area colored red corresponds to

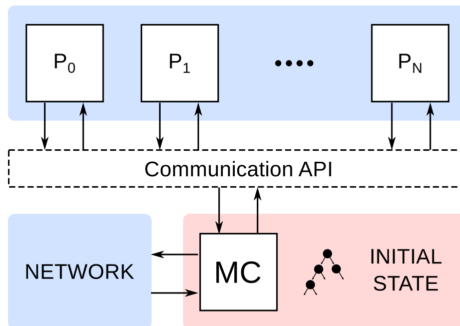


Fig. 4. SimGrid MC Architecture

the state of the model checker, which holds the snapshot of the initial state and the exploration stack. When a backtracking point is reached, the blue area is reset as described above, but the exploration history is preserved intact.

3.2 Partial Order Reduction for Multiple Communication APIs

The main problem in the verification of distributed programs, even using stateless model checking, lies in the enormous number of interleavings that these programs generate. Usually, many of these interleavings are equivalent in the sense that they lead to indistinguishable global states.

Algorithm 1. Depth-first search with DPOR

```
1: q := initial state
2: s := empty
3: for some p ∈ Proc that has an enabled transition in q do
4:    interleave(q) := {p}
5: end for
6: push(s,q)
7: while |s| > 0 do
8:    q := top(s)
9:    if |unexplored(interleave(q))| > 0 ∧ |s| < BOUND then
10:      t := nextinterleaved(q)
11:      q' := succ(t, q)
12:      for some p ∈ Proc that has an enabled transition in q' do
13:         interleave(q') := {p}
14:      end for
15:      push(s,q')
16:   else
17:      if ∃ i ∈ dom(s): Depend(tran(s_i), tran(q)) then
18:         j := max({i ∈ dom(s): Depend(tran(s_i), tran(q))})
19:         interleave(s_{j−1}) := interleave(s_{j−1}) ∪ {proc(tran(q))}
20:      end if
21:      pop(s)
22:   end if
23: end while
```

(Dynamic) Partial-Order Reduction [9] has proved to be efficient for avoiding the exploration of equivalent interleavings, and we also rely on this technique in SimGrid MC. The pseudo-code of the depth-first search algorithm implementing DPOR appears in Algorithm 1. With every scheduling history q on the exploration stack is associated a set $interleave(q)$ of processes enabled at q and whose successors will be explored. Initially, an arbitrary enabled process p is selected for exploration. At every iteration, the model checker considers the history q at the top of the stack. If there remains at least one process selected for exploration at q, but which has not yet been explored, and the search bound has not yet been reached, one of these processes (t) is chosen and scheduled for execution,

resulting in a new history q'. The model checker pushes q' on the exploration stack, identifying some enabled process that must be explored. Upon backtracking, the algorithm looks for the most recent history s_j on the stack for which the transition $tran(s_j)$ executed to generate s_j is dependent with the incoming transition $tran(q)$ of the state about to be popped. If such a history exists, the process executing $tran(q)$ is added to the set of transitions to be explored at the predecessor of s_j, ensuring its successor will be explored during backtracking (if it has not yet been explored). Our algorithm is somewhat simpler than the original presentation of DPOR [9] because it assumes that transitions remain enabled until they execute, which is the case for SimGrid.

The effectiveness of the reductions achieved by the DPOR algorithm is crucially affected by the precision with which the underlying dependency relation can be computed. The two extremes are to consider all or no transitions as dependent. In the first case, the DPOR algorithm degenerates to full depth-first search. In the second case, it will explore only one successor per state and may miss interleavings that lead to errors. A sound definition of dependence must ensure that two transitions are considered independent only if they commute, and preserve the enabledness of the other transition, at any (global) state where they are both enabled. Because processes do not share global memory in SimGrid, memory updates cannot contribute to dependence, and we need only consider the semantics of the communication actions. However, their semantics must be described formally enough to determine (in)dependence.

In the case of the SimGrid Simulation Framework, the programs can be written using one of its three communication APIs, which lack such formal specification, as they were not designed for formal reasoning. Palmer et al. [10] have given a formal semantics of a substantial part of MPI for use with DPOR, but this is a tedious and daunting task, which would have to be repeated for the other APIs in SimGrid.

Instead, our implementation of DPOR in SimGrid relies on the definition of a minimal internal networking API, for which we have given a fully formal semantics and for which we have proved independence theorems in our previous work [15]. The three communication APIs provided by SimGrid are implemented on top of this basic API, and the DPOR-based model checker presented in Algorithm 1 operates at the level of these elementary primitives.

The communication model used by this set of networking primitives is built around the concept of "mailbox". Processes willing to communicate queue their requests in mailboxes, and the actual communication takes place when a matching pair is found. The API provides just the four operations *Send*, *Recv*, *WaitAny* and *TestAny*. The first two post a send or receive request into a mailbox, returning a communication identifier. A *Send* matches any *Recv* for the same mailbox, and vice versa. The operation *WaitAny* takes as argument a set of communication identifiers and blocks until one of them has been completed. Finally, *TestAny* also expects a set of communication identifiers and checks if any of these communications has already completed; it returns a Boolean result and never blocks.

Listing 3.1. Inefficient WaitAll

```
1 void WaitAll(comm_list[])
2 {
3   while(len(comm_list) > 0){
4     comm = WaitAny(comm_list);
5     list_remove(comm, comm_list);
6   }
7 }
```

Listing 3.2. Efficient WaitAll

```
1 void WaitAll(comm_list[])
2 {
3   for(i=0;i<len(comm_list);i++){
4     WaitAny(comm_list[i]);
5   }
6 }
```

We specified these primitives formally in TLA$^+$ [4], and for every pair of communication operations we formally proved conditions ensuring their independence. Moreover, it was surprisingly easy to implement SimGrid's communication APIs in terms of these primitive operations.

However, it was not clear a priori that this approach would result in a satisfactory degree of reduction, as the implementations of two high-level operations in terms of the lower-level ones might falsely be considered dependent. Moreover, the implementation of the higher-level APIs may introduce additional non-determinism, generating spurious interleavings during model checking. For example, consider the implementation of listing 3.1: it expects a set of communications identifiers and repeatedly uses *WaitAny* for all unfinished communications, until no one is left. While correct, such an implementation would introduce a non-deterministic choice among the finished communications, which is irrelevant to the semantics of *WaitAll* but would be considered by the model checker. For our purposes, it is therefore better to issue *WaitAny* operations in sequence for all the communication operations as shown in listing 3.2.

4 Experimental Results

In this section we present a few verification experiments using two of the APIs supported by SimGrid. We thus illustrate the ability of our approach to use a generic DPOR exploration algorithm for different communication APIs through an intermediate communication layer. Each experiment aims to evaluate the effectiveness of the DPOR exploration at this lower level of abstraction compared to a simple DFS exploration. We use a depth bound fixed at 1000 transitions (which was never reached in these experiments), and run SimGrid SVN revision 9888 on a CPU Intel Core2 Duo T7200 2.0GHz with 1GB of RAM under Linux.

4.1 SMPI Experiments

The first case study is based upon two small C programs using MPI that are designed to measure the performance of our DPOR algorithm.

The first example, presented in Listing 4.1, shows an MPI program with N+1 processes. The process with rank 0 waits for a message from each of the other processes, while the other processes send their rank value to process 0. The

Listing 4.1. Example 1

```
1  if (rank == 0){
2    for (i=0; i < N-1; i++){
3      MPI_Recv(&val, MPI_ANY_SOURCE);
4    }
5    MC_assert(val == N);
6  } else {
7    MPI_Send(&rank, 0);
8  }
```

Listing 4.2. Example 2

```
1  if (rank % 3 == 0) {
2    MPI_Recv(&val, MPI_ANY_SOURCE);
3    MPI_Recv(&val, MPI_ANY_SOURCE);
4  } else {
5    MPI_Send(&rank, (rank / 3) * 3);
6  }
```

property to verify is coded as the assertion at line 5 that checks for the incorrect assumption of a fixed message receive order, where the last received message will be always from the process with rank N.

Table 1(a) shows the timing and the number of states visited before finding a violation of the assertion. In this case, the number of processes does not have a significant impact on the number of visited states because the error state appears early in the visiting order of the DFS. Still, using DPOR helps to reduce the number of visited states by more than 50% when compared to standard DFS.

Table 1. Timing, number of expanded states, and peak memory usage (a) to find the assertion violation in Listing 4.1; (b) for complete state space coverage of Listing 4.1 and (c) for complete state space coverage of Listing 4.2

(a)

#P	DFS			DPOR		
	States	Time	Peak Mem	States	Time	Peak Mem
3	119	0.097 s	23952 kB	43	0.063 s	23952 kB
4	123	0.114 s	25008 kB	47	0.064 s	25024 kB
5	127	0.112 s	26096 kB	51	0.072 s	26080 kB

(b)

#P	DFS			DPOR		
	States	Time	Peak Mem	States	Time	Peak Mem
2	13	0.054 s	21904 kB	5	0.046 s	18784 kB
3	520	0.216 s	23472 kB	72	0.069 s	23472 kB
4	60893	19.076 s	24000 kB	3382	0.913 s	24016 kB
5	-	-	-	297171	84.271 s	25584 kB

(c)

#P	DFS			DPOR		
	States	Time	Peak Mem	States	Time	Peak Mem
3	520	0.247 s	23472 kB	72	0.074 s	23472 kB
6	>10560579	>1 h	-	1563	0.595 s	26128 kB
9	-	-	-	32874	14.118 s	29824 kB

Table 1(b) shows the effectiveness of DPOR for a complete state space exploration of the same program (without the assertion). Here, the use of DPOR reduces the number of visited states by an order of magnitude.

The second example, presented in Listing 4.2, shows the relevance of performing reduction dynamically. This time the number of processes in the system should be a multiple of 3. Every process with a rank that is a multiple of three will wait for a message from the next two processes, thus process 0 will receive from processes 1 and 2, process 3 from processes 4 and 5, etc. It is quite obvious that each group of three processes is independent from the others, but standard static reduction techniques would not be able to determine this. Again, no property is verified, as we try to compare the reductions obtained by DPOR.

Table 1(c) shows the experimental results for a complete exploration of the state space. In this case the DFS with 6 processes was interrupted after one hour, and up to that point it had visited 320 times more states than the complete state space exploration of the same program for 9 processes with DPOR enabled.

4.2 MSG Experiment: CHORD

As our second case study we consider an implementation of Chord [16] using the MSG communication API. Chord is a well known peer-to-peer lookup service, designed to be scalable, and to function even with nodes leaving and joining the system. This implementation was originally developed to study the performance of the SimGrid simulator.

The algorithm works in phases to stabilize the lookup information on every node that form a logical ring. During each phase, nodes exchange messages to update their knowledge about who left and joined the ring, and eventually converge to a consistent global vision.

Listing 4.3 shows a simplified version of Chord's main loop. In MSG, processes exchange *tasks* containing the messages defined by the user. Each node starts an asynchronous task receive communication (line 3), waiting for petitions from the other nodes to be served. If there is one (the condition at line 4 is true), a handler is called to reply with the appropriate answer using the same received task (line 5). Otherwise, if the delay for the next lookup table update has passed, it performs the update in four steps: request the information (lines 7-9), wait for the answer (lines 12-14), update the lookup tables (line 19), and notify changes to other nodes (line 22).

Running Chord in the simulator, we occasionally spotted an incorrect task reception in line 14 that led to an invalid memory read, producing a segmentation fault. Due to the scheduling produced by the simulator, the problem only appeared when running simulations with more than 90 nodes. Although we thus knew that the code contained a problem, we were unable to identify the cause of the error because of the size of the instances where it appeared and the amount of debugging information that these generated.

We decided to use SimGrid MC to further investigate the issue, exploring a scenario with just two nodes and checking the property $task == update_task$ at line 15 of listing 4.3. In a matter of seconds we were able to trigger the bug and

Listing 4.3. Main loop of CHORD (simplified).

```
1  while(1) {
2    if (!rcv_comm)
3      rcv_comm = MSG_task_irecv(&task);
4    if (MSG_comm_test(rcv_comm)) {
5      handle(task);
6    } else if(time > next_update_time) {
7      /* Send update request task */
8      snd_comm = MSG_task_isend(&update_task);
9      MSG_task_wait(snd_comm);
10
11     /* Receive the answer */
12     if(rcv_comm == NULL)
13       rcv_comm = MSG_task_irecv(&task);
14     MSG_task_wait(rcv_comm);
15
16     MC_assert(task == update_task); /* <-- Assertion verified by the MC */
17
18     /* Update tables with received task */
19     update_tables(task);
20
21     /* Notify some nodes of changes */
22     notify();
23   } else {
24     sleep(5);
25   }
26 }
```

could understand the source of the problem by examining the counter-example trace, which appears in listing 4.4. It should be read top-down and the events of each node are tabulated for clarity. The Notify task sent by node 1 in line 22 of listing 4.3 is incorrectly taken by node 2 at line 14 as the answer to the update request sent by it line 8. This is due to an implementation error in the line 12: the code reuses the variable *recv_comm*, incorrectly assuming this to be safe because of the guard of that branch, but in fact the condition may change after the guard is evaluated.

Listing 4.4. Counter-example

```
 #line          Node 1                      #line          Node 2

 3: rcv_comm = MSG_task_irecv(&task)
 4: MSG_comm_test(rcv_comm) == FALSE
 8: snd_comm =
      MSG_MSG_task_isend(&update_task)
                                             3: rcv_comm = MSG_task_irecv(&task)
 9: MSG_task_wait(snd_comm)
                                             4: MSG_comm_test(rcv_comm) == TRUE
                                             5: handle(task)
                                             3: rcv_comm = MSG_task_irecv(&task)
                                             4: MSG_comm_test(rcv_comm) == FALSE
14: MSG_task_wait(recv_comm)
22: Notify()
 3: rcv_comm = MSG_task_irecv(&task)
                                             8: snd_comm =
                                                  MSG_MSG_task_isend(&update_task)
                                             9: MSG_task_wait(snd_comm)
                                            14: MSG_task_wait(recv_comm)
```

The Chord implementation that was verified has 563 lines of code, and the model checker found the bug after visiting just 478 states (in 0.280 s) using DPOR; without DPOR it had to compute 15600 states (requiring 24 s) before finding the error trace. Both runs had an approximate peak memory usage of 72 MB, measured with the `/usr/bin/time` program provided by the operating system.

5 Conclusions and Future Work

We have presented SimGrid MC, a model checker for distributed C programs that may use one of three different communication APIs. Like similar tools, SimGrid MC is based on the idea of stateless model checking, which avoids computing and storing the process state at interruptions, and relies on dynamic partial order reduction in order to make verification scale to realistic programs. One originality of SimGrid MC is that it is firmly integrated with the pre-existing simulation framework provided by SimGrid [11], allowing programmers to use the same code and the same platform for verification and for performance evaluation. Another specificity is the support for multiple communication APIs. We have implemented sensibly different APIs in terms of a small set of elementary primitives, for which we could provide a formal specification together with independence theorems with reasonable effort, rather than formalize three complete communication APIs. We have been pleasantly surprised by the fact that this approach has not compromised the degree of reductions that we obtain, which are roughly on a par with those reported in [10] for a DPOR algorithm specific to MPI.

The integration of the model checker in the existing SimGrid platform has been conceptually simple, because simulation and model checking share core functionality such as the virtualization of the execution environment and the ability to execute and interrupt user processes. However, model checking tries to explore all possible schedules, whereas simulation first generates a schedule that it then enforces for all processes. SimGrid benefitted from the development of SimGrid MC in that it led to a better modularization and reorganization of the existing code. The deep understanding of the execution semantics gained during this work lets us envision an efficient parallel simulation kernel in future work.

SimGrid MC is currently restricted to the verification of safety properties such as assertion violations or the detection of deadlock states. The verification of liveness properties would require us to detect cycles, which is currently impossible due to the stateless approach. For similar reasons, state exploration is limited by a (user-definable) search bound. We intend to investigate hybrid approaches between stateful and stateless model checking that would let us overcome these limitations.

Acknowledgment

The helpful comments of the anonymous reviewers are gratefully acknowledged. This work is partially supported by the ANR project USS SimGrid (08-ANR-SEGI-022).

References

1. Hennessy, M.: Algebraic Theory of Processes. MIT Press, Cambridge (1988)
2. Reisig, W.: A Primer in Petri Net Design. Springer, Heidelberg (1992)
3. Holzmann, G.J.: The model checker Spin. IEEE Trans. Softw. Eng. 23(5), 279–295 (1997)
4. Lamport, L.: Specifying Systems. Addison-Wesley, Boston (2002)
5. Visser, W., Havelund, K.: Model checking programs. Automated Software Engineering Journal, 3–12 (2000)
6. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: language support for building distributed systems. In: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation PLDI 2007, pp. 179–188. ACM, New York (2007)
7. Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation PLDI 2008, pp. 362–371. ACM Press, New York (2008)
8. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: A pragmatic approach to model checking real code. In: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, OSDI 2002 (2002)
9. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. SIGPLAN Not. 40(1), 110–121 (2005)
10. Palmer, R., Gopalakrishnan, G., Kirby, R.M.: Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In: Proceedings of the ACM workshop on Parallel and distributed systems: testing and debugging PADTAD 2007, pp. 43–53. ACM, New York (2007)
11. Casanova, H., Legrand, A., Quinson, M.: SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In: 10th IEEE International Conference on Computer Modeling and Simulation (March 2008)
12. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages POPL 1997, pp. 174–186. ACM, New York (1997)
13. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. SIGPLAN Not. 44(4), 261–270 (2009)
14. Iosif, R.: Exploiting heap symmetries in explicit-state model checking of software. In: Proceedings of the 16th IEEE international conference on Automated software engineering, ASE 2001, vol. 254, IEEE Computer Society, Washington (2001)
15. Rosa, C., Merz, S., Quinson, M.: A simple model of communication APIs – Application to dynamic partial-order reduction. In: 10th Intl. Workshop Automated Verification of Critical Systems, Düsseldorf, Germany, pp. 137–152 (2010)
16. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. 31, 149–160 (2001)

Ownership Types for the Join Calculus

Marco Patrignani[1], Dave Clarke[1], and Davide Sangiorgi[2,*]

[1] IBBT-DistriNet, Dept. Computer Sciences, Katholieke Universiteit Leuven
[2] Dipartimento di Scienze dell'Informazione, Università degli studi di Bologna

Abstract. This paper investigates ownership types in a concurrent setting using the Join calculus as the model of processes. Ownership types have the effect of statically preventing certain communication, and can block the accidental or malicious leakage of secrets. Intuitively, a channel defines a boundary and forbids access to its inside from outer channels, thus preserving the secrecy of the inner names from malicious outsiders. Secrecy is also preserved in the context of an untyped opponent.

1 Introduction

The Join calculus [16] is a message-based model of concurrency whose expressiveness is the same as the π-calculus up to weak barbed congruence [15]. It is a theoretical foundation of programming languages for distributed, mobile and concurrent systems, such as JOCaml [14], JErlang [22] and Scala's actor model [18]. The presence of several implementations of the calculus motivates the choice of Join calculus over π-calculus. The Join calculus has a notion of locality given by the channel definition construct: **def** `channels` **in** `scope`, but this can be broken by exporting a channel out of the environment it was created in. Ideally, scope boundaries are supposed to be crossed only by channels used for communication while channels defining secrets are supposed to remain within their scope. In practice, the malicious or accidental exportation of a channel name outside of a boundary it is not supposed to cross is a serious threat, because it can result in the leakage of secrets. Since crossing boundaries allows processes to communicate, it should not be eliminated, it has to be controlled.

Object-oriented programming suffers from a similar problem since object references can be passed around leading to issues like aliasing. As a remedy for this problem ownership types have been devised. Ownership types [11,9] statically enforce a notion of object-level encapsulation for object-oriented programming languages. Ownership types impose structural restrictions on object graphs based on the notions of *owner* and *representation*. The owner of an object is another object and its representation is the objects it owns. These two key concepts can be both defined and checked statically. The ultimate benefit imposed by ownership types is a statically-checkable notion of encapsulation: every object can have its own private collection of representation objects which are not accessible outside the object that owns them.

* Sangiorgi's reasearch partly supported by the MIUR project IPODS and the EU project FP7-231620 HATS

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 289–303, 2011.
© IFIP International Federation for Information Processing 2011

The aim of this paper is to adapt the strong encapsulation property imposed by ownership types, also known as *owners-as-dominators*, to a concurrent setting and use it for security enforcement. We do so by introducing an analogy between objects and channels: just as an object is owned by another object, so is a channel owned by another channel; just as an object owns other objects and limits access to them, so does a channel. As the Join calculus does not have references, the *owners-as-dominators* property is mimicked by limiting how channels are exported. The idea is that allowing a channel to be passed around as a parameter on another channel, enables other processes to refer to it. We impose access limitations by forbidding a channel to be exported outside the scope of the channel that owns it, thus allowing the kinds of restrictions analogous to those found in the object-oriented setting. A channel is however free to be passed around inside the area determined by its *owner*.

The contribution of this paper is a typed version of the Join calculus that enjoys the *owners-as-dominators* form of strong encapsulation. This property allows the calculus to define secrecy policies that are preserved both in a typed and in an untyped setting.

The paper is structured as follows. Section 2 reviews ownership and encapsulation related ideas. Section 3 introduces an ownership types system for the Join calculus and states its secrecy properties. Section 4 shows the secrecy results in the context of an untyped opponent. Section 5 presents related work and Section 6 concludes.

2 Ownership Types

We describe the notions of object ownership and of encapsulation based on it.

In a basic ownership types system [9] every object has an owner. The owner is either another object or the predefined constant *world* for objects owned by the system. Two notions are key in this system: *owner* and *representation*. The owner defines the entity to which the object belongs to. An object implicitly defines a representation consisting of the objects it owns. The aforementioned *owner* associated to an object is indicated using the function `owner`. Owners form a tree $(C, \prec:)$, where $\prec:$ is called *inside*, that represents the nesting of objects C. The tree has root *world* and is constructed according to the following scheme: $\iota \prec: \texttt{owner}(\iota)$, where ι is a newly created object, and $\texttt{owner}(\iota)$ refers to an already existing object. The following *containment invariant* determines when an object can refer to another one: ι *refers to* $\iota' \Rightarrow \iota \prec: \texttt{owner}(\iota')$. Owners and representations are encoded in a type system and the containment invariant holds for well-typed programs.

The property the containment invariant imposes on object graphs is called *owners-as-dominators* and it states that all access paths to an object from the root of a system pass through the object's owner. In effect, ownership types erect a boundary that protects an object's internal representation from external access. The idea is having a nest of boxes where there is no access from outside to inside a box, and every object defines one such box.

3 A Typed Join Calculus with Owners: J_{OT}

In this section we extend the classical Join calculus with ownership annotations. In this concurrent setting, channels will play an analogous role to objects in the object-oriented world. A channel will have another channel as *owner* and it may have a set of channels as *representation*.

3.1 Syntax

Following the notation of Pierce [21] we write $\overline{x}$ as a shorthand for $x_1, \ldots, x_n$ (similarly $\overline{T}, \overline{t}, \overline{o}$, etc), and $\overline{f\ g}$ for $f_1\ g_1 \ldots f_n\ g_n$. Let Σ be a denumerable set of channel names ranged over by: x, y, u, z. The name *world* is not in Σ since it is a constant. Let Ω be a denumerable set of owner variables ranged over by α, β, γ.

$$
\begin{array}{lllll}
P = \emptyset & D = \top & J = x\langle\overline{y}\rangle : T & T = {}^{o}_{o}\langle\!|\overline{t}|\!\rangle & o = x \\
\quad \mid x\langle\overline{u}\rangle & \quad \mid D \wedge D & \quad \mid J \mid J & & \quad \mid \mathit{world} \\
\quad \mid P \mid P & \quad \mid J \triangleright P & & t = \exists\alpha.T & \quad \mid \alpha \\
\quad \mid \mathbf{def}\ D\ \mathbf{in}\ P & & & &
\end{array}
$$

Fig. 1. Syntax, types and owners definition

Figure 1 presents the syntax of the calculus. A process P is either an empty process, a message, the parallel composition of processes or a defining process. A definition D is either an empty definition, the conjunction of definitions or a clause $J \triangleright P$, called a *reaction pattern*, which associates a guarded process P to a specific join pattern J. A join pattern J is either a typed message or the parallel composition thereof. A type $T \equiv {}^{o}_{o'}\langle\!|\overline{t}|\!\rangle$ is used to remember the *owner* (o) and *representation* (o') of a channel and to give types ($\overline{t}$) to the parameters the channel expects to send. Typing annotations explicitly establish ownership relations. The pair owner-representation identifies a single branch of the ownership tree. Define owners o as either a channel x, the constant *world* or an owner variable α. Variables are classified as *received (rv)*, *defined (dv)* and *free (fv)*, defined by structural induction in Figure 2. A *fresh name* is a name that does not appear free in a process.

The difference with the standard Join calculus [15] is that channels introduced in a message definition have a type annotation.

Existential quantification on owners is used to abstract over the representation of an expected parameter; thus a channel can send names that have different *representation* while their *owner* is the same. Existential types are used implicitly, thus there are no primitives for *packing* and *unpacking*. Existential pairs would have the form $\langle x, x\rangle$, but since the *witness* and the *value* coincide, we use the binding $x : T$, which is the implicit form of $\langle x, x\rangle : \exists\alpha.T[\alpha/x]$.

$$
\begin{aligned}
rv(x\langle\overline{y}\rangle) &= \{y_1\} \cup \cdots \cup \{y_n\} & rv(J \mid J') &= rv(J) \cup rv(J') \\
dv(x\langle\overline{y}\rangle) &= \{x\} & dv(J \mid J') &= dv(J) \cup dv(J') \\
dv(D \wedge D') &= dv(D) \cup dv(D') & fv(J \triangleright P) &= dv(J) \cup (fv(P) \setminus rv(J)) \\
fv(x\langle\overline{y}\rangle) &= \{x\} \cup \{y_1\} \cup \cdots \cup \{y_n\} & fv(\mathbf{def}\ D\ \mathbf{in}\ P) &= (fv(P) \cup fv(D)) \setminus dv(D) \\
fv({}^{o'}_{o}\langle\!|\overline{t}|\!\rangle) &= \{o\} \cup \{o'\} \cup fv(\overline{t}) & fv(\exists\alpha.T) &= fv(T) \setminus \{\alpha\}
\end{aligned}
$$

Fig. 2. Definition of received, defined and free variables (obvious cases omitted)

Example 1 (Typing and message sending). Consider the following two processes:

$$
\begin{aligned}
P = \mathbf{def}\ & o\langle a\rangle \triangleright \emptyset \\
\wedge\ \ & x\langle\rangle \mid y\langle\rangle \triangleright R \\
\mathbf{in}\ \ & o\langle x\rangle \mid o\langle y\rangle
\end{aligned}
\qquad
\begin{aligned}
P_T = \mathbf{def}\ & o\langle a\rangle : {}^{world}_{o}\langle\!|\exists\alpha.\ {}^{o}_{\alpha}\langle\!||\!\rangle|\!\rangle \triangleright \emptyset \\
\wedge\ \ & x\langle\rangle : {}^{o}_{x}\langle\!||\!\rangle \mid y\langle\rangle : {}^{o}_{y}\langle\!||\!\rangle \triangleright R \\
\mathbf{in}\ \ & o\langle x\rangle \mid o\langle y\rangle
\end{aligned}
$$

P defines three channels o, x, y and sends x and y on o. P_T is the typed version of P. R is in both cases an arbitrary process. Typing is used to enforce an ordering relation between the defined channels so that o owns x and y, thus $x, y \prec: o$. To allow both x and y to be sent on o, the type of o contains an abstraction over the parameter's representation: $\exists\alpha.\ {}^{o}_{\alpha}\langle\!||\!\rangle$. Since the type of x and y are concretizations of the abstraction obtained by replacing α with x and y respectively, $o\langle x\rangle$ and $o\langle y\rangle$ are well-typed processes.

3.2 Semantics

The expression $t[a/b]$ reads "where a is substituted for all occurrences of b in t". The substitution function $[a/b]$ is defined inductively on P, the most notable case being: $(x\langle\overline{z}\rangle : T)[u/y] \equiv x\langle\overline{z}\rangle : T[u/y]$. For the sake of simplicity we assume names that are to be substituted to be distinct from the defined ones. Substitution into types is also defined inductively and the most notable case is $(\exists\alpha.T)[o/\beta] = \exists\alpha.T$ if $\alpha = \beta$.

The matching of a join pattern and a parallel composition of messages ($\mathcal{J}$) is defined by structural induction below. The procedure identifies a substitution σ that replaces all received variables of the join pattern with the ones of the message sequence, namely if $J \stackrel{\circ}{=}_{\sigma} \mathcal{J}$ then $J\sigma \equiv \mathcal{J}$. The domains of σ_1 and σ_2 are disjoint for well-typed join patterns, so the union of σ_1 and σ_2 is well defined.

$$
\frac{}{x\langle\overline{y}\rangle : T \stackrel{\circ}{=}_{[\overline{u}/\overline{y}]} x\langle\overline{u}\rangle}
\qquad
\frac{J_1 \stackrel{\circ}{=}_{\sigma_1} \mathcal{J}_1 \qquad J_2 \stackrel{\circ}{=}_{\sigma_2} \mathcal{J}_2}{J_1 \mid J_2 \stackrel{\circ}{=}_{\sigma_1 \cup \sigma_2} \mathcal{J}_1 \mid \mathcal{J}_2}
$$

The semantics is specified as a reduced chemical abstract machine RCHAM [17]. The state of the computation is a chemical soup $\mathcal{D} \Vdash \mathcal{P}$ that consists of: $\mathcal{D}$, a set of definitions, and $\mathcal{P}$, a multiset of processes. Terms of soups will be called *molecules*.

$$\frac{}{\Vdash P_1 \mid P_2 \rightleftharpoons \Vdash P_1, P_2}\ \textit{S-PAR} \qquad \frac{dv(D) \text{ are fresh}}{\Vdash \textbf{def}\ D\ \textbf{in}\ P \rightleftharpoons D \Vdash P}\ \textit{S-DEF}$$

$$\frac{J \doteq_\sigma \mathcal{J}}{D \wedge J \triangleright P \wedge D' \Vdash \mathcal{J} \longrightarrow D \wedge J \triangleright P \wedge D' \Vdash P\sigma}\ \textit{R-BETA}$$

$$\frac{\mathcal{D}_1 \Vdash \mathcal{P}_1 \rightleftharpoons\!\!\!\!\!\Rightarrow \mathcal{D}_2 \Vdash \mathcal{P}_2 \qquad (fv(\mathcal{D}) \cup fv(\mathcal{P})) \cap (dv(\mathcal{D}_1) \setminus dv(\mathcal{D}_2) \cup dv(\mathcal{D}_2) \setminus dv(\mathcal{D}_1)) = \emptyset}{\mathcal{D}, \mathcal{D}_1 \Vdash \mathcal{P}_1, \mathcal{P} \rightleftharpoons\!\!\!\!\!\Rightarrow \mathcal{D}, \mathcal{D}_2 \Vdash \mathcal{P}_2, \mathcal{P}}\ \textit{CTX}$$

Fig. 3. Chemical rules for the RCHAM

Two kind of rules describe the evolution of the soup. Structural rules, *heating* $\rightharpoonup$, and *cooling* $\leftharpoondown$, (denoted together by $\rightleftharpoons$), are reversible and are used to rearrange terms. Reduction rules (denoted by $\longrightarrow$) represent the basic computational steps. Each reduction rule consumes a process and replaces it with another. The rules for the RCHAM are given in Figure 3. Rule *S-PAR* expresses that "|" is commutative and associative, as the soup is a multiset. Rule *S-DEF* describes the heating of a molecule that defines new names. This rule enforces that names defined in D are unique for the soup and limits the binding of such names to the process P. The side condition of this rule mimics the scope extrusion of the ν operator in π-calculus, and at the same time enforces a strict static scope for the definitions. The basic computational step is provided by rule *R-BETA*. Reduction consumes any molecule $\mathcal{J}$ that matches a given pattern J, makes a fresh copy of the guarded process P, substitutes the formal parameters in P with the corresponding actual sent names via the substitution σ, and releases this process into the soup as a new molecule. Rule *CTX* states a general evolution rule for soups. The symbol $\rightleftharpoons\!\!\!\!\!\Rightarrow$ denotes any of the above reduction steps. The side condition of *CTX* ensures that the names in the additional definitions $\mathcal{D}$ and processes $\mathcal{P}$ do not clash with those already in the soup.

3.3 The Type System

The type system needs to track the ownership relations along with the types of regular variables. These are recorded in an environment parallel to the typing one. When a channel is defined, the relationship between it and its *owner* is added to such an environment.

Define the environment Γ, which provides types for free channel variables, as $\Gamma = \emptyset \mid \Gamma, (x : T)$. Similarly, environment Δ tracks constraints on owners $\Delta = \emptyset \mid \Delta, (o \prec: o')$. The function $\mathtt{dom}(\Delta)$ is define inductively as follows: $\mathtt{dom}(\emptyset) = \emptyset$, $\mathtt{dom}(\Delta, (o \prec: o')) = \mathtt{dom}(\Delta) \cup \{o\}$.

We can now introduce the concept of ownership and how the environments of the type system keep track of it.

Definition 1 (Ownership). *A channel x is said to be* owned by *a channel o w.r.t Γ and Δ if either $(x \prec: o) \in \Delta$ or $(x : {}^{o}_{x}\langle\!|\overline{t}|\!\rangle) \in \Gamma$.*

Note that these two notions will coincide in the sense that if $x\langle\overline{u}\rangle$ is a well-typed message (via *P-msg*), then $(x \prec: o) \in \Delta$ iff $(x : {}^{o}_{x}\langle\!|\overline{t}|\!\rangle) \in \Gamma$.

A judgment $\Gamma \vdash \overline{j}$ is abbreviation for a sequence of judgments $\Gamma \vdash j_1, \ldots, \Gamma \vdash j_n$. The type system is specified via the typing judgments of Figure 4.

$\Delta \vdash \diamond$	well-formed environment Δ
$\Delta; \Gamma \vdash \diamond$	well-formed environments Γ and Δ
$\Delta; \Gamma \vdash o$	good owner o
$\Delta; \Gamma \vdash x : T$	channel x has type T
$\Delta; \Gamma \vdash T$	good type T
$\Delta; \Gamma \vdash oRo'$	o is R-related to o', where $R \in \{\prec:, \prec:^*, \prec:^+, =\}$
$\Delta; \Gamma \vdash P$	well-typed process P
$\Delta; \Gamma \vdash D :: \Delta'; \Gamma'$	well-typed definition D. Environments Δ' and Γ' contain context relations and bindings for $dv(D)$
$\Delta; \Gamma \vdash J :: \Delta'; \Gamma'$	well-typed join pattern J. Environments Δ' and Γ' contain context relations and bindings for $dv(J)$
$\Delta; \Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$	well-typed soup $\mathcal{D} \Vdash \mathcal{P}$

Fig. 4. Typing judgments for J_{OT}

Rules for owners and for the inside relation follow Clarke and Drossopolou [10]. Syntax related rules are more or less standard for the Join calculus [17], except for rules *P-msg* and *J-def*, where existential types are implicitly used.

Environment rules are standard.

Environments

$$\frac{}{\emptyset \vdash \diamond}\ \textit{Rel-n} \qquad \frac{o \notin \mathtt{dom}(\Delta) \cup \{\textit{world}\} \qquad \Delta; \emptyset \vdash o'}{\Delta, (o \prec: o') \vdash \diamond}\ \textit{Rel-c}$$

$$\frac{\Delta \vdash \diamond}{\Delta; \emptyset \vdash \diamond}\ \textit{Env-n} \qquad \frac{x \notin \mathtt{dom}(\Gamma) \qquad \Delta; \Gamma \vdash T}{\Delta; \Gamma, (x : T) \vdash \diamond}\ \textit{Env-c}$$

The following rules give the validity of owners.

Owners

$$\frac{\Delta; \Gamma \vdash \diamond \qquad o \in \mathtt{dom}(\Delta)}{\Delta; \Gamma \vdash o}\ \textit{C-rel} \qquad \frac{\Delta; \Gamma \vdash \diamond \qquad x \in \mathtt{dom}(\Gamma)}{\Delta; \Gamma \vdash x}\ \textit{C-ch}$$

The following rules capture properties of relations, based on the natural inclusions: $\prec: \ \subseteq \ \prec:^+ \ \subseteq \ \prec:^*$, $= \ \subseteq \ \prec:^*$, and $\prec:; \prec: \ \subseteq \ \prec:^+$; and equivalences: $\prec:; \prec:^* \equiv \prec:^*; \prec: \equiv \prec:^+$ and $=; R \equiv R; = \equiv R$ and $\prec:^*; \prec:^* \equiv \prec:^*$.

Inside relation

$$\frac{\Delta;\Gamma \vdash \diamond \qquad oRo' \in \Delta}{\Delta;\Gamma \vdash oRo'}\ \textit{In-rel} \qquad \frac{\Delta;\Gamma \vdash o}{\Delta;\Gamma \vdash o = o}\ \textit{In-ref} \qquad \frac{\Delta;\Gamma \vdash o}{\Delta;\Gamma \vdash o \prec: \textit{world}}\ \textit{In-wo}$$

$$\frac{\Delta;\Gamma \vdash oRo' \qquad R \subseteq R'}{\Delta;\Gamma \vdash oR'o'}\ \textit{In-weak} \qquad \frac{\Delta;\Gamma \vdash oRo' \qquad \Delta;\Gamma \vdash o'R'o''}{\Delta;\Gamma \vdash oR;R'o''}\ \textit{In-trans}$$

A type is valid (*Type-ch*) if the *representation* is *directly inside* the *owner*, and *inside* all the owners of the types that are sent. To access the *owner* of a type, use the function `own( )`, defined as: $\texttt{own}(\,{}^{o'}_{o}\langle\!|\overline{t}|\!\rangle) = o'$, $\texttt{own}(\exists\alpha.T) = \texttt{own}(T)$.

Types

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash o \prec: o' \\ \Delta;\Gamma \vdash o \prec:^{*} \texttt{own}(\overline{t}) \qquad \Delta;\Gamma \vdash \overline{t}\end{array}}{\Delta;\Gamma \vdash\ {}^{o'}_{o}\langle\!|\overline{t}|\!\rangle}\ \textit{Type-ch} \qquad \frac{\Delta,(\alpha \prec: o);\Gamma \vdash\ {}^{o}_{\alpha}\langle\!|\overline{t}|\!\rangle}{\Delta;\Gamma \vdash \exists\alpha.\ {}^{o}_{\alpha}\langle\!|\overline{t}|\!\rangle}\ \textit{Type-}\exists$$

Rule *P-msg* enforces that the channel name and the *representation* indicated in its type must coincide. The substitution $T_i[u_i/\alpha_i]$ in *P-msg* is an implicit unpacking of the witness u_i contained in the implicit existential pair $\langle u_i, u_i\rangle$ created in the eventual definition of u_i.

Processes

$$\frac{\Delta;\Gamma \vdash \diamond \qquad (x:T) \in \Gamma}{\Delta;\Gamma \vdash x : T}\ \textit{Chan}$$

$$\frac{\Delta;\Gamma \vdash \diamond}{\Delta;\Gamma \vdash \emptyset}\ \textit{P-null} \qquad \frac{\Delta;\Gamma \vdash P \qquad \Delta;\Gamma \vdash P'}{\Delta;\Gamma \vdash P \mid P'}\ \textit{P-par}$$

$$\frac{\begin{array}{l}\Delta,\Delta';\Gamma,\Gamma' \vdash P \\ \Delta,\Delta';\Gamma,\Gamma' \vdash D :: \Delta';\Gamma' \qquad \texttt{dom}(\Gamma') = dv(D) = \texttt{dom}(\Delta')\end{array}}{\Delta;\Gamma \vdash \textbf{def}\ D\ \textbf{in}\ P}\ \textit{P-def}$$

$$\frac{\Delta;\Gamma \vdash x :\ {}^{o}_{x}\langle\!|\exists\overline{\alpha}.\overline{T}|\!\rangle \qquad \Delta;\Gamma \vdash u_i : T_i[u_i/\alpha_i] \text{ for each } i \in 1..n}{\Delta;\Gamma \vdash x\langle\overline{u}\rangle}\ \textit{P-msg}$$

Rule *D-run* shows that bindings for defined channels $(\Delta_d;\Gamma_d)$ are collected and available while their scope lasts, while bindings for received channels (Δ_r,Γ_r) are collected only where they are used, namely in the started process P.

Definitions

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash D :: \Delta';\Gamma' \\ \Delta;\Gamma \vdash D' :: \Delta'';\Gamma'' \qquad dv(D) \cap dv(D') = \emptyset\end{array}}{\Delta;\Gamma \vdash D \wedge D' :: \Delta',\Delta'';\Gamma',\Gamma''}\ \textit{D-and} \qquad \frac{\Delta;\Gamma \vdash \diamond}{\Delta;\Gamma \vdash \top}\ \textit{D-top}$$

$$\frac{\begin{array}{ll} & \Delta_r = \Delta' \setminus \Delta_d \quad \Gamma_r = \Gamma' \setminus \Gamma_d \\ \Delta, \Delta_r; \Gamma, \Gamma_r \vdash J :: \Delta'; \Gamma' & \Delta_d = \{(x \prec: o) \in \Delta' \ \mid x \in dv(J)\} \\ \Delta, \Delta_r; \Gamma, \Gamma_r \vdash P & \Gamma_d = \{(x : T) \in \Gamma' \ \mid x \in dv(J)\} \end{array}}{\Delta; \Gamma \vdash J \triangleright P :: \Delta_d; \Gamma_d} \textit{ D-run}$$

An implicit packing of $\langle x, x \rangle$ and $\langle \overline{y}, \overline{y} \rangle$ is made in rule *J-def* dual to the unpacking that occurs in rule *P-msg*. Rule *J-def* also provides bindings for both the defined channel and its formal parameters.

Join patterns

$$\frac{\begin{array}{ll} \Delta; \Gamma \vdash J :: \Delta'; \Gamma' & dv(J) \cap dv(J') = \emptyset \\ \Delta; \Gamma \vdash J' :: \Delta''; \Gamma'' & rv(J) \cap rv(J') = \emptyset \end{array}}{\Delta; \Gamma \vdash J \mid J' :: \Delta', \Delta''; \Gamma', \Gamma''} \textit{ J-par}$$

$$\frac{\begin{array}{ll} \Delta; \Gamma \vdash x : T & \Delta; \Gamma \vdash y_i : T_i[y_i/\alpha_i] \text{ for each } i \in 1..n \\ T \equiv \ {}^{o}_{x}\langle\!| \exists \overline{\alpha}.\overline{T} |\!\rangle & T_i \equiv \ {}^{o_i}_{\alpha_i}\langle\!| \overline{t_i} |\!\rangle \end{array}}{\Delta; \Gamma \vdash x\langle \overline{y} \rangle : T :: (x \prec: o), (\overline{y} \prec: \overline{o}); (x : T), (\overline{y} : \overline{T}[\overline{y}/\overline{\alpha}])} \textit{ J-def}$$

Soups

$$\frac{\mathcal{P} = P_1, \ldots, P_n \qquad \Delta; \Gamma \vdash P_i \text{ for each } i = 1..n}{\Delta; \Gamma \vdash \mathcal{P}} \textit{ P-elim}$$

$$\frac{\begin{array}{ll} \Delta; \Gamma \vdash D_i :: \Delta'_i; \Gamma'_i \text{ for each } i = 1..n & \Gamma' = \Gamma'_1, \ldots, \Gamma'_n \\ dv(D_1) \cap \ldots \cap dv(D_n) = \emptyset & \Delta' = \Delta'_1, \ldots, \Delta'_n \end{array}}{\Delta; \Gamma \vdash \mathcal{D} :: \Delta'; \Gamma'} \textit{ D-elim}$$

$$\frac{\begin{array}{ll} \Delta, \Delta'; \Gamma, \Gamma' \vdash \mathcal{P} & \\ \Delta, \Delta'; \Gamma, \Gamma' \vdash \mathcal{D} :: \Delta'; \Gamma' & \mathtt{dom}(\Gamma') = dv(\mathcal{D}) = \mathtt{dom}(\Delta') \end{array}}{\Delta; \Gamma \vdash \mathcal{D} \Vdash \mathcal{P}} \textit{ Soup}$$

3.4 Soundness of the Type System

A type system for a concurrent language is correct whenever two standard theorems hold [24]. The first, subject reduction, ensures that typing is preserved by reduction. This means no typing error arise as the computation proceeds. The second one, no runtime errors, ensures that no error occurs as the computation progresses. An error may be sending a message with a different number of parameters than expected or, as is specific to our type system, breaking the *owners-as-dominators* property. Proofs can be found in a companion technical report [20].

Definition 2 (Typing environment agreement $\frown$). *Two typing environments* agree*, denoted with* $\Delta; \Gamma \frown \Delta'; \Gamma'$*, if the variables they have in common have the same type.*

Theorem 1 (Subject reduction for J_{OT}). *One step chemical reductions preserve typings. If* $\Delta; \Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$ *and* $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons \mathcal{D}' \Vdash \mathcal{P}'$*, then there exists* $\Delta'; \Gamma'$ *such that* $\Delta; \Gamma \frown \Delta'; \Gamma'$ *and* $\Delta'; \Gamma' \vdash \mathcal{D}' \Vdash \mathcal{P}'$*.*

Definition 3 (Runtime errors). *Consider a soup* $\mathcal{D} \Vdash \mathcal{P}$*. Say that a runtime error occurs if any of these kind of messages occurs in the processes set* $\mathcal{P}$*:*

- *a message* $x\langle\overline{y}\rangle$ *that is not defined in* $\mathcal{D}$*, i.e. no join pattern* J *in* $\mathcal{D}$ *has* x *in its defined variables;*
- *a message* $x\langle\overline{y}\rangle$ *that is defined in* $\mathcal{D}$ *but with different arity (e.g. the defined channel* x *wants four parameters while we call it with three);*
- *a message* $x\langle\overline{y}\rangle$ *where* x *is not* inside *some of its arguments' owners, i.e. there is a* y_i *owned by* o *such that* $x \not\prec:^* o$

Reduction of a well-typed soup never results in a runtime error.

Theorem 2 (No runtime errors for J_{OT}). *If* $\Delta; \Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$ *and* $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons^* \mathcal{D}' \Vdash \mathcal{P}'$*, then no runtime error occur in* $\mathcal{D}' \Vdash \mathcal{P}'$*.*

The following statement is a translation of the *owners-as-dominators* property ownership types enforce in the object-oriented setting. No reference from outside the ownership boundaries to an inner channel exists.

Theorem 3 (Owners as dominators). *A channel* y *owned by* o *may be sent over a channel* x *only if* x *is* transitively inside o*:* $\Delta; \Gamma \vdash x\langle y\rangle \Rightarrow \Delta; \Gamma \vdash x \prec:^* o$

Corollary 1 is a restatement of Theorem 3 from a secrecy perspective. The point of view of Theorem 3 highlights what can be sent on a channel. Dually, the perspective of Corollary 1 points out what cannot be sent on a channel.

Corollary 1 (Secrecy for J_{OT}). *Consider a well-typed soup* $\mathcal{D} \Vdash \mathcal{P}$ *that defines a channel* y *owned by* o*. However the soup evolves,* y *is not accessible from channels whose owner is not* transitively inside o*.*

Example 2 (Secrecy with a typed opponent). Consider the typed process P of Example 1. Suppose P is a private subsystem of a larger system O. Process P defines two secrets, namely x and y, which are intended to remain private to P. This privacy policy can be violated if, for example, a subsystem R can, after a sequence of reduction steps, send $l\langle x\rangle$, where l is a channel known to O. To typecheck the definition of l, O should know the name o. Fortunately o is not in the environment O uses to typecheck since o is defined in P. As the following proof tree shows, l cannot be typed in order to send channels owned by o. This means that the secrecy of x is preserved.

$$\dfrac{\dfrac{\dots \quad \dfrac{\text{Impossible since } o \text{ is unknown}}{\Delta; \Gamma \vdash l \prec:^* o} \quad \dots}{\Delta; \Gamma \vdash l : \ {}^{world}_{l}\langle\!|\exists\alpha.\ {}^{o}_{\alpha}\langle\!|\,|\!\rangle|\!\rangle \quad \dots}\ \textit{Type-ch}}{\Delta; \Gamma \vdash l\langle a\rangle : \ {}^{world}_{l}\langle\!|\exists\alpha.\ {}^{o}_{\alpha}\langle\!|\,|\!\rangle|\!\rangle :: \ \dots}\ \textit{Chan}$$

Example 3 (Code invariant definition). Consider the following process.

$$
\begin{aligned}
B = \mathbf{def}\ mb\langle pb\rangle : T_m \triangleright \mathbf{def}\ & guard\langle\rangle :\ {}^{world}_{guard}\llparenthesis\rrparenthesis \triangleright \emptyset \\
\wedge\quad & empty\langle\rangle :\ {}^{guard}_{empty}\llparenthesis\rrparenthesis \mid put\langle d\rangle : T_p \triangleright full\langle d\rangle \\
\wedge\quad & full\langle c\rangle :\ {}^{guard}_{full}\llparenthesis T_c\rrparenthesis \mid get\langle r\rangle : T_g \triangleright empty\langle\rangle \mid r\langle c\rangle \\
\mathbf{in}\quad & empty\langle\rangle \mid pb\langle put, get\rangle \\
\mathbf{in}\ \ \ldots &
\end{aligned}
$$

Process B is a one-place buffer where typing enforces the invariant: the internal representation of the buffer cannot be accessed except via *put* and *get*. The buffer is spawned by sending a channel *pb* over *mb*. Primitives for buffer usage, *put* and *get*, will be returned to the buffer creator on channel *pb*. Since *empty* and *full* are owned by channel *guard*, there is no possibility for them to be used outside the process boundaries, for the same reasons as Example 2. Channel *guard* and type annotations are the only additions to an untyped one-place buffer [16] which are needed to enforce the invariant. We do not fully specify all types involved in B, focussing the attention only on the types needed to define the invariant.

4 Secrecy in the Context of an Untyped Opponent

When computation is run on a collection of machines that we cannot have access to or simply rely on, we expect our programs to interact with code that possibly does not conform to our type system, which could result in a malicious attempt to access the secrets of our programs. The system must be strong enough to prevent such attacks.

We follow the approach of Cardelli et. al. [8] and formalize the idea that a typed process in J_{OT} can keep a secret from any opponent it interacts with, be it well- or ill-typed. The story reads as follows:

- consider a well-typed J_{OT} soup $\mathcal{D} \Vdash \mathcal{P}$ that defines a secret x owned by o;
- consider an untyped soup $\mathcal{D}' \Vdash \mathcal{P}'$ that knows neither o nor x a priori;
- erase typing annotation from $\mathcal{D} \Vdash \mathcal{P}$ and combine its molecules with those of $\mathcal{D}' \Vdash \mathcal{P}'$, the result is an untyped soup $\mathcal{D}'' \Vdash \mathcal{P}''$;
- then, in any way the untyped soup $\mathcal{D}'' \Vdash \mathcal{P}''$ can evolve, it does not leak the secret x.

We work with soups instead of processes as a process P is trivially a soup $\emptyset \Vdash P$.

The notation J_{UN} will be used to refer to the untyped Join calculus. The syntax for J_{UN} [15,16] is analogous to the one presented in Figure 1, except that typing annotations are dropped from channel definitions. The semantics of J_{UN} follows that of Figure 3 ignoring types.

In the next section a common framework where trusted and untrusted channels coexist is introduced. We give a precise definition of when an untyped process preserves the secrecy of a channel x from an opponent. The most important result presented in the paper is that the untyped process obtained by erasing type annotations from a well-typed J_{OT} process preserves the secrecy of x from *any* opponent it interacts with.

4.1 An Auxiliary Type System: J_{AU}

Proving the secrecy theorem in an untyped opponent context is based on an auxiliary type system. The auxiliary type system partitions the set of channels into untrusted and trusted ones with regards to one specific channel: the secret. The idea is that untrusted channels do not have access to the secret, trusted channels on the other side can handle such secret. Typing enforces such a distinction. Types have syntax: $T = Un \mid In\langle\!|\overline{T}|\!\rangle$. Untrusted channels have type Un. Trusted channels have type $In\langle\!|\overline{T}|\!\rangle$, where each T_i is either trusted or untrusted. The property enforced by the type system is that trusted channels cannot be sent over untrusted ones. Hence an opponent knowing only untrusted (Un) names cannot receive a trusted (In) one.

Define Γ as a list of bindings of variables to types: $\Gamma = \emptyset \mid \Gamma, (x : T)$. The typing judgments that define the type system are analogous to those in Figure 4, except that owners and environment Δ are dropped. Typing rules for message sending (*P-msg*) and channel definition (*J-def*) need to be replaced by rules *P-mun* and *P-min* and by rules *J-dun* and *J-din* below, respectively. Judgments for the J_{AU} type system are made against untyped processes, as defined channels in rules *J-dun* and *J-din* do not have typing annotation.

$$\frac{\Gamma \vdash x : Un \qquad \Gamma \vdash \overline{y} : Un}{\Gamma \vdash x\langle\overline{y}\rangle :: (x : Un), (\overline{y} : Un)}\ \textit{J-dun} \qquad \frac{\Gamma \vdash x :\ In\langle\!|\overline{T}|\!\rangle \qquad \Gamma \vdash \overline{y} : \overline{T}}{\Gamma \vdash x\langle\overline{y}\rangle :: (x :\ In\langle\!|\overline{T}|\!\rangle), (\overline{y} : \overline{T})}\ \textit{J-din}$$

Rules *P-mun* and *P-min* state that untrusted channels cannot send anything but untrusted ones while trusted channels can mention both kinds.

$$\frac{\Gamma \vdash x : Un \qquad \Gamma \vdash \overline{y} : Un}{\Gamma \vdash x\langle\overline{y}\rangle}\ \textit{P-mun} \qquad \frac{\Gamma \vdash x :\ In\langle\!|\overline{T}|\!\rangle \qquad \Gamma \vdash \overline{y} : \overline{T}}{\Gamma \vdash x\langle\overline{y}\rangle}\ \textit{P-min}$$

The auxiliary type system enjoys subject reduction, as Theorem 4 shows.

Theorem 4 (Subject reduction). *If $\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$ and $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons \mathcal{D}' \Vdash \mathcal{P}'$ then there exists a Γ' such that $\Gamma' \vdash \mathcal{D}' \Vdash \mathcal{P}'$ and $\Gamma \frown \Gamma'$.*

4.2 The Common Framework

The auxiliary type system is used as a meeting point for untyped and typed soups. It serves as a common framework for testing secrecy. Assigning trusted and untrusted types to channels makes J_{AU} a suitable system to reason about coexisting trusted and untrusted processes. From now on the notation $\vdash_{OT}$ will indicate a judgment in the J_{OT} system, while $\vdash_{AU}$ will indicate a judgment in the J_{AU} one. Firstly we must be able to typecheck any untyped opponent. The way of doing it is provided by Proposition 1.

Proposition 1. *For all untyped soups $\mathcal{D} \Vdash \mathcal{P}$, if $fv(\mathcal{D} \Vdash \mathcal{P}) = \{x_1, \ldots, x_n\}$, then $x_1 : Un, \ldots, x_n : Un \vdash_{AU} \mathcal{D} \Vdash \mathcal{P}$.*

Secondly we need a way to add elements from J_{OT} to the common framework. This is done by erasing all type annotations of such elements and by mapping typing environments of J_{OT} to J_{AU} ones. Type annotations can be erased via an erasure function (`erase( )`), which is defined by structural induction on P, the most notable case being $\mathtt{erase}(x\langle\overline{y}\rangle{:}T) = x\langle\overline{y}\rangle$. Translating the typechecking environment associated with a J_{OT} process to an J_{AU} environment is done via the mapping function $[\![\ ]\!]_o$ presented below.

$$[\![\Delta;\Gamma]\!]_o = [\![\Gamma]\!]_o \qquad [\![\emptyset]\!]_o = \emptyset \qquad [\![\Gamma,(x:T)]\!]_o = [\![\Gamma]\!]_o,(x:[\![T]\!]_o)$$

The environment Δ is dropped since it is not required. Γ is translated in an environment that contains bindings for the auxiliary type system. For any channel o that identifies a secret, map J_{OT} types to J_{AU} ones as follows.

$$[\![\ {}^{o'}_{o}\langle\!|\overline{t}|\!\rangle]\!]_o = \begin{cases} In\langle\!|[\![\overline{t}]\!]_o|\!\rangle & \text{if } o \in fv(\ {}^{o'}_{o}\langle\!|\overline{t}|\!\rangle) \\ Un & \text{else} \end{cases} \qquad [\![\exists\alpha.T]\!]_o = [\![T]\!]_o$$

Types that do not mention the secret o are translated as untrusted: Un. All others preserve their structure and are translated as trusted: $In\langle\!|\overline{T}|\!\rangle$. Note that trusted channels are the ones *transitively inside* the given channel o, while untrusted ones are those that are outside the ownership boundaries o defines.

Proposition 2 shows how to combine the erasure and the mapping functions to obtain a well-typed soup in J_{AU} starting from a well-typed one in J_{OT}. The secret o is any channel, be it a free name in the environment, a bound name in the soup or a fresh name.

Proposition 2. *If $\Delta;\Gamma \vdash_{OT} \mathcal{D} \Vdash \mathcal{P}$ then, for any o, $[\![\Delta;\Gamma]\!]_o \vdash_{AU} \mathtt{erase}(\mathcal{D} \Vdash \mathcal{P})$.*

Now we have all the machinery to bring together trusted and untrusted processes in a common framework and show that the former ones cannot leak secrets to the latter ones.

4.3 Secrecy Theorem

Before stating the secrecy results in the untyped setting we need to introduce some related concepts.

Soup combination allows us to merge the molecules of two soups if the type-checking environments agree. The agreement implies that names that are common to the two soups have the same degree of trust, there is no name that is considered trusted in a soup but is untrusted in the other one.

Definition 4 (Soups combination). *Consider two untyped soups $\mathcal{D} \Vdash \mathcal{P}$ and $\mathcal{D}' \Vdash \mathcal{P}'$. Suppose they are well-typed in J_{AU}, so there exist Γ, Γ' such that $\Gamma \vdash_{AU} \mathcal{D} \Vdash \mathcal{P}$ and $\Gamma \vdash_{AU} \mathcal{D}' \Vdash \mathcal{P}'$. If $\Gamma \frown \Gamma'$, the molecules of the two soups can be combined into a single one: $\mathcal{D},\mathcal{D}' \Vdash \mathcal{P},\mathcal{P}'$.*

The definition of secret leakage we use is inspired by Abadi [1] for the untyped spi calculus [3]. The underlying idea is attributed to Dolev and Yao [13]: a name is kept secret from an opponent if after no series of interactions is the name transmitted to the opponent.

Definition 5 (Secret leakage). *A soup* $\mathcal{D} \Vdash \mathcal{P}$ *leaks secrets whenever* $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons^* \mathcal{D}' \Vdash \mathcal{P}'$ *and in* $\mathcal{P}'$ *there is an emission of a trusted channel on an untrusted one.*

The following proposition is the crux of the proof of Theorem 5: an opponent who knows only untrusted names cannot learn any trusted one.

Proposition 3. *Suppose* $y_1 : Un, \ldots, y_n : Un, o : T', x : T \vdash_{AU} \mathcal{D} \Vdash \mathcal{P}$*, where* $T, T' \neq Un$*. Then, the untyped soup* $\mathcal{D} \Vdash \mathcal{P}$ *does not leak secrets.*

Theorem 5 shows that an opponent, which does not know any trusted channel a priori, does not learn any trusted name by interacting with a well-typed J_{OT} soup whose type annotations have been erased. Definition 4 allows the two soups to have names in common and to communicate as long as the shared channels have the same degree of trust.

Theorem 5 (Secrecy in unsafe context). *Consider a well-typed* J_{OT} *soup* $\mathcal{D} \Vdash \mathcal{P}$ *and an untyped soup* $\mathcal{D}' \Vdash \mathcal{P}'$ *that does not know a priori any trusted name of the typed one. Let* $\mathcal{D}'' \Vdash \mathcal{P}''$ *be the combination of* $\texttt{erase}(\mathcal{D} \Vdash \mathcal{P})$ *and* $\mathcal{D}' \Vdash \mathcal{P}'$*. Then* $\mathcal{D}'' \Vdash \mathcal{P}''$ *does not leak secrets.*

5 Related Work

Ownership types have had several applications, mostly unrelated to security. They have been combined with effects for reasoning about programs [10]. Ownership types were used to detect data races and deadlocks [5], to allow safe persistent storage of objects [6] and to allow safe region-based memory management in real time computation [7]. Preventing uncontrolled accesses to EJBs [12] was obtained using a notion of containment similar to ownership types. Another similar but more lightweight idea is that of confined types [23], which provide a per-package notion of object encapsulation. To our knowledge ownership types have never been used before as a direct way to enforce secrecy policies. Furthermore, this is the first attempt to translate them from the object-oriented setting to a process calculus. A translation to the π-calculus appears to be straightforward. Ownership types have also been encoded into System F [19].

As stated before, types are not present in standard Join calculus [15]. Absence of type annotations is a difference also with the typed Join calculus [17], where typing is implicit and polymorphic. Since it loosens the constraints imposed by typing, polymorphism is not used in the current work.

In process algebra, security has been achieved via encryption both for the Join calculus [2] and the π-calculus [3]. The cited works require encryption and decryption primitives while the presented work does not. The control flow analysis for the π-calculus [4] testifies that a process provides a certain encapsulation property, on the other side our type system allows the programmer to specify such a property.

Cardelli et al.'s Groups [8] is the closest work to the results presented here. Groups were created for the π-calculus, a translation in the Join calculus seems

however straightforward. Groups are created with a specific operator: νG that mimics the scope extrusion principle of channel definition νx. Channels have type $T = G[\overline{T}]$ which indicates the group G a channel belongs to, and its parameters' types $\overline{T}$. $\overline{T}$ can only mention groups which are in scope when T is defined. The notion of secrecy of Groups is thus based on scoping and there are policies that cannot be expressed using Groups, as Example 4 shows.

Example 4 (Expressiveness). Consider a process that is supposed to match the password provided from a client, sent on channel c, and the corresponding database entry, sent on channel d. If the client is able to access any entry, it can impersonate any user. Database entries must be protected from the client but they must be accessible from the database. P_G and P_O specify such a policy using Groups or using ownership types respectively.

$$\begin{aligned} P_G = {} & \nu G \\ & \mathbf{def}\ d\langle\overline{a}\rangle : T_d \mid c\langle\overline{b}\rangle : T_c \triangleright R \\ & \mathbf{in}\ \ \ldots \end{aligned} \qquad \begin{aligned} P_O = {} & \mathbf{def}\ d\langle\overline{a}\rangle : {}^{world}_{d}\langle\!|\,|\!\rangle \mid c\langle\overline{b}\rangle : {}^{world}_{c}\langle\!|\,|\!\rangle \triangleright R \\ & \mathbf{in}\ \ \ldots \end{aligned}$$

To design the policy using Groups we can create a group G that is supposed to protect the database entries. In order for the database to access the entries via channel d, the declaration of G should appear before the join pattern that starts the password checking process, since such join pattern defines d. This would allow the type of the client, T_c, to mention G. The client could then access the data hidden within G via the channel c, violating the intended security policy. The designed policy can be expressed using ownership types as P_O shows. Channel c would not be able to access the entries owned by d because $c \not\prec:^* d$. On the other side d would still be able to access the database entries since d owns them.

6 Conclusion

This paper shows how to prevent the leakage of secrets in a mobile setting using ownership types. We provide a type system that encodes ownership types concepts in the Join calculus. The type system enforces the *owners-as-dominators* property and the consequent strong form of encapsulation. A typed process protects its secrets against malicious or accidental leakage. Secrecy is also preserved even in the context of an untyped opponent.

A formal comparison between ownership types and Cardelli et al.'s Groups remains to be done. Additionally, in order to allow a JOCaml implementation of ownership, polymorphism and type inference need to be investigated.

References

1. Abadi, M.: Security protocols and specifications. In: Thomas, W. (ed.) FOSSACS 1999. LNCS, vol. 1578, pp. 1–13. Springer, Heidelberg (1999)
2. Abadi, M., Fournet, C., Gonthier, G.: Secure implementation of channel abstractions. Inf. Comput. 174, 37–83 (2002)
3. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Inf. Comput. 148(1), 1–70 (1999)

4. Bodei, C., Degano, P., Nielson, F., Riis Nielson, H.: Control flow analysis for the π-calculus. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 84–98. Springer, Heidelberg (1998)
5. Boyapati, C., Lee, R., Rinard, M.C.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA 2002, pp. 211–230 (2002)
6. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL 2003, pp. 213–223 (2003)
7. Boyapati, C., Salcianu, A., Beebee, W.S., Rinard, M.C.: Ownership types for safe region-based memory management in real-time java. In: PLDI 2003, pp. 324–337 (2003)
8. Cardelli, L., Ghelli, G., Gordon, A.D.: Secrecy and group creation. Inf. Comput. 196(2), 127–155 (2005)
9. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales (July 2001)
10. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA 2002, pp. 292–310 (2002)
11. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA 1998, pp. 48–64 (1998)
12. Clarke, D., Richmond, M., Noble, J.: Saving the world from bad beans: deployment-time confinement checking. In: OOPSLA 2003, pp. 374–387. ACM, New York (2003)
13. Dolev, D., Yao, A.C.: On the security of public key protocols. In: SFCS 1981, pp. 350–357. IEEE Computer Society Press, Washington, DC, USA (1981)
14. Fournet, C., Fessant, F.L., Maranget, L., Schmitt, A.: JoCaml: a Language for Concurrent Distributed and Mobile Programming. In: Proceedings of the 4th Summer School on Advanced Functional Programming. LNCS, pp. 129–158. Springer, Heidelberg (2002)
15. Fournet, C., Gonthier, G.: The reflexive CHAM and the Join-calculus. In: POPL 1996, pp. 372–385 (1996)
16. Fournet, C., Gonthier, G.: The join calculus: A language for distributed mobile programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002)
17. Fournet, C., Laneve, C., Maranget, L., Rémy, D.: Implicit typing à la ML for the Join-calculus. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 196–212. Springer, Heidelberg (1997)
18. Haller, P., Van Cutsem, T.: Implementing joins using extensible pattern matching. In: Wang, A.H., Tennenholtz, M. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 135–152. Springer, Heidelberg (2008)
19. Krishnaswami, N.R., Aldrich, J.: Permission-based ownership: encapsulating state in higher-order typed languages. In: PLDI 2005, pp. 96–106 (2005)
20. Patrignani, M., Clarke, D., Sangiorgi, D.: Ownership types for the Join calculus. CW Reports CW603, Dept. of Computer Science, , K.U.Leuven (March 2011)
21. Pierce, B.: Types and Programming Languages. MIT Press, Cambridge (2002)
22. Plociniczak, H., Eisenbach, S.: JErlang: Erlang with joins. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 61–75. Springer, Heidelberg (2010)
23. Vitek, J., Bokowski, B.: Confined types. In: OOPSLA 1999, Denver, Colorado, United States, pp. 82–96. ACM Press, New York (1999)
24. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115, 38–94 (1994)

Contracts for Multi-instance UML Activities

Vidar Slåtten and Peter Herrmann

Norwegian University of Science and Technology (NTNU),
Department of Telematics, N-7491 Trondheim, Norway
{vidarsl,herrmann}@item.ntnu.no

Abstract. We present a novel way of encapsulating UML activities using interface contracts, which allows to verify functional properties that depend on the synchronization of parallel instances of software components. Encapsulated UML activities can be reused together with their verification results in SPACE, a model-driven engineering method for reactive systems. Such compositional verification significantly improves the scalability of the method. Employing a small example of a load balancing system, we explain the semantics of the contracts using the temporal logic TLA. Thereafter, we propose a more easily comprehensible graphical notation and clarify that the contracts are able to express the variants of multiplicity that we can encounter using UML activities. Finally, we give the results of verifying some properties of the example system using the TLC model checker.

1 Introduction

A key to efficient software engineering is the reuse of existing software components that ideally are independently developed and marketed as commercial off-the-shelf (COTS) products. To enable a seamless combination of the components, one needs meaningful descriptions of the component interfaces that specify properties to be kept by both the components themselves and their environments. Due to the tight interaction with their environment, this requirement holds in particular for reactive systems [6]. To support the reuse of software components for reactive, distributed software systems, we use collaborative building blocks described by multi-partition UML activities, each augmented with an interface contract in the form of a UML state machine [12,13]. We call these contracts External State Machines (ESMs). The contracts not only enable reuse of problem solutions, but also make for a reuse of verification effort as the user can verify a composed system using only the contracts, which in turn have been verified to be correct abstractions of the underlying solutions. This compositional approach helps to reduce the complexity and state space of the system models significantly [12]. The ESMs also help another problem with reuse: It may not always be straightforward to look at a reusable activity and see what it does and how to compose it correctly with the rest of the system. As the ESM only describes behaviour visible to the outside of the block, it aids both these tasks.

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 304–318, 2011.
© IFIP International Federation for Information Processing 2011

ESMs constitute what Beugnard et al. [2] call synchronization contracts, meaning that they can specify the effect of interleaving operations on a component, not just sequential operations. However, up until now we have been limited to collaborations in which only one instance of each type participates, as the contracts could not support collaborations featuring multiple component instances of the same type. If, for instance, a client request may be routed to one of several servers, we could not express an interface behaviour that permits server S to receive the request only if none of the other servers have received it. In systems that employ load balancing or fault-tolerance mechanisms, however, to specify and guarantee this kind of behaviour is crucial. Thus, compared with the ESMs, we need additional concepts and notation for behavioural interfaces.

Any extension of ESMs should ideally keep a key characteristic of SPACE [14]: The underlying formalism is hidden to the user. According to Rushby [21], this is a key quality of practical development methods. SPACE relies on automatic model checking to verify system models. To mitigate the problem of state-space explosion, we limit our scope to verifying properties dependent only on the control flow of the system designs. While we could very well include data in the model, the model checker would not be able to verify properties dependent on data for realistic systems, as the state space would grow exponentially with every data element we include. Nevertheless, as pointed out in [13,14], also the model checking of just control flows is of great practical help, for single-instance activities.

The ESMs are basically Finite State Machines (FSMs) with special annotations of their transitions. To model multiple entities in a suitable way, we use Extended Finite State Machines (EFSMs) [3] instead, which allow to refer to the indexes of instances in the form of auxiliary variables. The semantics of these Extended ESMs (EESMs) is formalized using the Temporal Logic of Actions, TLA [15]. To relieve the software engineer from too much formalism, we further present a more compact graphical notation in the form of UML state machines where statements closer to programming languages are used to describe variable operations.

The next section discusses related work on component contracts, particularly work using UML. Our load balancing example system is presented in Sect. 3. In Sect. 4, we formalize the EESM semantics for many-to-many activities in TLA, and present the graphical notation. We give EESMs for the other types of activities, one-to-one and one-to-many, in Sect. 5. Some results, in particular about the effects on model checking, as well as future work is discussed in Sect. 6, where we also conclude.

2 Related Work

There are several other works that define a formal semantics for UML activities [4,5,23], but neither of them include contracts for use in hierarchical activities. Eshuis [4] explicitly argues to leave this concept out, as any hierarchical activity can be represented as a flat one. However, this results in a much bigger state space.

As Beugnard et al. [2] point out, we can only expect software components to be reused for mission-critical systems if they come with clear instructions on how to be correctly reused and what guarantees they give under those conditions. UML has the concept of Protocol State Machines [19] to describe the legal communication on a port of a component. Mencl [18] identifies several shortcomings of these, for example that they do not allow to express dependencies between events on a provided and required interface, nor nesting or interleaving method calls. To remedy this, he proposes Port State Machines, where method calls are split into atomic request and response events. These Port State Machines are restricted to pure control flow, as transition guards are not supported. Bauer and Hennicker [1] describe their protocol style as a hybrid of control flow and data state. However, they also cannot express the dependency between provided and required interfaces, and they currently lack verification support for whether two components fit together.

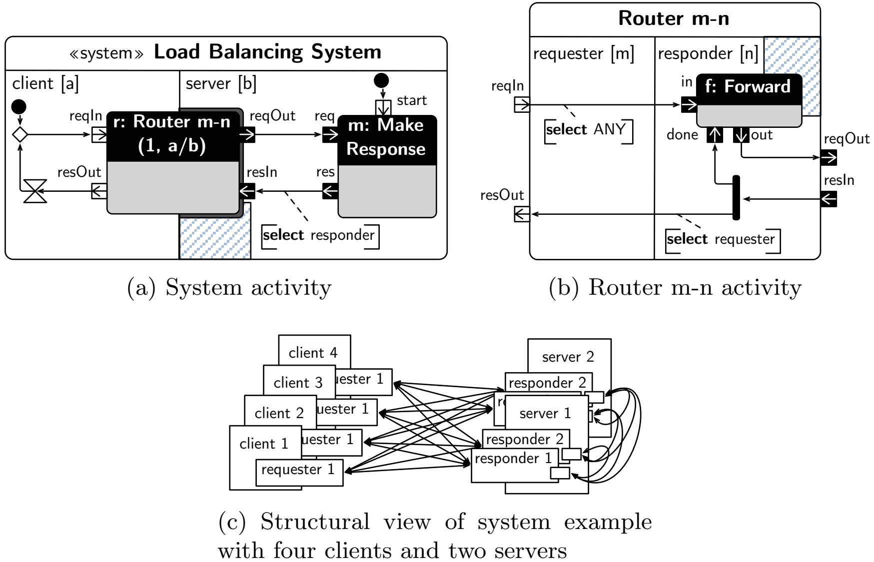
The ESMs have similar properties to Port State Machines in that all interface events are atomic, i.e., an operation is split into a request and response event, to allow for expressing nesting and interleaving of operations. They also essentially combine provided and required interfaces in the same contract, hence allowing to express dependencies between them. The EESMs combine this with data state to allow for compact representations of parametrized components and collaborations.

Sanders et al. [22] present what they call semantic interfaces of service components, both described as finite state machines. These interfaces support both finding complementary components and implementations of an interface, hence also compositional verification. While they provide external interfaces for each local component and then asynchronously couple these to other, possible remote, components, our activity contracts act as internal interfaces that can be synchronously coupled with other local behaviour in the fashion of activity diagrams.

Our approach differs from all the above in that it permits the encapsulation of both local components and distributed collaborations between components, described by single-partition or multi-partition activities respectively. Further, our extended interfaces allow to constrain behaviour based on the state of parallel component instances, giving a powerful abstraction of distributed collaborations.

3 A Load Balancing Client – Server Example

In SPACE, the main units of composition are collaborative building blocks in the form of UML activities that can automatically be transformed to executable code [14]. A system specification consists of a special system activity as the outermost block. This system activity can contain any number of inner activities, referenced by call behaviour actions, as well as glue logic between them. Figure 1(a) shows the system activity for our example, a set of clients that can send requests via a Router m-n block to a set of servers. Thus, Router m-n is an inner block, its activity depicted in Fig. 1(b). Each activity partition is named in



(a) System activity

(b) Router m-n activity

(c) Structural view of system example with four clients and two servers

Fig. 1. System example: A load balancing client – server system

the upper left corner and the bracketed parameter behind the name, a for client and b for server, denotes the number of component instances of this type. While each client only sees one router, each server partition has a/b instances of the router block, as denoted by parameters *(1, a/b)* after its name and the shade around the server side of the block. This is also illustrated in Fig. 1(c), where we see that each client component only has a single requester sub-component, whereas each server has two responders. Note that the structural view is completely redundant and only serves to illustrate the information in the activities and the EESMs introduced below. The diagonally striped area inside the server partition represents other components of the same type, i.e., other servers. This gives a visual indication that the Router m-n block, in addition to collaborating with clients, also collaborates with other server instances. Each server also makes use of an inner activity called Make Response, which turns requests into responses. We have omitted it in Fig. 1(c), as it is completely local.

It is the job of the Router m-n block in Fig. 1(b) to balance the load so that a single server does not have to serve requests from all clients at the same time. All requesters can send requests to all responders, as illustrated by the full mesh connectivity in Fig. 1(c). Each responder uses the Forward block to forward requests to other responders, if it is currently busy itself. In the structural view, the component of the Forward block is shown as a small nameless rectangle on each responder that can communicate with every other such component. It is important to note that the Router m-n activity encapsulates asynchronous communication between components, while the synchronous interaction between an outer and an

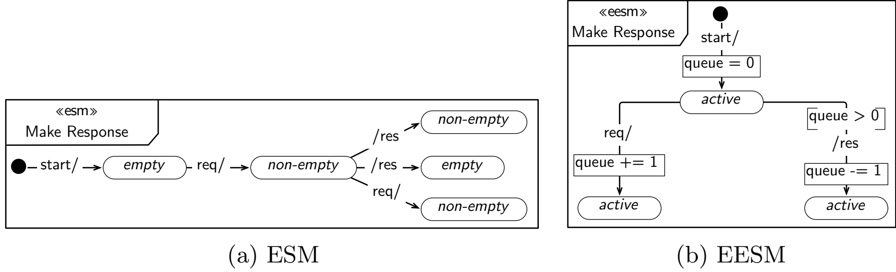
inner activity takes place via pins[1] linking flows of the two activities. For instance, the block Router m-n is linked with the system activity by the pins reqIn, reqOut, resIn and resOut. Here, reqIn is a start pin initiating an activity instance (really, the corresponding requester instance), whereas resOut is a stop pin terminating the instance. The remaining pins with black background are streaming pins for interacting with active instances.

The semantics of UML activities is similar to Petri-nets, where the state is encoded as tokens resting in token places and then moving along the directed edges to perform a state transition [19]. In our approach, all behaviour takes place in run-to-completion steps [9]. That is, all token movements are triggered by either receptions of external events (for instance, tokens resting between partitions) or expiration of local timers, and the tokens move until such an event is needed to make progress again.

Initial nodes start the behaviour of each system-level partition instance. They are fired one by one, but we make the assumption that no token will enter a partition before its initial node has fired. The initial node of the server partition can fire at any time, releasing a token into the start pin of the Make Response block. In the client partition, the token emitted from the initial node will enter the Router m-n block via the reqIn pin. Afterwards, it will be forwarded to a server instance and leave the block via pin reqOut, to enter pin req of Make Response. The Make Response block will eventually emit a token via its res pin, and the server partition will choose one of the Router m-n instances to receive it via its resIn pin, as denoted by the select statement [10]. A select statement takes some data carried by the token and uses it to select either among various instances of a local inner block or of remote partitions. The Router m-n block will eventually emit a token through its resOut pin in one of the client partitions, which will follow the edge to the timer on the client side, where the step will stop. Later, the timer will expire, causing it to emit the token so that the client can perform another request. In this example, the timer is simply an abstraction for whatever the client might be doing between receiving a response and sending a new request. The behaviour of the Router m-n block is described in Sect. 4.

When we compose a system by creating new activities and reusing existing ones, we want to be able to verify properties of it. Given that SPACE models have a formal semantics [11,12], we can express them as temporal logic specifications and use a model checker to verify properties automatically. To mitigate the problem of the state space growing exponentially with every stateful element, each activity is abstracted by an External State Machine (ESM), which is a description of the possible ordering of events visible on the activity pins. This allows us to do compositional verification of the system specification: We first verify that the activity and its ESM are consistent, then we only use that ESM when verifying properties for a higher-level or system-level block. Note that verifying the consistency of an ESM and its activity cannot be done automatically

[1] They are really activity parameter nodes when seen from the inner activity itself, but are called pins when an activity is reused via a call behaviour action. We use *pins* to denote both, to keep it simple.



(a) ESM (b) EESM

Fig. 2. Contracts for the Make Response block

for all blocks, as some will constrain their control-flow behaviour according to data stored in the activity tokens. In this case, the model allows all possible behaviours, and the potentially false positives reported by a model checker (that is, error traces that cannot occur in the real system) can be inspected manually, reducing the verification task. A select statement is an example where data constrains the destination of a token.

Figure 2(a) shows the ESM of the local building block Make Response. As discussed in the introduction, the ESM notation has the same expressive power as a finite state machine or Mealy machine [17], to be precise. The transition labels correspond to pins, and the slash character separates the transition trigger event, as seen from the perspective of the block, from the effect. Hence, *start/* means that the transition is triggered from outside by a token entering pin start and that no tokens will pass any other pins in the same step. The ESM shows that a response is not output until a request has been sent in, and that this will not happen in the same step. Once the non-empty state is reached, further requests may enter and responses may be emitted from the block. Just looking at the ESM, however, we cannot know exactly how many responses will be sent out, as there is no way of knowing if a /res event has caused a transition to the empty state, or if the ESM is still in the non-empty state. This is because there is no way to track the actual number of buffered requests.[2] When verifying properties of a system, such information may sometimes be necessary. For example, if this block was used in a system that sends three requests, we would like to infer from the ESM that exactly three responses can be emitted back out.

4 Contracts for Multi-instance Activities

To support multi-instance activities, we extend the ESMs to include transition guards, variables, arithmetic and predicate logic. Hence, they are now formally EFSMs [3]. This enables us to specify event constraints that relate to the state of parallel component instances. Moreover, this increases the general expressiveness, so that we are able to better handle the case of the Make Response block.

[2] An ESM could of course track the number of buffered requests in explicit states, up to some finite number, but it would quickly grow syntactically very large.

Figure 2(b) shows the EESM of the Make Response block. We have here added a variable, *queue*, that tracks the number of requests buffered. To constrain the behaviour based on the queue size, as well as update it, this EESM also contains transition guards in square brackets and variable operations in lined boxes.

Figure 1(b) shows the internal activity of the Router m-n block. A request enters through the reqIn pin of the requester partition and is forwarded to a responder partition. The select statement, along with the fact that there are n responder partitions, tells us that a requester expects to have a choice of responders to communicate with, when forwarding the token. When a token crosses a partition border, the corresponding activity step ends, as remote communication is asynchronous. When the token is received by the responder partition, it is passed on to an inner block, Forward. This block may emit the token directly through its out pin to be passed on through the reqOut pin of Router m-n, or it may forward the token to another responder, if this one is busy already serving a request.[3] When a response token is received via the resIn pin, it is duplicated in the fork node and passed both to pin done and the channel for the originating requester partition.

We now describe the EESM of block Router m-n using the language TLA$^+$ of the temporal logic TLA [15], as shown in Fig. 3. The TLA$^+$ module starts by defining the module name on the first line. The EXTENDS keyword states that this module imports the modules Naturals, which defines various arithmetic operators on natural numbers, and MatrixSum, defining operators for summing matrices. The variables of the module are declared using the VARIABLES keyword, where *req* and *res* represent the requester and responder partitions respectively. Constants are declared by the CONSTANTS keyword. They are the parameters of the model. When creating the building block Router m-n, we do not know how it will be reused in an enclosing activity. Another developer may choose to put multiple instances in both, one or none of the enclosing partitions. So, we need constants for the number of requesters and responders per enclosing partition instance, as well as the number of enclosing partition instances on each side. Hence, the global number of requesters is really $no_req * no_req_encl$.

A TLA$^+$ specification describes a labelled transition system. This is given as a set of states, a subset of the states that are initial states, a set of actions (labels) and a transition relation describing whether the system can make a transition from one state to another via a given action. The set of initial states is described by the *Init* construct. Here, the *req* and *res* variables are each given records for their corresponding pins (except pin resOut, see below), which in turn are functions in two dimensions stating whether a token has passed the pin for each requester or responder instance. That is, a requester or responder instance is identified by an enclosing instance number combined with an inner instance number.

Next in the TLA$^+$ module follow the actions, which formally are predicates on pairs of states. Variables denoting the state before carrying out an action use

[3] This behaviour is described by the EESM of the Forward block, which, due to space constraints, we do not show.

```
---------------------------- MODULE router_m_n ----------------------------
EXTENDS Naturals, MatrixSum
VARIABLES req, res
CONSTANTS no_req, no_res, no_req_encl, no_res_encl

Init ≜
∧ req = [reqIn ↦ [e ∈ 1 .. no_req_encl, i ∈ 1 .. no_req ↦ 0]]
∧ res = [reqOut ↦ [e ∈ 1 .. no_res_encl, i ∈ 1 .. no_res ↦ 0],
   resIn ↦ [e ∈ 1 .. no_res_encl, i ∈ 1 .. no_res ↦ 0]]

reqIn(e, i) ≜ req.reqIn[e, i] = 0
∧ req' = [req EXCEPT !.reqIn[e, i] = 1] ∧ UNCHANGED ⟨res⟩

reqOut(e, i) ≜ res.reqOut[e, i] = 0
∧ Sum(req.reqIn, no_req_encl, no_req) > Sum(res.reqOut, no_res_encl, no_res)
∧ res' = [res EXCEPT !.reqOut[e, i] = 1] ∧ UNCHANGED ⟨req⟩

resIn(e, i) ≜ res.reqOut[e, i] > 0 ∧ res.resIn[e, i] = 0
∧ res' = [res EXCEPT !.resIn[e, i] = 1] ∧ UNCHANGED ⟨req⟩

resOut(e, i) ≜ req.reqIn[e, i] > 0
∧ ∃ f ∈ 1 .. no_res_encl, k ∈ 1 .. no_res :
   ∧ res.resIn[f, k] > 0
   ∧ res' = [res EXCEPT !.reqOut[f, k] = 0, !.resIn[f, k] = 0]
∧ req' = [req EXCEPT !.reqIn[e, i] = 0]
============================================================================
```

Fig. 3. TLA$^+$ module for the EESM of Router m-n

their common identifiers while those referring to the state afterwards are given a prime. The action *reqIn* states that for a requester$\langle \mathrm{e,i} \rangle$ identified by enclosing instance e and inner instance i, a token can enter pin reqIn only if the given instance has not yet had a token pass through this pin. The next conjunct of the reqIn action says that the values of the req variable will be the same as now, except that the counter for tokens having passed through pin reqIn will be set to 1 for requester$\langle \mathrm{e,i} \rangle$. The UNCHANGED keyword states which variables are not changed by an action, as TLA$^+$ requires all next-state variable values to be set explicitly.

The *reqOut* action also represents that a token is only allowed through pin reqOut of responder$\langle \mathrm{e,i} \rangle$ if it has not already had a token pass through. The second line constrains this further by stating that a token passing event can only happen if the sum of all tokens having passed any reqIn pin is greater than the sum of all tokens having passed any reqOut pin. Hence, this event on responder$\langle \mathrm{e,i} \rangle$ is constrained by the state of parallel components. Action *resIn* states that a token may only enter a responder$\langle \mathrm{e,i} \rangle$ via pin resIn if the same instance has already emitted a token via pin reqOut, but not already sent a response through pin resIn. The *resOut* action states that a requester$\langle \mathrm{e,i} \rangle$ can only emit a token through pin resOut if it has received a token through pin reqIn. This is constrained further by requiring there to be a responder$\langle \mathrm{f,k} \rangle$, that has received a token through its resIn pin. All counters belonging to requester$\langle \mathrm{e,i} \rangle$ and responder$\langle \mathrm{f,k} \rangle$ are then set to 0, to reset their state. As the resOut action also performs the reset, there is no TLA$^+$ variable for its corresponding pin.

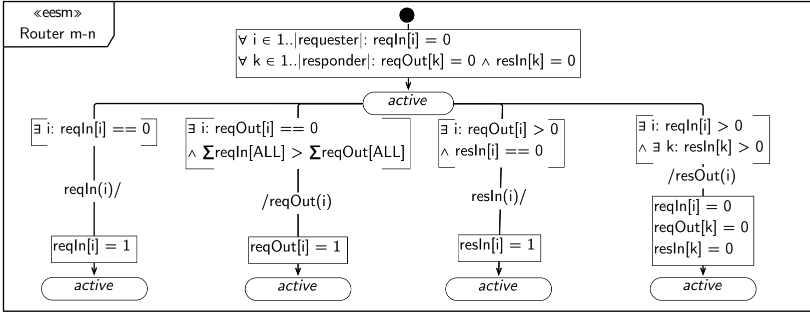


Fig. 4. UML notation for the EESM of Router m-n

This behaviour is easier seen looking at the graphical notation in Fig. 4. Here, the style of the transition operations is closer to programming languages like Java. The number of partition instances is denoted |partition name|. We omit the domain of ∃ statements where this is obvious from the context, and the keyword ALL denotes all indexes in a domain. We also omit specifying which partition a pin belongs to if there is only one pin by that name in the activity. The transition from the initial node to the *active* state represents the *Init* construct in TLA$^+$, and the remaining transitions represent the actions.

Since we do not model that a token keeps the index of its requester instance as data while located at a server, we cannot fully automatically verify that the activity and EESM for Router m-n are consistent. The model checker finds counterexamples where a response is simply sent to a requester that has not yet issued a request, instead of to a requester that has. What we can verify automatically, however, is that whenever a token is sent back to a requester, the EESM is in a state where the token would be allowed through the resOut pin of at least one of them.

Once a building block is complete, we can reuse it like we have reused Router m-n in our system example from Fig. 1(a). To verify properties of the system, we generate the TLA$^+$ module in Fig. 5 (see [13]). This module instantiates other modules, namely Router m-n and Make Response. The constants *no_clients* and *no_servers* represent the parameters a and b from the system activity. We express the actions of the system activity as a composition of constraints and operations on the variables of the system activity, and actions of ESMs of the inner activities. Hence, the Init construct not only sets the timer in all client instances to 0 and all initial nodes to 1, but also calls the Init construct of Router m-n and Make Response as shown by $r!Init$ and $m!Init$. Note also that since this is a system activity, we do not need to add an extra dimension for enclosing partition instances when identifying activity elements like the timer, as it cannot be reused in other activities. For a description of each action, we refer back to Sect. 3.

```
──────────────── MODULE load_sharing_system ────────────────
EXTENDS Naturals, MatrixSum
VARIABLES r_req, r_res, m_state, m_queue, client, server
CONSTANTS no_clients, no_servers
no_res ≜ no_clients ÷ no_servers

r ≜ INSTANCE router_m_n WITH no_req ← 1, no_req_encl ← no_clients,
  no_res ← no_res, no_res_encl ← no_servers, req ← r_req, res ← r_res
m ≜ INSTANCE make_response WITH no_make_response ← 1, no_enclosing ← no_servers,
   state ← m_state, queue ← m_queue

Init ≜ client = [timer ↦ [i ∈ 1 .. no_clients ↦ 0], initial ↦ [i ∈ 1 .. no_clients ↦ 1]]
 ∧ server = [initial ↦ [i ∈ 1 .. no_servers ↦ 1]] ∧ r!Init ∧ m!Init

start_client(p) ≜ client.initial[p] = 1 ∧ client′ = [client EXCEPT !.initial[p] = 0]
 ∧ r!reqIn(p, 1) ∧ UNCHANGED ⟨server, m_state, m_queue⟩

start_server(p) ≜ server.initial[p] = 1 ∧ server′ = [server EXCEPT !.initial[p] = 0]
 ∧ m!start(p, 1) ∧ UNCHANGED ⟨client, r_req, r_res⟩

r_reqOut_m_req(p, i) ≜ r!reqOut(p, i) ∧ m!req(p, 1) ∧ UNCHANGED ⟨client, server⟩

m_res_r_resIn(p, i) ≜ m!res(p, 1) ∧ r!resIn(p, i) ∧ UNCHANGED ⟨client, server⟩

r_resOut_client_timer(p) ≜ r!resOut(p, 1) ∧ client.timer[p] = 0
 ∧ client′ = [client EXCEPT !.timer[p] = 1] ∧ UNCHANGED ⟨server, m_state, m_queue⟩

client_timer_r_reqIn(p) ≜ client.timer[p] = 1 ∧ client′ = [client EXCEPT !.timer[p] = 0]
 ∧ r!reqIn(p, 1) ∧ UNCHANGED ⟨server, m_state, m_queue⟩

Next ≜
 ∨ ∃ p ∈ 1 .. no_clients : start_client(p)
 ∨ ∃ p ∈ 1 .. no_servers : start_server(p)
 ∨ ∃ p ∈ 1 .. no_servers, i ∈ 1 .. no_res : r_reqOut_m_req(p, i)
 ∨ ∃ p ∈ 1 .. no_servers, i ∈ 1 .. no_res : m_res_r_resIn(p, i)
 ∨ ∃ p ∈ 1 .. no_clients : r_resOut_client_timer(p)
 ∨ ∃ p ∈ 1 .. no_clients : client_timer_r_reqIn(p)

Spec ≜ Init ∧ □[Next]_⟨r_req, r_res, m_state, m_queue, client, server⟩
├──────────────────────────────────────────────────────────────┤
P1 ≜ □(∀ p ∈ 1 .. no_servers : m_queue[p, 1] ≤ no_res)
P2 ≜ □(Sum(r_res.reqOut, no_servers, no_res) ≤ Sum(r_req.reqIn, no_clients, 1))
P3 ≜ □(∀ p ∈ 1 .. no_servers, i ∈ 1 .. no_res :
        (server.initial[p] = 0 ∧ ENABLED r!reqOut(p, i)) ⇒ ENABLED m!req(p, 1))
P4 ≜ □(∀ p ∈ 1 .. no_servers : ENABLED m!res(p, 1) ⇒
         ∃ i ∈ 1 .. no_res : ENABLED r!resIn(p, i))
P5 ≜ □(∀ p ∈ 1 .. no_clients : ENABLED r!resOut(p, 1) ⇒ client.timer[p] = 0)
└──────────────────────────────────────────────────────────────┘
```

Fig. 5. TLA$^+$ module for the system activity

The whole system specification is written as a single formula $Spec \triangleq Init \land \Box[Next]_{\langle vars \rangle}$. This formula states that the transition system has initial states(s) as specified by *Init* and that every change to the variables listed in *vars* is done via one of the actions listed in the next-state relation, *Next*. The box-shaped symbol ($\Box$) in front of $[Next]$ is the temporal logic operator *always*. It means that what follows must be true for every reachable state of the system model.

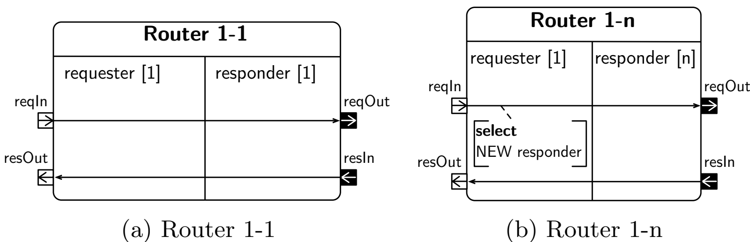
The small example system of this paper is chosen to allow us to show the formal semantics of the EESMs in TLA$^+$ and clarify that they are unambiguous,

yet expressive enough for our needs. Therefore, the properties that we can verify for this system might seem rather trivial, but for more complex systems, variations of these properties may be very hard to verify without a formal model. The properties we want to verify are written formally below the horizontal bar in Fig. 5. All the properties can be verified by model checking. See Sect. 6 for further discussion of the results.

P1. The number of requests queued in any Make Response block is at most equal to the number of responders per server, i.e., the inner queue is finite.
P2. There are at most as many ongoing requests on the server side as there are on the client side.
P3. Whenever a server is started and a responder instance of that server is ready to emit a token through the reqOut pin, the Make Response instance of that server is ready to accept a token through its req pin.
P4. Whenever a token can be emitted from the Make Response block of a server, at least one of the responder instances on that server is able to accept it.
P5. Whenever a token can be emitted via the resOut pin of a requester instance, the corresponding timer is empty, hence ready to receive a token.

5 Other Types of Multiplicity

Our formalism for expressing contracts of multi-instance activities also works for one-to-one building blocks without any internal select statements, like Router 1-1 shown in Fig. 6(a). This is a special case, where each requester instance is statically mapped to a responder instance and vice versa. As each binary collaboration cannot have any constraints on its behaviour in terms of the state of parallel instances, we can simplify the EESM as shown in Fig. 7 without loss of information. This is, in fact, the same notation that we have been using already for ESMs of activities with one instance of each type [14], only augmented with an index i. The difference is that the formal semantics now supports multiple instances globally, instead of requiring such a block to be used in a system with only one instance of each enclosing partition as well.



(a) Router 1-1 (b) Router 1-n

Fig. 6. The two other variants of the router activity, with respect to select statements

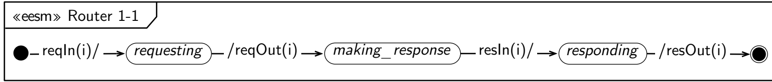


Fig. 7. UML notation for the EESM of Router 1-1

Finally, we present a one-to-many variation of the router block, Router 1-n, where a requester is statically mapped to a set of responders, as shown in Fig. 6(b).[4] This could be used in a setting where each server from Fig. 1(a) has one responder instance per client, so that each client has a choice of any server when issuing a request. When the response is to be routed back, the corresponding requester is already given, due to the static mapping.

The EESM of Router 1-n is shown in Fig. 8. Due to the mapping between requester and responder instances, the notation is a bit more complex than for the other variants. For example, the /reqOut(i) transition states that a token may only leave the reqOut pin through instance i if this has not happened already. The rest of the guard constrains this further by stating that a token passing can only occur if the corresponding requester instance has gotten more tokens through its reqIn pin than the sum of tokens having already passed through pin reqOut in all the responders mapping to that requester. The expression $reqIn[responder[i]]$ means the value of the reqIn variable for the requester who can be found by mapping $responder[i]$ to its corresponding requester. Hence, $\Sigma\ reqOut[requester[responder[i]][k]]$ means the sum of reqOut values for the k different responders found by mapping responder[i] to its requester and then mapping that requester to the set of corresponding responders.

Note that it is the EESM that holds the information on whether there is a constrained static mapping or not. In contrast, the EESM of Router m-n, given in Fig. 4, has no references to any particular parallel instance or set of instances, only to the current instance and the keyword ALL.

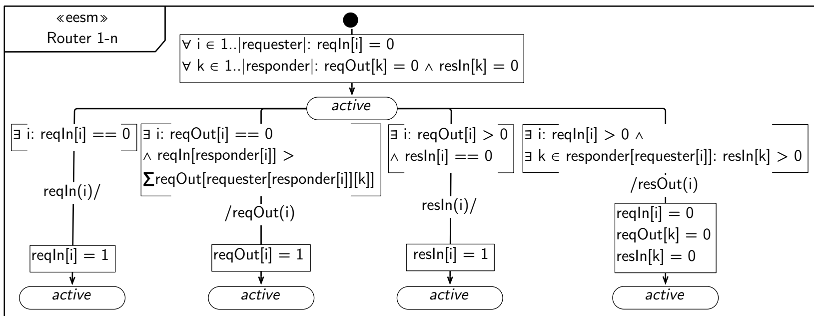


Fig. 8. UML notation for the ESM of Router 1-n

[4] In addition, there could naturally be a mirror version of the Router 1-n activity, a Router n-1, but this is also a one-to-many activity.

Table 1. Number of states found and time required to verify properties P1–P5

# of servers → # of clients ↓	1	2	3
1	7 states, < 1 sec		
2	37 states, 1 sec	70 states, 1 sec	
3	241 states, 2 sec		707 states, 3 sec
4	1713 states, 4 sec	3410 states, 5 sec	
5	12617 states, 9 sec		
6	94513 states, 48 sec	188962 states, 99 sec	283411 states, 155 sec
7	715001 states, 651 sec		

6 Concluding Remarks

All the properties from Fig. 5 have been verified by the TLC model checker [24], for the parameter values shown in Table 1.[5] Model checking only verifies properties for a model with some exact parameters. It does not say whether those properties will still hold for different parameters. However, if the model changes behaviour with respect to a property for some specific parameter values, it is often when a parameter is changed from 1 to >1, or it is likely due to highly intentional design decisions. Hence, the fundamental problem remains, but it is not always that great in practice.

Given that model checking is automatic, one could say that time is not an issue, as we can just leave a computer at it and check for up to, for example, a thousand instances of each partition. However, as Table 1 shows, the time needed grows exponentially as we increase the number of client instances. The linear increase from server instances comes from the fact that more servers reduce the number of responders per server.

There is a high level of parallelism in our system example. This is also the case for other systems using EESMs that we have verified. Hence, we expect partial order reduction [7] to alleviate the state-space blowup from increasing the number of instances. We therefore plan to also formalize our semantics in Promela, so we can use the Spin [8] model checker, which implements partial order reduction. The formalisms are compatible, as there is already work for transforming another TLA derivative, cTLA, into Promela automatically [20]. For relatively simple blocks, where the contract must be verified for any number of instances, the TLA formalism allows for writing manual refinement proofs as well [16].

We already have a tool for generating TLA$^+$ from SPACE models [13]. This tool greatly reduces the time required to specify systems, and it automatically generates many types of general properties to ensure the well-formedness

[5] We are aware that the TLA$^+$ specification for the given example can be optimized by only storing the aggregate number of tokens having passed through a pin on any of the responders in a server. However, this optimization would not work if the EESM required two tokens to pass pin reqOut before a token is allowed though pin resIn.

of SPACE models. We will extend the tool to work with EESMs, outputting Promela specifications as well. To hide the formalism when specifying application-specific properties, there is work in progress to express them in UML.

To verify properties like "Every request is eventually responded to", would require adding data to identify each request and adding liveness constraints to the model. Being based on TLA, the formalism can accommodate this quite easily in the form of weak or strong fairness assumptions. The limiting factor is still time needed for model checking.

Having formalized extended ESMs, we are eager to use them in the setting of fault-tolerant systems, where multiple instances of the same type often collaborate to mask failures, and conditions such as a majority of the instances being reachable are often essential to precisely describe the behaviour of a block.

To conclude, contracts encapsulate software components and facilitate both reuse and compositional verification. The SPACE method uses collaborations detailed by UML activities as the unit of reuse. We introduce EESMs, which allow to describe the global behaviour of multi-instance activities, abstracting away internal state, while still having the expressive power to detail when an external event can take place. An example from the load balancing Router m-n block is that a request will only arrive at a server that has free capacity in the form of free responder instances, and only if the number of requests received from all clients is greater than the number of requests forwarded to any server. While the EESMs have a formal semantics in TLA, we give graphical UML state machines as specification tools, so that software engineers themselves need not be experts in temporal logic.

References

1. Bauer, S.S., Hennicker, R.: Views on behaviour protocols and their semantic foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 367–382. Springer, Heidelberg (2009)
2. Beugnard, A., Jezequel, J.M., Plouzeau, N., Watkins, D.: Making Components Contract Aware. Computer 32, 38–45 (1999)
3. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: Proc. 30th Int. Design Automation Conf. DAC 1993, pp. 86–91. ACM Press, New York (1993)
4. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
5. Guelfi, N., Mammar, A.: A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In: Proc. 12th Asia-Pacific SE Conf., pp. 283–290 (2005)
6. Harel, D., Pnueli, A.: On the development of reactive systems. In: Logics and models of concurrent systems, pp. 477–498. Springer New York, Inc., Heidelberg (1985)
7. Holzmann, G., Peled, D.: An improvement in formal verification. In: Proc. 7th IFIP WG6.1 Int. Conf. on Formal Description Techniques, pp. 197–211 (1995)
8. Holzmann, G.J.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2003)

9. Kraemer, F.A., Herrmann, P.: Reactive Semantics for Distributed UML Activities. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 17–31. Springer, Heidelberg (2010)
10. Kraemer, F.A., Bræk, R., Herrmann, P.: Synthesizing components with sessions from collaboration-oriented service specifications. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 166–185. Springer, Heidelberg (2007)
11. Kraemer, F.A., Herrmann, P.: Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In: Proc. Networking and Electronic Conf. (2007)
12. Kraemer, F.A., Herrmann, P.: Automated encapsulation of UML activities for incremental development and verification. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 571–585. Springer, Heidelberg (2009)
13. Kraemer, F.A., Slåtten, V., Herrmann, P.: Engineering Support for UML Activities by Automated Model-Checking — An Example. In: Proc. 4th Int. Workshop on Rapid Integration of Software Engineering Techniques, RISE 2007 (2007)
14. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. Journal of Systems and Software 82(12), 2068–2080 (2009)
15. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)
16. Lamport, L.: Refinement in state-based formalisms. Tech. rep., Digital Equipment Corporation, Systems Research Center, Palo Alto, California (1996)
17. Mealy, G.H.: A Method to Synthesizing Sequential Circuits. Bell Systems Technical Journal 34(5), 1045–1079 (1955)
18. Mencl, V.: Specifying Component Behavior with Port State Machines. Electronic Notes in Theoretical Computer Science 101, 129–153 (2004)
19. OMG. Unified Modeling Language: Superstructure, Version 2.3 (2010)
20. Rothmaier, G., Poh1, A., Krumm, H.: Analyzing Network Management Effects with Spin and cTLA. In: Security and Protection in Information Processing Systems, ch. 5. IFIP AICT, vol. 147, pp. 65–81. Springer, Heidelberg (2004)
21. Rushby, J.: Disappearing formal methods. In: Fifth IEEE International Symposim on High Assurance Systems Engineering, pp. 95–96 (2000)
22. Sanders, R.T., Bræk, R., von Bochmann, G., Amyot, D.: Service Discovery and Component Reuse with Semantic Interfaces. In: Prinz, A., Reed, R., Reed, J. (eds.) SDL 2005. LNCS, vol. 3530, pp. 1244–1247. Springer, Heidelberg (2005)
23. Storrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. Electronic Notes in Theoretical Computer Science 127(4), 35–52 (2005)
24. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999)

Annotation Inference for Separation Logic Based Verifiers

Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans*

Katholieke Universiteit Leuven, Leuven, Belgium
{frederic.vogels,bart.jacobs,frank.piessens,jans.smans}@cs.kuleuven.be

Abstract. With the years, program complexity has increased dramatically: ensuring program correctness has become considerably more difficult with the advent of multithreading, security has grown more prominent during the last decade, etc. As a result, static verification has become more important than ever.

Automated verification tools exist, but they are only able to prove a limited set of properties, such as memory safety. If we want to prove full functional correctness of a program, other more powerful tools are available, but they generally require a lot more input from the programmer: they often need the code to be verified to be heavily annotated.

In this paper, we attempt to combine the best of both worlds by starting off with a manual verification tool based on separation logic for which we develop techniques to automatically generate part of the required annotations. This approach provides more flexibility: for instance, it makes it possible to automatically check as large a part of the program as possible for memory errors and then manually add extra annotations only to those parts of the code where automated tools failed and/or full correctness is actually needed.

1 Introduction

During the last decade, program verification has made tremendous progress. However, a key issue hindering the adoption of verification is that a large amount of annotations is required for tools to be able to prove programs correct, in particular if correctness involves not just memory safety, but also program-specific properties. In this paper, we propose three annotation inference and/or reduction techniques in the context of separation logic-based verifiers: (1) automatic predicate folding and unfolding, (2) predicate information extraction lemmas and (3) automatic lemma application via shape analysis. All aforementioned contributions were developed in the context of the VeriFast program verifier in order to reduce annotation overhead for practical examples.

VeriFast [17] is a verification tool being developed at the K.U. Leuven. It is based on separation logic [21] and can currently be used to verify a multitude of correctness-related properties of C and Java programs. Example usages are

* Jan Smans is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 319–333, 2011.
© IFIP International Federation for Information Processing 2011

- ensuring that C code does not contain any memory-related errors, such as memory leaks and dereferencing dangling pointers;
- checking that functions or methods satisfy contracts describing their intended semantics;
- preventing the occurrence of data races in multi-threaded code.

VeriFast heavily relies on programmer-provided annotations: this makes the tool very efficient, but the need for annotations can make its use quite cumbersome. To give the reader an idea of the quantity of annotations needed, we provide some quick statistics in Figure 1: the first column contains a count of the number of lines of actual C code. The second column expresses the number of annotations in number of lines. The numbers between parentheses correspond to the number of open and close statements, respectively, which will be further explained in Section 3.1. The third column shows the amount of annotation overhead.

We have developed three different techniques to (partially) automate verification by mechanizing the generation of (some of) the necessary annotations. In this paper, we describe these three approaches in detail. We will make the distinction between two layers:

- VeriFast's core, which requires all annotations and performs the actual verification. This core must be as small and uncomplicated as possible, as the verification's soundness relies on it.
- The automation layer, which generates as large a portion of the necessary annotations as possible, which will then in a second phase be fed to VeriFast's core for verification.

This approach maximizes robustness: we need only trust the core and can freely experiment with different approaches to automation without having to worry about introducing unsound elements, since the generated annotations will still be fully verified by the core, thus catching any errors.

In order to be able to discuss and compare our three annotation generation techniques, we need the reader to be familiar with VeriFast, therefore we included a small tutorial (Section 2) which explains the basic concepts. Next, we explain our different approaches to automation in Section 3. We then put them side by side in Section 4 by comparing how many of the necessary annotations they are able to generate.

	LOC	**LOAnn**	**LoAnn/LOC**
stack (C)	88	198 (18/16)	2.3
sorted binary tree (C)	125	267 (16/23)	2.1
bank example program (C)	405	127 (10/22)	0.31
chat server (C)	130	114 (20/26)	0.88
chat server (Java)	138	144 (19/28)	1.0
game server (Java)	318	225 (47/63)	0.71

Fig. 1. Some line count statistics

2 VeriFast: A Quick Tutorial

This section contains a quick introduction to VeriFast. It is not our intention to teach the reader how to become proficient in using VeriFast, but rather to provide a basic understanding of certain concepts on which VeriFast is built. A full tutorial is available at [1].

2.1 A Singly Linked List

Figure 2 shows a `struct`-definition for a singly linked list node together with a function `new` which creates and initializes such a node. In order to verify this function in VeriFast, we need to provide a contract. The precondition, `emp`, indicates that the function does not require anything to be able to perform its task. The postcondition is a separating conjunction (`&*&`) of the "heap fragments":

- `malloc_block_list(result)` means that the returned pointer will point to a malloc'ed block of memory the size of a `list`. It is produced by a call to `malloc` and it is meant to be eventually consumed by a matching `free`. Failure to do so eventually leads to an unreachable `malloc_block_list` which corresponds to a memory leak.
- `result->next |-> n` means two things: it grants the permission to read and write to the `node`'s `next` field, and it tells us that the `next` field contains the value of the argument `n`. Idem for `result->value |-> v`.
- `result != 0` guarantees that the returned pointer is not null.

If this function verifies, it will mean that it is both memory safe and functionally correct. We defer the explanation of the separating conjunction `&*&` until later.

VeriFast uses symbolic execution [7] to perform verification. The precondition determines the initial symbolic state, which in our case is empty. VeriFast then proceeds by symbolically executing the function body.

```
struct list { struct list* next; int value; };
struct list* new(struct list* n, int v)
 //@ requires emp;
 /*@ ensures malloc_block_list(result) &*& result->next  |-> n &*&
             result->value |-> v       &*& result != 0         @*/
{
 struct list* node = malloc( sizeof( struct list ) );
 if ( node == 0 ) abort();
 node->next = n; node->value = v;
 return node;
}
```

Fig. 2. A singly linked list node in C

1. `malloc` can either fail or succeed. In the case of `list`, its contract is
```
requires emp;
ensures  result == 0 ? emp
                     : malloc_block_list(result) &*&
                       result->next  |-> _       &*&
                       result->value |-> _;
```
   Either it will return `0`, which represents failure, and the heap is left unchanged. Otherwise, it returns a non-null pointer to a block of memory the size of a `struct list`, guaranteed by `malloc_block_list(result)`. It also provides access to both `next` and `value` fields of this newly created node, but does not make any promises about their values.
2. The `if` statement will intercept execution paths where `malloc` has failed and calls `abort`, causing VeriFast not to consider these paths any further.
3. Next, we assign `n` to the `next` field. This operation is only allowed if we have access to this field, which we do since the previous `if` statement filtered out any paths where allocation failed, and a successful `malloc` always provides the required access rights. This statement transforms the `result->next |-> _` heap fragment to `result->next |-> n`.
4. Assigning `v` to `node->value` works analogously.
5. Finally, we return the pointer to the node. At this point, VeriFast checks that the current execution state matches the postcondition. This is the case, which concludes the successful verification of `new`.

A full explanation of the separating conjunction `&*&` can be found in [21]. In short, `P &*& Q` means that the heap consists of two disjoint subheaps where `P` and `Q` hold, respectively. It is used to express that blocks of memory do not overlap, and therefore changes to one object do not influence another. Separating conjunction enables the frame rule, which allows us to reason "locally". It expresses the fact that if an operation behaves some way in a given heap (`{P} op {Q}`), it will behave the same way in an extended heap (`{P &*& R} op {Q &*& R}`). For example, if we were to call `malloc` twice in a row, how can we be sure that it didn't return the same block of memory twice? We know this since, thanks to the frame rule, we get two `malloc_block_list` heap fragments joined by a separating conjunction, guaranteeing us that we are dealing with two different objects.

2.2 Predicates

As shown in the previous section, freely working with a single node requires carrying around quite a bit of information. This can become tiresome, especially if one considers larger structures with more fields whose types might be other structures. For this reason, VeriFast provides the ability to perform abstractions using predicates [20] which make it possible to "fold" (or close, in VeriFast terminology) multiple heap fragments into one.

Figure 3 shows the definition for the `Node` predicate and an updated version of the `new` function. The `close` statement removes three heap fragments (`malloc_block_list` and the two field access permissions) and replaces them

```
predicate Node(struct list* n, struct list* m, int v) =
 malloc_block_list(n) &*& n->next |-> m &*& n->value |-> v &*& n != 0;

struct list* new(struct list* n, int v)
 //@ requires emp;
 //@ ensures Node(result, n, v);
{
 struct list* node = malloc( sizeof( struct list ) );
 if ( node == 0 ) abort();
 node->next = n; node->value = v;
 //@ close Node(node, n, v);
 return node;
}
```

Fig. 3. Predicates

by a single `Node` fragment, on condition that it can ascertain that `node` is not `0` and the three fragments are present on the symbolic heap, otherwise verification fails. This closing is necessary in order to have the final execution state match the postcondition. Closing must happen last, for otherwise the field access permissions would be hidden when they are needed to initialize the node's fields (third line of the procedure body.) At a later time, whenever the need arises to access one of a node's fields, the `Node` fragment can be opened up (replacing `Node` by the three separate heap fragments on the symbolic heap) so as to make the field access permission available again.

2.3 Recursive Predicates

A linked list consists of a chain of nodes each pointing to the next. Currently, we can only express linked lists of fixed maximum length:

```
p == 0 ? emp
        : Node(p, q, v1) &*& (q == 0 ? emp
                                        : Node(q, 0, v2)) // len 0-2
```

We can solve this problem using recursive predicates: we need to express that a list is either empty or a node pointing to another list. Figure 4 shows the definition for `LSeg(p, q, xs)`, which stands for a cycleless singly linked list segment where `p` points to the first node, `q` is the one-past-the-end node and `xs` stands for the contents of this list segment.

Figure 4 also contains the definition for a `prepend` function. The contract fully describes its behaviour, i.e. that a new element is added in front of the list, and that the pointer passed as argument becomes invalid; instead the returned pointer must be used.

```
/*@ inductive List = Nil | Cons(int, List);
    predicate LSeg(struct list* p, struct list* q, List Xs) =
    p == q ? Xs == Nil
           : Node(p,?t,?y) &*& LSeg(t,q,?Ys) &*& Xs == Cons(y,Ys); @*/

struct list* prepend(struct list* xs, int x)
 //@ requires LSeg(xs, 0, ?Xs);
 //@ ensures LSeg(result, 0, Cons(x, Xs));
{
 struct list* n = new(xs, x);
 //@ open Node(n, xs, x);
 //@ close Node(n, xs, x);
 //@ close LSeg(n, 0, Cons(x, Xs));
 return n;
}
```

Fig. 4. Recursive predicates

2.4 Lemmas

The reader might wonder why a `Node` is consecutively opened and closed in Figure 4. Let us first examine what happens without it. When VeriFast reaches the closing of the `LSeg`, execution forks due to the conditional in the `LSeg` predicate:

- `n` might be equal to `0`, in which case `xs` must be `Nil` instead of `Cons(x, Xs)`, so that closing fails. This needs to be prevented.
- `n` could also be a non-null pointer: the `Node` and original `LSeg` get merged into one larger `LSeg`, which is exactly what we want.

Therefore, we need to inform VeriFast of the fact that `n` cannot be equal to `0`. This fact is hidden within the `Node` predicate; opening it exposes it to VeriFast. After this we can immediately close it again in order to be able to merge it with the `LSeg` heap fragment.

The need to prove that a pointer is not null occurs often, and given the fact that an `open`/`close` pair is not very informative, it may be advisable to make use of a lemma, making our intentions clearer, as shown in Figure 5. In Section 3, we will encounter situations where lemmas are indispensable if we are to work with recursive data structures.

3 Automation Techniques

We have implemented three automation techniques which we discuss in the following sections. For this, we need a running example: Figure 7 contains a fully annotated list-copying function, for which we will try to automatically infer as many of the required annotations as possible.

```
/*@ lemma void NotNull(struct list* p)
      requires Node(p, ?pn, ?pv);
      ensures  Node(p, pn, pv) &*& p != 0;
    {
      open Node(p, pn, pv); close Node(p, pn, pv);
    }
@*/
struct list* prepend(struct list* xs, int x)
 //@ requires LSeg(xs, 0, ?Xs);
 //@ ensures  LSeg(result, 0, Cons(x, Xs));
{
 struct list* n = new(xs, x);
 //@ NotNull(n);
 //@ close LSeg(n, 0, Cons(x, Xs));
 return n;
}
```

Fig. 5. The `NotNull` lemma

In our work, we have focused on verifying memory safety; automation for verifying functional properties is future work. Therefore, we simplify the `Node` and `LSeg` predicates we defined earlier by having the predicates throw away the data-related information, as shown in Figure 6.

```
predicate Node(struct list* P, struct list* Q) =
 P != 0 &*& malloc_block_list(P) &*& P->next  |-> Q &*& P->value |-> ?v;

predicate LSeg(struct list* P, struct list* Q) =
 P == Q ? emp : Node(P, ?R) &*& LSeg(R, Q);
```

Fig. 6. Simplified `Node` and `LSeg` predicates

While it is not strictly necessary to understand the code in Figure 7, we choose to clarify some key points:

- The `new()` function produces a new `Node(result, 0)` and always succeeds. It is just another function defined in terms of `malloc` and `aborts` on allocation failure (comparable to Figure 3).
- `NoCycle`, `Distinct`, `AppendLSeg` and `AppendNode` are lemmas whose contracts are shown in Figure 8.

The `copy` function comprises 12 statements containing actual C code, while the annotations consist of 31 statements, not counting the lemmas since these can be shared by multiple function definitions. We now proceed with a discussion of how to generate some of these annotations automatically.

3.1 Auto-Open and Auto-Close

As can be seen in the examples, a lot of annotations consist of opening and closing predicates. This is generally true for any program: the values between parentheses in Figure 1 indicate how many open and close statements respectively were necessary for the verification of other larger programs.

These annotations seem to be ideal candidates for automation: whenever the execution of a statement fails, the verifier could take a look at the current execution state and try opening or closing predicates to find out whether the right heap fragments are produced.

For example, assume we are reading from the `next` field of a variable `x`, which requires a heap fragment matching `x->next |-> v`. However, only `Node(x, y)` is available. Without automation, verification would fail, but instead, the verifier could try opening `Node(x, y)` and find out that this results in the required heap fragment. Of course, this process could go on indefinitely given that predicates can be recursively defined. Therefore, some sort of heuristic is needed to guide the search.

We have added support for automatic opening and closing of predicates [20] to VeriFast. Without delving too much into technical details, VeriFast keeps a directed graph whose nodes are predicates and whose arcs indicate how predicates are related to each other. For example, there exists an arc from `LSeg` to `Node` meaning that opening an `LSeg` yields a `Node`. However, this depends on whether or not the `LSeg` does represent the empty list. To express this dependency, we label the arcs with the required conditions. These same conditions can be used to encode the relationships between the arguments of both predicates. For the predicate definitions from Figure 6, the graph would contain the following:

$$\text{LSeg}(a,b) \quad \overset{\substack{a \neq b \\ a = p}}{\longrightarrow} \quad \text{Node}(p,q) \quad \overset{p = x}{\longrightarrow} \quad x \rightarrow \text{next} \mapsto y$$

When, during verification, some operation requires the presence of a `Node(p, q)` heap fragment but which is missing, two possible solutions are considered: we can either attempt to perform an auto-open on an `LSeg(p, b)` for which we know that `p != b`, or try to close `Node(p, q)` if there happens to be a `p->next |-> ?` on the current heap.

Using this technique yields a considerable decrease in the amount of necessary annotations: each `open` or `close` indicated by `// a` is inferred automatically by VeriFast. Out of the 31 annotation statements, 17 can be generated, which is more than a 50% reduction.

3.2 Autolemmas

We now turn our attention to another part of the annotations, namely the lemmas. On the one hand, we have the lemma definitions. For the moment, we

```
struct list* copy(struct list* xs)
 //@ requires LSeg(xs, 0);
 //@ ensures LSeg(xs, 0) &*& LSeg(result, 0);
{
 if ( xs == 0 ) {
  //@ close LSeg(0, 0);                                  // a
  return 0; }
 else {
  struct list* ys = new();
  //@ open LSeg(xs, 0);
  //@ open Node(xs, _);                                  // a
  //@ open Node(ys, 0);                                  // a
  ys->value = xs->value;
  struct list *p = xs->next, *q = ys;
  //@ close Node(ys, 0);                                 // a
  //@ close Node(xs, p);                                 // a
  //@ NoCycle(xs, p);
  //@ close LSeg(p, p);                                  // a
  //@ close LSeg(xs, p);                                 // a
  //@ close LSeg(ys, q);                                 // a
  while ( p != 0 )
   //@ invariant LSeg(xs,p) &*& LSeg(p,0) &*& LSeg(ys,q) &*& Node(q,0);
  {
   //@ struct list *oldp = p, *oldq = q;
   struct list* next = new();
   //@ Distinct(q, next);
   //@ open Node(q, 0);                                  // a
   q->next = next; q = q->next;
   //@ close Node(oldq, q);                              // a
   //@ open LSeg(p, 0);
   //@ assert Node(p, ?pn);
   //@ NoCycle(p, pn);
   //@ open Node(p, _);                                  // a
   //@ open Node(q, 0);                                  // a
   q->value = p->value; p = p->next;
   //@ close Node(q, 0);                                 // a
   //@ close Node(oldp, p);                              // a
   //@ AppendLSeg(xs, oldp); AppendNode(ys, oldq);
  }
  //@ open LSeg(p, 0);                                   // a
  //@ NotNull(q);                                        // b
  //@ close LSeg(0, 0);                                  // a
  //@ AppendLSeg(ys, q);
  //@ open LSeg(0, 0);                                   // a
  return ys;
 }
}
```

Fig. 7. Copying linked lists

```
lemma void NoCycle(struct list* P, struct list* Q)
  requires Node(P, Q) &*& LSeg(Q, 0);
  ensures  Node(P, Q) &*& LSeg(Q, 0) &*& P != Q;
lemma void Distinct(struct list* P, struct list* Q)
  requires Node(P, ?PN) &*& Node(Q, ?QN);
  ensures  Node(P, PN)  &*& Node(Q, QN) &*& P != Q;
lemma void AppendLSeg(struct list* P, struct list* Q)
  requires LSeg(P, Q) &*& Node(Q, ?R) &*& Q != R &*& LSeg(R, 0);
  ensures  LSeg(P, R) &*& LSeg(R, 0);
lemma void AppendNode(struct list* P, struct list* Q)
  requires LSeg(P, Q) &*& Node(Q, ?R) &*& Node(R, ?S);
  ensures  LSeg(P, R) &*& Node(R, S);
```

Fig. 8. Lemmas

have made no efforts to automate this aspect as lemmas need only be defined once, meaning that automatic generation would only yield a limited reduction in annotations.

On the other hand we have the lemma applications, which is where our focus lies. Currently, we have only implemented one very specific and admittedly somewhat limited way to automate lemma application. While automatic opening and closing of predicates is only done when the need arises, VeriFast will try to apply all lemmas regarding a predicate `P` each time `P` is produced, in an attempt to accumulate as much extra information as possible. This immediately gives rise to some obvious limitations:

- It can become quite inefficient: there could be many lemmas to try out and many matches are possible. For example, imagine a lemma operates on a single `Node`, then it can be applied to every `Node` on the heap, so it is linear with the number of `Node`s on the heap. If however it operates on two `Node`s, matching becomes quadratic, etc. For this reason, two limitations are imposed: lemmas need to be explicitly declared to qualify for automatic application, and they may only depend on one heap fragment.
- Applying lemmas can modify the execution state so that it becomes unusable. For example, if the `AppendLSeg` lemma were applied indiscriminately, `Node`s would be absorbed by `LSeg`s, effectively throwing away potentially crucial information (in this case, we "forget" that the list segment has length 1.) To prevent this, autolemmas are not allowed to modify the symbolic state, but instead may only extend it with extra information.

Given these limitations, in the case of our example, only one lemma qualifies for automation: `NotNull`. Thus, every time a `Node(p, q)` heap fragment is added to the heap, be it by closing a `Node` or opening an `LSeg` or any other way, VeriFast will immediately infer that `p != 0`. Since we only needed to apply this lemma once, we decrease the number of annotations by just one line (Figure 7, indicated by `// b`).

3.3 Automatic Shape Analysis

Ideally, we would like to get rid of all annotations and have the verifier just do its job without any kind of interaction from the programmer. However, as mentioned before, the verifier cannot just guess what behaviour a piece of code is meant to exhibit, so that it can only check for things which are program-independent bugs, such as data races, dangling pointers, etc.

Our third approach for reducing annotations focuses solely on shape analysis [13], i.e. it is limited to checking for memory leaks and invalid pointers dereferences. Fortunately, this limitation is counterbalanced by the fact that it is potentially able to automatically generate all necessary annotations for certain functions, i.e. the postcondition, loop invariants, etc.

In order to verify a function by applying shape analysis, we need to determine the initial program state. The simplest way to achieve this is to require the programmer to make his intentions clear by providing preconditions. Even though it appears to be a concession, it has its advantages. Consider the following: the function `length` requires a list, but `last` requires a non-empty list. How does the verifier make this distinction? If `length` contains a bug which makes it fail to verify on empty lists, should the verifier just deduce it is not meant to work on empty lists?

We could have the verifier assume that the buggy `length` function is in fact correct but not supposed to work on empty lists. The verification is still sound: no memory-related errors will occur. A downside to this approach is that the `length` function will probably be used elsewhere in the program, and the unnecessary condition of non-emptiness will propagate. At some point, verification will probably fail, but far from the actual location of the bug. Requiring contracts thus puts barriers on how far a bug's influence can reach.

One could make a similar case for the postconditions: shape analysis performs symbolic execution and hence ends up with the final program state. If the programmer provides a postcondition, it can be matched against this final state. This too will prevent a bug's influence from spreading.

Our implementation of shape analysis is based on the approach proposed by Distefano et al. [13]. The idea is simple and very similar to what has been explained earlier in Section 3.1: during the symbolic execution of a function, it will open and close the predicates as necessary to satisfy the precondition of the operations it encounters. However, the analysis has a more thorough understanding of the lemmas: it will know in what circumstances they need to be applied. A good example of this is the inference of the loop invariant where shape analysis uses the lemmas to abstract the state, which is necessary to prevent the symbolic heap from growing indefinitely while looking for a fixpoint. Consider the following pseudocode:

$$p' := p; \textbf{while } p \neq 0 \textbf{ do } p := p{\rightarrow}\text{next } \textbf{end}$$

Initially, the symbolic heap contains `LSeg(p, 0)`. To enter the loop, `p` needs to be non-null, hence it is a non-empty list and can be opened up to `Node(p', p1) &*& LSeg(p1, 0)`. During the next iteration, `p1` can be null (the loop ends) or

without abstraction	with abstraction
`Node(p', p) &*& LSeg(p, 0)`	`LSeg(p', p) &*& LSeg(p, 0)`
`Node(p', p1) &*& Node(p1, p) &*& LSeg(p, 0)`	`LSeg(p', p) &*& LSeg(p, 0)`

Fig. 9. Finding a fixed point

non-null (a second node). Thus, every iteration adds the possibility of an extra node. This way, we'll never find a fixed point. Performing abstraction will fold nodes back into `LSeg`s. The difference is shown in Figure 9. One might wonder why the abstraction doesn't also merge both `LSeg`s into a single `LSeg`. The reason for this is that the local variable `p` points to the start of the second `LSeg`: folding would throw away information deemed important.

For our purposes, the algorithms defined in [13] need to be extended so that apart from the verification results of a piece of code and final program states which determine the postcondition, they also generate the necessary annotations to be added to the verified code. This way, the results can be checked by VeriFast, keeping our trusted core to a minimum size (i.e. we do not need to trust the implementation of the shape analysis tool), and extra annotations can be added later on if we wish to prove properties other than memory safety.

For our example, shape analysis is able to deduce all `open` and `close` annotations, the lemma applications, the loop invariant and the postcondition (in our implementation, we chose to require only the precondition and we manually check that the generated postcondition is as intended). Hence, the number of necessary annotations for Figure 7 is reduced to 1, namely the precondition.

4 Comparison

In order to get a better idea of by how much we managed to decrease the number of annotations, we wrote a number of list manipulation functions. There are four versions of the code:

C-code	#code	A	B	C	D
`length`	10	12	9	9	1
`sum`	11	11	7	7	1
`destroy`	9	6	4	4	1
`copy`	23	32	15	14	1
`reverse`	12	9	5	5	1
`drop_last`	28	28	13	13	1
`prepend`	7	5	3	3	1
`append`	13	20	11	11	1

lemma	A	B	C
`Distinct`	9	7	7
`NotNull`	7	6	6
`AppendNode`	19	16	16
`AppendLSeg`	27	19	18
`AppendNil`	9	7	6
`NoCycle`	11	10	9

	#code	A	B	C	D
total	113	205	132	128	8

Fig. 10. Annotation line count comparison

(A) A version with all annotations present.
(B) An adaptation of (A) where we enabled auto-open and auto-close.
(C) A version where we take (B) and make `NotNull` an autolemma (Section 3.2).
(D) Finally, a minimal version with only the required annotations to make our shape analysis implementation (Section 3.3) able to verify the code.

Figure 10 shows how the annotation line counts relate to each other.

5 Related Work

Smallfoot [6] is a verification tool based on separation logic which given pre- and post-conditions and loop invariants can fully automatically perform shape analysis. It has been extended for greater automation [24], for termination proofs [8,12], fine-grained concurrency [10] and lock-based concurrency [15].

jStar [14] is an another automatic separation logic based verification tool which targets Java. One only needs to provide the pre- and post-conditions for each method, after which it attempts to verify it without extra help. It is able to infer loop invariants, for which it uses a more generalized version of the approach described by Distefano et al. [13]. This is achieved by allowing the definition of user-defined rules (comparable to our lemmas) which are then used by the tool to perform abstraction on the heap state during the fixed point computation.

A third verification tool based on separation logic is SpaceInvader [13,24], which performs shape analysis on C programs. Abductor, an extension of this tool, uses a generalized form of abduction [9], which gives it the ability not only to infer loop invariants and postconditions, but also preconditions.

Other tools which don't rely on separation logic are for example KeY [3] (dynamic logic [16]), Spec♯ [5], Chalice [19], Dafny [2], and VCC [11], the latter three being based on Boogie2 (verification condition generation [4,18]). Still other alternatives to separation logic are implicit dynamic frames [23] and matching logic [22], the latter being an approach where specifications are expressed using patterns which are matched against program configurations.

6 Conclusion

We can divide verifiers in two categories.

– Fully automatic verifiers which are able to determine whether code satisfies certain conditions without any help of the programmer. Unfortunately, this ease of use comes with a downside: these tools can only check certain properties for certain patterns of code. More ambitious verifications such as ensuring full functional correctness remains out of the scope of these automatic verifiers, since correctness only makes sense with respect to a specification, which needs to be provided by the programmer.
– Non-automatic tools are able to perform more thorough verifications (such as full functional correctness), but these require help from the programmer.

In practice, given a large body of code, it is often sufficient to check only automatically provable properties except for a small section of critical code, where a proof of full functional correctness is necessary. Neither of the above two options is then ideal. Our proposed solution is to combine the best of both worlds by using the following verification framework: at the base lies the non-automatic "core" verifier (in our case VeriFast), which will be responsible for performing the actual verification. To achieve this, it requires code to be fully annotated, but in return, it has the potential of checking for a wide variety of properties. On this base we build an automation layer, consisting of specialized tools able to automatically verify code for specific properties. Instead of just trusting the results of these tools, we require them to produce annotations understood by the core verifier.

A first advantage is that only the core verifier needs to be trusted. Indeed, in the end, all automatically produced annotations are fed back to the core verifier, so that unsoundnesses introduced by buggy automation tools will be caught.

A second advantage is that it allows us to choose which properties are checked for which parts of the code. For example, in order to verify a given program, we would start with unannotated code, on which we would apply an automatic verification tool, such as the shape analysis tool discussed in Section 3.3. This produces a number of annotations, which are fed to the core verifier. If verification succeeds, we know the application contains no memory-related errors.

Now consider the case where a certain function `foo` appears to be troublesome and shape analysis fails to verify it, which could mean that all other parts of the code which call this function also remain unverified. In order to deal with this problem the programmer can manually add the necessary annotations for `foo`, let the core verifier check them, and then re-apply the shape analysis tool, so that it can proceed with the rest of the code.

After the whole program has been proved memory-safe, one can proceed with the critical parts of the code where a proof of full functional correct is required. Thus, it makes an iterative incremental approach to verification possible where manually added annotations aid the automatic tools at performing their task.

In this paper, we presented preliminary experience gained in our work in progress towards this goal. Future work includes gaining additional experience with larger programs, gaining experience with the usability of an iterative infer-annotate process, and improving the power of the inference algorithm.

Acknowledgments. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

1. http://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf
2. Dafny: An Automatic Program Verifier for Functional Correctness. LPAR-16 (2010)
3. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and System Modeling 4(1) (2005)

4. Barnett, M., Chang, B.E., Deline, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO (2006)
5. Barnett, M., Leino, Schulte, W.: The Spec# Programming System: An Overview (2005)
6. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO (2005)
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic Execution with Separation Logic. In: APLAS (2005)
8. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: POPL (2008)
9. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
10. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: SAS. LNCS. The MIT Press, Cambridge (2007)
11. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOLs, pp. 23–42 (2009)
12. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: POPL (2007)
13. Distefano, D., O'Hearn, P.W., Yang, H.: A Local Shape Analysis based on Separation Logic. In: TACAS (2006)
14. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: OOPSLA (2008)
15. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: APLAS (2007)
16. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: Handbook of Philosophical Logic (1984)
17. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: APLAS (2010)
18. Leino, K., Rümmer, P.: A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In: TACAS (2010)
19. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: ESOP (2009)
20. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL (2005)
21. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS (2002)
22. Rosu, G., Ellison, C., Schulte, W.: Matching Logic: An Alternative to Hoare/Floyd Logic. In: AMAST (2010)
23. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
24. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In: CAV (2008)

Analyzing BGP Instances in Maude

Anduo Wang[1], Carolyn Talcott[2], Limin Jia[3], Boon Thau Loo[1], and Andre Scedrov[1]

[1] University of Pennsylvania
[2] SRI International
[3] Carnegie-Mellon University
{anduo,boonloo}@cis.upenn.edu, clt@csl.sri.com,
liminjia@cmu.edu, scedrov@math.upenn.edu

Abstract. Analyzing Border Gateway Protocol (BGP) instances is a crucial step in the design and implementation of safe BGP systems. Today, the analysis is a manual and tedious process. Researchers study the instances by manually constructing execution sequences, hoping to either identify an oscillation or show that the instance is safe by exhaustively examining all possible sequences. We propose to automate the analysis by using Maude, a tool based on rewriting logic. We have developed a library specifying a generalized path vector protocol, and methods to instantiate the library with customized routing policies. Protocols can be analyzed automatically by Maude, once users provide specifications of the network topology and routing policies. Using our Maude library, protocols or policies can be easily specified and checked for problems. To validate our approach, we performed safety analysis of well-known BGP instances and actual routing configurations.

1 Introduction

The Internet today runs on a complex routing protocol called the *Border Gateway Protocol* or *BGP* in short. BGP enables Internet-service providers (ISP) world-wide to exchange reachability information to destinations over the Internet, and simultaneously, each ISP acts as an autonomous system that imposes its own import and export routing policies on route advertisements exchanged among neighboring ISPs.

Over the past few years, there has been a growing consensus on the complexity and fragility of BGP routing. Even when the basic routing protocol converges, conflicting policy decisions among different ISPs have led to route oscillation and slow convergence. Several empirical studies [11] have shown that there are prolonged periods in which the Internet cannot reliably route data packets to specific destinations due to routing errors induced by BGP. In response, the networking community has proposed several alternative Internet architectures [18] and policy constraints (or "safety guidelines") that guarantee protocol convergence if universally adopted [8,6,16].

One of the key requirements for designing new routing architectures and policy guidelines is the ability to study BGP network instances. These instances can come in the form of small topology configurations (called "gadgets"), which serve as examples of safe systems, or counterexamples showing the lack of convergence. They can

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 334–348, 2011.
© IFIP International Federation for Information Processing 2011

also come from actual internal router (iBGP) and border gateway (eBGP) router configurations. Today, researchers and network operators analyze these instances by manually examining execution sequences. This process is tedious and error-prone.

The main contribution of this paper is that we present an automated tool for analyzing BGP instances, and thus relieve researchers and network operators of manual analysis. Our tool uses Maude [4], a language and tool based on rewriting logic. We encode in Maude the BGP protocol as a transition system driven by rewriting rules. Consequently, we can use the high-performance rewriting engine provided by Maude to analyze BGP instances automatically. Our tool can simulate execution runs, as well as exhaustively explore all execution runs for possible divergence.

More specifically, we developed a set of Maude libraries specifying a generalized path vector protocol that is common to all BGP instances. The generalized path vector protocol utilizes a set of unspecified routing policy functions. These unspecified functions serve as the interface for specific routing policies which are formalized as *Stable Path Problems (SPP)* [10]. To use our library, users only need to input the network topology and customize routing policies functions in the form of SPP. We illustrate the use of our tool by analyzing various BGP instances.

2 Background

2.1 BGP

BGP assumes a network model in which routers are grouped into various Autonomous Systems (*AS*) administrated by Internet Server Provider (*ISP*). An individual AS exchanges route advertisements with neighboring ASes using the *path-vector* protocol. Upon receiving a route advertisement, a BGP node may choose to accept or ignore the advertisement based on its *import policy*. If the route is accepted, the node stores the route as a possible candidate. Each node selects among all candidate routes the best route to each destination based on its local route rankings. Once a best route is selected, the node advertises it to its neighbors. A BGP node may choose to export only selected routes to its neighboring ASes based on its *export policy*.

BGP systems come in two flavors: external BGP (eBGP), which establishes routes between ASes; and internal BGP (iBGP), which distributes routes within an AS. At the AS-level, a BGP system can be viewed as a network of AS nodes running eBGP. Each AS is represented by one single *router* node in the network (its internal structure ignored), and its *network state* includes its neighbors (ASes), selected best path and a routing table. Route advertisements constitute the *routing messages* exchanged among them. Within an AS, a BGP system can be viewed as a network of two kinds of network nodes running iBGP: gateway routers and internal routers whose network states are similar to eBGP routers. iBGP allows internal routers to learn external routes (to destinations outside the AS) from gateway routers.

We model both eBGP and iBGP systems as network systems with two components: routing dynamics and routing policies. Routing dynamics specify how routers exchange routing messages, and how they update their network states accordingly. Routing policies are part of the static configuration of each router, by which the ISP operator expresses his local traffic interests and influences route decisions.

In our library, we adopt the use of Stable Paths Problems (SPP) [10] as the formal model of routing policies. An instance of the SPP S is a tuple (G, o, P, Λ), where G is a graph, o is a specific destination node [1], P is the set of permitted (usable) paths available for each node to reach o, and Λ is the ranking functions for each node. For each node v, λ^v is its ranking function, mapping its routes to natural numbers (ranks), and P^v are its permitted paths, the set of available paths to reach o. A path assignment is a function π that maps each network node $v \in V$ to a path $\pi(v) \in P^v$. A path assignment is *stable* if each node u selects a path $\pi(u)$ which is (1) the highest ranked path among its permitted paths, and (2) is consistent with the path chosen by the next-hop node. Consistency requires if $\pi(u) = (uv)P$ then for the next-hop node v, we must have $\pi(v) = P$. A solution to the SPP is a stable path assignment.

In this paper, we are interested in analyzing BGP convergence (safety) property in the SPP formalism. A BGP system converges and is said to be safe, if it produces stable routing tables, given any sequence of routing message exchanges. We can study BGP convergence by analyzing its SPP representation: SPP instance for a safe BGP system converges to a solution in all BGP executions. Note that, the existence of an SPP solution does not guarantee convergence.

For example, Figure 1 presents an SPP instance called the Disagree "gadget". The per-node ranking functions are $\lambda^1(\texttt{[n1 n2 n0]}) = 1$, $\lambda^1(\texttt{[n1 n0]}) = 2$, $\lambda^2(\texttt{[n2 n1 n0]}) = 1$, and $\lambda^2(\texttt{[n2 n0]}) = 2$. The permitted paths for each node are listed besides the corresponding node. The order in which the paths are listed is based on the ranking function: Nodes prefer higher ranked routes, e.g. node `n1` prefers route `[n1 n2 n0]` over `[n1 n0]`. Disagree has two stable path assignment solutions: `([n1 n2 n0], [n2 n0])` and `([n2 n1 n0], [n1 n0])`. However, Disagree is not guaranteed to convergebecause there exists an execution trace where route assignments keep oscillating. Consider the execution where node `n1` and `n2` update and exchange routing messages in a synchronized manner, and their network states oscillate between two unstable path assignments `([n1 n0]) ([n2 n0])` and `([n1 n2 n0] [n2 n1 n0])` forever.

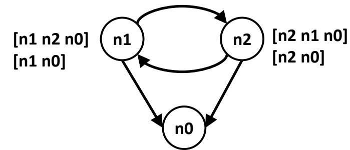


Fig. 1. Disagree Gadget

2.2 Rewriting Logic and Maude

Rewriting logic [13] is a logical formalism that is based on two simple ideas: states of a system can be represented as elements of an algebraic data type, and the behavior of a system can be given by transitions between states described by *rewrite rules*. By algebraic data type, we mean a set whose elements are constructed from atomic elements by application of constructors. Functions on data types are defined by equations that allow one to compute the result of applying the function. A rewrite rule has the form $t \Rightarrow t'$ *if* c where t and t' are patterns (terms possibly containing variables) and c is a condition (a boolean term). Such a rule applies to a system state s if t can be matched

[1] Assuming the Internet is symmetric, we can study its routing behavior by studying routing to a single destination.

to a part of s by supplying the right values for the variables, and if the condition c holds when supplied with those values. In this case the rule can be applied by replacing the part of s matching t by t' using the matching values for variables in t'.

Maude [4] is a language and tool based on rewriting logic [12]. Maude provides a high performance rewriting engine featuring matching modulo associativity, commutativity, and identity axioms; and search and model-checking capabilities. Given a specification S of a concurrent system, Maude can execute this specification and allows one to observe possible behaviors of the system. One can also use the search functionality of Maude to check if a state meeting a given condition can be reached during the system's execution. Furthermore, one can model-check S to check if a temporal property is satisfied, and if not, Maude will produce a counter example. Maude also supports object-oriented specifications that enable the modeling of distributed systems as a multiset of objects that are loosely coupled by message passing. As a result, Maude is particularly amenable to the specification and analysis of network routing protocols.

3 A Maude Library for Encoding BGP Protocols

This section presents our Maude library for analyzing BGP instances. This library provides specification of the protocol dynamics that are common to BGP instances, and defines a routing policy template in terms of the Stable Path Problem (SPP) so that network designers can customize it to analyze a specific instance. Our library also provides support for detecting route oscillation.

Table 1. Overview and Interpretation of Maude Library

BGP system	Maude interpretation
Network nodes	(Router) objects
Routing messages	Terms of type `Msg`
Global Network	Multiset of router objects and terms representing messages
Protocol dynamics	Local rewriting rules
Global network behaviors	Concurrent rewrites using local rules
Route oscillation support	A Logger object recording the histories of route assignments and rewriting rules updating the Logger object

Our library is organized into a hierarchy of Maude modules. Table 1 presents the correspondence between concepts in BGP protocols and the Maude code. We first show how our library represents a single network state of BGP system (Section 3.1). Then we explain how to capture the dynamic behavior of a local BGP router using rewrite rules. In doing so, the global network behaviors can be viewed as concurrent applications of the local rewriting rules (Section 3.2). Finally, we discuss the component in the library that detects route oscillation (Section 3.3).

3.1 Network State

A network state is represented by a multiset of network nodes (routers) and routing messages used by routers to exchange routing information. Each network node is represented by a Maude object, whose attributes consist of its routing table, best path and neighboring table. We omit the detailed Maude sort definitions, but provide an example encoding of the network node `n1` in Disagree gadget show in Figure 1 as follows.

```
[n1 : router |
      routingTable: (source: n1,dest: n0,pv:(n1 n0),metric: 2),
      bestPath: (source: n1,dest: n0,pv:(n1 n0),metric: 2),
      nb: (mkNeigh(n0,2) mkNeigh(n2,1))]
```

The constructor for a node is `[_:_|_,_,_]`. The first two elements (`n1:router`) specify the node's id `n1`, and its object class `router`. The next three elements are the attributes. At a given state, the routing table attribute constructed from `routingTable:_` contains `n1`'s current available routes. Each routing table entry stores the routing information for one particular next-hop. Here, the routing table attribute only contains one entry `(source: n1, dest: n0, pv:(n1 n0), metric: 2)`. This route is specified by its source (`source: n1`), destination (`dest: n0`), the path vector that contains the list of nodes along the path (`pv: (n1 n0)`), and the cost of the route (`metric: 2`). This route is also used for the best path attribute, constructed from `bestPath:_`, which contains `n1`'s current best path. The last attribute is the neighbor table, constructed from `nb:_`. To extract a node's local neighbor table from the network topology, we further introduce an operator `mkNeigh`. The first argument of `mkNeigh` is the identifier of the neighboring node, and the second argument the metric associated with the link to that node. Node `n1` has two neighbors, node `n0`, the cost to which is 2 (`mkNeigh(n0,2)`); and node `n2`, the cost to which is 1 (`mkNeigh(n2,1)`).

Besides router objects, the second part of a network state is routing messages in the network. Typically, network nodes exchange routing information by sending each other routing messages carrying newly-learned routes. In our library, a routing message is constructed from `sendPacket(_,_,_,_,_)`. For example, in the Disagree gadget, the initial routing message sent by node `n1` to its neighbor `n2` is represented by message term: `sendPacket(n1,n2,n0,2,n1 n0)`. This message carries `n1`'s routes to destination `n0` with path vector `n1 n0` at cost `2`. In general, the first two arguments of `sendPacket(_,_,_,_,_)` denote the sender's identifier (node `n1`), and the receiver's identifier (node `n2`) respectively. The rest of the arguments specify the identifier of the destination (node `n0`), the metric representing the cost of the route (2), and the path vector of the routing path (`n1 n0`).

3.2 Protocol Dynamics

We now show how to specify network system dynamics in Maude. By modeling a BGP system as a concurrent system consisting of router objects (and the routing messages), to specify the global BGP evolution, we only need to specify the local rewrite rules governing the state transition of each BGP router.

A BGP node's dynamics can be captured by various equivalent state transitions. To reduce search space in analysis, we adopt a one-state transition: for each BGP node `N`, when it receives routing messages from a neighbor `S`, `N` computes the new path from the received message, updates `N`'s routing table and re-selects best path accordingly, and

finally sends out routing messages carrying its new best path information if a different best path is selected. This state transition is encoded as a single rewrite rule of the following form:

```
rl [route-update] :
  sendPacket(S, N, D, C, PV)
  [ N : router | routingTable: RT, bestPath: Pb, nb: NB ]
 =>
if  (case 1) then best path re-selects (promotion)
else (if (case 2) then best path remains same
      else (if (case 3) then best path re-selection (withdraw)
            else error processing
            fi) fi) fi.
```

Here, `rl` is the identifier of this rule, and `route-update` is the name of this rule. Rule `rl` is fired when the left-hand side is matched; that is, when a node `N` consists of routingTable `RT`, bestPath `Pb`, and neighboring table `NB` receives a route advertisement message from neighbor `S`. The result of applying the rule is shown on the right-hand side: the routing message is consumed, and attributes of router `N` are updated. Based on the result of the re-selected bestPath attribute, there are three different cases for `N` to update its state as specified in the three branches. Next, we explain these three cases.

Best path promotion. In any case, node `N` needs to first compute the new path based on its neighbor `S`'s message asserting that `S` can reach `D` via a path `PV`. We define a function `newPath` that takes a routing message and the neighbor table as arguments, and returns the new path by first prepending `N` to the path announced by `S`, setting the new path attribute according to the local ranking function `lookUpRank`, and then imposing the import policy by modifying the path metric according to BGP routing policy configuration (`import` function). Here `import` and `lookUpRank` are unspecified routing policy functions. Together with `export` that we will introduce shortly, they constitute our library's specification interface for defining BGP routing policy. To specify a particular BGP instance's routing policy, the user only needs to specify `import`, `lookUpRank` and `export` accordingly.

The first branch (case 1) is specified below. The newly computed path is compared with the current bestPath `Pb`, if the new one is preferred over the old value `Pb`, the `bestPath` attribute will be updated to this new path. Furthermore, if the export policy allows, the new best path value will be re-advertised to all of `N`'s neighbors by sending them routing messages.

```
if getDest(newPath(sendPacket(S,N,D,C,PV),NB))==getDest(Pb) and
   prefer?(newPath(sendPacket(S,N,D,C,PV),NB),Pb)==true
then
([ N : router |
   routingTable: updatedRT(newPath(sendPacket(S,N,D,C,PV),NB),RT),
   bestPath: newPath(sendPacket(S,N,D,C,PV),NB),
   nb: NB ]
 multiCast(NB, export(newPath(sendPacket(S,N,D,C,PV),NB))))
```

Here the new state of `N` is obtained by updating the old `routingTable` attribute `RT` (`updateRT` function), and updating the `bestPath` attribute by setting it to the new value of `bestPath`. The `updateRT` function recursively checks the routing table, and for each next-hop entry, it either inserts the new path (`newPath(...)`) if no available route is presented; or replaces the old value with the new path. To complete the state transition, for all `N`'s neighbors, routing messages carrying the new path are generated by `multiCast` function.

To impose the export routing policy, before sending the new best path, `export` is applied to the new path to filter out the routes which are intended to be hidden from neighbors. Similar to `import`, `export` is to be instantiated by the user when analyzing a particular BGP instance. If the export routing policy prohibits the new path to be announced, `export` will transform it to `emptyPath`, which `multiCast` will not generate any message.

Best path remains the same. In the second branch (`case 2`), a new path `newPath(...)` is computed from the received message as before. However, the new path is no better than the current bestPath `Pb`. But the next-hop node of the new path and `Pb` are different, implying that the new path is just an alternative path [2] for `N` to reach the destination. As a result, the current `bestPath` value `Pb` is unchanged, and only the `routingTable` will be updated with this alternative path (`newPath(...)`). No routing messages will be generated:

```
if getDest(newPath(sendPacket(S,N,D,C,PV),NB))==getDest(Pb) and
   getNext(newPath(sendPacket(S,N,D,C,PV),NB))=/=getNext(Pb) and
   prefer?(Pb,newPath(sendPacket(S,N,D,C,PV),NB))==true
then
[ N : router |
   routingTable: updateRT(newPath(sendPacket(S,N,D,C,PV),NB),RT),
   bestPath: Pb,
   nb: NB ]
```

Best path withdraw. The same as in the second branch, in case 3, the newly computed path `newPath(...)` is worse than the current bestPath `Pb`, but it is now routed through the same next-hop `S` as current bestPath `Pb`. The fact that `S` now sends a less preferred path indicates that the previous learned route `Pb` is no longer available at `S`. Therefore, we need to withdraw `Pb` by dropping `Pb` from routing table, shown as follows:

```
if getDest(newPath(sendPacket(S,N,D,C,PV),NB))==getDest(Pb) and
   getNext(newPath(sendPacket(S,N,D,C,PV),NB))==getNext(Pb) and
   prefer?(Pb, newPath(sendPacket(S,N,D,C,PV),NB))==true
then
([ N : router |
   routingTable: updateRT(newPath(sendPacket(S,N,D,C,PV),NB),RT),
   newBest(newPath(sendPacket(S,N,D,C,PV),NB),
            updateRT(newPath(sendPacket(S,N,D,C,PV),NB),RT)),
   nb: NB ]
 multiCast(NB,export(newBest(newPath(sendPacket(S,N,D,C,PV),NB),
                     updateRT(newPath(sendPacket(S,N,D,C,PV),NB),
                              RT))))
```

Here, `updateRT` replaces (therefore removes) the outdated `Pb` with the new path (`newPath(...)`), and `newBest` function re-computes the best path from `newPath(...)` and the remaining paths in routing table. As in case 1, to complete the state transition, the newly selected best path is sent to its neighbors by `multiCast(...)`.

3.3 Route Oscillation Detection Support

Our library also provides extra definitions to help detect route oscillation. Our method is based on the observation that if route oscillation occurs during network system evolution, there is at least one *path assignment* (at a given state for a BGP system, we define

[2] Different next-hop implies the route is learned from a different neighbor.

the path assignment to be the collection of best paths currently selected by all network nodes) that is visited twice. Therefore, we use the following simple heuristic: we maintain a record of all path assignments for all visited states in BGP execution, and check for recurring path assignment. Note that a path assignment (best path attribute of `router` object) only constitutes a sub-set of the entire system state (the `router` objects attributes and routing messages), consequently our heuristic based on this partial system state can have false positives: our analysis may report a false route oscillation when two states are identical only in path assignments, but not the entire system states. Nevertheless, our heuristic is sound and is still helpful in detecting all potential route oscillation: when route oscillation occurs, a recurring path assignment state must occur.

More concretely, in our Maude library, we create a global *logger object* to keep track of the history of path assignments. For each snapshot of the network state, i.e. whenever a network node makes a local state transition and updates its best path attribute, the logger object is synchronized to create a new path assignment entry that corresponds to the updated best path. We then provide a function that checks for recurring entries in the list of visited path assignments, which can be used directly in Maude's exhaustive search to detect route oscillation.

Logger object. The global logger is represented by an object `pa` of `Logger` class which has one attribute `history`. At a given state, this attribute contains a list (history) of path assignments, each entry of which contains the snapshot of the network's collection of best paths in a visited state.An example logger object for the disagree gadget is the following:

```
{pa : Logger | history: ({[n1 n2 n0] [n2 n0]}
                         {[n1 n2 n0] [n2 n1 n0]}
                         {[n1 n2 n0] [n2 n0]}
                         {[n1 n0]   [n2 n0]})}
```

The above logger records four snapshots of the Disagree's best paths. For example, the first path assignment `{[n1 n2 n0] [n2 n0]}` denotes the network latest state where node `1`'s best path to `0` is `[n1 n2 n0]` and node `2`'s best path is `[n2 n0]`. And line 4 `{[n2 n0] [n2 n0]}` records Disagree's path assignment at its initial (oldest) state. Note that, this object content actually exhibits route oscillation (line 1 and line 3) described in Section 3.2.

Synchronized logging. To log all path assignment changes, we only need to slightly modify the single rewrite rule for route update, such that whenever the rule is fired to apply local state transition for some node, the global object `pa` is synchronized and its path assignment is updated to reflect changes in the local node's best path attribute, shown as follows:

```
rl [route-update-logging] :
  sendPacket(S, N, D, C, PV)
  [ N : router | routingTable: RT, bestPath: Pb, nb: NB ]
  { pa : Logger | history: HIS }
=>
*** first branch: bestPath re-selects (promotion)
if ... then ...
 { pa : Logger | history:
   historyAppend(updateAt(index(N),
          [getPV(newPath(sendPacket(S,N,D,C,PV),NB))],
          head(HIS)),HIS)})
else ... fi .
```

On the left-hand side, two objects: a router `N` and the global logger `pa` are matched to trigger the transition. As described in 3.2, in the first branch of route update where the node's best path attribute is set to `newPath(...)`, the logger `pa` updates its path assignment attribute as follows: First, it creates a new path assignment entry to record `newPath(...)` by function `updateAt(...)`. Then, the new entry `updateAt(...)` is inserted into the list of previous path assignments `HIS` by function `historyAppend`. Here, the new path assignment entry `updateAt(...)` is computed by updating the latest path assignment entry `head(HIS)` with `newPath(...)`. The rest of branches 2 and 3 are modified similarly.

Route oscillation detection. A network state is now a multiset of router objects, routing messages, and one global logger object. The function `detectCycle` detects re-curring path assignments, as follows:

```
eq detectCycle([ N : router | routingTable: RT,
                              bestPath: Pb,nb: NB] cf)
  = detectCycle (cf) .
eq detectCycle(message cf) = detectCycle (cf) .
eq detectCycle({ pa : Logger | history: HIS } cf)
  = containCycle? (HIS) .
```

The first two equations ignore router objects and routing messages in the network state, and the last equation examines logger `pa` by function `containCycle?` to check for recurring path assignment entries in `HIS`. We will revisit the use of `detectCycle` to search for route oscillation in Section 5.

4 Specifying BGP Instance

Given a BGP instance with its *network topology* and *routing policies*, we show how to specify the instance as a SPP in our library. We discuss examples for both eBGP and iBGP.

4.1 eBGP Instance

An eBGP instance can be directly modeled by an SPP instance $S = (G, o, P, \Lambda)$: G, o specifies the instance's *network topology*, and P, Λ specifies the resulting per-node route ranking function after applying the eBGP instance's *routing policies*. Our library provides Maude definitions for each SPP element.

Network topology. An eBGP instance's initial network state is generated from its network topology, which is represented by a list of network nodes and links. Our library declares two constants `top-Nodes` and `top-BGP` to represent network nodes and links. For example, to specify the topology of the Disagree gadget, the user defines `top-Nodes`, `top-BGP` as follows:

```
eq top-Nodes = n1 n2 .
eq top-BGP = (n1,n0 : 2) (n1,n2 : 1) (n2,n1 : 1) (n2,n0 : 2) .
```

Here, `n0` is the identifier of the destination node (o). Each link is associated with its cost. Based on the value of `top-Nodes` and `top-BGP` that are input by the user, our library automatically generates Disagree's initial state by `init-config` function:

```
eq gadget = init-config (top-Nodes, top-BGP) .
```

The resulting `gadget` is a network state which consists of the two network router objects `n1,n2`, the four initial routing messages, and the initial logger `pa`, as shown in Section 5.1. In this initial state, the three attributes of each network node – the routing table and best-path and neighbor tables are computed as follows: `init-config` parses the BGP links in network topology (`top-BGP`), for each link `(ni,nj : M)`, a new routing table entry for `nj` with cost `M` is created, and if `nj == n0`, then set `ni`'s best path to the one-hop direct path `ni n0`, and its routing tables containing this one-hop direct route; otherwise if there is no direct link from `ni` to `n0`, set `ni`'s best path and the routing table to `emptyPath`. Initial routing messages and logger `pa` are computed in a similar manner.

Routing policy. The route ranking function Λ and permitted paths P are the result of applying three BGP policies functions: `import`, `export` and `lookUpRank`. As we have discussed in Section 3, `import,export,lookUpRank` are three user-defined functions that serve as the specification interface for routing policies.

Functions `import` and `lookUpRank` are used to compute new routing paths from a neighbor's routing message: `import` filters out un-wanted paths, and `lookUpRank` assigns a rank to the remaining permitted paths. Note that the metric value `lookUpRank (N PV)` assigned by `lookUpRank` also determines the route's preference in route selection. `export` is used to filter out routes the router would like to hide.

As an example, the policy functions for Disagree are defined as follows.

```
eq export (P) = P . eq import (P) = P .
eq lookUpRank (n1 n2 n0) = 1 . eq lookUpRank (n1 n0) = 2 .
eq lookUpRank (n2 n1 n0) = 1 . eq lookUpRank (n2 n0) = 2 .
```

The first line says Disagree does not employ additional import/export policies. Whereas the second and third line asserts that Disagree's two nodes prefers routes through each other: For example the second line encodes node `n1`'s ranking policy that it prefers path `(n1 n2 n0)` (with higher rank 1) through `n2` over the direct path `(n1 n0)` (rank 2).

4.2 iBGP Instance

Our technical report [19] shows our SPP encoding of iBGP instances. The main differences between an iBGP and eBGP instances are: (1) iBGP network topology distinguishes between internal routers and gateway routers. Gateway routers runs eBGP to exchange routing information with (gateway routers of) other ISPs, while simultaneously running iBGP to exchange the external routing information with internal routers in the AS. (2) iBGP routing policy utilizes a separate IGP protocol to select best route. Internal to an AS, the ISP uses its own IGP protocol to compute shortest paths among all routers. The shortest path distance between internal routers and gateway routers are used in iBGP route selection: iBGP policy requires the internal routers to pick routes with shortest distance to its gateway router.

As a result, iBGP requires encoding two types of topologies: a *signaling* topology for gateway routers and internal routers to exchange routes within the AS, and a *physical*

topology on which the IGP protocol is running. Further, an additional *destination router* denoting the special SPP destination *o* is added as an external router which is connected with all gateway routers. In our library, we implement and run separately in Maude an IGP protocol (for computing all-pairs shortest paths) and pass the resulting shortest path distances to iBGP protocol.

5 Analysis

To analyze BGP instances, our library allows us to (1) execute the Maude specification to simulate possible execution runs; and (2) exhaustively search all execution runs to detect route oscillation.

5.1 Network Simulation

Network initialization. For any analysis, we need to first generate a BGP instance's initial network state. For a given BGP instance, we have shown how to generate its initial state `gadget` from its network topology and routing policy, as described in section 4. For example, the initial state generated for Disagree is as follows:

```
{pa : Logger | history:{[n1 n0] [n2 n0]}}
[n1 : router | routingTable: (source: n1, dest: n0,
                              pv:(n1 n0), metric: 2),
               bestPath: (source: n1, dest: n0,
                          pv:(n1 n0), metric: 2),
               nb: (mkNeigh(n0,2) mkNeigh(n2,1))]
[n2 : router | ... ]
sendPacket(n1,n0,n0,n2,n1 n0) sendPacket(n1,n2,n0,n2,n1 n0)
sendPacket(n2,n0,n0,n2,n2 n0) sendPacket(n2,n1,n0,n2,n2 n0)
```

This state consists of Disagree's initial logger object `pa` that holds the initial path assignment `[n1 n0] [n2 n0]`, two router objects `n1`,`n2`, and four initial routing messages.

Execution. Unlike many formal specification paradigms used in static network analysis, a Maude specification is executable. To explore *one* possible execution run from a given initial state `gadget`, we can directly use Maude's `rewrite` and `frewrite` (fair rewriting) commands. For example, we could tell Maude to execute the Disagree gadget with the following command: `frew gadget .` This command terminates and returns the following final state:

```
{pa : Logger |
    history: ({[n1 n0] [n2 n1 n0]} ... {[n1 n0] [n2 n0]})}
[n1 : router |...
    bestPath: (source: n1,dest: n0,pv:(n1 n0),metric: 2), ...]
[n2 : router |...
    bestPath: (source: n2,dest: n0,pv:(n2 n1 n0),metric: 1),...]
```

Note that this final state corresponds to one of the stable path assignments of Disagree described in Section 2, where node `n1` sets its best path to `[n1 n0]`, and node `n2` sets its best path to `[n2 n1 n0]`.

On the other hand, with the `rew` command which employs a different rewriting strategy, divergence scenario is simulated and route oscillation is observed in the simulation. This is because `frewrite` employs a depth-first position-fair rewriting strategy, while `rewrite` employs a left-most, outer-most strategy that coincides with the execution trace that leads to divergence.

5.2 Route Oscillation Detection

While Maude commands `frew/rew` explore a small portion of possible runs of the instance, the `search` command allows us to exhaustively explore the entire execution space. To exhaustively search BGP execution for route oscillation, we only need to first input the BGP instance's network topology and routing policy to generate the corresponding initial state, as described in Section 4; and then use the `search` command to automatically search for oscillation. For example, for Disagree, we run:

```
search [1] gadget =>+ X such that detectCycle(X) = true .
```

Here, `gadget` is Disagree's initial state, and `=>+ X` tells Maude to search for any reachable network state `X` such that at that state, the logger `pa` contains recurring path assignment (`detectCycle(X)=true`). `search` command exhaustively explores Disagree runs and returns with the first Disagree state that exhibits oscillation:

```
{pa : Logger | history: ({[n1 n2 n0] [n2 n0]}
                         {[n1 n2 n0] [n2 n1 n0]}
                         {[n1 n2 n0] [n2 n0]}
                         {[n1 n0] [n2 n0]})}
[n1 : router |...] [n2 : router |...] ...
```

Here, the resulting path assignment content in `pa` exhibits an oscillation (line 1, line 3).

In general, Maude allows us to exhaustively search for violation of a safety property `P` by running the following command:

```
search initialNetwork =>+ X:Configuration such that P(X) == false.
```

which tells Maude to exhaustively search for a network state `X` that violates `P` along all possible execution traces from the initial state `initialNetwork`. If Maude returns with `No solution`, we can conclude property `P` holds for all execution traces.

5.3 Case Studies

We have analyzed well-known eBGP instances, including good gadget, bad gadget, disagree [10]. In addition, we analyze two iBGP configuration instances: a 9-node iBGP gadget [7] that is known to oscillate, and a 25-node configuration randomly extracted from the Rocketfuel [17] dataset. Rocketfuel is a well-known dataset on actual iBGP configurations that are made available to the networking community. Given that an ISP has complete knowledge of its internal router configurations, the Rocketfuel experiment presents a practical use case for using our tool to check an actual BGP configuration instance for safety.

For each BGP instance, we simulate its possible executions using rewriting commands (*Simulation*), and check for route oscillation using exhaustive search (*Exhaustive*). We summarize our analysis results are as follows:

We have carried out these analysis on a laptop with 1.9 GB memory and 2.40GHz dual-cores running Debian 5.0.6. The version of Maude is Maude 2.4. While route oscillation detection explores the entire state space of the instance execution, the analysis time for rewriting based execution are measured for only one possible terminating execution (that converges to a stable path assignment).

Table 2. Summary of BGP analysis in Maude. In the first row, each entry shows the simulation time in milliseconds. In the second row, for each entry, the first value denotes exhaustive search time in milliseconds, the second denotes number of states explored, and the third on whether our tool determines the instances to be safe ("Yes") or unsafe ("No").

	Disagree	**Bad**	**Good**	**9-node iBGP**	**25-node iBGP**
Simulation	2	NA	4	20	31
Exhaustive	2,10,No	181,641,No	10997,37692,Yes	20063,52264,No	723827,177483,Yes

Here we summarize findings from our case studies. Single-trace simulation is helpful in finding permanent routing oscillation. When simulating the execute trace that diverges, Maude does not terminate (e.g., in executing Bad gadget [3]). However, simulation can miss temporary oscillations which are only manifested on a particular executing trace. When Maude terminates, single-trace simulation time increases when network size grows. On the other hand, exhaustive search always provides a solid safety proof. For instances of similar network size, the search time for a safe instance (`good`) is considerably longer than that of an unsafe instance (`bad`). For instances of different sizes, as network size grows, exhaustive search time grows exponentially. Nevertheless, even for the 25-node scenario, exhaustive search can be completed in 12 minutes. As future work, we are going to scale our analysis technique to larger networks.

6 Related Work

Maude is a widely used tool for a variety of protocol analysis. In addition to our use of Maude for analyzing BGP instances, there is also a huge literature of using Maude for other complex systems, such as security protocols [9] , real-time systems [14], and active networking [5].

Theorem proving and model checking techniques have been applied to formal verification of network protocols. For instance, in [3], a routing protocol standard is formalized in the SPIN model checker and HOL theorem prover, where SPIN is used to check convergence of small network instances, and HOL is used to generalize the convergence proof for arbitrary network instances. Their work focuses on basic intra-AS routing protocols such as the distance-vector protocol, and does not consider policy interactions that occur in inter-AS routing protocols such as BGP. However, while our proofs are automated by Maude's built-in simulation and exhaustive search capabilities, we are restricted to analyzing specific network instances. As future work, we plan to generalize our instance-based proofs towards more general properties on BGP stability, by leveraging Maude's connection with existing theorem provers such as PVS [15].

Arye *et al.* [2] has attempted a similar formalization of eBGP gadgets in SPP using the Alloy [1] tool. Our approach differs from theirs in the overall goal of the formalization: Ayre *et al.* uses Alloy to synthesize eBGP instances that exhibit certain behaviors

[3] Bad gadget always diverges and does not have any stable path assignment, therefore, when we simulate bad gadget with rewriting, Maude does not terminate, and we do not record the statistics.

such as divergence, whereas our approach takes an eBGP instance as input and analyzes it via simulation runs and exhaustive search. Our overall goal is to provide an easy-to-use library in Maude that eases the entire process of specifying and analyzing a BGP instance. Besides, in addition to eBGP gadgets, our library also supports iBGP instances and handles iBGP route ranking generation based on a separate IGP protocol 4.2.

7 Conclusion and Future Work

This paper presents our development of a Maude library for specifying and analyzing BGP instances. Our work aims to automate an important task for network designers when designing BGP protocols and safe policy guidelines. Our library uses Maude's object-based specification language and enables the user to easily generate Maude specification by only requiring them to define the network topology and routing policies. To validate the feasibility of our library, we explored a variety of well-known BGP gadgets and an actual BGP instance obtained from the Rocketfuel dataset, and demonstrated the use of Maude's analysis capabilities to detect possible divergence. All Maude code described in this paper is available at `http://netdb.cis.upenn.edu/discotec11`.

In addition to integrating our framework with the PVS theorem prover, our ongoing work includes: (1) more case studies on BGP instances and recent guidelines to explore the limits of our library, leading to possible extensions of our Maude library; and (2) releasing our tool for network designers to use.

Acknowledgment

The authors would like to thank Yiqing Ren and Wenchao Zhou for their valuable help in generating the iBGP instances (from the Rocketfuel dataset) used in our analysis. This research was supported by NSF grants IIS-0812270, CNS-0830949, CPS 0932397,Trusted Computing 0905607; AFOSR Grant No: FA9550-08-1-0352; NSF CAREER CNS-0845552; and Office of Naval Research Grant N00014-10-1-0365.

References

1. Alloy, `http://alloy.mit.edu/community/`
2. Arye, M., Harrison, R., Wang, R.: The Next 10,000 BGP Gadgets: Lightweight Modeling for the Stable Paths Problem. Princeton COS598D course project report
3. Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. J. ACM 49(4), 538–576 (2002)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: A High-Performance Logical Framework. Springer, Heidelberg (2007)
5. Denker, G., Meseguer, J., Talcott, C.: Formal specification and analysis of active networks and communication protocols: The maude experience. DARPA Information Survivability Conference and Exposition (2000)
6. Feamster, N., Johari, R., Balakrishnan, H.: Implications of Autonomy for the Expressiveness of Policy Routing. In: ACM SIGCOMM, Philadelphia, PA (August 2005)
7. Flavel, A., Roughan, M.: Stable and flexible iBGP. In: ACM SIGCOMM (2009)

8. Gao, L., Rexford, J.: Stable internet routing without global coordination. In: SIGMETRICS (2000)
9. Goodloe, A., Gunter, C.A., Stehr, M.-O.: Formal prototyping in early stages of protocol design. In: Proc. ACM WITS 2005 (2005)
10. Griffin, T.G., Shepherd, F.B., Wilfong, G.: The stable paths problem and interdomain routing. IEEE Trans. on Networking 10, 232–243 (2002)
11. Labovitz, C., Malan, G.R., Jahanian, F.: Internet Routing Instability. TON (1998)
12. Maude, `http://maude.cs.uiuc.edu/`
13. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
14. Ölveczky, P.C., Meseguer, J.: Real-time maude: A tool for simulating and analyzing real-time and hybrid systems. Electr. Notes Theor. Comput. Sci. 36 (2000)
15. PVS Specification and Verification System, `http://pvs.csl.sri.com/`
16. Schapira, M., Zhu, Y., Rexford, J.: Putting BGP on the right path: A case for next-hop routing. In: HotNets (October 2010)
17. Spring, N., Mahajan, R., Wetherall, D.: Measuring ISP topologies with Rocketfuel. In: SIGCOMM 2002 (2002)
18. Subramanian, L., Caesar, M., Ee, C.T., Handley, M., Mao, M., Shenker, S., Stoica, I.: HLP: A Next-generation Interdomain Routing Protocol. In: SIGCOMM (2005)
19. Wang, A., Talcott, C., Jia, L., Loo, B.T., Scedrov, A.: Analyzing BGP instances in maude. University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-11-08 (2011), `http://netdb.cis.upenn.edu/papers/bgpMaudeTR.pdf`

Author Index

Ábrahám, Erika 75
Acciai, Lucia 29
Asirelli, Patrizia 44

Becker, Bernd 75
Bogdoll, Jonathan 59
Boiten, Eerke 121
Boreale, Michele 29
Braitling, Bettina 75
Bravetti, Mario 90

Castagna, Giuseppe 1
Clarke, Dave 289

De Nicola, Rocco 29
Deng, Yuxin 106
Derrick, John 121
Dezani-Ciancaglini, Mariangiola 1
Di Giusto, Cinzia 90

Fantechi, Alessandro 44
Ferrer Fioriti, Luis María 59
Filipiuk, Piotr 138
Fratani, Séverine 153

Giachino, Elena 168
Gnesi, Stefania 44
Graf, Susanne 183
Grumbach, Stéphane 106

Hartmanns, Arnd 59
Hermanns, Holger 59
Herrmann, Peter 304
Honda, Kohei 228

Jacobs, Bart 319
Jansen, Nils 75
Jia, Limin 334

Kalyon, Gabriel 198
Kouzapas, Dimitrios 213, 228

Laneve, Cosimo 168
Le Gall, Tristan 198

Loo, Boon Thau 334
Lu, Tianxiang 244

Marchand, Hervé 198
Martos-Salgado, María 259
Massart, Thierry 198
Merz, Stephan 244, 274
Monin, Jean-François 106

Nielson, Flemming 138
Nielson, Hanne Riis 138

Padovani, Luca 1
Patrignani, Marco 289
Peled, Doron 183
Pérez, Jorge A. 90
Philippou, Anna 213
Piessens, Frank 319

Quinson, Martin 274
Quinton, Sophie 183

Rosa, Cristian 274
Rosa-Velardo, Fernando 259

Sangiorgi, Davide 289
Scedrov, Andre 334
Slåtten, Vidar 304
Smans, Jan 319

Talbot, Jean-Marc 153
Talcott, Carolyn 334
ter Beek, Maurice H. 44
Terepeta, Michał 138

Vogels, Frédéric 319

Wang, Anduo 334
Weidenbach, Christoph 244
Wimmer, Ralf 75

Yoshida, Nobuko 228

Zavattaro, Gianluigi 90