Holger Giese
Grigore Rosu (Eds.)

LNCS 7273

Formal Techniques for Distributed Systems

Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012
and 32nd IFIP WG 6.1 International Conference, FORTE 2012
Stockholm, Sweden, June 2012, Proceedings

ifip

Springer

Holger Giese Grigore Rosu (Eds.)

Formal Techniques for Distributed Systems

Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012
and 32nd IFIP WG 6.1 International Conference, FORTE 2012
Stockholm, Sweden, June 13-16, 2012
Proceedings

Springer

Volume Editors

Holger Giese
Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Strasse 2-3, 14482, Potsdam, Germany
E-mail: holger.giese@hpi.uni-potsdam.de

Grigore Rosu
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin, Urbana, IL 61801, USA
E-mail: grosu@illinois.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-30792-8 e-ISBN 978-3-642-30793-5
DOI 10.1007/978-3-642-30793-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012938642

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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 2012, the seventh International Federated Conferences on Distributed Computing Techniques (DisCoTec) took place in Stockholm, Sweden, during June 13–16. It was hosted and organized by KTH Royal Institute of Technology. The DisCoTec 2012 federated conference was one of the major events sponsored by the International Federation for Information Processing (IFIP) and it acted as an umbrella event for the following conferences:

– The 14th International Conference on Coordination Models and Languages (Coordination)
– The 12th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)
– The 14th Formal Methods for Open Object-Based Distributed Systems and 32nd Formal Techniques for Networked and Distributed Systems (FMOODS/-FORTE)

Together, these conferences cover the complete spectrum of distributed computing subjects ranging from theoretical foundations to formal specification techniques to systems research issues.

At a plenary session of the conferences, Schahram Dustdar of Vienna University of Technology and Bengt Jonsson of Uppsala University gave invited talks. There was also a poster session, and a session of invited talks from Swedish companies involved in distributed computing: Spotify, Peerialism, and severalnines.com. In addition to this, there were three workshops:

– The Third International Workshop on Interactions Between Computer Science and Biology (CS2BIO) with keynote talks by Jane Hillston (University of Edinburgh, UK) and Gianluigi Zavattaro (University of Bologna, Italy)
– The 5th Workshop on Interaction and Concurrency Experience (ICE) with keynote lectures by Marcello Bonsague (Leiden University, The Netherlands) and Ichiro Hasuo (Tokyo University, Japan)
– The 7th International Workshop on Automated Specification and Verification of Web Systems (WWV) with a keynote talk by José Luiz Fiadeiro (University of Leicester, UK)

I would like to thank the Program Committee Chairs of each conference and workshop for their effort. The organization of DisCoTec 2012 was only possible thanks to the dedicated work of the Publicity Chair Ivana Dusparic (Trinity College Dublin, Ireland), the Workshop Chair Rui Oliveira (Universidade do Minho, Portugal), the Poster Chair Sarunas Girdzijauskas (Swedish Institute of Computer Science, Sweden), the Industry-Track Chair György Dán (KTH Royal College of Technology, Sweden), and the members of the Organizing Committee from KTH Royal Institute of Technology and the Swedish Institute of

Computer Science: Amir H. Payberah, Fatemeh Rahimian, Niklas Ekström, Ahmad Al-Shishtawy, Martin Neumann, and Alex Averbuch. To conclude I want to thank the sponsorship of the International Federation for Information Processing (IFIP) and KTH Royal Institute of Technology.

June 2012 Jim Dowling

Preface

This volume contains the proceedings of the FMOODS/FORTE 2012 conference, a joint conference combining the 14th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS) and the 32nd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE) held during June 13–14, 2012, in Stockholm.

FMOODS/FORTE was hosted together with the 14th International Conference on Coordination Models and Languages (COORDINATION) and the 12th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS) by the federated conference event DisCoTec 2012, 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 FMOODS/FORTE 2012 program consisted of 16 regular papers which were selected by the Program Committee (PC) out of 42 submissions. Each submitted paper was evaluated on the basis of at least four detailed reviews from 31 PC members and 56 external reviewers. The final decision of acceptance was preceded by a thorough online discussion of the PC members. The selected papers constituted a strong program of stimulating, timely, and diverse research.

We are deeply indebted to the PC members and external reviewers for their hard and conscientious work in preparing 166 reviews. We thank Jim Dowling, 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, the reviewing process, and the production of the proceedings.

April 2012

Holger Giese
Grigore Rosu

Organization

Program Committee

Additional Reviewers

Abraham, Erika
Ancona, Davide
Arusoaie, Andrei
Axelsen, Holger Bock
Bae, Kyungmin
Becker, Steffen
Bocchi, Laura
Boström, Pontus
Ciobaca, Stefan
Combaz, Jacques
Delzanno, Giorgio
Duggan, Jerry
Erdogan, Gencer
Giachino, Elena
Gonnord, Laure
Griesmayer, Andreas
Göthel, Thomas
Haidar, May
Hallal, Hesham
Hansen, Hallstein A.
Heckel, Reiko
Helouet, Loic
Hermerschmidt, Lars
Kassios, Ioannis
Kurpick, Thomas
Lanese, Ivan
Legay, Axel
Lluch Lafuente, Alberto
Lund, Mass Soldal
Merro, Massimo
Merz, Stephan
Montesi, Fabrizio
Mostrous, Dimitris
Mueller, Klaus
Noll, Thomas
Omerovic, Aida
Phillips, Iain
Piterman, Nir
Refsdal, Atle
Ridge, Tom
Russo, Alejandro
Sammartino, Matteo
Schlatte, Rudolf
Schneider, Sven
Schremmer, Alexander
Seehusen, Fredrik
Solhaug, Bjørnar
Stolz, Volker
Summers, Alexander J.
Taylor, Ramsay
Timm, Nils
Ulrich, Andreas
Vogler, Walter
Willemse, Tim
Wortmann, Andreas
Ziegert, Steffen

Table of Contents

A Reversible Abstract Machine and Its Space Overhead 1
Michael Lienhardt, Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani

A Small Model Theorem for Rectangular Hybrid Automata Networks 18
Taylor T. Johnson and Sayan Mitra

Analysis of May-Happen-in-Parallel in Concurrent Objects 35
Elvira Albert, Antonio E. Flores-Montoya, and Samir Genaim

Behavioural Equivalences over Migrating Processes with Timers 52
Bogdan Aman, Gabriel Ciobanu, and Maciej Koutny

Checking Soundness of Business Processes Compositionally Using Symbolic Observation Graphs 67
Kais Klai and Jörg Desel

Beyond Lassos: Complete SMT-Based Bounded Model Checking for Timed Automata 84
Roland Kindermann, Tommi Junttila, and Ilkka Niemelä

Conformance Testing of Boolean Programs with Multiple Faults 101
Pavithra Prabhakar and Mahesh Viswanathan

Knowledge-Based Distributed Conflict Resolution for Multiparty Interactions and Priorities 118
Saddek Bensalem, Marius Bozga, Jean Quilbeuf, and Joseph Sifakis

Modelling Probabilistic Wireless Networks (Extended Abstract) 135
Andrea Cerone and Matthew Hennessy

Noninterference via Symbolic Execution 152
Dimiter Milushev, Wim Beck, and Dave Clarke

Defining Distances for All Process Semantics 169
David Romero Hernández and David de Frutos Escrig

Secure Multi-Execution through Static Program Transformation 186
Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas

Synchronous Interface Theories and Time Triggered Scheduling 203
Benoît Delahaye, Uli Fahrenberg, Thomas A. Henzinger, Axel Legay, and Dejan Ničković

TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs 219
Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha

Verification of Ad Hoc Networks with Node and Communication Failures 235
Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro

Verification of Timed Erlang Programs Using McErlang 251
Clara Benac Earle and Lars-Åke Fredlund

Author Index 269

A Reversible Abstract Machine and Its Space Overhead*

Michael Lienhardt[1], Ivan Lanese[1],
Claudio Antares Mezzina[2], and Jean-Bernard Stefani[3]

[1] Focus Team, University of Bologna/INRIA, Italy
{lienhard,lanese}@cs.unibo.it
[2] SOA Unit, FBK, Trento, Italy
mezzina@fbk.eu
[3] INRIA Grenoble-Rhône-Alpes, France
jean-bernard.stefani@inria.fr

Abstract. We study in this paper the cost of making a concurrent programming language reversible. More specifically, we take an abstract machine for a fragment of the Oz programming language and make it reversible. We show that the overhead of the reversible machine with respect to the original one in terms of space is at most linear in the number of execution steps. We also show that this bound is tight since some programs cannot be made reversible without storing a commensurate amount of information.

1 Introduction

There has recently been renewed interest in the notion of reversible computation [4] and new studies initiated on reversible programming languages [7,12,18]. This is sparked by the potential usefulness of reversible computation in a number of areas, including low-power computation [10], quantum computing [1] and building recoverable systems, typically using some form of undo [5].

In a previous paper [12], we have studied how to make the higher-order π-calculus (HOπ) reversible, i.e. how to equip this small paradigmatic concurrent higher-order language with a reduction semantics that comprises both forward steps (the usual reductions of HOπ) and backward ones, which precisely undo previous forward reductions. Specifically, if M, N are two reversible HOπ program configurations and M can reduce to N in one forward step, noted $M \twoheadrightarrow N$, then N can reduce to M in one backward step, noted $N \rightsquigarrow M$. The paper also presented a faithful encoding of reversible HOπ into HOπ, which can be seen as a first step towards understanding how to implement such a reversible language. This encoding, however, was quite wasteful in terms of resources, leading in

* This work has been partially supported by the French National Research Agency (ANR), project REVER n. ANR 11 INSE 007.

particular to a potential space overhead, compared to standard (forward only) HOπ executions, which can be exponential in the number computation steps.[1]

In this paper we initiate a study of the *implementation* of a *reversible higher-order concurrent language* and of its attendant costs. We start with a subset of the Oz kernel programming language [16]. This fragment of Oz, called μOz, is very close to HOπ, and its formal operational semantics is specified as a simple and rather classical *stack-based abstract machine*, itself directly inspired by the abstract machine of the Oz kernel programming language, which provides an interesting and well-known point of reference. We then define a reversible variant of μOz by means of an extended abstract machine, and we prove (i) that this new machine implements exactly the forward reductions of the initial one, and (ii) that it indeed implements reversibility for μOz, as characterized above for HOπ. Finally we study the *space overhead* that the reversible abstract machine adds to a forward execution compared to the same execution carried out by the μOz abstract machine. We prove that this overhead is *at worst linear* in the number of execution steps, and that this linear *upper bound is tight*: we show that some reversible μOz programs cannot execute with less than an amount of additional information – required to allow reversing their execution – that is linear in the number of execution steps. It would have been difficult to carry on a similar analysis directly in HOπ, since there is no largely accepted abstract machine for HOπ to be used as a reference. In fact, HOπ operational semantics is not precise enough on the use of memory space to act as a reference in such a context. To the best of our knowledge this is the first study of its kind. There is work investigating the time and space complexity of simulating irreversible computations by reversible ones e.g. [6,17], as well as recent work investigating the compilation of a reversible sequential language [2], but we do not know of work focusing as we do on the implementation or simulation of a reversible concurrent language and the analysis of its space costs.

Outline. The paper is organized as follows. Section 2 presents the syntax and abstract machine for μOz, the fragment of Oz we consider; Section 3 presents the reversible extension of the μOz abstract machine; Section 4 describes its

[1] To explain this without going into details of our reversible HOπ, let us just mention that a forward computation step in this calculus requires retaining in a so-called memory the message $a\langle P\rangle$ and the receiver process $a(X) \triangleright Q$ that participated in it (in HOπ and its reversible variant the only – forward – computation steps are message receipts). Thus the space overhead of a computation step in reversible HOπ compared to standard HOπ is at least $\|P\|$, the size of the payload of message $a\langle P\rangle$. Now consider the following recursive programs: $P = c(X) \triangleright P \mid a\langle X \mid X\rangle$ and $Q = a(X) \triangleright Q \mid c\langle X \mid X\rangle$. We have $a\langle R\rangle \mid P \mid Q \to P \mid Q \mid c\langle R \mid R\rangle$ so the space overhead of this first step starting from $a\langle R\rangle \mid P \mid Q$ is at least $\|R\|$. On the second step we have $P \mid Q \mid c\langle R \mid R\rangle \to P \mid Q \mid a\langle R \mid R \mid R \mid R\rangle$, so the space overhead of this second step is at least $2\|R\|$. By induction, one can see that the space overhead associated with making the program $a\langle R\rangle \mid P \mid Q$ reversible is at least $2^{n-1}\|R\|$, where n is the number of computation steps taken from the initial state $a\langle R\rangle \mid P \mid Q$.

S ::=		Statements
	skip	Empty statement
|	S_1 S_2	Sequential composition
|	**let** $x = v$ **in** S **end**	Variable declaration
|	**if** x **then** S_1 **else** S_2 **end**	Conditional statements
|	**thread** S **end**	Thread creation
|	**let** $x = c$ **in** S **end**	Procedure declaration
|	$\{ x\ x_1 \ldots x_n \}$	Procedure call
|	**let** $x =$ `NewPort` **in** S **end**	Port creation
|	$\{$ `Send` $x\ y \}$	Send on a port
|	**let** $x = \{$ `Receive` $y \}$ **in** S **end**	Receive from a port
v ::=	**true** | **false**	Simple values
c ::=	**proc** $\{ x_1 \ldots x_n \}$ S **end**	Procedure

Fig. 1. μOz Syntax

reversibility properties; Section 5 studies the overhead reversibility adds compared to the μOz abstract machine; Section 6 discusses related work; and Section 7 concludes the paper.

2 The μOz Language

In this section we present the syntax and semantics of μOz, a strict but non-trivial subset of Oz, to be extended with reversibility mechanisms in the next sections. We define the operational semantics of μOz by specifying an abstract machine for executing μOz programs which is directly inspired by the stack-based abstract machine in Chapter 13 of [16]. We refer to [16] for a description of the whole Oz language. We chose Oz because it came with a simple well documented abstract machine, and this subset of Oz because it was very close to HOπ. We do not know of a similarly simple stack-based abstract machine for HOπ in the literature. We could of course have come up with our own abstract machine for HOπ, but starting from on a non reversible abstract machine not devised by us is more appealing.

The syntax of μOz is in Figure 1. μOz is a higher-order language with thread-based concurrency and asynchronous communication via ports. For the sake of simplicity, the only values we consider in μOz are booleans, ports and closures. We eschew Oz logical variables in favor of simple immutable variables, i.e. read-only variables that are initialized at the time of their declaration. Dealing with the full Oz kernel programming language would not have posed more conceptual difficulties but would have obscured the technical details. The statements in μOz are fairly classical. Let us just point out that communication on a port, by means of send and receive operations, is asynchronous and by way of a FIFO queue. Variable declaration, procedure declaration, port creation and receiving are binders. Specifically, x is bound in **let** $x = v$ **in** S **end**, **let** $x = c$ **in** S **end**, **let** $x =$ `NewPort` **in** S **end**, and **let** $x = \{$ `Receive` $y \}$ **in** S **end**.

$T ::=$	Thread
$\langle\rangle$	Null thread
$\mid \langle S\ T\rangle$	Thread with statement
$U, V ::=$	Task
0	No task
$\mid T$	Thread
$\mid U \parallel V$	Parallel composition
$w ::= v \mid \xi$	Extended value
$\sigma ::=$	Store
0	Empty store
$\mid x = w$	Binding
$\mid \xi : c$	Closure
$\mid \xi : Q$	Port
$\mid \sigma \parallel \sigma'$	Conjunction
$Q ::=$	Port queue
$\perp$	Empty queue
$\mid x \mid Q; Q'$	Sequence of variables

Fig. 2. μOz Runtime Syntax

$$S =_\alpha S' \Rightarrow S \equiv S' \qquad \langle (S_1\ S_2)\ T\rangle \equiv \langle S_1\ \langle S_2\ T\rangle\rangle$$

$$Q_1; (Q_2; Q_3) \equiv (Q_1; Q_2)\ Q_3 \qquad Q; \perp \equiv \perp; Q$$

$$E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3 \qquad E \parallel 0 \equiv E \qquad E_1 \parallel E_2 \equiv E_2 \parallel E_1$$

Fig. 3. μOz Structural Congruence

Runtime Syntax. To describe the abstract machine defining μOz semantics we need a runtime syntax defining tasks and threads, used for statement execution, port queues, for communication, and the store. The runtime syntax of μOz is presented in Figure 2. Tasks are a parallel composition of threads. Threads are stacks of statements. The store (or heap) is a conjunction of bindings, closures, and ports (essentially implemented as named FIFO queues).

Structural Congruence. We consider tasks, threads, statements and the store up to a structural congruence relation. Structural congruence, written $\equiv$, is defined as the smallest congruence that validates the rules presented in Figure 3, where $=_\alpha$ stands for equality up to α-conversion. We write E for an *execution term*, i.e. either a task U or a store σ. The neutral element of concatenation for queues is $\perp$.

Reduction Rules. The μOz semantics is defined as a reduction relation, noted $\rightarrow$, between configurations of the form (U, σ). To follow Oz notation, the relation $\rightarrow$ is defined by a set of rules of the form below, specifying that (U, σ) reduces to (U', σ') if condition G is satisfied:

$$\text{R:skp} \qquad \frac{\langle \textbf{skip}\ T\rangle}{0} \,\Bigg\|\, \frac{T}{0}$$

$$\text{R:var} \qquad \frac{\langle \textbf{let}\ x = v\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{0} \,\Bigg\|\, \frac{\langle S\{^{x'}/_{x}\}\ T\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

$$\text{R:npr} \qquad \frac{\langle \textbf{let}\ x = c\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{0} \,\Bigg\|\, \frac{\langle S\{^{x'}/_{x}\}\ T\rangle}{x' = \xi \parallel \xi : c} \text{ if } x', \xi \text{ fresh}$$

$$\text{R:npt} \qquad \frac{\langle \textbf{let}\ x = \texttt{NewPort}\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{0} \,\Bigg\|\, \frac{\langle S\{^{x'}/_{x}\}\ T\rangle}{x' = \xi \parallel \xi : \bot} \text{ if } x', \xi \text{ fresh}$$

$$\text{R:if1} \qquad \frac{\langle \textbf{if}\ x\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\ T\rangle}{x = \textbf{true}} \,\Bigg\|\, \frac{\langle S_1\ T\rangle}{x = \textbf{true}}$$

$$\text{R:if2} \qquad \frac{\langle \textbf{if}\ x\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\ T\rangle}{x = \textbf{false}} \,\Bigg\|\, \frac{\langle S_2\ T\rangle}{x = \textbf{false}}$$

$$\text{R:nth} \qquad \frac{\langle \textbf{thread}\ S\ \textbf{end}\ T\rangle}{0} \,\Bigg\|\, \frac{T \parallel \langle S\ \langle\rangle\rangle}{0}$$

$$\text{R:pc} \qquad \frac{\langle \{\ x\ x_1 \ldots x_n\ \}\ T\rangle}{x = \xi \parallel \xi : \textbf{proc}\ \{\ y_1 \ldots y_n\ \}\ S\ \textbf{end}} \,\Bigg\|\, \frac{\langle S\{^{x_1}/_{y_1}\} \ldots \{^{x_n}/_{y_n}\}\ T\rangle}{x = \xi \parallel \xi : \textbf{proc}\ \{\ y_1 \ldots y_n\ \}\ S\ \textbf{end}}$$

$$\text{R:snd} \qquad \frac{\langle \{\ \texttt{Send}\ x\ y\ \}\ T\rangle}{x = \xi \parallel \xi : Q} \,\Bigg\|\, \frac{T}{x = \xi \parallel \xi : y; Q}$$

$$\text{R:rcv} \qquad \frac{\langle \textbf{let}\ x = \{\ \texttt{Receive}\ y\ \}\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{y = \xi \parallel \xi : Q; z \parallel z = w} \,\Bigg\|\, \frac{\langle S\{^{x'}/_{x}\}\ T\rangle}{y = \xi \parallel \xi : Q \parallel z = w \parallel x' = w} \text{ if } x' \text{ fresh}$$

Fig. 4. μOz Abstract Machine Semantics

$$\frac{U}{\sigma} \,\Bigg\|\, \frac{U'}{\sigma'} \text{ if } G$$

The reduction relation $\rightarrow$ is closed under evaluation contexts, and is defined modulo structural congruence. We capture this with the notion of evaluation-closed relation.

Definition 1. *A relation R between configurations is said to be* evaluation-closed *if it satisfies the following inference rules:*

$$\left(\frac{U}{\sigma} \,\Bigg\|\, \frac{U'}{\sigma'} \Rightarrow \frac{U \parallel U_k}{\sigma \parallel \sigma_k} \,\Bigg\|\, \frac{U' \parallel U_k}{\sigma' \parallel \sigma_k} \right)$$

$$\begin{pmatrix} U_1 \equiv U_1' \wedge U_2 \equiv U_2' \\ \sigma_1 \equiv \sigma_1' \wedge \sigma_2 \equiv \sigma_2' \end{pmatrix} \wedge \frac{U_1}{\sigma_1} \,\Bigg\|\, \frac{U_2}{\sigma_2} \Rightarrow \frac{U_1'}{\sigma_1'} \,\Bigg\|\, \frac{U_2'}{\sigma_2'}$$

Intuitively, the first rule corresponds to closure under evaluation contexts (parallel composition of threads), while the second rule is closure under structural congruence.

We define the *reduction relation* $\rightarrow$ to be the smallest evaluation-closed relation on configurations that validates the rules in Figure 4.

Rule R:var creates a new binding $x' = v$ in the store. Also, x' is substituted for x in the scope S to avoid variable capture. Rule R:npr is similar, but deals with closures. The closure c is assigned a fresh name ξ, and the name is bound to the fresh variable x'. Similarly, rule R:npt creates a new port, and initializes the queue to $\bot$. Rule R:nth creates a new thread. Rule R:pc executes a procedure call, substituting the actual parameter variables for the formal ones in the code to be executed. Rule R:snd performs a send, by enqueuing a variable in the corresponding queue. Rule R:rcv performs a receive, dequeuing the corresponding element z and fetching its value w. The value w is assigned to the fresh variable x' that substitutes the formal variable x.

3 A Reversible Abstract Machine for μOz

In this section we extend the μOz abstract machine presented in Section 2 to make it reversible. Since our language is concurrent, we aim at defining a causally consistent form of reversibility [7], where the execution can go back to any state that could have been reached in the forward execution by changing the order of execution of concurrent steps. We start by extending μOz runtime syntax.

μOz Reversible Runtime Syntax. The runtime syntax used in the definition of μOz reversible semantics is in Figure 5.

With respect to μOz runtime syntax (Figure 2) we now consider extended stacks C, which may also contain the scope delimiter **esc** as a statement. This is needed to reverse the let and if statements, as well as procedure calls. Consider for instance the case of procedure calls: **esc** is needed to find out where the procedure code ends (the body of the procedure should be removed to reverse the call) and the caller code begins (the caller code should be preserved).

Threads now have a name t (which is unique) and a history H, and execute an extended statement stack C. The history stores information about executed statements. Note that sent variables are actually stored in the queue, not in the history. Also, for an if statement just the discarded branch has to be stored, since the other one is available in the thread code. History is needed also inside ports, to remember the order of communications.

Structural congruence. Structural congruence has to be extended from threads T to extended stacks C, and from store σ to store θ. The rules are however the same as in Figure 3.

Reduction Rules. We define the two reduction relations $\twoheadrightarrow$ (forward execution) and $\rightsquigarrow$ (backward execution) to be the smallest evaluation-closed relations that validate the rules presented in Figure 6 and Figure 7, respectively. In the rules and in the following, we will abbreviate a sequence (possibly empty) of variables

$C ::=$		Extended stack
	$\langle\rangle$	Null stack
	$\mid \langle S\ C\rangle$	Stack with statement
	$\mid \langle \mathbf{esc}\ C\rangle$	Stack with scope
$M, N ::=$		Task
	0	No task
	$\mid t[H]C$	Thread
	$\mid M \parallel N$	Parallel composition
$H ::=$		History
	$\perp$	Empty history
	$\mid \mathbf{skip}$	Executed a skip
	$\mid H\ H'$	Sequential composition
	$\mid \uparrow x$	Sent on port x
	$\mid \downarrow x(y)$	Received y from port x
	$\mid *x$	Created variable x
	$\mid \{\ x\ x_1 \ldots x_n\ \}$	Called procedure x
	$\mid *t$	Created thread t
	$\mid \mathbf{if}(x)S$	Executed an if statement
	$\mid \mathbf{esc}$	Scope statement
$\theta ::=$		Store
	0	Empty store
	$\mid x = w$	Binding
	$\mid \xi : c$	Closure
	$\mid \xi : K \mid K_h$	Port
	$\mid \theta \parallel \theta'$	Parallel composition
$K ::=$		Message queue
	$\perp$	Empty queue
	$\mid t{:}x \quad \mid \quad K; K'$	Messages
$K_h ::=$		Queue history
	$\perp$	Empty queue history
	$\mid t{:}x, t'; K_h$	Message in queue history

Fig. 5. μOz Reversible Runtime Syntax

$x_1 \ldots x_n$ as $(x_i)_1^n$ (with $n \geq 0$). Similarly, we will abbreviate a sequence of substitutions $\{^{x_1}/_{y_1}\} \ldots \{^{x_n}/_{y_n}\}$ as$(\{^{x_i}/_{y_i}\})_1^n$. We define the reduction relation $\rightarrow_{vm}$ as $\rightarrow_{vm} = \twoheadrightarrow \cup \rightsquigarrow$.

Rule R:fw:var stores $*x'$ in the history, meaning that x' has been used as fresh variable, and uses the scope delimiter **esc** to recall the scope of the binding. Rules R:fw:npr and R:fw:npt are similar. In rule R:fw:npt the created queue comes with an empty history $\perp$. Rules R:fw:if1 and R:fw:if2 store the branch discarded by the choice in the history. In rule R:fw:nth the new thread is given a fresh name t' and an empty history $\perp$. The fresh name t' is also stored in the history of the creating thread. In rule R:fw:pc the name and actual parameters of the invoked procedure are stored in the history. In rule R:fw:snd we store in the queue the name of the thread sending the value, to avoid that a different thread takes the value when rolling back. In rule R:fw:rcv the read value is also kept in the

R:fw:skp
$$\frac{t[H]\langle \mathbf{skip}\ C\rangle}{0} \;\Big\|\; \frac{t[H\ \mathbf{skip}]C}{0}$$

R:fw:var
$$\frac{t[H]\langle \mathbf{let}\ x = v\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{0} \;\Big\|\; \frac{t[H\ \ast x']\langle S\{^{x'}\!/_{x}\}\ \langle \mathbf{esc}\ C\rangle\rangle}{x' = v}\ \text{if } x' \text{ fresh}$$

R:fw:npr
$$\frac{t[H]\langle \mathbf{let}\ x = c\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{0} \;\Big\|\; \frac{t[H\ \ast x']\langle S\{^{x'}\!/_{x}\}\ \langle \mathbf{esc}\ C\rangle\rangle}{x' = \xi \parallel \xi : c}\ \text{if } x', \xi \text{ fresh}$$

R:fw:npt
$$\frac{t[H]\langle \mathbf{let}\ x = \mathtt{NewPort}\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{0} \;\Big\|\; \frac{t[H\ \ast x']\langle S\{^{x'}\!/_{x}\}\ \langle \mathbf{esc}\ C\rangle\rangle}{x' = \xi \parallel \xi : \bot|\bot}\ \text{if } x', \xi \text{ fresh}$$

R:fw:if1
$$\frac{t[H]\langle \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ C\rangle}{x = \mathbf{true}} \;\Big\|\; \frac{t[H\ \mathbf{if}(x)S_2]\langle S_1\ \langle \mathbf{esc}\ C\rangle\rangle}{x = \mathbf{true}}$$

R:fw:if2
$$\frac{t[H]\langle \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ C\rangle}{x = \mathbf{false}} \;\Big\|\; \frac{t[H\ \mathbf{if}(x)S_1]\langle S_2\ \langle \mathbf{esc}\ C\rangle\rangle}{x = \mathbf{false}}$$

R:fw:nth
$$\frac{t[H]\langle \mathbf{thread}\ S\ \mathbf{end}\ C\rangle}{0} \;\Big\|\; \frac{t[H\ \ast t']C \parallel t'[\bot]\langle S\ \langle\rangle\rangle}{0}\ \text{if } t' \text{ fresh}$$

R:fw:pc
$$\frac{t[H]\langle\{\ x\ (x_i)_1^n\ \}\ C\rangle}{x = \xi \parallel \xi : \mathbf{proc}\ \{\ (y_i)_1^n\ \}\ S\ \mathbf{end}} \;\Big\|\; \frac{t[H\ \{\ x\ (x_i)_1^n\ \}]\langle S(\{^{x_i}\!/_{y_i}\})_1^n\ \langle \mathbf{esc}\ C\rangle\rangle}{x = \xi \parallel \xi : \mathbf{proc}\ \{\ (y_i)_1^n\ \}\ S\ \mathbf{end}}$$

R:fw:snd
$$\frac{t[H]\langle\{\ \mathtt{Send}\ x\ y\ \}\ C\rangle}{x = \xi \parallel \xi : K|K_h} \;\Big\|\; \frac{t[H\ \uparrow x]C}{x = \xi \parallel \xi : t\!:\!y; K|K_h}$$

R:fw:rcv
$$\frac{t[H]\langle \mathbf{let}\ y = \{\ \mathtt{Receive}\ x\ \}\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{\theta \parallel \xi : K; t'\!:\!z|K_h} \;\Big\|\; \frac{t[H\ \downarrow x(y')]\langle S\{^{y'}\!/_{y}\}\ \langle \mathbf{esc}\ C\rangle\rangle}{\theta \parallel \xi : K|t'\!:\!z, t; K_h \parallel y' = w}$$
$$\text{if } y' \text{ fresh } \wedge\ \theta \triangleq x = \xi \parallel z = w$$

R:fw:scp
$$\frac{t[H]\langle \mathbf{esc}\ C\rangle}{0} \;\Big\|\; \frac{t[H\ \mathbf{esc}]C}{0}$$

Fig. 6. μOz Abstract Machine Forward Semantics

queue history, with information on who read it. Rule R:fw:scp is new, allowing to record the scope delimiter **esc** in the history.

The backward rules in Figure 7 are in one to one correspondence with the forward ones, and use the stored information to get back to the original state. Notably, rules R:bk:var, R:bk:npr, R:bk:npt and R:bk:rcv go back to a term which is not the starting one, but which is equivalent up to α-conversion. Also, rules R:bk:var, R:bk:npr, R:bk:npt, R:bk:if1, R:bk:if2, R:bk:pc and R:bk:rcv exploit the scope delimiter **esc** to identify the scope of the statement to be reversed. Note that the occurrence of **esc** in the rule is always matched by the nearest occurrence in the term. In rule R:bk:nth the $\bot$ in the history of the second thread ensures that the thread is rolled-back before its creation is rolled-back. The same happens for ports in R:bk:npt.

R:bk:skp $$\begin{array}{c||c} t[H\ \mathbf{skip}]C & t[H]\langle \mathbf{skip}\ C\rangle \\ \hline 0 & 0 \end{array}$$

R:bk:var $$\begin{array}{c||c} t[H\ \ {*}x]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle & t[H]\langle \mathbf{let}\ x = v\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle \\ \hline x = v & 0 \end{array}$$

R:bk:npr $$\begin{array}{c||c} t[H\ \ {*}x]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle & t[H]\langle \mathbf{let}\ x = c\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle \\ \hline x = \xi \parallel \xi : c & 0 \end{array}$$

R:bk:npt $$\begin{array}{c||c} t[H\ \ {*}x]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle & t[H]\langle \mathbf{let}\ x = \texttt{NewPort}\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle \\ \hline x = \xi \parallel \xi : \bot|\bot & 0 \end{array}$$

R:bk:if1 $$\begin{array}{c||c} t[H\ \mathbf{if}(x)S_2]\langle S_1\ \langle \mathbf{esc}\ C\rangle\rangle & t[H]\langle \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ C\rangle \\ \hline x = \mathbf{true} & x = \mathbf{true} \end{array}$$

R:bk:if2 $$\begin{array}{c||c} t[H\ \mathbf{if}(x)S_1]\langle S_2\ \langle \mathbf{esc}\ C\rangle\rangle & t[H]\langle \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ C\rangle \\ \hline x = \mathbf{false} & x = \mathbf{false} \end{array}$$

R:bk:nth $$\begin{array}{c||c} t[H\ \ {*}t']C \parallel t'[\bot]\langle S\ \langle\rangle\rangle & t[H]\langle \mathbf{thread}\ S\ \mathbf{end}\ C\rangle \\ \hline 0 & 0 \end{array}$$

R:bk:pc $$\begin{array}{c||c} t[H\ \{\ x\ (x_i)_1^n\ \}]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle & t[H]\langle\{\ x\ (x_i)_1^n\ \}\ C\rangle \\ \hline 0 & 0 \end{array}$$

R:bk:snd $$\begin{array}{c||c} t[H\ \ \uparrow x]C & t[H]\langle\{\ \texttt{Send}\ x\ y\ \}\ C\rangle \\ \hline x = \xi \parallel \xi : t\!:\!y;K|K_h & x = \xi \parallel \xi : K|K_h \end{array}$$

R:bk:rcv $$\begin{array}{c||c} t[H\ \ \downarrow x(z)]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle & t[H]\langle \mathbf{let}\ z = \{\ \texttt{Receive}\ x\ \}\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle \\ \hline z = w \parallel x = \xi \parallel \xi : K|t'\!:\!y,t;K_h & x = \xi \parallel \xi : K;t'\!:\!y|K_h \end{array}$$

R:bk:scp $$\begin{array}{c||c} t[H\ \mathbf{esc}]C & t[H]\langle \mathbf{esc}\ C\rangle \\ \hline 0 & 0 \end{array}$$

Fig. 7. μOz Abstract Machine Backward Semantics

4 Properties of the Reversible Abstract Machine

In this section we prove that the forward and backward semantics actually define a causally consistent reversible version of μOz, following the lines of [7,12].

First of all we show that the reversible language is a conservative extension of μOz, i.e. forward computations of our reversible machine are indeed a decorated version of the μOz reductions. To this end we define a function, called **erase**, which takes a reversible μOz configuration and erases all the information needed only for reversibility purposes. The function is defined in Figure 8.

Lemma 1. *Let (M,θ) and (N,θ') be two reversible configurations such that $(M,\theta) \twoheadrightarrow (N,\theta')$. We have either*

$$\mathbf{erase}((M,\theta)) \rightarrow \mathbf{erase}((N,\theta'))\ \ \textit{or}\ \ \mathbf{erase}((M,\theta)) = \mathbf{erase}((N,\theta'))$$

Proof. By case analysis on the reduction rule applied. The equality case holds for rule R:fw:scp only. $\square$

$$\mathbf{erase}(\langle\rangle) \triangleq \langle\rangle \qquad \mathbf{erase}(\langle S\ C\rangle) \triangleq \langle S\ \mathbf{erase}(C)\rangle \qquad \mathbf{erase}(\langle \mathbf{esc}\ C\rangle) \triangleq \mathbf{erase}(C)$$

$$\mathbf{erase}(0) \triangleq 0 \qquad \mathbf{erase}(t[H]C) \triangleq \mathbf{erase}(C)$$

$$\mathbf{erase}(M \parallel N) \triangleq \mathbf{erase}(M) \parallel \mathbf{erase}(N)$$

$$\mathbf{erase}(x = w) \triangleq x = w \qquad \mathbf{erase}(\xi : c) \triangleq \xi : c \qquad \mathbf{erase}(\xi : K|K_h) \triangleq \xi : \mathbf{erase}(K)$$

$$\mathbf{erase}(\theta \parallel \theta') \triangleq \mathbf{erase}(\theta) \parallel \mathbf{erase}(\theta')$$

$$\mathbf{erase}(\bot) \triangleq \bot \qquad \mathbf{erase}(t{:}x) \triangleq x \qquad \mathbf{erase}(K;K') \triangleq \mathbf{erase}(K);\mathbf{erase}(K')$$

Fig. 8. Function Erasing Reversibility Information

Lemma 2. *Let (T, σ) and (M, θ) be a simple and a reversible configuration such that* $\mathbf{erase}((M, \theta)) \to (T, \sigma)$. *Then, there exists a reversible configuration (N, θ') with* $\mathbf{erase}((N, \theta')) = (T, \sigma)$ *and such that $(M, \theta) \twoheadrightarrow^+ (N, \theta')$ where $\twoheadrightarrow^+$ denotes one or more applications of $\twoheadrightarrow$.*

Proof. By case analysis on the rule applied. Each rule is matched by the corresponding reversible rule. Auxiliary steps using rule R:fw:scp may be needed. □

The Loop Lemma below ensures that every step can be reversed.

Lemma 3 (Loop Lemma). *For all configurations (M, θ) and (M', θ') the following double implication holds:*

$$(M, \theta) \twoheadrightarrow (M', \theta') \quad \Leftrightarrow \quad (M', \theta') \rightsquigarrow (M, \theta)$$

Proof. By case analysis on the used rule. Each rule R:fw:* is reversed by the corresponding rule R:bk:* and viceversa. Relevant issues have been highlighted in the description of the backward rules. □

We call *transition* a pair of the form $(M, \theta) \to_{vm} (N, \theta')$, where (M, θ) and (N, θ') are two configurations. We indicate (M, θ) as the *source* of the transition and (N, θ') as the target of the transition. Two transitions are said to be *coinitial* if they have the same source, *cofinal* if they have the same target, and *composable* if the target of the first is the source of the second. A sequence of pairwise composable transitions is called a *trace*. We let r and its decorated variants range over transitions, τ and its decorated variants range over traces. If r is a transition, we set $r_\bullet$ as its inverse. A transition $(M, \theta) \twoheadrightarrow (N, \theta')$ is said to be forward, while a transition $(M, \theta) \rightsquigarrow (N, \theta')$ is said to be backward. Notions of target, source and composability extend naturally to traces. We denote with $\epsilon_{(M,\theta)}$ the empty trace with source (M, θ), and with $\tau_1;\tau_2$ the composition of two composable traces τ_1 and τ_2.

In order to show that reversibility is causally consistent, we now define the notion of concurrency in our language.

Definition 2 (Concurrent transitions)
Two coinitial transitions $r_1 = (M, \theta) \rightarrow_{vm} (M_1, \theta_1)$ and $r_2 = (M, \theta) \rightarrow_{vm} (M_2, \theta_2)$ are said to be concurrent *unless r_1 and r_2:*

- *are executed by the same thread;*
- *are sends or a send and an undo of a send to the same queue;*
- *are receives and/or undo of receives from the same queue;*
- *are an action of a thread and the undo of the thread creation;*
- *are a use of a variable and the undo of its creation;*
- *are a receive on a queue with one element and the undo of its send.*

The definition of concurrent transitions enables the following result.

Lemma 4 (Square lemma). *Given two coinitial concurrent transitions $r_1 = (M, \theta) \rightarrow_{vm} (M_1, \theta_1)$ and $r_2 = (M, \theta) \rightarrow_{vm} (M_2, \theta_2)$, there exist two cofinal transitions $r_2/r_1 = (M_1, \theta_1) \rightarrow_{vm} (N, \theta_3)$ and $r_1/r_2 = (M_2, \theta_2) \rightarrow_{vm} (N, \theta_3)$.*

Proof. By case analysis on the form of transitions r_1 and r_2. $\square$

We finally define the notion of causal equivalence between traces, noted $\asymp$, as the least equivalence relation between traces closed under composition that obeys the following rules (where r is forward):

$$r_1; r_2/r_1 \asymp r_2; r_1/r_2 \qquad r; r_\bullet \asymp \epsilon_{\mathtt{source}(r)} \qquad r_\bullet; r \asymp \epsilon_{\mathtt{target}(r)}$$

Following the same proof schema as that used in [7,12], we can now prove:

Theorem 1 (Causal consistency). *Let τ_1 and τ_2 be coinitial traces, then $\tau_1 \asymp \tau_2$ if and only if τ_1 and τ_2 are cofinal.*

Informally, this means that, if we consider different computations from the same starting process, we never distinguish processes obtained by causal equivalent computations, while we always distinguish processes obtained by computations which are not causally equivalent, since they should have different backward behaviors. Together, the loop lemma and the causal consistency theorem express that our reversible machine correctly implements step wise reversibility (loop lemma), and that it does so with the maximum amount of flexibility by not distinguishing, as far as reversing a computation is concerned, between causally equivalent traces (causal consistency theorem).

5 Memory Overhead

We turn now to the analysis of implementation costs. We prove two results in this section. First, we show that the space overhead imposed by our reversible abstract machine compared to the standard μOz machine is linear in the number of steps of a given computation. Second, we show that this cannot be improved, as far as the order of magnitude is concerned, since the amount of information required to reverse certain μOz programs is indeed linear in the number of steps in their executions.

$$\mathbf{size}(\mathbf{skip}) \triangleq 1 \qquad \mathbf{size}(S_1\ S_2) \triangleq \mathbf{size}(S_1) + \mathbf{size}(S_2)$$
$$\mathbf{size}(\mathbf{let}\ x = v\ \mathbf{in}\ S\ \mathbf{end}) \triangleq 3 + \mathbf{size}(S)$$
$$\mathbf{size}(\mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}) \triangleq 2 + \mathbf{size}(S_1) + \mathbf{size}(S_2)$$
$$\mathbf{size}(\mathbf{thread}\ S\ \mathbf{end}) \triangleq 1 + \mathbf{size}(S)$$
$$\mathbf{size}(\mathbf{let}\ x = c\ \mathbf{in}\ S\ \mathbf{end}) \triangleq 2 + \mathbf{size}(c) + \mathbf{size}(S)$$
$$\mathbf{size}(\mathbf{proc}\ \{\ x_1 \dots x_n\ \}\ S\ \mathbf{end}) \triangleq n + 1 + \mathbf{size}(S) \qquad \mathbf{size}(\{\ x\ x_1 \dots x_n\ \}) \triangleq n + 2$$
$$\mathbf{size}(\mathbf{let}\ x = \texttt{NewPort}\ \mathbf{in}\ S\ \mathbf{end}) \triangleq 3 + \mathbf{size}(S) \qquad \mathbf{size}(\{\ \texttt{Send}\ x\ y\ \}) \triangleq 3$$
$$\mathbf{size}(\mathbf{let}\ x = \{\ \texttt{Receive}\ y\ \}\ \mathbf{in}\ S\ \mathbf{end}) \triangleq 3 + \mathbf{size}(S)$$

$$\mathbf{size}((U, \sigma)) \triangleq \mathbf{size}(U) + \mathbf{size}(\sigma) \qquad \mathbf{size}(U \parallel V) \triangleq \mathbf{size}(U) + \mathbf{size}(V)$$
$$\mathbf{size}(\sigma \parallel \sigma') \triangleq \mathbf{size}(\sigma) + \mathbf{size}(\sigma') \qquad \mathbf{size}(0) \triangleq 0 \qquad \mathbf{size}(\langle\rangle) \triangleq 1$$
$$\mathbf{size}(\langle S\ T\rangle) \triangleq \mathbf{size}(S) + \mathbf{size}(T) \qquad \mathbf{size}(x = w) \triangleq 3 \qquad \mathbf{size}(\xi : c) \triangleq 2 + \mathbf{size}(c)$$
$$\mathbf{size}(\xi : Q) \triangleq 2 + \mathbf{size}(Q) \qquad \mathbf{size}(\perp) \triangleq 0 \qquad \mathbf{size}(x) \triangleq 1$$
$$\mathbf{size}(Q; Q') \triangleq \mathbf{size}(Q) + \mathbf{size}(Q')$$

Fig. 9. Size of a Simple Configuration

5.1 Overhead of the Reversible Abstract Machine

In order to measure the space overhead of our reversible abstract machine we define a function **size** computing the size of a simple configuration (Figure 9) and a function **rsize** computing the size of a reversible configuration (Figure 10). Essentially the size of a term is computed as follows: we count 1 for the operator, 1 for each name it can have as argument, plus the size of subterms, if any.

Definition 3. *The* overhead *of a reversible configuration* (M, θ) *is defined as:*

$$\mathbf{overhead}(M, \theta) \triangleq \mathbf{rsize}((M, \theta)) - \mathbf{size}(\mathbf{erase}((M, \theta)))$$

To show that the space overhead of our abstract machine is linear in the number of reduction steps, we prove first that the maximal amount of information stored in a single step (computed by function **stsize** defined in Figure 11) is bounded by a constant during the execution of any fixed program:

Lemma 5. *Let* (U, σ) *be a simple configuration. Then for all* (U', σ') *such that* $(U, \sigma) \to (U', \sigma')$ *we have* $\mathbf{stsize}((U', \sigma')) \leq \mathbf{stsize}((U, \sigma))$.

Proof. By case analysis on the reduction rule. □

$$\mathbf{rsize}(\bot) \triangleq 0 \qquad \mathbf{rsize}(\mathbf{skip}) \triangleq 1 \qquad \mathbf{rsize}(H_1\ H_2) \triangleq \mathbf{rsize}(H_1) + \mathbf{rsize}(H_2)$$

$$\mathbf{rsize}(\uparrow x) \triangleq 2 \qquad \mathbf{rsize}(\downarrow x(y)) \triangleq 3 \qquad \mathbf{rsize}(*x) \triangleq 2 \qquad \mathbf{rsize}(*t) \triangleq 2$$

$$\mathbf{rsize}(\{\ x\ x_1 \dots x_n\ \}) \triangleq n + 2 \quad \mathbf{rsize}(\mathbf{if}(x)S) \triangleq 2 + \mathbf{size}(S) \quad \mathbf{rsize}(\mathbf{esc}) \triangleq 1$$

$$\mathbf{rsize}((M, \theta)) \triangleq \mathbf{rsize}(M) + \mathbf{rsize}(\theta) \qquad \mathbf{rsize}(0) \triangleq 0$$

$$\mathbf{rsize}(M \parallel N) \triangleq \mathbf{rsize}(M) + \mathbf{rsize}(N) \qquad \mathbf{rsize}(\theta \parallel \theta') \triangleq \mathbf{rsize}(\theta) + \mathbf{rsize}(\theta')$$

$$\mathbf{rsize}(t[H]C) \triangleq 1 + \mathbf{rsize}(H) + \mathbf{rsize}(C) \qquad \mathbf{rsize}(\langle\rangle) \triangleq 1$$

$$\mathbf{rsize}(\langle \mathbf{esc}\ C\rangle) \triangleq 1 + \mathbf{rsize}(C) \qquad \mathbf{rsize}(\langle S\ C\rangle) \triangleq \mathbf{size}(S) + \mathbf{rsize}(C)$$

$$\mathbf{rsize}(x = w) \triangleq 3 \qquad \mathbf{rsize}(\xi : c) \triangleq 2 + \mathbf{size}(c)$$

$$\mathbf{rsize}(\xi : K|K_h) \triangleq 2 + \mathbf{rsize}(K) + \mathbf{rsize}(K_h) \qquad \mathbf{rsize}(\bot) \triangleq 0 \qquad \mathbf{rsize}(t{:}x) \triangleq 3$$

$$\mathbf{rsize}(K; K') \triangleq \mathbf{rsize}(K) + \mathbf{rsize}(K') \qquad \mathbf{rsize}(t{:}x, t'; K_h) \triangleq 4 + \mathbf{rsize}(K_h)$$

Fig. 10. Size of a Reversible Configuration

$$\mathbf{stsize}(S_1\ S_2) \triangleq \max(\mathbf{stsize}(S_1), \mathbf{stsize}(S_2)) \qquad \mathbf{stsize}(\mathbf{skip}) \triangleq 1$$

$$\mathbf{stsize}(\mathbf{let}\ x = v\ \mathbf{in}\ S\ \mathbf{end}) \triangleq \max(2, \mathbf{stsize}(S))$$

$$\mathbf{stsize}(\mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}) \triangleq 3 + \max(\mathbf{size}(S_1)), \mathbf{size}(S_2))$$

$$\mathbf{stsize}(\mathbf{thread}\ S\ \mathbf{end}) \triangleq \max(3, \mathbf{stsize}(S))$$

$$\mathbf{stsize}(\mathbf{let}\ x = c\ \mathbf{in}\ S\ \mathbf{end}) \triangleq \max(3, \mathbf{stsize}(c), \mathbf{stsize}(S))$$

$$\mathbf{stsize}(\mathbf{proc}\ \{\ x_1 \dots x_n\ \}\ S\ \mathbf{end}) \triangleq \mathbf{stsize}(S) \qquad \mathbf{stsize}(\{\ x\ x_1 \dots x_n\ \}) \triangleq n + 3$$

$$\mathbf{stsize}(\mathbf{let}\ x = \texttt{NewPort}\ \mathbf{in}\ S\ \mathbf{end}) \triangleq \max(3, \mathbf{stsize}(S)) \qquad \mathbf{stsize}(\{\ \texttt{Send}\ x\ y\ \}) \triangleq 4$$

$$\mathbf{stsize}(\mathbf{let}\ x = \{\ \texttt{Receive}\ y\ \}\ \mathbf{in}\ S\ \mathbf{end}) \triangleq \max(7, \mathbf{stsize}(S))$$

$$\mathbf{stsize}((U, \sigma)) \triangleq \max(\mathbf{stsize}(U), \mathbf{stsize}(\sigma)) \qquad \mathbf{stsize}(0) \triangleq 0 \qquad \mathbf{stsize}(\langle\rangle) \triangleq 0$$

$$\mathbf{stsize}(U \parallel V) \triangleq \max(\mathbf{stsize}(U), \mathbf{stsize}(V)) \qquad \mathbf{stsize}(x = w) \triangleq 0$$

$$\mathbf{stsize}(\sigma \parallel \sigma') \triangleq \max(\mathbf{stsize}(\sigma), \mathbf{stsize}(\sigma')) \qquad \mathbf{stsize}(\xi : c) \triangleq \mathbf{stsize}(c)$$

$$\mathbf{stsize}(\langle S\ T\rangle) \triangleq \max(\mathbf{stsize}(S), \mathbf{stsize}(T)) \qquad \mathbf{stsize}(\xi : Q) \triangleq 0$$

Fig. 11. Maximal Overhead added by one Forward Step

We can now bound the overhead of a computation:

Lemma 6. *Let (M, θ) and (N, θ') be configurations such that $(M, \theta) \twoheadrightarrow (N, \theta')$. Then we have*

$$\mathbf{overhead}(N, \theta') \leq \mathbf{overhead}(M, \theta) + \mathbf{stsize}(\mathbf{erase}((M, \theta)))$$

Proof. By case analysis on the reduction rule. □

Theorem 2. *Assume* $(t[\bot]\langle S\ \langle\rangle\rangle, 0) \rightarrow^n_{vm} (M, \theta)$*, where* $\rightarrow^n_{vm}$ *denotes* $n \rightarrow_{vm}$ *steps. Then*

$$\mathbf{overhead}(M, \theta) \leq n \cdot \mathbf{stsize}(S)$$

Proof. By induction on the number of forward transitions, using Lemma 5 and Lemma 6. There is no need to consider backward transitions, since they reduce the overhead. □

5.2 Lower Bound on the Cost of Reversing μOz Programs

In this section we show that to ensure causally consistent reversibility of μOz programs we need a space overhead which is at least linear in the number of execution steps. Specifically, we prove the following:

Theorem 3. *The amount of information to be stored to ensure causally consistent reversibility of* μ*Oz programs is at least linear in the number of execution steps.*

Proof. Consider the following program:

```
let a = NewPort in
let x = true in
let y = false in
let p1 = proc {} {Send a x} { p1 } end in
let p2 = proc {} {Send a y} { p2 } end in
let p3 = proc {} let z = {Receive a} in { p3 } end end in
   thread { p1 } end thread { p2 } end thread { p3 } end
end end end end end end
```

The program launches three threads $p1$, $p2$ and $p3$. The threads $p1$ and $p2$ send messages over port a, while $p3$ receives them. Let us consider the set of traces which start with the initial program configuration, and where all the sends are performed before all the receives. Let us further restrict our attention to those traces consisting only of sequences of sends where for each $i \in \{1, \ldots, n\}$ the sends $2i - 1$ and $2i$ are from different threads, for some natural number n. Call T_n this set of traces. For a fixed n, in all the traces in T_n exactly n values **true** and n values **false** are sent. Thus the same program state S is reached in all the traces after all the values have been received, according to the standard (non-reversible) abstract machine semantics, up to α-conversion of the names of the receiving variables.

Now, all the traces in T_n above are coinitial, and they are not causally equivalent since different send actions to the same queue are not concurrent. To have a causally consistent rollback, coinitial traces which are not causally equivalent should lead to different states (cf Theorem 1), thus the state S obtained must be complemented with additional information $c(t)$ to distinguish between the different traces $t \in T_n$. Each trace $t \in T_n$ is uniquely identified by the state of the queue just before the first receive is executed by thread `p3`. We can encode each pair of elements in the queue with one bit, since the only possible values are (**true**, **false**) and (**false**, **true**). Thus, for a sequence of $2i$ sends we need i bits for distinguishing the different possibilities. All the words of i bits can be obtained, including the ones whose description is an incompressible string according to Kolmogorov complexity theory [14], so this number of bits cannot be improved. This is a lower bound on the size of information $c(t)$ that has to be stored to ensure causally consistent reversibility. The number of execution steps of the computations described above is linear in the number of bits, since a bounded number of steps is required to create the threads and perform two sends and two receives. $\square$

The result above shows that space is needed for keeping track of communications. A linear lower bound can also be proved using the number of created threads. In fact, in a program involving many threads, one has to remember how many steps each of them actually performed. If one chooses a program where the number of steps performed by each thread is bounded, and the number of threads is linear in the number of steps, a space bound which is linear in the number of steps can be proved.

5.3 Discussion

A few remarks concerning our reversible abstract machine and our results are in order. First, as should be clear from the abstract machine rules and the accompanying explanations, our reversibility machinery is based on an explicit history mechanism, in contrast, say, to reversible operators used in [19]. This may seem a naive choice, but our lower bound in Section 5.2 and the proof of the result there suggest that, in presence of non-deterministic concurrency as is the case with μOz, we have to resort to some sort of history, at least to be able to revert the effects of communication (message sends and receives) and thread creation.

Second, we avoid the exponential blowup in space overhead that we had in [12]. The main reason for this is that in our histories we only store pointers to values that are already present in the store, created by normal forward computation. We can recreate in μOz the equivalent of the sample HOπ program given in the Introduction, by creating appropriate additional closures corresponding to the values $R \mid R$, $R \mid R \mid R \mid R$, etc. But the reversibility machinery would only add pointers to these closures, created by normal forward computation, in thread histories, thus avoiding the exponential overhead.

Finally, one can note that our result in Section 5.2 is fairly robust, since it really is dependent only on the non-deterministic occurrences of non-concurrent events such as putting messages in a port queue from different threads. Such features are likely to be present in any physically distributed program or any concurrent program performing I/O operations, regardless of the actual language constructs used.

6 Related Work

Different works have dealt with designing reversible programming languages both in the *sequential* and *concurrent* settings.

In the sequential setting there is no need to save causal information among events. A framework for adding a general undo capability (and hence reversibility) to a programming language is presented in [13]. Computational history is saved by means of undo-lists, storing previous states of the execution. In [3,19,2] a reversible programming language, its virtual machine and compilation are presented. The key aspect of this language is that all its constructs (including assignments) are *bijective* (and hence reversible). In order to have reversible assignments a syntactic restriction on the possible expressions is imposed. The language is sequential and first-order, however, compared to our higher-order concurrent one. [19] contains several references to reversible sequential languages. A reversible abstract machine which implements the linear head reduction strategy for λ-calculus is presented in [8], where it is related to the well-known Krivine abstract machine.

In the concurrent setting, the works closer to ours are those dealing with programming languages with transactional constructs, such as [15], which exploits a form of undo to implement transactions (but only in a mono-processor setting), or those dealing with explicit checkpointing mechanisms, such as [9], but which implements an imperfect form of reversibility (e.g. rollbacks may not reach a global checkpoint).

A general upper bound on the trade-off between space and time to simulate irreversible computations by means of reversible ones is given in [6]. Moreover, [6] also provides a lower bound on the extra storage space required by step-wise reversible simulation of irreversible computations. Our lower bound on the space overhead that the reversible mechanism requires with respect to a non-reversible computation, is consistent with, though not reducible to, the one presented in [6].

7 Conclusion

We have presented a reversible abstract machine for a small higher-order concurrent programming language, μOz, which is a fragment of the Oz kernel programming language. We have shown that its space overhead on a program execution, compared to a non-reversible abstract machine for μOz directly inspired by the Oz abstract machine, is at most linear in the number of execution steps. This result cannot be much improved for we have also shown that reversing a μOz program requires at least such an amount of information.

There are however a number of ways that our abstract machine can be improved. Notice in particular that the information absolutely required for reversibility is related to potential sources of non-determinism in the execution of μOz programs. One can thus aim to reduce the information stored pertaining to deterministic steps, in particular trading space costs for time costs in backward steps. This would also improve the time costs incurred by forward executions in our abstract machine. It would also be interesting to see whether insights on lambda-machines in [8] can be leveraged to optimize the deterministic and sequential part of our machine.

On a different track, it would be interesting to study in more detail the costs of implementing *controlled* reversibility as introduced in [11], and to see how we can leverage the presence of explicit instructions for reversibility for different time and space trade-offs.

References

1. Altenkirch, T., Grattage, J.: A functional quantum programming language. In: Proc. of LICS 2005 (2005)
2. Axelsen, H.B.: Clean Translation of an Imperative Reversible Programming Language. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 144–163. Springer, Heidelberg (2011)
3. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible Machine Code and Its Abstract Processor Architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007)
4. Bennett, C.H.: Notes on the history of reversible computation. IBM Journal of Research and Development 32(1) (1988)
5. Brown, A.B., Patterson, D.A.: Undo for operators: Building an undoable e-mail store. In: USENIX Annual Technical Conference, General Track. USENIX (2003)
6. Buhrman, H., Tromp, J., Vitányi, P.M.B.: Time and Space Bounds for Reversible Simulation. In: Yu, Y., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 1017–1027. Springer, Heidelberg (2001)
7. Danos, V., Krivine, J.: Reversible Communicating Systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004)
8. Danos, V., Regnier, L.: Reversible, irreversible and optimal lambda-machines. Theor. Comput. Sci. 227(1-2) (1999)
9. Field, J., Varela, C.A.: Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In: Proc. of POPL 2005. ACM (2005)
10. Frank, M.P.: Introduction to reversible computing: motivation, progress, and challenges. In: 2nd Conference on Computing Frontiers. ACM (2005)
11. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.B.: Controlling Reversibility in Higher-Order Pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011 – Concurrency Theory. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011)
12. Lanese, I., Mezzina, C.A., Stefani, J.B.: Reversing Higher-Order Pi. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 478–493. Springer, Heidelberg (2010)
13. Leeman, G.B.: A formal approach to undo operations in programming languages. ACM Trans. Program. Lang. Syst. 8(1) (1986)
14. Li, M., Vitanyi, P.: An Introduction to Kolmogorov Complexity and Its Applications, 3rd edn. Springer (2008)
15. Ringenburg, M.F., Grossman, D.: AtomCaml: first-class atomicity via rollback. In: Proc. of ICFP 2005. ACM (2005)
16. Van Roy, P., Haridi, S.: Concepts, Techniques and Models of Computer Programming. MIT Press (2004)
17. Vitányi, P.M.B.: Time, space, and energy in reversible computing. In: 2nd Conference on Computing Frontiers. ACM (2005)
18. Yokoyama, T.: Reversible computation and reversible programming languages. Electr. Notes Theor. Comput. Sci. 253(6) (2010)
19. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Proc. of PEPM 2007. ACM (2007)

A Small Model Theorem for Rectangular Hybrid Automata Networks

Taylor T. Johnson and Sayan Mitra

Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{johnso99,mitras}@illinois.edu

Abstract. Rectangular hybrid automata (RHA) are finite state machines with additional skewed clocks that are useful for modeling real-time systems. This paper is concerned with the uniform verification of safety properties of networks with arbitrarily many interacting RHAs. Each automaton is equipped with a finite collection of pointers to other automata that enables it to read their state. This paper presents a small model result for such networks that reduces the verification problem for a system with arbitrarily many processes to a system with finitely many processes. The result is applied to verify and discover counterexamples of inductive invariant properties for distributed protocols like Fischer's mutual exclusion algorithm and the Small Aircraft Transportation System (SATS). We have implemented a prototype tool called Passel relying on the satisfiability modulo theories (SMT) solver Z3 to check inductive invariants automatically.

Keywords: hybrid automata, parameterized verification, small model theorem, uniform verification.

1 Introduction

Distributed systems are naturally modeled as a collection of interacting building-blocks or *modules*. For example, distributed computing systems are built from communicating computing processes, distributed traffic control protocols involve the interaction of individual vehicles, and neural networks arise from the interaction of neurons. This paper presents a result for *parameterized verification* of systems composed of such modules. For parameterized verification, a property P and a module template $\mathcal{A}_i$ are given. The property must be independent of the number and the identities of the modules, and we must verify that P holds for any system built from arbitrarily many instances of $\mathcal{A}_i$. That is,

$$\forall\ N \in \mathbb{N},\ \mathcal{A}_1 \| \mathcal{A}_2 \| \dots \| \mathcal{A}_N \models P, \tag{1}$$

where N is not fixed and the precise meaning of the parallel composition of operator $\|$ depends on the particular modeling framework.

The rectangular hybrid automaton (RHA) modeling framework [5,29] combines finite state machines with continuous variables. It has proven to be useful for modeling protocols and control logics with timers, and for approximating more complex linear and nonlinear dynamics with piecewise constant dynamics. Several subclasses of RHA have been identified for which safety verification is decidable [29] and model checking tools have been developed [28,22,23].

In this paper, we consider parameterized verification of RHA networks where modules communicate by reading one another's state and through globally shared variables. A RHA may read the variables of another RHA through the use of a pointer variable tracking the identifier of that automaton. Such communication allows us to model distributed traffic control systems like the Small Aircraft Transportation System (SATS) [1,34,38]. In SATS, aircraft communicate by reading the valuations of discrete variables and continuous positions through pointer variables. The pointer variables allow us to model systems where the communication topology is dynamic. Thus, a variety of other distributed cyber-physical systems (DCPS) can be modeled in this manner through the use of pointer variables. For instance, in the automated highway system, a car may only need to keep track of the positions of the cars immediately ahead and behind it requiring two pointer variables, and similar scenarios arise in robotic swarm protocols in one-dimensional lanes [30]. However, at a four-way intersection of single lane roads, an autonomous car may need to track the positions of cars coming from every other direction, requiring three pointer variables. All of these scenarios fit into the communication model and verification framework developed in this paper.

To perform verification of an infinite number of such infinite-state RHA, we develop and use a small model result for RHA networks. Small model theorems are used to prove decidability of checking satisfiability of first-order logic (FOL) formulas. The philosophy behind applying them in verification is to identify classes of systems and specifications with the small model property, which reduces an infinite problem to a finite one. Many prefix classes of FOL were shown to be decidable by showing that the class has the *finite model property*: every satisfiable formula also has a finite model [11]. In many cases the proof of the finite model property comes with an explicit bound on the size of the finite structure that may satisfy (or violate) the property in question. For parameterized verification, small model theorems provide a finite threshold N_o such that if, for all $N \leq N_o$, $A^N \triangleq \mathcal{A}_1 \| \ldots \| \mathcal{A}_N \models P$, then Equation 1 also holds.

The contributions of this paper are:

(a) A small model theorem for RHA networks that guarantees the existence of a bound N_o, such that, if an instantiation of A^N violating P exists for some $N > N_o$, then $\mathcal{A}^{N_o}$ must also violate P. Thus, the verification problem from Equation 1 is solved if, for all $N \leq N_o$, no instantiation A^N violates P.

(b) The theorem is applied in a software tool called Passel that we use to automatically check inductive invariants up to the bound N_o using the satisfiability modulo theories (SMT) solver Z3 [16].

The input to Passel is a hybrid automaton specification $\mathcal{A}_i$ and a candidate safety property P. Then, the bound N_o is computed from the syntactic description of

$\mathcal{A}_i$ and P. In addition to these hybrid protocols, we have also verified several purely discrete algorithms (cache coherence protocols and mutual exclusion algorithms like the simplified bakery algorithm) studied in [18,2]. The experiments have indicated that it is feasible to develop automatic methods relying on our small model result that apply both to real-time distributed systems and classic distributed algorithms. The success of our experiments in part relies on the strengths of state-of-the-art SMT solvers like Z3, which allow for quantified formulas and have quantifier elimination and instantiation procedures for real and integer arithmetic [24,10].

Related Work. To the best of our knowledge, the automatic parameterized verification problem has not been addressed previously for RHA, but there are several works addressing parameterized verification for networks of the special subclass of timed automata [4,3,25,14,21,13]. Parameterized verification of RHA networks is useful to show, for instance, that for arbitrarily many aircraft participating in a given distributed air traffic control protocol like the Small Aircraft Transportation System (SATS), no two aircraft ever collide [34,38,31]. We use a simplified version of SATS as a case study to illustrate the concepts and results developed in this paper. There are several partly manual works for parameterized verification of timed and hybrid systems using theorem provers [20,34,38,35,36,32].

An overview of automatic approaches for parameterized verification of discrete systems appears in [15, Ch. 15]. The parameterized verification problem is in general undecidable, even for finite-state modules [7]. However, for restricted classes of systems under various communications constraints, the problem has been shown to be decidable. For instance, parameterized verification is decidable for safety properties of timed networks considered in [4,14]. However, each timed automaton in the network is assumed to have either (a) a single real-valued clock, or (b) any finite number of discrete-valued clocks [3]. If the timed automata each have more than a single real-valued clock, then the problem is undecidable [3]. The previous undecidability result prevents using the standard initialized rectangular hybrid automata (IRHA)-to-timed automata conversion algorithm [6,29] because it adds two clocks for every continuous variable evolving with rectangular dynamics.

There are a variety of automatic and semi-automatic approaches to parameterized verification, but the most closely related are network invariants [40]. Finding invariants was automated with the invisible invariants approach [37,8], which provides a heuristic method to automatically compute inductive invariants, such as implemented in [9]. A small model theorem like the one presented in this paper was introduced in [37] for a class of discrete parameterized systems with bounded data. Network invariants have previously been developed for timed parameterized systems in [25]. Another automated approach for parameterized verification of timed systems is presented in [21]. One can view cutoffs—an instantiation of the system that has all the behaviors of additional compositions—like those in [26] like small model results, in that it is sufficient to check the composition of a protocol up to the cutoff or small model bound to verify the parameterized specification.

2 Modeling Framework

In this section, we present the syntax for a class of assertions we call *LH-assertions*, introduce a modeling framework for networks of rectangular hybrid automata (RHA), and then show how inductive invariants for such networks can be asserted in terms of LH-assertions. We will then develop the small model theorem for LH-assertions. Let $N \geq 2$ be a natural number. The set $[N] \triangleq \{1, \ldots, N\}$ is used for indexing RHA. For a set S, we define $S_\perp \triangleq S \cup \{\perp\}$.

2.1 LH-Assertions

LH-assertions are built-up from constants, variables, arrays, and terms of several different types. We introduce LH-assertions first because we will use LH-assertions for specifying the syntactic description of individual RHA and RHA networks below. Throughout, natural numbers are used for indexing arrays. The numbers 1 and N are *constants* of type $\mathbb{N}$ and $L = \{L_1, \ldots, L_k\}$ is a fixed finite type. The signature of LH-assertions involves a finite number of variables of the following types: (a) *Index variables*: $i_1, \ldots, i_b : [N]_\perp$, (b) *Discrete variables*: $l_1, \ldots, l_c : L$, (c) *Real variables*: $x_1, \ldots, x_d, t_1, \ldots, t_d : \mathbb{R}$, (d) *Index-valued array variables*: $\bar{p}_1, \ldots, \bar{p}_e : [N] \to [N]_\perp$, (e) *Discrete array variables*: $\bar{l}_1, \ldots, \bar{l}_f : [N] \to L$, and (f) *Real array variables*: $\bar{x}_1, \ldots, \bar{x}_g : [N] \to \mathbb{R}$.

The grammar for constructing LH-assertions is defined as follows.

$$\begin{aligned} \mathsf{ITerm} &::= 1 \mid N \mid i_j \mid \bar{p}_k[\mathsf{ITerm}] \\ \mathsf{DTerm} &::= L_j \mid l_k \mid \bar{l}_j[\mathsf{ITerm}] \\ \mathsf{RTerm} &::= x_j \mid \bar{x}_k[\mathsf{ITerm}] \end{aligned}$$

An index term (ITerm) is either (a) one of the constants 1, N, or an index variable i_j, or (b) an index array $\bar{p}_k$ referenced at 1, N, or i_j. We use the notation $\bar{y}[i]$ to denote the valuation of the array $\bar{y}$ at the value of the index variable i. Discrete terms (DTerm) and real terms (RTerm) are defined as specified above. Here, L_j is constant from L, l_k is a discrete variable, $\bar{l}_j$ is a discrete array, x_j is a real variable, and $\bar{x}_k$ is a real array. Using these terms, formulas are defined next.

$$\begin{aligned} \mathsf{Atom} &::= \mathsf{ITerm}_1 < \mathsf{ITerm}_2 \mid \mathsf{DTerm} = L_k \mid a_1\mathsf{RTerm}_1 + a_2\mathsf{RTerm}_2 + a_3 < 0 \\ \mathsf{Formula} &::= \mathsf{Atom} \mid \neg\mathsf{Formula} \mid \mathsf{Formula}_1 \wedge \mathsf{Formula}_2 \end{aligned}$$

Here, a_1, a_2, and a_3 are real-valued numerical constants used to specify some real linear arithmetic constraint, and $\mathsf{Formula}_1$ and $\mathsf{Formula}_2$ are shorter formulas that are joined by boolean operators to obtain a longer formula. By combining the boolean operators $\wedge$ and $\neg$ with the $<$ operator, other comparison operators, such as $=$, $\neq$, $\leq$, $>$, and $\geq$, can be expressed in formulas for both indices and reals. For example, $\bar{p}_1[i_j] = \bar{p}_2[i_k]$ can be written as $\neg(\bar{p}_1[i_j] < \bar{p}_2[i_k]) \wedge \neg(\bar{p}_2[i_k] < \bar{p}_1[i_j])$.

Given a formula $\mathsf{Formula}$, a *sentence*—a formula with no free variables—is obtained by quantifying all the free index and real variables. An LH-assertion is a sentence of the form $\forall t_1 \in \mathbb{R} : \forall i_1, \ldots, i_k \in [N] : \exists t_2 \in \mathbb{R} : \exists j_1, \ldots, j_m \in [N] : \varphi$, where φ is a formula. We mention that t_1 and t_2 are only used in practice to respectively model an elapse of time of length t_1 and enforcing invariants for all trajectories of lengths $0 \leq t_2 \leq t_1$. We provide several example quantified sentences and LH-assertions:

$$\forall i, j : i \neq j \implies (\bar{l}[i] = \bar{l}[j] \implies |\bar{x}[j] - \bar{x}[i]| > a), \quad (2)$$

$$\forall i, j : i = j \ \vee \ \bar{l}[i] = \bar{l}[j] \ \vee \ (\bar{x}[j] - \bar{x}[i] - a < 0) \ \vee \ (\bar{x}[i] - \bar{x}[j] - a < 0), \quad (3)$$

$$\forall i \ \exists j : \bar{p}[i] = j \wedge |\bar{x}[i] - \bar{x}[\bar{p}[i]]| > a. \quad (4)$$

We only use LH-assertions with t_1 and t_2 for checking continuous transitions as shown in Subsection 3.1. Reading these assertions as statements about networks of automata, the first one states that all automata with identical values of the discrete variable $\bar{l}$ have a minimum gap of a between the values of their $\bar{x}$ variables. The first assertion is an abbreviation of the second. The last assertion states every automaton has a pointer $\bar{p}$ to another automaton and that there is a minimum separation of a between its $\bar{x}$ value and the $\bar{x}$ value of the automaton it points to.

Models for Sentences and Assertions. A *model* for an assertion provides interpretation to the elements appearing in the assertion. Specifically, an n-model M for an LH-assertion ψ is denoted $M(n, \psi)$ and provides an interpretation of each the *free* variables in ψ as follows:

- the constants 1 and N are assigned the values 1 and n,
- each index variable is assigned a value in the set $\{1, \ldots, n\}$,
- each discrete variable is assigned a value in L,
- each real variable is assigned a value in $\mathbb{R}$, and
- each index, discrete, and real array is assigned respectively a $\{1, \ldots, n\}_\perp$-valued, L-valued, and real-valued array of length $[1, \ldots, n]$.

For example, a 2-model for the assertion of Equation 4 is specified by the assignments $N = 2$, $\bar{p} = \langle 2, 1 \rangle$, and $\bar{x} = \langle 0, 10.0 \rangle$, assuming any choice for the real constant $a < 10$. Given an assertion ψ and a model $M(n, \psi)$, if ψ evaluates to true with the interpretations of the free variables given by $M(n, \psi)$, then $M(n, \psi)$ is said to *satisfy* ψ. If all models of ψ satisfy it, then the assertion is said to be *valid*. If there exist models that satisfy ψ, then the assertion is said to be *satisfiable*. Fixing $a = 9.0$, the above model satisfies assertion Equation 4, but it is not valid because the 1-model $\bar{p} = \langle 1 \rangle, \bar{x} = \langle 10.0 \rangle$ does not satisfy it, since $|\bar{x}[1] - \bar{x}[1]| = 0 \not> a$.

2.2 Networks of Rectangular Hybrid Automata

Informally, in the context of networks of rectangular hybrid automata (RHA), the arrays of discrete, real, and index variables respectively represent discrete, continuous, and pointer variables of individual automata, while the ordinary (non-array) variables represent globally shared variables[1]. In this section, we

[1] A real-typed variable may be updated continuously and/or discretely, while variables of other types are only updated discretely—we do not explicitly partition the sets of real and real array variables for simplicity of presentation, but will mention it when defining the semantics.

introduce the syntax of a language for specifying networks of rectangular hybrid automata, and then introduce the semantics of the language and show how the networks can be modeled with LH-assertions.

A RHA $\mathcal{A}_i$ is a (possibly nondeterministic) finite state machine augmented with skewed real-valued clocks. A *RHA Network* is a collection of RHA in which the transitions of each RHA can depend on the the state of certain other RHA. In particular, these certain other RHA are specified through the use of pointer (index-valued) variables. In this paper, we consider RHA Networks that are composed of *arbitrarily many* identical RHA. If a module contains several RHA subsystems $\mathcal{A}_1$, $\mathcal{B}_1$, $\mathcal{C}_1$, ..., the composition of these subsystems can be taken first prior to composing the RHA network.

Syntax of Individual RHA. Syntactically, a RHA $\mathcal{A}_i$ for $i \in [N]$ is specified by the following components: (a) a list of variable names Var_i, (b) a list of action names Act_i, (c) a list of mode names Mode_i, (d) an assertion Init_i on Var_i, (e) a collection of precondition-effect statements—one for each element in Act_i, and (f) a collection of invariant-stop-flow statements—one for each element in Mode_i. For example, Figure 1 shows a complete specification for a RHA modeling the simplified SATS air traffic control protocol.

Now we describe the syntactic structure of each of these components. Each variable in Var_i is associated with a type. The type can be (a) *a finite set* L, (b) *the set of indices* (augmented with the special element $\bot$) $[N]_\bot$, or (c) *the set of reals*. Each variable in Var_i has a name of the form $\langle \mathit{variable\_name} \rangle[i]$. For example, $l[i] : L$, $p[i] : [N]_\bot$, and $x[i] : \mathbb{R}$, define discrete, index, and real variables in Var_i. Looking ahead, this naming convention will be consistent with the syntax of a class of assertions called LH-assertions, that are used for defining the RHA network. One array type variable $\bar{l}$ will encode valuation of all the $l[i]$ variables in the network. That is, the state variables of a RHA network will be represented by arrays of discrete, continuous, and index variables.

The initial assertion Init_i is specified by a sentence involving the variables in Var_i. IN SATS for instance, $\mathsf{Init}_i : l[i] = F \wedge \mathit{next}[i] = \bot \wedge \mathit{last} = \bot$, where last is a global variable. For each action $\mathsf{a} \in \mathsf{Act}_i$, the precondition—denoted by $\mathbf{pre}(\mathsf{a}, i)$—is a sentence involving the variables in Var_i with possibly additional quantified index variables. The effect, denoted by $\mathbf{eff}(\mathsf{a}, i)$, is a sentence involving both the variables in Var_i as well as their primed versions. For example, the following are precondition-effect statements for some labels setPtr and chkGap:

$$\begin{aligned}
\mathbf{pre}(i, \mathsf{setPtr}) &: \forall j : l[i] = L_1 \ \wedge \ (j \neq i \implies j \neq p[i]), \\
\mathbf{eff}(i, \mathsf{setPtr}) &: \exists j : l'[i] = L_2 \ \wedge \ x'[i] = 0 \ \wedge \ j \neq i \ \wedge \ p'[i] = j, \\
\mathbf{pre}(i, \mathsf{chkGap}) &: l[i] = L_2 \ \wedge \ x[p[i]] > 20, \\
\mathbf{eff}(i, \mathsf{chkGap}) &: l'[i] = L_3 \ \wedge \ x'[i] = 0.
\end{aligned}$$

Note that the precondition and effect sentences may have additional quantified index variables apart from i with the $\exists\ \forall$ quantifier ordering.

For each mode $\mathsf{m} \in \mathsf{Mode}_i$, the invariant—denoted by $\mathbf{inv}(\mathsf{m}, i)$—and the stopping condition—denoted by $\mathbf{stop}(\mathsf{m}, i)$—are sentences involving the variables in

Var_i with possibly additional quantified index variables. For each $\mathsf{m} \in \mathsf{Mode}_i$, the flow statement associates two real constants with each real-valued, continuously updated variable in Var_i. For variable $x[i]$, these constants are denoted by $\mathbf{lflowrate}(\mathsf{m}, i, x)$ and $\mathbf{uflowrate}(\mathsf{m}, i, x)$. This allows for modeling the usual rectangular dynamics $\mathbf{flowrate}(\mathsf{m}, i, x) \in [a, b]$ for $a \leq b$, where a equals the lower rate and b equals the upper. If $a = b$, we say the dynamics are timed. For example, for mode Base of SATS (see Figure 1),

$$\begin{aligned}
\mathbf{inv}(\mathsf{Base}, i) &: l[i] = B \ \wedge \ x[i] \leq L_B \\
\mathbf{stop}(\mathsf{Base}, i) &: l[i] = B \ \wedge \ x[i] = L_B \\
\mathbf{lflowrate}(\mathsf{Base}, i, x) &: v_{min} \ \wedge \ \mathbf{uflowrate}(\mathsf{Base}, i, x) : v_{max}
\end{aligned}$$

so that $\mathbf{flowrate}(\mathsf{Base}, i, x) \in [v_{min}, v_{max}]$.

2.3 Example: Simple Air Traffic Landing Protocol

We use a simplified version of the Small Aircraft Transportation System (SATS), a distributed air traffic control protocol, as a running example for the remainder of the paper [1,34,38,31]. SATS is a program aimed at increasing throughput at small airports without air traffic controllers by allowing aircraft to communicate between themselves along with a centralized communication component at the airport used to determine a landing sequence order [1,39]. Aircraft in SATS communicate by reading the continuous position of any aircraft immediately ahead of it in the landing sequence, where the aircraft immediately ahead is tracked using a pointer. The example is parameterized on the number of aircraft involved in the landing attempt, and is naturally modeled as a RHA network. The system models a single airport and N flying aircraft that are attempting to land. After determining the landing sequence order from a centralized airport management module (AMM) located at the airport—which we model using the global shared variable *last*—the remainder of the protocol is decentralized and each aircraft communicates with the aircraft immediately ahead of it (if one exists) to determine if it is safe to attempt landing.

All aircraft begin in the Flying mode, and when an aircraft is ready to attempt landing, it initiates the approach to the airport by making a discrete transition to the $\mathsf{Holding}$ mode. The $\mathsf{Holding}$ mode physically represents that the aircraft is flying in a cyclic holding pattern, and it is assumed the aircraft maintain a safe separation in this mode. On entering $\mathsf{Holding}$, an aircraft is either designated as the first one in the landing sequence (and $next[i] = \bot$), or the aircraft is assigned the identifier of the last aircraft that began its approach to the runway (and $next[i] = last$). Subsequently, an aircraft may nondeterministically transition from the $\mathsf{Holding}$ mode to the Base mode and is now physically approaching the runway. The position of the i^{th} aircraft is modeled using a single continuous variable ($x[i]$) of real type, representing the position along a line measured starting from the geographic location of the cyclic holding zone (that is, the beginning of the base region). This transition is only enabled for aircraft i if there is at least L_S distance between its position $x[i]$ and the position of the

```
1  Var: $l[i]$:{$F,H,B,R$}, $x[i]$:$\mathbb{R}$, $next[i]$:$[N]_\perp$
   Global Var: $last$:$[N]_\perp$
3  Init: $l[i] = F \wedge next[i] = \perp \wedge last = \perp$
   Mode: Fly, Holding, Base, Runway
5
   Fly: Inv: $l[i] = F$
7    Flowrate: $\dot{x}[i] = 0$

9  Holding: Inv: $l[i] = H$
     Flowrate: $\dot{x}[i] = 0$
11
   Base: Inv: $l[i] = B \wedge x[i] \le L_B$
13   Stop: $x[i] = L_B$
     Flowrate: $\dot{x}[i] \in [v_{min}, v_{max}]$
15
   Runway: Inv: $l[i] = R$
17   Flowrate: $\dot{x}[i] = 0$

Act: FtoH, HtoB, BtoH, BtoR
FtoH: Pre: $l[i] = F$                                              20
  Eff: $l'[i] = H \wedge next'[i] = last \wedge last' = i$
                                                                   22
HtoB: Pre: $l[i] = H \wedge (next[i] = \perp$
    $\vee\, l[\ next[i]\ ] = B \wedge x[\ next[i]\ ] \ge L_S)$     24
  Eff: $l'[i] = B \wedge x'[i] = 0$
                                                                   26
BtoH: Pre: $l[i] = B$
  Eff: $l'[i] = H \wedge x'[i] = 0\ \wedge$                        28
    $(last \ne i \Rightarrow next'[i] = last) \wedge last' = i$
    $\wedge\, \forall\ j \ne i$ (if $next[j] = i$ then $next'[j] =$  30
    $\perp$else $next'[j] = next[j]$)
                                                                   32
BtoR: Pre: $l[i] = B \wedge x[i] \ge L_B\ \wedge next[i] = \perp$
  Eff: $l'[i] = R \wedge (last = i \Rightarrow last' = \perp)$     34
    $\wedge\, \forall\ j \ne i$ (if $next[j] = i$ then $next'[j] =$
    $\perp$else $next'[j] = next[j]$)                              36
```

Fig. 1. RHA $\mathcal{A}_i$ for simplified SATS protocol

aircraft ahead of it, $x[next[i]]$ (if one exists). Once in the Base mode, the aircraft is approaching the runway and after traversing L_B distance, the aircraft may either (a) cancel the landing attempt and return to the cyclic holding pattern in mode Holding, in which case it becomes the last aircraft in the sequence, or (b) the aircraft may succeed in landing and set its mode to Runway.

SATS Properties. We checked the following properties for SATS with memory and time required as indicated in Table 1. The properties specifying a safe separation are D and E. Note that we checked property D only for rectangular dynamics (that is, Figure 1, line 14 is as written) and E only for timed dynamics (that is, the rectangular dynamics in Figure 1, line 14 are replaced by $\dot{x}[i] = 1$). This is because SATS with rectangular dynamics does not satisfy E. The properties are:

(A) $\forall i \in [N] : l[i] = F \Rightarrow last \ne i$,
(B) $\forall i, j \in [N] : next[j] = i \Rightarrow l[i] \ne F$,
(C) $\forall i, j \in [N] : l[i] = H \wedge next[j] = i \Rightarrow l[j] = H$,
(D) $\forall i, j \in [N] : l[i] = B \wedge l[j] = B \wedge next[j] = i \Rightarrow x[i] \ge L_S + (v_{max} - v_{min})\frac{L_B - x[j]}{v_{min}}$, and
(E) $\forall i, j \in [N] : i \ne j \wedge l[i] = B \wedge l[j] = B \wedge next[j] = i \Rightarrow x[i] - x[j] \ge L_S$.

2.4 Semantics of RHA Networks

Given the syntactic specification of a single RHA $\mathcal{A}_i$, the semantic definition of a RHA network, where arbitrarily many $\mathcal{A}_i$'s execute in parallel, is obtained as follows. We note this is essentially $N-1$ parallel compositions like those considered in [27]. For any $N \in \mathbb{N}$, the automaton A^N is the tuple $\langle V_N, Q_N, \Theta_N, D_N, T_N \rangle$. We define each of these components as follows.

V_N is a set of variables, with a real array $\bar{x}_1$ corresponding to each real variable x_1 in the Var_i list, an index array $\bar{p}_1$ corresponding to each index variable p_1 in the Var_i list, and so on. Additionally, V_N contains a set of global (non-array) variables of assorted types, corresponding to the non-array variables in the LH-assertions.

Q_N is the set of all possible valuations of the variables in V_N. Elements of Q_N are called *states* and are denoted by boldface $\mathbf{v}$, $\mathbf{v}'$, etc. At a state $\mathbf{v}$, the valuation of a particular array variable $\bar{p}$ is denoted by $\mathbf{v}.\bar{p}_1$, and $\mathbf{v}.g$ for some non-array variable g in V_N. The valuation of the variables in Var_i at state $\mathbf{v}$ is denoted by $\mathbf{v}[i]$. Given a state $\mathbf{v}$ and a quantified sentence Sent involving the arrays in V_N and index variables, we say that $\mathbf{v}$ *satisfies* Sent iff fixing the values of the variables in Sent yields an assertion that is valid. In that case, we write $\mathbf{v} \models \mathsf{Sent}$.

$\Theta_N \subseteq Q_N$ is called the *set of initial states*, and is the set of states that satisfy the assertion Init_i for every index $i \in [N]$,

$$\{\mathbf{v} \in Q_N \mid \forall\, i \in [N] : \mathbf{v}[i] \models \mathsf{Init}_i\}.$$

$D_N \subseteq Q_N \times Q_N$ is called the set of *discrete transitions* and is defined as follows: $(\mathbf{v}, \mathbf{v}') \in D_N$ iff:

$$\exists\, i \in [N] :\ \exists\, \mathsf{a} \in \mathsf{Act}_i : \mathbf{v}\ \models \mathbf{pre}(\mathsf{a}, i)\ \wedge\ (\mathbf{v}, \mathbf{v}') \models \mathbf{eff}(\mathsf{a}, i) \wedge \\ \forall\, j \in [N] :\ j \neq i \wedge j \notin M \implies \mathbf{v}'[j] = \mathbf{v}[j],$$

where M is a (possibly empty) subset of $[N]$ corresponding to any indices (excluding i) of the valuations of array variables modified by the $\mathbf{eff}(\mathsf{a}, i)$ statement. For instance, in SATS (see Figure 1, line 31), if i transitions from Base back to Hold, then if $next[m] = i$ for any $m \neq i$, then $next'[m] = \perp$, and $M = \{m\}$.

$T_N \subseteq Q_N \times Q_N$ is called the set of *trajectories*. To define T_N we first define the relation $\mathbf{flow}(\mathsf{m}, \mathbf{v}[i], t)$ that returns a set of valuations $\mathbf{v}'[i]$, such that for each $x \in \mathsf{Var}_i$, if x's type is not real (or is real, but is only updated discretely), then $\mathbf{v}'[i].x = \mathbf{v}[i].x$, but otherwise, $\mathbf{v}[i].x + \mathbf{lflowrate}(\mathsf{m}, i, x)t \leq \mathbf{v}'[i].x \leq \mathbf{v}[i].x + \mathbf{uflowrate}(\mathsf{m}, i, x)t$. A pair $(\mathbf{v}, \mathbf{v}') \in T_N$ iff:

$$\exists t_1 \in \mathbb{R}_{\geq 0} :\ \forall i \in [N] : \exists \mathsf{m} \in \mathsf{Mode}_i : \forall t_2 \in \mathbb{R}_{\geq 0} : t_2 \leq t_1 \wedge \\ (\mathbf{flow}(\mathsf{m}, \mathbf{v}[i], t_2) \models \mathbf{inv}(\mathsf{m}, i) \wedge \mathbf{flow}(\mathsf{m}, \mathbf{v}[i], t_2) \models \mathbf{stop}(\mathsf{m}, i) \Rightarrow t_2 = t_1) \\ \wedge\, \mathbf{v}'[i] \in \mathbf{flow}(\mathsf{m}, \mathbf{v}[i], t_1).$$

Informally, a discrete transition from $\mathbf{v}$ to $\mathbf{v}'$ models the discrete transition of a particular RHA $\mathcal{A}_i$ by some action $\mathsf{a} \in \mathsf{Act}_i$. The precondition of action a may depend on the variables in Var_j for $j \neq i$. The effect is usually defined in terms of the variables in Var_i, but may also set variables in Var_j if j is the valuation of some index-valued variable of i. A trajectory models a transition from $\mathbf{v}$ to $\mathbf{v}'$ that occurs over some interval of time with length t_1. All the non-continuously updated variables (discrete variables, pointers, and any real variable updated only discretely) remain unchanged. For each $i \in [N]$ and each continuously updated real variable $x \in \mathsf{Var}_i$, $\mathbf{v}[i].x$ must evolve to the valuations $\mathbf{v}'[i].x$, in exactly t_1 time in some mode $m \in \mathsf{Mode}_i$. All intermediate states along the trajectory must also satisfy the invariant $\mathbf{inv}(\mathsf{m}, i)$, and if an intermediate state satisfies $\mathbf{stop}(\mathsf{m})$, then that state must be $\mathbf{v}'$ (that is, the end of a trajectory).

The behavior of a network of RHA is defined as sequences of states that are related by transitions and trajectories. An *execution* of A^N is a sequence of states $\mathbf{v}_0, \mathbf{v}_1, \ldots$ such that $\mathbf{v}_0 \in \Theta_N$, and for each index k appearing in the sequence, if k is even then $(\mathbf{v}_k, \mathbf{v}_{k+1}) \in T_N$, and otherwise $(\mathbf{v}_k, \mathbf{v}_{k+1}) \in D_N$.

3 Small Model Theorem

We begin this section with the main small model result, Lemma 1, for LH-assertions with the signature introduced in the previous section. Then we show how inductive invariants for networks of RHAs can be encoded as LH-assertions. Thus, for a specific inductive invariant I, Lemma 1 provides a threshold on size of models, written $n(I)$. If for all $N \leq n(I)$, I is an inductive invariant for A^N, then I is indeed an inductive invariant for all $N \in \mathbb{N}$. This makes it possible to verify inductive invariants for parameterized networks of hybrid automata by verifying I for a finite number of instances of A^N.

Lemma 1. *Let ψ be a LH-assertion of the form $\forall t_1 \in \mathbb{R}\ \forall i_1, \ldots, i_k \in [N]\ \exists t_2 \in \mathbb{R}\ \exists j_1, \ldots, j_m \in [N] : \varphi$, where φ is a quantifier-free formula involving the index variables $i_1, \ldots, i_k$, $j_1, \ldots, j_m$, real variables t_1 and t_2, and global and array variables in V_N. Then, ψ is valid iff for all $n \leq N_0 = (e+1)(k+2)$, ψ is satisfied by all n-models, where e is the number of index array variables in φ and k is the largest subscript of the universally quantified index variables in ψ.*

Proof. We assume that all models of size n, for $n \leq (e+1)(k+2)$, satisfy ψ. It suffices to show that ψ is valid. Suppose for the sake of contradiction that ψ is not valid. Then there exists a model M of size $n > (e+1)(k+2)$ that satisfies $\neg\psi = \exists\ t_1, i_1, \ldots, i_k : \forall t_2, j_1, \ldots, j_m : \neg\varphi$. We will show that for any model of size $n > (e+1)(k+2)$, there exists a model of size $n-1$ that contradicts the assumption.

The n-model M assigns a real value to the variable t_1 and values in $\{1, \ldots, n\}$ to the index variables $i_1, \ldots, i_k$ (in addition to providing interpretations for the other bounded variables and arrays). The values assigned to the universally quantified variables $t_2, j_1, \ldots, j_m$ in the model M are not important, because any value of these variables would satisfy $\neg\psi$. The set of values assigned to $i_1, \ldots, i_k$ can contain at most k distinct values. Consider an index term with one of the forms 1, N, or i_m, where i_m is an existentially quantified index variable in $\neg\psi$: any such term can take at most $k+2$ distinct index values. Thus, an index array term $\bar{p}[i_m]$ can take at most $k+2$ distinct values. Since there are at most e index arrays, the set of all possible index arrays and terms can take at most $(e+1)(k+2)$ distinct values. Therefore, there exists a value in $\{1, \ldots, n\}$, say

u, that is not assigned to any index variable or to any of the referenced values of the index arrays, in M.

Now, we define an $(n-1)$-model M' by removing u from $\{1, \ldots, n\}$ and shifting values appropriately. The constants n is interpreted as $n-1$ in M'. For each index variable i_j, if $i_j < u$ then we assign $M'(i_j) = M(i_j)$, and otherwise we assign $M'(i_j) = M(i_j) - 1$. For each (index, discrete, or real) array $\bar{z}$, for each $i \in \{1, \ldots, n-1\}$, if $i < u$ then we assign $M'(\bar{z}[i]) = M(\bar{z}[i])$, and otherwise we assign $M'(\bar{z}[i]) = M(\bar{z}[i+1])$. Finally, it is routine to check that M' assigns the same binary value to each Atom in φ as M, and therefore M' also satisfies $\neg\psi$.

3.1 Applying the Small Model Result to Check Inductive Invariants

For an automaton network A^N, an *invariant assertion* is a logical sentence involving the variables in V_N (and possibly the global variables). In this paper we will consider *regular invariants* in which (a) the indices of all the arrays are index variables (and not constants), and (b) index variables can only be compared with other index variables (not constants). Furthermore, we require the regular invariants to have all the universal quantifiers precede the existential quantifiers. Thus, a regular invariant is of the form $\psi \triangleq \forall\, i_1, \ldots, i_k \in [N] : \exists j_1, \ldots, j_m \in [N] : \varphi$, where φ is a quantifier-free formula involving the index variables $i_1, \ldots, i_k$, $j_1, \ldots, j_m$, and the global and array variables in V_N.

In the case of SATS, the regular invariant specifying a safe separation of aircraft is $\forall i, j \in [N] \;:\; (i \neq j \wedge l[i] = B \wedge l[j] = B \wedge next[j] = i) \Rightarrow x[i] - x[j] \geq L_S$. That is, if there is an aircraft i attempting to land, the aircraft immediately ahead of it is at least L_S physical distance away.

In the remainder of this section, we show how inductive invariant assertions for networks of RHA can be stated as LH-assertions. For the purposes of this presentation we assume that there are no global variables. It can be checked in a straightforward manner that these derivations hold for systems with global variables. An assertion ψ is an invariant assertion for the parameterized network A^N if, for all $N \in \mathbb{N}$,

(A) **initiation**: for each state $\mathbf{v} \in \Theta_N \Rightarrow \mathbf{v} \models \psi$,
(B) **transition consecution**: for each $(\mathbf{v}, \mathbf{v}') \in D_N$, $\mathbf{v} \models \psi \Rightarrow \mathbf{v}' \models \psi$, and
(C) **trajectory consecution**: for each $(\mathbf{v}, \mathbf{v}') \in T_N$, $\mathbf{v} \models \psi \Rightarrow \mathbf{v}' \models \psi$.

We derive an LH-assertion for each of the above conditions.

From the definition of the initial states, $\mathbf{v} \in \Theta_N$ iff $\forall\, i \in [N] : \mathbf{v}[i] \models \mathsf{Init}_i$, where recall that Init_i is a formula involving the variables in Var_i. Thus, condition A is equivalent to checking:

$$(\forall\, i \in [N] : \mathsf{Init}_i) \Rightarrow (\forall\, i_1, \ldots, i_k \in [N] : \exists j_1, \ldots, j_m \in [N] : \varphi)$$

Moving the quantifiers of ψ to the front, we obtain:

$$\forall\, i_1, \ldots, i_k \in [N] : \exists\, j_1, \ldots, j_m \in [N] : (\forall\, i \in [N] : \mathsf{Init}_i \Rightarrow \varphi),$$
$$\forall\, i_1, \ldots, i_k \in [N] : \exists\, i, j_1, \ldots, j_m \in [N] : (\mathsf{Init}_i \Rightarrow \varphi),$$

which is in the required LH-assertion form.

From the definition of discrete transitions D_N, condition B can be written:

$$(\psi \wedge (\exists h \in [N] : \exists \mathsf{a} \in \mathsf{Act} : \mathbf{pre}(\mathsf{a}, h) \ \wedge \ \mathbf{eff}(\mathsf{a}, h)) \\ \wedge \forall g \in [N] : \ g \neq h \Rightarrow \mathbf{id}(g)]) \Rightarrow \psi'.$$

Here $\mathbf{id}(g)$ is a shorthand for the formula $\bigwedge_{x \in \mathsf{Var}_g} x'[g] = x[g]$ and ψ' is the formula obtained by replacing each variable in ψ with its primed version. Moving the quantifier to the front and rearranging we obtain the LH-assertion:

$$\forall h, \mathsf{a} : \exists g : (\psi \ \wedge \ \mathbf{pre}(\mathsf{a}, h) \ \wedge \ \mathbf{eff}(\mathsf{a}, h) \wedge \ (g \neq h \Rightarrow \mathbf{id}(g)) \Rightarrow \psi').$$

Exposing the quantifier in ψ and ψ':

$$\forall h, \mathsf{a} : \exists k : ((\forall \ i_1, \ldots, i_k : \exists \ j_1, \ldots, j_m : \varphi) \wedge \ \mathbf{pre}(\mathsf{a}, h) \ \wedge \ \mathbf{eff}(\mathsf{a}, h) \\ \wedge \ (g \neq h \Rightarrow \mathbf{id}(g)) \Rightarrow (\forall \ i'_1, \ldots, i'_k : \exists \ j'_1, \ldots, j'_m : \varphi)).$$

Moving quantifiers to the front of the formula, we obtain:

$$\forall h, \mathsf{a}, j_1, \ldots, j_m, i'_1, \ldots, i'_k : \exists \ g, i_1, \ldots, i_k, j'_1, \ldots, j'_m : \\ ((\varphi \ \wedge \ \mathbf{pre}(\mathsf{a}, h) \ \wedge \ \mathbf{eff}(\mathsf{a}, h) \wedge \ (g \neq h \Rightarrow \mathbf{id}(g))) \Rightarrow \varphi).$$

As a is universally quantified over the finite set of actions Act_i, it is removed by writing the above as a finite conjunction of LH-assertions.

Finally, by the definition of trajectories T_N, condition C can be written as:

$$\psi \wedge (\exists t_1 \in \mathbb{R} : \forall h \in [N], t_2 \in \mathbb{R} : \exists \mathsf{m} \in \mathsf{Loc} : \mathbf{inv}(\mathsf{m}, h) \wedge (\mathbf{stop}(\mathsf{m}, h) \Rightarrow t_2 = t_1) \\ \bigwedge_{x \in \mathsf{Var}^c_h} x[h] + t_1 \mathbf{lflowrate}(\mathsf{m}, h, x) \leq x'[h] \leq x[h] + t_1 \mathbf{uflowrate}(\mathsf{m}, h, x)) \\ \Rightarrow \psi', \tag{5}$$

where **inv** and **stop** have all continuously updated real array variables replaced with primed versions using a time-elapse of t_2, and Var^c_h are the continuously updated real array variables. The conversion of Equation 5 to an LH-assertion is essentially the same as the discrete case, but more tedious. The easiest way to do this is first to convert to prenex normal form. In summary, these derivations show how we can check inductive invariants as LH-assertions for networks of RHA using the small model result introduced above.

4 Passel: Tool Implementation and Results

We developed a prototype tool called Passel—which is a large group of people or things of indeterminate number—for verifying that properties are inductive invariants. Passel utilizes the satisfiability modulo theories (SMT) [17] solver Z3 [16] to prove validity of an LH-assertion by checking unsatisfiability of the

Table 1. Example properties and results for $2 \leq N \leq 100$, which exceeds the threshold from Lemma 1 for each property and example. SATS properties are shown in Subsection 2.3 and Fischer properties are shown in the last paragraph of Section 4. A check in the "Correct" column means that a property was shown to be an inductive invariant, while an "X" indicates not, and similarly for buggy versions of the protocols as indicated in the "Buggy" column. Time is the runtime in seconds. QI is the number of quantifier instantiations. The results in this table had both MBQI and quantifier elimination enabled.

Example	Property	Correct	Time (s)	QI	Buggy	Time (s)	QI
SATS	A	✓	0.47	166	✓	24.429	63
	B	✓	0.591	373	✓	0.595	197
	C	✓	0.586	703	✗	1.041	113485
	D	✓	0.757	8298	✗	1.256	1659
Timed SATS	A	✓	0.349	66	✓	0.34	61
	B	✓	0.304	673	✓	0.317	460
	C	✓	0.244	373	✗	1.763	140512
	E	✓	0.467	3032	✗	2.204	26958
Fischer	a	✓	0.498	305	✓	0.491	305
	b	✓	0.33	204	✓	0.325	204
	c	✓	0.376	544	✓	0.33	544
	d	✓	0.396	618	✗	0.533	2548
	e	✓	0.435	1306	✗	0.532	1202
	f	✓	0.414	1036	✗	0.437	3162

negation of the assertion. Our program is written in C# and we used the managed .NET API to version 3.2 of Z3. The input to Passel comes from the HyXML format for describing hybrid automata developed for Hylink [33]. We configured Z3 to use a variety of options, in particular, our method requires either having model-based quantifier instantiation (MBQI) enabled or quantifier elimination enabled, as otherwise we may receive unknown as a response from Z3 for some satisfiability checks. Within Z3, we model array variables of processes as functions mapping a subset of the integers (i.e., the set of process indices) to the type of the variable. We did not need to use any special encoding to represent our systems for Z3, so the queries we ask are almost exactly the same as the formulas appearing in Subsection 3.1. Given the finite bound N_0 from Lemma 1, we could potentially have composed the system for each instantiation $2 \leq N \leq N_0$ and used existing tools (for instance, HyTech [28] or PHAVer [22]), but the LH-assertions specifications were more natural to state in an environment that allows quantifiers. Additionally, our prior experience in model checking such parameterized systems [31] indicated that the bound allowed in practice due to memory requirements may be less than the bound N_0, and may prevent verification.

The operation of checking an inductive invariant is as follows. The user specifies a set $\mathcal{I}$ of candidate inductive invariants. Based on the protocol specification, we receive from Lemma 1 a bound N_0 on the number of processes N for which we must check each property. Many of the invariant properties we are interested in are not inductive (e.g., mutual exclusion in Fischer's protocol is not inductive, nor even k-inductive [20]), so having a set of candidate invariants allows us to discharge each until we have a set of proven lemmas that imply the desired invariant. For each candidate invariant $I \in \mathcal{I}$, we check if I is an inductive invariant by attempting to prove I' after each transition, where I' is I with all variables replaced with their primed counterparts (i.e., post-states). If I is successfully proved, we assert I as an assumption lemma and check some other $J \in \mathcal{I}$ for $J \neq I$ until we have proved—or failed to prove—each property in $\mathcal{I}$. We emphasize that if we do not prove a property, it does not necessarily mean that the property does not hold, only that it may not be inductive.

The main difficulty is specifying a rich enough set of properties $\mathcal{I}$. We perform satisfiability checks with model construction enable, thus if a transition or time elapse violates the candidate property, we record it and display it to the user so she/he may use this information to refine the candidate manually. For Fischer, a detailed refinement is performed in [20], and we used several of these refined properties in the set $\mathcal{I}$. In addition to SATS and Fischer, we considered several other examples[2] We performed the verification on a laptop with a quad-core Intel I7 2.0GHz processor and 8GB RAM.

The buggy version of SATS replaces the precondition $l[next[i]]$ with $l[last]$ and $x[next[i]]$ with $x[last]$ in Figure 1, line 24, which ensures the spacing between i and the last aircraft is large enough. However, this may not be the aircraft immediately ahead of i, for instance, if two aircraft have moved to the base, so the safe separation properties do not hold. The correct version of Fischer's mutual exclusion protocol has a constraint $A < B$, and the buggy version has $A \geq B$ [20]. We checked the following properties for Fischer (see also Table 1):

(a) $\forall i, j \in [N] : x[i] = x[j]$,
(b) $\forall i \in [N] : q[i] = set \Rightarrow last[i] \leq x[i] + A$,
(c) $\forall i \in [N] : q[i] = set \Rightarrow x[i] \leq last[i]$,
(d) $\forall i, j \in [N] : (q[i] = check \wedge g = i \wedge q[j] = set) \Rightarrow first[i] > last[j]$,
(e) $\forall i, j \in [N] : q[i] = crit \Rightarrow (g = i \wedge q[j] = set)$, and
(f) $\forall i, j \in [N] : (i = j) \Rightarrow (q[i] = crit \vee q[j] = crit)$,

where f specifies mutual exclusion.

5 Conclusion and Future Work

In this paper, we developed a small model theorem for networks of rectangular hybrid automata (RHA), and used the theorem to establish inductive invariant properties for several case studies. To the best of our knowledge, this is the first positive result on automatic parameterized verification of hybrid automata, beyond previous results for timed automata [4,3,25,14,21,13]. The modeling framework and process of inductive invariant checking are amenable to automation, so we implemented a prototype called Passel using the SMT solver Z3. We plan

[2] Passel and case study specification files are available from:
`http://www.taylortjohnson.com/research/forte2012/`

to continue development of Passel, and investigate other methods of verifying such RHA networks like automated abstraction and invariant generation. One weakness of the current method is that the user is required to specify all the inductive invariants that will imply the desired invariant property (for instance, mutual exclusion in Fischer is not an inductive invariant, so one must find a stronger property that implies mutual exclusion and is an inductive invariant; a similar process was necessary for the SATS example). While we aid the user in this task by providing counterexample models, she/he must still manually perform the strengthening, so methods to automatically generate or strengthen invariants—such as invisible invariants [37,8,9]—or k-induction [12,19] would be interesting to investigate. It will also be interesting to investigate parameterized verification for hybrid automata with linear or nonlinear dynamics.

Acknowledgments. This paper is based upon work supported by the National Science Foundation under CAREER Grant No. 1054247. The authors thank the anonymous reviewers for their suggestions that helped improve the presentation of this paper.

References

1. Abbott, T.S., Jones, K.M., Consiglio, M.C., Williams, D.M., Adams, C.A.: Small aircraft transportation system, higher volume operations concept: Normal operations. Tech. Rep. NASA/TM-2004-213022, NASA (August 2004)
2. Abdulla, P., Delzanno, G., Rezine, A.: Parameterized Verification of Infinite-State Processes with Global Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
3. Abdulla, P.A., Deneux, J., Mahata, P.: Multi-clock timed networks. In: Proc. of 19th Annual IEEE Symposium Logic in Computer Science, pp. 345–354 (July 2004)
4. Abdulla, P.A., Jonsson, B.: Model checking of systems with many identical timed processes. Theoretical Computer Science 290(1), 241–264 (2003)
5. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
6. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)
7. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (1986)
8. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.: Parameterized Verification with Automatically Computed Inductive Assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
9. Balaban, I., Fang, Y., Pnueli, A., Zuck, L.: IIV: An Invisible Invariant Verifier. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 408–412. Springer, Heidelberg (2005)
10. Bjørner, N.: Linear Quantifier Elimination as an Abstract Decision Procedure. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 316–330. Springer, Heidelberg (2010)

11. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Springer (2001)
12. Brown, G., Pike, L.: Easy Parameterized Verification of Biphase Mark and 8N1 Protocols. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 58–72. Springer, Heidelberg (2006)
13. Bruttomesso, R., Carioni, A., Ghilardi, S., Ranise, S.: Automated Analysis of Parametric Timing-Based Mutual Exclusion Algorithms. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 279–294. Springer, Heidelberg (2012)
14. Carioni, A., Ghilardi, S., Ranise, S.: MCMT in the land of parameterized timed automata. In: Proc. of VERIFY 2010 (July 2010)
15. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
16. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM 54, 69–77 (2011)
18. Delzanno, G.: Automatic Verification of Parameterized Cache Coherence Protocols. In: Emerson, E., Sistla, A. (eds.) CAV 2000. LNCS, vol. 1855, pp. 53–68. Springer, Heidelberg (2000)
19. Donaldson, A., Haller, L., Kroening, D., Rümmer, P.: Software Verification Using k-Induction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011)
20. Dutertre, B., Sorea, M.: Timed systems in sal. Tech. Rep. SRI-SDL-04-03, SRI International (October 2004)
21. Faber, J., Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: Automatic Verification of Parametric Specifications with Complex Topologies. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 152–167. Springer, Heidelberg (2010)
22. Frehse, G.: PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
23. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
24. Ge, Y., de Moura, L.: Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
25. Grinchtein, O., Leucker, M.: Network invariants for real-time systems. Formal Aspects of Computing 20, 619–635 (2008)
26. Hanna, Y., Samuelson, D., Basu, S., Rajan, H.: Automating Cut-off for Multi-parameterized Systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 338–354. Springer, Heidelberg (2010)
27. Henzinger, T.A.: The theory of hybrid automata. In: IEEE Symposium on Logic in Computer Science (LICS), p. 278. IEEE Computer Society, Washington, DC (1996)
28. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: Hytech: a model checker for hybrid systems. Journal on Software Tools for Technology Transfer 1, 110–122 (1997)
29. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? Journal of Computer and System Sciences 57, 94–124 (1998)

30. Johnson, T.T., Mitra, S.: Safe flocking in spite of actuator faults using directional failure detectors. Journal of Nonlinear Systems and Applications 2(1-2), 73–95 (2011)
31. Johnson, T.T., Mitra, S.: Parameterized verification of distributed cyber-physical systems: An aircraft landing protocol case study. In: ACM/IEEE 3rd International Conference on Cyber-Physical Systems (April 2012)
32. Loos, S.M., Platzer, A., Nistor, L.: Adaptive Cruise Control: Hybrid, Distributed, and Now Formally Verified. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 42–56. Springer, Heidelberg (2011)
33. Manamcheri, K., Mitra, S., Bak, S., Caccamo, M.: A step towards verification and synthesis from simulink/stateflow models. In: Proc. of the 14th Intl. Conf. on Hybrid Systems: Computation and Control, pp. 317–318. ACM (2011)
34. Muñoz, C., Carreño, V., Dowek, G.: Formal analysis of the operational concept for the small aircraft transportation system. In: Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.) Fault-Tolerant Systems, LNCS, vol. 4157, pp. 306–325. Springer Berlin / Heidelberg (2006)
35. Platzer, A.: Quantified Differential Dynamic Logic for Distributed Hybrid Systems. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 469–483. Springer, Heidelberg (2010)
36. Platzer, A.: Quantified differential invariants. In: Proc. of the 14th ACM Intl. Conf. on Hybrid Systems: Computation and Control, pp. 63–72. ACM (2011)
37. Pnueli, A., Ruah, S., Zuck, L.: Automatic Deductive Verification with Invisible Invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
38. Umeno, S., Lynch, N.: Safety Verification of an Aircraft Landing Protocol: A Refinement Approach. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 557–572. Springer, Heidelberg (2007)
39. Viken, S., Brooks, F.: Demonstration of four operating capabilities to enable a small aircraft transportation system. In: The 24th Digital Avionics Systems Conference, DASC 2005, vol. 2 (October 2005)
40. Wolper, P., Lovinfosse, V.: Verifying Properties of Large Sets of Processes with Network Invariants. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 68–80. Springer, Heidelberg (1990)

Analysis of May-Happen-in-Parallel in Concurrent Objects[⋆]

Elvira Albert, Antonio E. Flores-Montoya, and Samir Genaim

Complutense University of Madrid, Spain

Abstract. This paper presents a *may-happen-in-parallel* (MHP) analysis for OO languages based on *concurrent objects*. In this concurrency model, objects are the concurrency *units* such that, when a method is invoked on an object o_2 from a task executing on object o_1, statements of the current task in o_1 may run in parallel with those of the (asynchronous) call on o_2, and with those of transitively invoked methods. The goal of the MHP analysis is to identify pairs of statements in the program that may run in parallel in any execution. Our MHP analysis is formalized as a method-level (*local*) analysis whose information can be modularly composed to obtain application-level (*global*) information.

1 Introduction

The actor-based paradigm [2] on which concurrent objects are based has lately regained attention as a promising solution to concurrency in OO languages. For many application areas, standard mechanisms like threads and locks are too low-level and have been shown to be error-prone and, more importantly, not *modular* enough. The concurrent objects model is based on considering objects as the concurrency units i.e., each object conceptually has a dedicated processor. Communication is based on asynchronous method calls with standard objects as targets. An essential feature of this paradigm is that task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. Data-driven synchronization is possible by means of so-called *future* variables [7] as follows. Consider an asynchronous method call m on object o, written as f=o.m(). Here, the variable f is a future which allows synchronizing with the result of executing task m. In particular, the instruction **await** f? allows checking whether m has finished, and lets the current task release the processor to allow another available task to take it.

This paper develops a *may-happen-in-parallel* (MHP) analysis for concurrent objects. The goal of an MHP analysis is to identify pairs of statements that

⋆ This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

can execute in parallel (see, e.g., [10]). In the context of concurrent objects, an asynchronous method invocation $\mathsf{f{=}o_2.m()}$; within a task t_1 executing in an object $\mathsf{o_1}$ implies that the subsequent instructions of t_1 in $\mathsf{o_1}$ may execute in parallel with the instructions of m within $\mathsf{o_2}$. However, if the asynchronous call is synchronized with an instruction **await** f?, after executing such an *await*, it is ensured that the execution of the call to m has terminated and hence, the instructions after the **await** cannot execute in parallel with those of m. Inferring precise MHP information is challenging because, not only does the current task execute in parallel with m, but also with other tasks that are *transitively* invoked from m. Besides, two tasks can execute in parallel even if they do not have a transitive invocation relation. For instance, if we add an instruction $\mathsf{f_3{=}o_3.p()}$; below the previous asynchronous invocation to m in t_1, then instructions in p may run in parallel with those of m. This is a form of *indirect* MHP relation in which tasks run in parallel because they have a common ancestor. The challenge is to precisely capture in the analysis all possible forms of MHP relations.

It is widely recognized that MHP is an analysis of utmost importance [10] to understand the behaviour and verify the soundness of concurrent programs. On one hand, it is a basic analysis to later construct different kinds of verification and testing tools which build on it in order to infer more complex properties. For example, in order to prove termination (or infer the cost) of a simple loop of the form **while** (l!=**null**) {f=o.process(l.data); **await** f?; l=l.next;}, assuming l is a shared variable (i.e., field), we need to know the tasks that can run in parallel with the body of the loop to check whether the length of the list l can be modified during the execution of the loop by some other task when the processor is released (at the **await**). For concurrent languages which are not data-race free, MHP is fundamental in order to verify the absence of data-races. On the other hand, it provides very useful information to automatically extract the maximal level of parallelism for a program and improve performance. In the context of concurrent objects, when the methods running on two different objects may run in parallel, it can be profitable to deploy such objects on different machines in order to improve the overall performance. As another application, the programmer can use the results of the MHP analysis to identify bugs in the program related to fragments of code which should not run in parallel, but where the analysis spots possible parallel execution.

This paper proposes a novel MHP analysis for concurrent objects. The analysis has two main phases: we first infer *method-level* MHP information by locally analyzing each method and ignoring transitive calls. This local analysis, among other things, collects the *escape* points of method calls, i.e., those program points in which the asynchronous calls terminate but there might be transitive asynchronous calls not finished. In the next step, we modularly compose the method-level information in order to obtain *application-level* (*global*) MHP information. The composition is achieved by constructing an *MHP analysis graph* which over-approximates the parallelism –both implicit and through transitive calls– in the application. Then, the problem of inferring if two statements x and y can run in parallel amounts to checking certain *reachability* conditions between x and y in the MHP analysis graph. We have implemented our analysis in COSTABS [3], a cost and termination analyzer for the ABS language. ABS [8] is an actor-like language which has been recently proposed to model distributed concurrent objects. The implementation has been evaluated on small applications which are classical examples of concurrent programming and on two industrial case studies. Results on the efficiency and accuracy of the analysis, in spite of being still prototypical, are promising.

2 Concurrent Objects

We describe the syntax and semantics of the simple imperative language with concurrent objects on which we develop our analysis. It is basically the subset of the ABS language [8] relevant to the MHP analysis. Class, method, field, and variable names are taken from a set $\mathcal{X}$ of valid *identifiers*. A *program* consists of a set of classes $\mathcal{K} \subseteq \mathcal{X}$. The set *Types* is the set of possible types $\mathcal{K} \cup \{\textsf{int}\}$, and the set $Types_{\mathcal{F}}$ is the set of future variable types defined as $\{Fut\langle t\rangle \mid t \in Types\}$. A *class declaration* takes the form:

$$\textsf{class}\ \kappa_1\ \{t_1\ fn_1; \ldots\ t_n\ fn_n;\ M_1\ \ldots\ M_k\}$$

where each "$t_i\ fn_i$" declares a field fn_i of type $t_i \in Types$, and each M_i is a method definition. A *method definition* takes the form

$$t\ m(t_1\ w_1, \ldots, t_n\ w_n)\ \{t_{n+1}\ w_{n+1}; \ldots t_{n+p}\ w_{n+p};\ s\}$$

where $t \in Types$ is the type of the return value; $w_1, \ldots, w_n \in \mathcal{X}$ are the formal parameters of types $t_1, \ldots, t_n \in Types$; $w_{n+1}, \ldots, w_{n+p} \in \mathcal{X}$ are local variables of types $t_{n+1}, \ldots, t_{n+p} \in Types \cup Types_{\mathcal{F}}$; and s is a sequence of instructions which adhere to the following grammar:

$$\begin{aligned}
e &::= \textsf{null} \mid \textsf{this}.f \mid x \mid n \mid e+e \mid e*e \mid e-e\\
b &::= e>e \mid e=e \mid b \wedge b \mid b \vee b \mid !b\\
s &::= instr \mid instr; s\\
instr &::= x{=}\textsf{new}\ \kappa(\bar{x}) \mid x{=}e \mid \textsf{this}.f{=}e \mid y{=}x.m(\bar{z}) \mid \textsf{return}\ x\\
&\quad\ \textsf{if}\ b\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \mid \textsf{while}\ b\ \textsf{do}\ s \mid \textsf{await}\ y? \mid x{=}y.\textsf{get}
\end{aligned}$$

There is an implicit local variable called **this** that refers to the current object. x and y represent variables of types $t \in Types$ and $ft \in Types_{\mathcal{F}}$ respectively. Observe that only fields of the current object **this** can be accessed (this, together with the semantics, make the language be data-race free [8]). We assume the program includes a method called main without parameters, which does not belong to any class and has no fields, from which the execution will start. Data synchronization is by means of future variables as follows. An **await** $y?$ instruction is used to synchronize with the result of executing task $y{=}x.m(\bar{z})$ such that the **await** $y?$ is executed only when the future variable y is available (i.e., the

$$(1)\quad \frac{(O', l', s') = eval(instr, O, l, oid) \quad instr \in \{x{=}e, \textsf{this}.fn{=}e, x{=}\textsf{new}\ \kappa(\bar{x}), \textsf{if}\ b\ \textsf{then}\ s_1\ \textsf{else}\ s_2, \textsf{while}\ b\ \textsf{do}\ s_3\}}{\langle O, \{\langle tid, m, oid, \top, l, instr; s\rangle \parallel T\}\rangle \rightsquigarrow \langle O', \{\langle tid, m, oid, \top, l', s'; s\rangle \parallel T\}\rangle}$$

$$(2)\quad \frac{l(x) = oid_1 \neq \textsf{null}, l' = l[y \rightarrow tid_1], l_1 = buildLocals(\bar{x}, m)), tid_1 \text{is a fresh id}}{\begin{array}{c}\langle O, \{\langle tid, m, oid, \top, l, y{=}x.m_1(\bar{x}); s\rangle \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{\langle tid, m, oid, \top, l', s\rangle, \langle tid_1, m_1, oid_1, \bot, l_1, body(m_1)\rangle \parallel T\}\rangle\end{array}}$$

$$(3)\quad \frac{\langle oid, \bot, f\rangle \in O, O' = O[\langle oid, \bot, f\rangle / \langle oid, \top, f\rangle], v = l(x)}{\langle O, \{\langle tid, m, oid, \top, l, \textsf{return}\ x\rangle\} \parallel T\}\rangle \rightsquigarrow \langle O', \{\langle tid, m, oid, \bot, l, \epsilon(v)\rangle \parallel T\}\rangle}$$

$$(4)\quad \frac{l_1(y) = tid_2}{\begin{array}{c}\langle O, \{\langle tid_1, m_1, oid_1, \top, l_1, \textsf{await}\ y?; s_1\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v)\rangle \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{\langle tid_1, m_1, oid_1, \top, l_1, s_1\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v)\rangle \parallel T\}\rangle\end{array}}$$

$$(5)\quad \frac{}{\begin{array}{c}\langle O, \{\langle tid_1, m_1, oid_1, lk, l_1, \textsf{await}\ y?; s_1\rangle \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{\langle tid_1, m_1, oid_1, lk, l_1, \textsf{release}; \textsf{await}\ y?; s_1\rangle \parallel T\}\rangle\end{array}}$$

$$(6)\quad \frac{\langle oid, \bot, f\rangle \in O, O' = O[\langle oid, \bot, f\rangle / \langle oid, \top, f\rangle]}{\langle O, \{\langle tid, m, oid, \top, l, \textsf{release}; s\rangle \parallel T\}\rangle \rightsquigarrow \langle O', \{\langle tid, m, oid, \bot, l, s\rangle \parallel T\}\rangle}$$

$$(7)\quad \frac{\langle oid, \top, f\rangle \in O, O' = O[\langle oid, \top, f\rangle / \langle oid, \bot, f\rangle], s \neq \epsilon(v)}{\langle O, \{\langle tid, m, oid, \bot, l, s\rangle \parallel T\}\rangle \rightsquigarrow \langle O', \{\langle tid, m, oid, \top, l, s\rangle \parallel T\}\rangle}$$

$$(8)\quad \frac{l_1(y) = tid_2, l_1' = l_1[x \rightarrow v]}{\begin{array}{c}\langle O, \{\langle tid_1, m_1, oid_1, \top, l_1, x{=}y.\textsf{get}; s_1\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v)\rangle \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{\langle tid_1, m_1, oid_1, \top, l_1', s_1\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v)\rangle \parallel T\}\rangle\end{array}}$$

Fig. 1. Summarized semantics

task is finished). In the meantime, the processor can be released and some other pending task on this object can take it. In contrast, the instruction y.**get** unconditionally blocks the processor (no other task of the same object can run) until y is available, i.e., the execution of $m(\bar{z})$ on x is finished. Note that class fields and methods parameters cannot have future types, i.e, future variables are defined locally in each method and cannot be passed over. This is a restriction of the approach, however, programs that pass futures over can still be analyzed with some loss of precision by ignoring the non-local future variables.

W.l.o.g, we assume that all methods in the program have different names. As notation, we use $body(m)$ for the sequence of instructions defining method m, $P_{\mathcal{M}}$ for the set of method names defined in a program P, $P_{\mathcal{F}}$ for the set of future variable names defined in a program P.

2.1 Operational Semantics

A program state S is a tuple $S = \langle O, T\rangle$ where O is a set of objects and T is a set of tasks. Only one task can be *active* (running) in each object and has the object's *lock*. All other tasks are *pending* to be executed or *finished* if they terminated and released the lock. The set of objects O includes all available objects. An object takes the form $\langle oid, lk, f\rangle$ where oid is a unique identifier taken from an infinite set of identifiers $\mathcal{O}$, $lk \in \{\top, \bot\}$ indicates whether the object's lock is free ($\top$) or not ($\bot$), and $f : \mathcal{X} \rightarrow \mathcal{O} \cup \mathbb{Z} \cup \{\textbf{null}\}$ is a partial mapping from object fields to values. The set of tasks T represents those tasks that are being executed. Each task takes the form $\langle tid, m, oid, lk, l, s\rangle$ where tid is a unique identifier of the task taken from an infinite set of identifiers $\mathcal{T}$, m is the method name executing in the task, oid identifies the object to which the task belongs, $lk \in \{\top, \bot\}$ is a flag that indicates if the task has the object's lock or not, $l : \mathcal{X} \rightarrow \mathcal{O} \cup \mathcal{T} \cup \mathbb{Z} \cup \{\textbf{null}\}$ is a partial mapping from local (possibly future) variables to their values, and s is the sequence of instructions still to be executed. Given a task tid, we assume that $object(tid)$ returns the object identifier oid of the corresponding task. The execution of a program starts from the initial state $S_0 = \langle\{\langle 0, \bot, f\rangle\}, \{\langle 0, \mathsf{main}, 0, \top, l, body(\mathsf{main})\rangle\}\rangle$ where f is an empty mapping (since main had no fields), and l maps local references and future variables to **null** and integer variables to 0.

The execution proceeds from S_0 by applying *non-deterministically* the semantic rules depicted in Fig. 1. We use the notation $\{t \parallel T\}$ to represent that task t is non-deterministically selected for execution. The operational semantics is given in a rewriting-based style where at each step a subset of the state is rewritten according to the rules as follows: (1) executes an instruction in a task that has its object lock. These instructions may change the heap (global state), the local state and the sequence of instructions that are left to execute (in the case of an if-then-else or a while instruction). Such changes are captured in function *eval*. As the instructions executed in this rule are standard, they are summarized. (2) A method call creates a new task (the initial state is created by *buildLocals*) with a fresh task identifier which is associated to the corresponding future variable. (3) When **return** is executed, the return value is stored in v so that it can be obtained by the future variables that point to that task. Besides, the lock is released and will never be taken again by that task (the notation $O[o/o']$ is used to replace o by o' in O). Consequently, that task is *finished* (marked by adding the instruction $\epsilon(v)$), though it does not disappear as other tasks might need to access its return value. (4) If the future variable we are awaiting for points to a finished task, the await can be completed. (5) The await can be substituted by a release plus an await. This allows us to await until rule (4) can be applied. (6) A task executes a release and yields the lock so that any other task of the same object can take it. (7) A non finished task can obtain its object lock if it is unlocked. (8) A y.**get** instruction waits for the future variable but without yielding the lock. It then retrieves the value associated with the future variable y.

3 Definition of MHP

We first formally define the concrete property "MHP" that we want to approximate using static analysis. In what follows, we assume that instructions are

A	B	C	D	E
1 **int** m() {	11 **int** m() {	21 **int** m() {	31 **int** m() {	41 **int** p() {
2 ...	12 ...	22 ...	32 ...	42 y=x.r();
3 y=x.p();	13 y=**this**.r();	23 **while** b **do**	33 **if** b **then**	43 ...
4 z=x.q();	14 z=x1.p();	24 y=x.q();	34 y=x.p();	44 }
5 ...	15 z=x2.p();	25 **await** y?;	35 **else**	45 **int** q() {
6 **await** z?;	16 z=x3.q();	26 z=x.p();	36 y=x.q();	46 y=x.r();
7 ...	17 w=z.**get**;	27 ...	37 ...	47 **await** y?;
8 **await** y?;	18 ...	28 ...	38 **await** y?;	48 ...
9 ...	19 **await** y?;	29 ...	39 ...	49 ...
10 }	20 }	30 }	40 }	50 }

Fig. 2. Simple examples for different MHP behaviours

labelled such that it is possible to obtain the corresponding program point identifiers. We also assume that program points are globally different. We use $p_{\mathring{m}}$ to refer to the entry program point of method m, and $p_{\dot{m}}$ to all program points after its **return** instruction. The set of all program points of P is denoted by $P_{\mathcal{P}}$. We write $p \in m$ to indicate that program point p belongs to method m. Given a sequence of instructions s, we use $pp(s)$ to refer to the program point identifier associated with its first instruction, $pp(\epsilon(v)) = p_{\dot{m}}$ and and $pp(\textbf{release}; s) = pp(s)$.

Definition 1 (concrete MHP). *Given a program P, its MHP is defined as $\mathcal{E}_P = \cup\{\mathcal{E}_S | S_0 \rightsquigarrow^* S\}$ where for the state $S = \langle O, Tk\rangle$, the set $\mathcal{E}_S$ is $\mathcal{E}_S = \{(pp(s_1), pp(s_2)) \mid \langle id_1, m_1, o_1, lk_1, l_1, s_1\rangle \in Tk, \langle id_2, m_2, o_2, lk_2, l_2, s_2\rangle \in Tk, id_1 \neq id_2\}$.*

Observe in the above definition that, as execution is non-deterministic (and different MHP behaviours can actually occur using different task scheduling strategies), the union of the pairs obtained from all derivations from S_0 is considered.

Let us explain first the notions of *direct* and *indirect* MHP and *escaped* methods, which are implicit in the definition of MHP above, on the simple representative patterns in Fig. 2. There are 4 versions of m which use the methods p, q and r. We consider a call to m with no other processes executing. Only the parts of p and q useful for explaining the MHP behavior are shown (the code of r is irrelevant). We implicitly assume that the last instruction of each method is a **return**. The global MHP behavior of executing each main (separately) is as follows.

(A) p is called from m, then r is called from p and q. The await instruction in program point 6 (L6 for short) ensures that q will have finished afterwards. If q has finished executing, its call to r has to be finished as well because there is an await in L47. The await instruction in L8 waits until p has finished before continuing. That means that at L9, p is not longer executing. However, the call to r from p might be still executing. We say that r might *escape* from p. Method calls that might escape need to be considered.

(B) In example B, both q and p are called from m, but p is called twice. Any program point of p, for example L43, might execute in parallel with q even if they do not call each other, i.e., they have an *indirect* MHP relation. Furthermore, L43 might execute in parallel with any point of m after the method call, L15 – 17. We say that m is a common *ancestor* of p and q. Two methods execute indirectly in parallel if they have a common ancestor. Note that m is also a common ancestor of the two instances of p, so p might execute in parallel with itself. r is called in L13. However, as r belongs to the same object as m, it will not be able to start executing until m reaches a release point (L19). We say that r is *pending* from L14 up to L19.

(C) In the third example we have a **while** loop. If we do not estimate the number of iterations, we can only assume that q and p are called an arbitrary number of times. However, as every call to q has a corresponding await, q will not execute in parallel with itself. At L28, we might have any number of p instances executing but none of q. Note that if any method escaped from q, it could also be executing at L28.

(D) The last example illustrates an **if** statement. Either p or q is executed but not both. At L37, p or q might be executing but p and q cannot run in parallel even if m is a common ancestor. Furthermore, after the await instruction (L38) neither q or p might be executing. This information will be extracted from the fact that both calls use the same future variable.

4 MHP Analysis

The problem of inferring $\mathcal{E}_P$ is clearly undecidable in our setting [9], and thus we develop a MHP analysis which statically approximates $\mathcal{E}_P$. The analysis is done in two main steps, first it infers method-level MHP information. Then, in order to obtain application-level MHP, it composes this information by building a MHP graph whose paths provide the required global MHP information.

4.1 Inference of Method-Level MHP

The method-level MHP analysis is used to infer the local effect of each method on the global MHP property. In particular, for each method m, it infers, for each program point $p \in m$, the status of all tasks that (might) have been invoked (within m) so far. The status of a task can be (1) *pending*, which means that it has been invoked but has not started to execute yet, i.e., it is at the entry program point; (2) *finished*, which means that it has finished executing already, i.e., it is at the exit program point; and (3) *active*, which means that it can be executing at any program point (including the entry and the exit). As we explain later, the distinction between these statuses is essential for precision.

The analysis of each method abstractly executes its code such that the (abstract) state at each program point is a multiset of symbolic values that describes the status of all tasks invoked so far. Intuitively, when a method is invoked, we add it to the multiset (as pending or active depending if it is a call on the same object or on a different object); when an **await** $y?$ or y.**get** instruction is executed, we change the status of the corresponding method to finished; and when the execution passes through a release point (namely **await** $y?$ or **return**), we change the status of all pending methods to active.

Example 1. Consider programs A and B in Fig. 2. The call to p (resp. q) at L3 (resp. L4) creates an *active* task that becomes *finished* at L8 (resp. L6). In B, the call to r at L13 creates a *pending* task that becomes *active* at L19 and *finished* after L19. p is an *active* task from L14 up to the end of the method. p will never become a *finished* task as its associated future variable is reused in L16.

The symbolic values used to describe the status of a task, referred to as MHP *atoms*, can be one of the following: (1) $y{:}\tilde{m}$, which represents an *active* task that is an instance of method m; (2) $y{:}\hat{m}$, which represents a *finished* task that is an instance of method m; and (3) $y{:}\check{m}$, which represents a *pending* task that is an instance of method m. In the three cases the task is associated to the future variable y. In addition, since it is not always possible to relate tasks to future variables (e.g., if they are reused), we also allow symbolic values in which y is replaced by $\star$, i.e., $\star$ represents any future variable.

Intuitively, an abstract state M is a multiset of MHP atoms which represents the following information: each $y{:}x \in M$ (resp. $\star{:}x \in M$) represents *one* task that *might* be available and associated to future variable y (resp. to any future variable). The status of the task is active, pending or finished, resp., if $x = \tilde{m}$, $x = \check{m}$ or $x = \hat{m}$. In addition, we can have several tasks associated to the same future variable meaning that at most one of them can be available at the same time (since only one task can be associated to a future variable in the semantics).

Example 2. Consider programs A, B and D. The multisets $\{y{:}\tilde{p}, z{:}\tilde{q}\}$, $\{y{:}\tilde{p}, z{:}\hat{q}\}$, $\{y{:}\hat{p}, z{:}\hat{q}\}$, $\{y{:}\check{r}, z{:}\tilde{p}\}$, $\{y{:}\check{r}, \star{:}\tilde{p}, \star{:}\tilde{p}, z{:}\hat{q}\}$ and $\{y{:}\tilde{p}, y{:}\tilde{q}\}$ resp. describe the abstract states at L5, L7, L9, L15, L18 and L37. An important observation is that, in the multiset of L18, when the future variable is reused, its former association is lost (and hence becomes $\star$). However, multiple associations to one future variable can be kept when they correspond to disjunctive branches, as in L37.

For a given program P, the set of all MHP atoms $\mathcal{A} = \{y{:}x \mid m \in P_{\mathcal{M}}, x \in \{\tilde{m}, \hat{m}, \check{m}\}, y \in P_{\mathcal{F}} \cup \{\star\}\}$ is a partially order set w.r.t. the partial order relation $\preceq$ defined as in the diagram below (we use $\prec$ for strict inequality and $=$ for syntactic equality). The meaning of $a \preceq a'$ is that concrete scenarios described by a, are also described by a'. For example, $y{:}\check{m} \preceq y{:}\tilde{m}$ because $y{:}\check{m}$ is included in the description of $y{:}\tilde{m}$ since an active task can be at any program point (including the entry program point). The set of all multisets over $\mathcal{A}$ is denoted by $\mathcal{B}$. We write $(a, i) \in M$ to indicate that a appears exactly $i > 0$ times in M. In the examples, we omit i when it is 1. Given $M_1, M_2 \in \mathcal{B}$, we say that $a \in M_2$ *covers* $a' \in M_1$ if $a' \preceq a$. Thus, $M_1 \sqsubseteq M_2$ if all elements of M_1 are covered by *different* elements from M_2.

$$\begin{tikzcd}
 & & \star{:}\tilde{m} & & \\
\star{:}\check{m} \arrow[urr] & & y{:}\tilde{m} \arrow[u] & & \star{:}\hat{m} \arrow[ull] \\
 & y{:}\check{m} \arrow[ul] \arrow[ur] & & y{:}\hat{m} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

$$\begin{array}{ll}
(1) & \tau(y{=}x.m(\bar{x}), M) = M[y{:}x/\star{:}x] \cup \{y{:}\tilde{m}\} \qquad x \in \{\check{m}, \tilde{m}, \hat{m}\} \\
(2) & \tau(y{=}\textbf{this}.m(\bar{x}), M) = M[y{:}x/\star{:}x] \cup \{y{:}\check{m}\} \qquad x \in \{\check{m}, \tilde{m}, \hat{m}\} \\
(3) & \tau(\textbf{await}\ y?, M) = \tau(x{=}y.\textbf{get}, M) = M[y{:}z/y{:}\hat{m}] \qquad z \in \{\check{m}, \tilde{m}\} \\
(4) & \tau(\textbf{release}, M) = \tau(\textbf{return}, M) = M[y{:}\check{m}/y{:}\tilde{m}] \qquad \forall y \\
(5) & \tau(b, M) = M \qquad \text{otherwise}
\end{array}$$

Fig. 3. Method-level MHP transfer function: $\tau : s \times \mathcal{B} \mapsto \mathcal{B}$

Note that for two different $M_1, M_2 \in \mathcal{B}$, it might be the case that $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_1$, in such case they represent the same concrete states. This happens because when $(a, \infty) \in M$, then any $(a', i) \in M$ is redundant if $a' \preceq a$. The join (or upper bound) of M_1 and M_2, denoted $M_1 \sqcup M_2$, is an operation that calculates a multiset $M_3 \in \mathcal{B}$ such that $M_1 \sqsubseteq M_3$ and $M_2 \sqsubseteq M_3$. It is not guaranteed that least upper bound exists, as we show in the following example.

Example 3. Let $M_1 = \{y{:}\hat{m}, y{:}\check{m}\}$ and $M_2 = \{y{:}\tilde{m}\}$. Both $M_3 = \{y{:}\hat{m}, y{:}\tilde{m}\}$ and $M_3' = \{y{:}\check{m}, y{:}\tilde{m}\}$ are upper bounds for M_1 and M_2. However, there is no other upper bound M_3'' such that $M_3'' \sqsubseteq M_3$ and $M_3'' \sqsubseteq M_3'$. Thus, the least upper bound of M_1 and M_2 does not exist.

The above example shows that there are several possible ways of computing an upper bound M_3 of two given abstract states M_1 and M_2. Assuming that multisets are normalized in the sense that redundant elements are removed, the following steps define a possible algorithm:

1. any atom in M_1 (resp. M_2) with ∞ multiplicity is added to M_3 and removed from M_1 (resp. M_2);
2. the atoms of M_1 or M_2 that are covered by an element of M_3 with infinite multiplicity are removed;
3. the atoms $M_1 \cap M_2$ are added to M_3, and removed from M_1 and M_2;
4. let $a \in M_1$ be an atom covered by an atom $a' \in M_2$ ($a \preceq a'$). Both a and a' are removed from M_1 and M_2 and a' is added to M_3. Respectively, if $a' \preceq a$, a is the one added to M_3. Note there are several possible ways to compute the covering as we have seen in the example above; and
5. $M_1 \cup M_2$ is added to M_3.

In what follows, for the sake of simplicity, we assume that the program to be analyzed has been instrumented to have a **release** instruction before every **await** $y?$. This is required to simulate the auxiliary instruction **release** introduced in the semantics described in Sec. 2.1 (we could simulate it implicitly in the analysis also). The analysis of a program P is done as follows. For each method $m \in P_{\mathcal{M}}$, it starts from an abstract state $\emptyset \in \mathcal{B}$, which assumes that there are no tasks executing (since we are looking at the locally invoked tasks), and propagates the information to the different program points by applying the transfer function τ defined in Fig. 3 on the code *body*(m). The transfer function defines the effect of executing each (simple) instruction on a given abstract state $M \in \mathcal{B}$. Let us explain the different cases of τ: Case 1 adds an active instance of m to the abstract state; Case 2 adds a pending instance of m to the abstract state; Case 3 changes the status all active tasks that are guaranteed to be finished; Case 4 changes all pending tasks to active tasks; and Case 5 applies to the remaining instructions which do not have any effect on the MHP information.

Example 4. Consider program B. The abstract state at L13 is $\emptyset$ since we have not invoked any method yet. Executing L13 adds $y{:}\check{r}$ since the call is to a method in the same object; executing L14 adds $z{:}\tilde{p}$; executing L16 renames one $z{:}\tilde{p}$ to $\star{:}\tilde{p}$ since the future variable z is reused, and adds $z{:}\tilde{q}$; executing L17 renames $z{:}\tilde{q}$ to $z{:}\hat{q}$ since it is guaranteed that q has finished. The auxiliary **release** between L18 and L19 renames $y{:}\check{r}$ to $y{:}\tilde{r}$, since the current task might suspend and thus any pending task might become active. Finally, L19 renames $y{:}\tilde{r}$ to $y{:}\hat{r}$.

The analysis merges abstract states at branching points (i.e., after **if** and at loop entries) using the join operation $\sqcup$. The analysis of while loops requires iterating the corresponding code several times until a fixpoint is reached. To guarantee convergence in such cases we employ the following widening operator $\triangle : \mathcal{B} \times \mathcal{B} \mapsto \mathcal{B}$ after some predetermined number of iterations. Briefly, assuming that M_2 is the current abstract state at the loop entry program point, and that $M_1 \sqsubseteq M_2$ is the abstract state at the previous iteration, then $M_1 \triangle M_2$ replaces each element $(a, i) \in M_2$ by (a, ∞) if $(a, j) \in M_1$ and $i > j$, i.e., it replaces unstable elements by infinite number of occurrences in order to stabilize them.

Example 5. Let us demonstrate the analysis of the **if** and **while** statements on programs C and D. (**if**) At L37, the information that comes from the **then** and **else** branches is joined using $\sqcup$, namely $\{y{:}\tilde{p}\} \sqcup \{y{:}\tilde{q}\} = \{y{:}\tilde{p}, y{:}\tilde{q}\}$. Note that this state describes that either q or p are running at L37, but not both (as they share the same future variable); (**while**) In the first visit to L23, we have the abstract state $M_0 = \emptyset$, abstractly executing the body we reach L23 again with $M_1 = \{y{:}\hat{q}, z{:}\tilde{p}\}$ and joining it with M_0 results in M_1 itself. Similarly, if we apply two more iterations we respectively get $M_2 = \{\star{:}\hat{q}, \star{:}\tilde{p}, y{:}\hat{q}, z{:}\tilde{p}\}$ and $M_3 = \{(\star{:}\hat{q}, 2), (\star{:}\tilde{p}, 2), y{:}\hat{q}, z{:}\tilde{p}\}$. Inspecting M_2 and M_3, we see that $\star{:}\hat{q}$ and $\star{:}\tilde{p}$ are unstable, thus, we apply the widening operator $M_2 \triangle M_3$ obtaining $M_3' = \{(\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty), y{:}\hat{q}, z{:}\tilde{p}\}$. Executing the loop body starting with the new abstract state does not add any new MHP atoms since $\star{:}\hat{q}$ and $\star{:}\hat{p}$ already appear an infinite number of times.

In what follows, we assume that the result of the analysis is a mapping $\mathcal{L}_P{:}P_{\mathcal{P}} \mapsto \mathcal{B}$ from each program point p (including entry and exit points) to an abstract state $\mathcal{L}_P(p) \in \mathcal{B}$ that describes the status of the tasks that might be executing at p.

Example 6. The following table summarizes $\mathcal{L}_P$ for some selected program points of interest (from Fig. 2) that we will use in the next section:

4:$\{y{:}\tilde{p}\}$	16:$\{y{:}\check{r}, z{:}\tilde{p}, \star{:}\tilde{p}\}$	25:$\{y{:}\tilde{q}, (\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty)\}$	44 :$\{y{:}\tilde{r}\}$
6:$\{y{:}\tilde{p}, z{:}\tilde{q}\}$	17:$\{y{:}\check{r}, (\star{:}\tilde{p}, 2), z{:}\tilde{q}\}$	26:$\{y{:}\hat{q}, (\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty)\}$	47 :$\{y{:}\tilde{r}\}$
8:$\{y{:}\tilde{p}, z{:}\hat{q}\}$	18:$\{y{:}\check{r}, (\star{:}\tilde{p}, 2), z{:}\hat{q}\}$	30:$\{y{:}\hat{q}, (\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty)\}$	50 :$\{y{:}\hat{r}\}$
10:$\{y{:}\hat{p}, z{:}\hat{q}\}$	20:$\{y{:}\hat{r}, (\star{:}\tilde{p}, 2), z{:}\hat{q}\}$	38:$\{y{:}\tilde{p}, y{:}\tilde{q}\}$	
14:$\{y{:}\check{r}\}$	24:$\{y{:}\hat{q}, (\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty)\}$	40:$\{y{:}\hat{p}, y{:}\hat{q}\}$	

Recall that the state associated to a program point represents the state before the execution of the corresponding instruction. In addition, the results for the entry points L2, L12, L22, L32, L42 and L46 are all $\emptyset$. Also note that L10, L20, L30, L40, L44 and L50 are exit points for the corresponding methods. Those will allow us to capture tasks that escape from the methods. Observe that L24, L26 and L30 contain redundant information because $y{:}\hat{q}$ is redundant w.r.t. $(\star{:}\hat{q}, \infty)$.

4.2 The Notion of MHP Graph

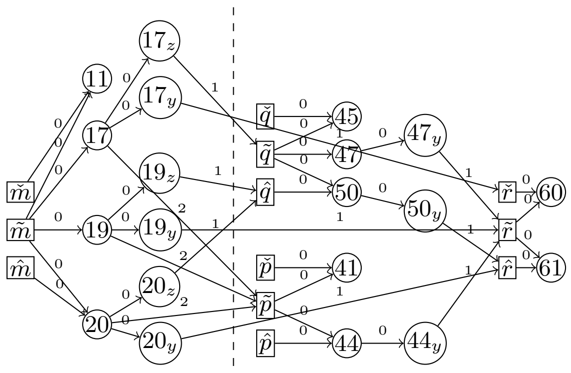
We now introduce the notion of *MHP graph* from which it is possible to extract precise information on which program points might globally run in parallel (according to Def. 1). A MHP graph has different types of nodes and different types of edges. There are nodes that represent the status of methods (active, pending or finished) and nodes which represent the program points. Outgoing edges from method nodes represent points of which at most one might be executing. In contrast, outgoing edges from program point nodes represent tasks such that any of them might be running. The information computed by the method-level MHP analysis is required to construct the MHP graph. When two nodes are directly connected by $i > 0$ edges, we connect them with a single edge of weight i. We start by formally constructing the MHP graph for a given program P, and then explain the construction in detail.

Definition 2 (MHP Graph). *Given a program P, and its method-level MHP analysis result $\mathcal{L}_P$, the MHP graph of P is a directed graph $\mathcal{G}_P = \langle V, E\rangle$ with a set of nodes V and a set of edges $E = E_1 \cup E_2 \cup E_3$ defined as follows:*

$$
\begin{aligned}
V &= \{\tilde{m}, \hat{m}, \check{m} \mid m \in P_{\mathcal{M}}\} \cup P_{\mathcal{P}} \cup \{p_y \mid p \in P_{\mathcal{P}}, y{:}m \in \mathcal{L}_P(p)\}\\
E_1 &= \{\tilde{m} \xrightarrow{0} p \mid m \in P_{\mathcal{M}}, p \in P_{\mathcal{P}}, p \in m\} \cup \{\hat{m} \xrightarrow{0} p_{\dot{m}}, \check{m} \xrightarrow{0} p_{\mathring{m}} \mid m \in P_{\mathcal{M}}\}\\
E_2 &= \{p \xrightarrow{i} x \mid p \in P_{\mathcal{P}}, (\star{:}x, i) \in \mathcal{L}_P(p)\}\\
E_3 &= \{p \xrightarrow{0} p_y, p_y \xrightarrow{1} x \mid p \in P_{\mathcal{P}}, (y{:}x, i) \in \mathcal{L}_P(p)\}
\end{aligned}
$$

Let us explain the different components of $\mathcal{G}_P$. The set of nodes V consists of several kinds of nodes:

1. *Method Nodes*: Each $m \in P_{\mathcal{M}}$ contributes three nodes $\tilde{m}$, $\hat{m}$, and $\check{m}$. These nodes will be used to describe the program points that can be reached from active, finished or pending tasks which are instances of m.
2. *Program Point Nodes*: Each $p \in P_{\mathcal{P}}$ contributes a node p that will be used to describe which other program points might be running in parallel with it.



	11	17	19	20	45	47	50	41	44	60	61
11											
17					●	●	●	●	●	●	●
19							●	●	●	●	●
20							●	●	●	●	●
45		●						○	○	○	○
47		●						○	○	●	●
50		●	●	●				○	○	○	●
41		●	●	●	○	○	○	○	○	○	○
44		●	●	●	○	○	○	○	○	●	●
60		●	●	●	○	●	○	○	●	○	○
61		●	●	●	○	●	●	○	●	○	○

Fig. 4. The $\mathcal{G}_P$ of example B (left) and its corresponding $\tilde{\mathcal{E}}_P$ (right)

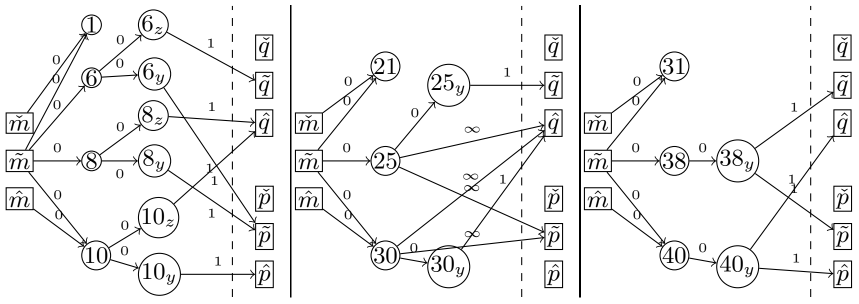


Fig. 5. Partial $\mathcal{G}_P$ for examples: A, C and D

3. *Future variable nodes*: These nodes are a refinement of program point nodes for improving precision in the presence of branching constructs. Each future variable y that appears in $\mathcal{L}_P(p)$ contributes a node p_y. These nodes will be used to state that if there are several MHP atoms in $\mathcal{L}_P(p)$ that are associated to y, then at most one of them can be running.

What gives the above meaning to the nodes are the edges $E = E_1 \cup E_2 \cup E_3$:

1. Edges in E_1 describe the program points at which each task can be depending on its status. Each m contributes the edges (a) $\tilde{m} \xrightarrow{0} p$ for each $p{\in}m$, which means that if m is active it can be in a state in which any of its program points is executing (but only one of them); (b) $\check{m} \xrightarrow{0} p_{\mathring{m}}$, which means that when m is pending, it is at the entry program point; and (c) $\hat{m} \xrightarrow{0} p_{\grave{m}}$, which means that when m is finished, it is at the exit program point;
2. Edges in E_2 describe which tasks might run in parallel with such program point. For every program point $p \in P_{\mathcal{P}}$, if $(\star{:}x, i) \in \mathcal{L}_P(p)$ then $p \xrightarrow{i} x$ is added to E_2. This edges means, if $x = \tilde{m}$ for example, that up to i instances of m might be running in parallel when reaching p;

3. Edges in E_3 enrich the information for each program point given in E_2. An edge $p_y \xrightarrow{1} x$ is added to E_3 if $(y{:}x, i) \in \mathcal{L}_P(p)$. For each future variable y that appears in $\mathcal{L}_P(p)$ an edge $p \xrightarrow{0} p_y$ is also added to E_3. This allows us to accurately handle cases in which several MHP atoms in $\mathcal{L}_P(p)$ are associated to the same future variable. Recall that in such cases at most one of the corresponding tasks can be available (see Ex. 2).

Note that MHP graphs might have cycles due to recursion.

Example 7. Using the method-level MHP information of Ex. 6 we obtain the MHP graphs for the four examples. Fig. 4 contains (to the left) the graph of example B and Fig. 5 contains incomplete graphs of examples A, C and D. The omitted parts of the graphs for A, C and D, marked with dashed lines, should be identical to the subgraph to the right of the dashed line in the graph of B. Besides, for readability, the graphs do not include all program points, but rather only those that correspond to entry, get and release points.

4.3 Inference of Global MHP

Given the MHP graph $\mathcal{G}_P$, two program points $p_1, p_2 \in P_{\mathcal{P}}$ may run in parallel (i.e., it might be that $(p_1, p_2) \in \mathcal{E}_P$) if one of the following conditions hold:

1. there is a non-empty path in $\mathcal{G}_P$ from p_1 to p_2 or vice-versa; or
2. there is a program point $p_3 \in P_{\mathcal{P}}$, and non-empty paths from p_3 to p_1 and from p_3 to p_2 that are either different in the first edge, or they share the first edge but it has weight $i > 1$.

The first case corresponds to *direct MHP* scenarios in which, when a task is running at p_1, there is another task running from which it is possible to *transitively* reach p_2, or vice-versa. This is the case, for example, of program points 17 and 50 in Fig. 4. The second case corresponds to *indirect MHP* scenarios in which a task is running at p_3 and there are two other tasks p_1 and p_2 executing in parallel and both are reachable from p_3. This is the case, for example, of program points 50 and 44 that are both reachable from program point 19 in Fig. 4 through paths that start with a different edge. Observe that the first edge can only be shared if it has weight $i > 1$ because it represents that there might be more than one instance of the same type of task running. This allows us to infer that 41 may run in parallel with itself because the edge from 17 to $\tilde{p}$ has weight 2 and, besides, that 41 can run in parallel with 44. Note that program points 45, 47, and 50 of method q do not satisfy any of the above conditions, which implies, as expected, that they cannot run in parallel.

The following definition formalizes the above intuition. We write $p_1 \rightsquigarrow p_2 \in \mathcal{G}_P$ to indicate that there is a path of length at least 1 from p_1 to p_2 in $\mathcal{G}_P$, and $p_1 \xrightarrow{i} x \rightsquigarrow p_2$ to indicate that such path starts with an edge to x with weight i.

Definition 3. *Given a program P, we let $\tilde{\mathcal{E}}_P = directMHP \cup indirectMHP$ where*

$$\begin{aligned} directMHP &= \{(p_1, p_2) \mid p_1, p_2 \in P_{\mathcal{P}}, p_1 \rightsquigarrow p_2 \in \mathcal{G}_P)\} \\ indirectMHP &= \{(p_1, p_2) \mid p_1, p_2, p_3 \in P_{\mathcal{P}}, p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_P, p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P, \\ &\qquad x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)\} \end{aligned}$$

Example 8. The table on the right side of Fig. 4 represents the $\tilde{\mathcal{E}}_P$ obtained from the graph on the left side. Empty cells mean that the corresponding points cannot run in parallel. Cells marked by $\bullet$ indicate that the pair is in *directMHP*. Cells marked with $\circ$ indicate that the pair is in *indirectMHP*. Note that the table captures the MHP relations informally discussed in Sec. 3.

4.4 Soundness and Complexity

The following theorem states the soundness of the analysis, namely, that $\tilde{\mathcal{E}}_P$ is an over-approximation of $\mathcal{E}_P$.

Theorem 1 (Soundness). $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.

As regards complexity, we distinguish its three phases:

1. The $\mathcal{L}_P$ computation can be performed independently for each method m. The transfer function τ only needs to be applied a constant number of times for each program point, even for loops, due to the use of widening. It is possible to represent multisets such that the cost of all multiset operations is linear w.r.t. their sizes which are at most $nm_m \cdot fut_m$, where nm_m is the number of different methods that can be called from m and fut_m is the number of future variables in m. Therefore, the cost of computing $\mathcal{L}_P$ for a method m is in $O(pp_m \cdot nm_m \cdot fut_m)$ where pp_m is the number of program points in the method.
2. The cost of creating the graph $\mathcal{G}_P$ is linear with respect to the number of edges. The number of edges originating from a method m is in $O(pp'_m \cdot nm_m \cdot fut_m)$ where pp'_m is the number of program points of interest. A strong feature of our analysis is that most of the program points can be ignored in this phase without affecting correctness or precision. Only points that correspond to **await** and **get** instructions and exit points are required for correctness. This happens due to the definition of τ in which the abstract states always grow (in the domain) except for those points.
3. Once the graph has been created, computing $\tilde{\mathcal{E}}_P$ is basically a graph reachability problem. Therefore, a straightforward algorithm for inferring of $\tilde{\mathcal{E}}_P$ is clearly in $O(n^3)$ where n is the number of nodes of the graph. However, a major advantage of this analysis is that for most applications there is no need to compute the complete $\tilde{\mathcal{E}}_P$; rather, this information can be obtained *on demand*.

5 Experimental Evaluation

We have implemented our analysis as a module of COSTABS [4], a cost analyzer of ABS programs. A standalone version of the MHP analysis can be tried out online at: `http://costa.ls.fi.upm.es/costabs/mhp`. Experimental evaluation has been carried out using two industrial case studies: ReplicationSystem and TradingSystem, which can be found at `http://www.hats-project.eu`, as

Table 1. Statistics about the analysis execution (times are in milliseconds)

Code	**Ns**	$\mathbf{NP}_{\mathcal{P}}$	$\mathbf{E_p}$	$\tilde{\mathcal{E}}_{\mathbf{P}}$	$\mathbf{PPs^2}$	$\mathbf{R}\varepsilon$	$\mathbf{T_G}$	$\mathbf{T}_{\tilde{\mathcal{E}}_{\mathbf{P}}}$
RepSystem	496	213	-	7724	45369	-	360	23020
TradingSystem	360	137	-	14829	18769	-	120	18120
MailServer	23	8	17	34	64	26.5%	10	< 10
BookShop	35	21	66	66	196	0%	< 10	10
PeerToPeer	75	36	385	487	1296	7.87%	20	100
BBuffer	22	7	36	36	49	0%	< 10	< 10
Chat	120	45	552	1219	2025	32.9%	< 10	190
DistHT	51	24	83	151	573	11.8%	< 10	20

well as a few typical concurrent applications: PeerToPeer, a peer to peer protocol implementation; Chat, a client-server implementation of a chat program; MailServer, a simple model of a Mail server; BookShop, a web shop client-server application; BBuffer, a classical bounded-buffer for communicating several producers and consumers; and DistHT, a distributed hash-table.

Table 1 summarizes our experiments. They have been performed on an Intel Core i5 at 2.4GHz with 3.7GB of RAM, running Linux. For each program, $\mathcal{G}_{\mathcal{P}}$ is built and the relation $\tilde{\mathcal{E}}_P$ is completely computed using only the program points required for soundness. **Ns** is the number of nodes of $\mathcal{G}_{\mathcal{P}}$ and $\mathbf{NP}_{\mathcal{P}}$ is the number of program point nodes. $\mathbf{E_p}$ is the number of MHP pairs obtained by running the program using a random scheduler, i.e., one which randomly chooses the next task to execute when the processor is released. These executions are bounded to a maximum number of interleavings as termination in some examples is not guaranteed. Observe that $\mathbf{E_p}$ does not capture all possible MHP pairs but just gives us an idea of the level of real parallelism. It gives us a lower bound of $\mathcal{E}_P$ which we will use to approximate the error. $\tilde{\mathcal{E}}_{\mathbf{P}}$ is the number of pairs inferred by the analysis. $\mathbf{PPs^2}$ is the square of the number of program points, i.e., the number of pairs considered in the analysis. $\mathbf{PPs^2} - \tilde{\mathcal{E}}_{\mathbf{P}}$ gives us the number of pairs that are guaranteed not to happen in parallel. $\mathbf{R}\varepsilon = 100(\tilde{\mathcal{E}}_{\mathbf{P}} - \mathbf{E_p})/\mathbf{PPs^2}$ is the approximated error percentage taking $\mathbf{E_p}$ as reference, i.e., $\mathbf{R}\varepsilon$ is an upper bound of the real error of the analysis. $\mathbf{T_G}$ is the time (in milliseconds) taken by the method-level analysis and in the graph construction. $\mathbf{T}_{\tilde{\mathcal{E}}_{\mathbf{P}}}$ is the time needed to infer all possible pairs of program points that may happen in parallel.

Although the MHP analysis has been successfully applied to both industrial case studies, it has not been possible to capture their runtime parallelism due to limitations in the simulator which could not treat all parts of these applications. Thus, there is no measure of error in these cases. We argue that the analyzer achieves high precision, with the approximated error less than 32.9% (bear in mind that $\mathbf{E_p}$ is a lower bound of the real parallelism) and up to 0% in other cases. As regards efficiency, both the method-level analysis and the graph construction are very efficient (just 0.36 sec. for the largest case study). The $\tilde{\mathcal{E}}_{\mathbf{P}}$ inference takes notably more time. But, as explained in Sec. 4.4, for most applications only a subset of pairs is of interest and, besides, those pairs can be computed on demand.

6 Conclusions, Related and Future Work

We have proposed a novel and efficient approach to infer MHP information for concurrent objects. The main novelty is that MHP information is obtained by means of a local analysis whose results can be modularly composed by using a MHP *analysis graph* in order to obtain global MHP relations. Concurrent objects operate similarly to Actors [2] and Erlang processes [5]. Therefore, the main ideas of our approach could be adapted to these languages.

When compared to the MHP analysis for X10 proposed in [10,1], we should first note that the async-finish model simplifies the inference of *escape* information, since the finish construct ensures that all methods called within its scope terminate before the execution continues to the next instruction. Moreover, it is important to note that our approach would achieve the same precision as their context-sensitive motivating example (Sec. 2.2 in [10]). This is because we do not merge calling contexts, but rather leave them explicit in the MHP graph. In addition, by splitting the analysis in two phases we achieve: (1) a higher degree of modularity and incrementality, since when a method is modified (or added, deleted, etc.), we only need to re-analyze that method locally, and replace its corresponding sub-graph in the global MHP graph accordingly; and (2) on demand MHP analysis, since we do not need to compute all MHP pairs in order to check if two given program points might run in parallel, but rather just check the relevant conditions for those two program points only.

An MHP analysis for Ada has been presented in [13], and extended later for Java in [14]. These works have been superseded later by [11,6]. In [6], Java programs are abstracted to an *abstract thread model* which is then analyzed in two phases. MHP graphs are used as well despite being substantially different from ours. A main difference is that our first phase infers local information for each method, while that of [6] infers a thread-level MHP from which it is possible to tell which threads might *globally* run in parallel. In addition, unlike our method-level analysis, it does not consider any synchronization between the threads in the first phase, but rather in the second phase. In future work, we plan to investigate if our analysis can be adapted to this thread abstract model.

An important application of MHP analysis is for understanding if two program points that belong to different tasks *in the same object* might run in parallel (i.e., interleave). We refer to this information as object-level MHP. This information is valuable because, in any static analysis that aims at approximating the objects' states, when a suspended task resumes, the (abstract) state of the corresponding object should be refined to consider modifications that might have been done by other tasks that interleave with it. Our approach can be directly applied to infer object-level MHP pairs by incorporating points-to information [15,12].

References

1. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May happen-in-parallel analysis of x10 programs. In: PPOPP 2007, pp. 183–193. ACM (2007)

2. Agha, G.A.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
3. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO Programs. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 238–254. Springer, Heidelberg (2011)
4. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: A Cost and Termination Analyzer for ABS. In: PEPM 2012, pp. 151–154. ACM Press (January 2012)
5. Armstrong, J., Virding, R., Wistrom, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall (1996)
6. Barik, R.: Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 152–169. Springer, Heidelberg (2006)
7. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
8. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
9. Lee, J.K., Palsberg, J., Majumdar, R.: Complexity results for may-happen-in-parallel analysis (2010) (manuscript)
10. Lee, J.K., Palsberg, J.: Featherweight X10: A Core Calculus for Async-Finish Parallelism. In: PPoPP 2010, pp. 25–36. ACM, New York (2010)
11. Li, L., Verbrugge, C.: A Practical MHP Information Analysis for Concurrent Java Programs. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 194–208. Springer, Heidelberg (2005)
12. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In: ISSTA, pp. 1–11 (2002)
13. Naumovich, G., Avrunin, G.S.: A conservative data flow algorithm for detecting all pairs of statements that happen in parallel. In: SIGSOFT FSE 1998, vol. 23(6), pp. 24–34 (1998)
14. Naumovich, G., Avrunin, G.S., Clarke, L.A.: An efficient algorithm for computing MHP information for concurrent java programs. In: ESEC / SIGSOFT FSE 1999, vol. 24(6), pp. 338–354 (1999)
15. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI 2004, pp. 131–144. ACM (2004)

Behavioural Equivalences over Migrating Processes with Timers

Bogdan Aman[1], Gabriel Ciobanu[1], and Maciej Koutny[2]

[1] Romanian Academy, Institute of Computer Science
and "A.I.Cuza" University, 700506 Iaşi, Romania
bogdan.aman@gmail.com, gabriel@info.uaic.ro

[2] School of Computing Science, Newcastle University
Newcastle upon Tyne, NE1 7RU, United Kingdom
maciej.koutny@newcastle.ac.uk

Abstract. The temporal evolution of mobile processes is governed by independently operating local clocks and their migration timeouts. We define a formalism modelling such distributed systems allowing (maximal) parallel execution at each location. Taking into account explicit timing constraints based on migration and interprocess communication, we introduce and study a number of timed behavioural equivalences, aiming to provide theoretical underpinnings of verification methods. We also investigate relationships between such behavioural equivalences.

Keywords: mobility, timer, process algebra, bisimulation, behaviour, equivalence.

1 Introduction

Process calculi are a family of formalisms used to model distributed systems. They provide algebraic laws allowing a high-level description and analysis of concurrent processes, behavioural equivalences (e.g., bisimulations) between processes, and automated tools for the verification of interaction, communication, and synchronization between processes. During the past couple of decades, a number of calculi supporting process mobility were defined and studied; in particular, π-calculus [16] and mobile ambients [5]. Various specific features were introduced to obtain such formalisms, including explicit locations in distributed π-calculus [14], explicit migration and timers in timed distributed π-calculus [12], and timed mobile ambients [1]. Time is an important aspect of distributed computing systems, and can play a key role in their formal description. Since time is a complex subject, its introduction to the domain of process calculi has received a lot of attention in, e.g., [2,3,6,15,17,21]. Papers like these assume the existence of a global clock which is usually required in the description of complex systems. However, there are several applications and systems for which considering a global clock would be inappropriate. This paper follows such an approach, in which local clocks operate independently, and so processes at different locations evolve asynchronously. Overall, temporal evolution of mobile processes is governed by independent local clocks and their migration timeouts.

The ever increasing complexity of mobile processes calls for the development of effective techniques and tools for the automated analysis and verification of their properties including, in particular, behavioural equivalences between systems. Bisimulation is the most common mathematical concept used to capture behavioural equivalence between processes. The corresponding equivalence relation, called bisimilarity, is used to abstract from certain details of the systems, and is widely accepted as a standard behavioural equivalence for different kinds of computational processes. Several kinds of bisimulations had been defined (e.g, strong or weak bisimulation for π-calculus [16]).

In the paper [9], we defined TiMo, a basic language for mobile systems in which it is possible to add timers to control process mobility and interaction. After that, in [11], a local clock was assigned to each location of a system modelled in TiMo. Each such clock determines the timing of actions executed at the corresponding location. Then, starting from TiMo, we created a flexible software platform supporting the specification of agents and their physical distribution, allowing also a timed migration in a distributed environment [8]. We obtained this implementation by using an advanced software technology, creating a platform for mobile agents with time constraints.

In this paper, we consider TiMo as the specification language for mobile agents with timeouts, and define various behavioural equivalences taking into account the timers which control the execution of communication and migration actions. We study these bisimulations over networks, and then relax them by applying the so-called 'up-to' technique. In [10] we discussed the *TravelShop* example, in which clients buy tickets to predefined destinations from travel agents. One can use the bisimilarities defined in this paper to differentiate, for example, between two travel agents using the same databases for prices, but having different delays for providing the answer (in an urgent situation, the faster one would be preferred). On the other hand, it is possible to define equivalence classes of agents offering similar services with respect to the waiting time (possibly up to an acceptable time difference).

The paper is organised as follows. We start in Section 2 with a brief presentation of TiMo, including its syntax and operational semantics, and introduce an example to illustrate the basic features of TiMo. In Sections 3 and 4, we formally define a number of timed bisimulations, and present some of their properties. Section 5 discusses the 'up-to' technique in the context of the bisimulations introduced in this paper. Conclusion and references end the paper.

2 TiMo

Timing constraints for migration allow one to specify what is the longest time it takes a mobile process to move to another location. A timer denoted by $\Delta 3$ associated to a migration action $\mathsf{go}^{\Delta 3} work$ indicates that the process moves to location *work* after at most 3 time units. It is also possible to constrain the waiting for a communication on a channel; if a communication action does not happen before a deadline, the process gives up and switches its operation to an alternative. E.g., a timer $\Delta 5$ associated to an output action $a^{\Delta 5}!\langle 10\rangle$ makes the channel available for communication only for the period of 5 time units.

2.1 Syntax

We assume suitable data sets including a set *Loc* of locations and a set *Chan* of communication channels. We use a set *Id* of process identifiers, and each $id \in Id$ has the arity m_{id}. In what follows, we use $\boldsymbol{x}$ to denote a finite tuple of elements $(x_1, \ldots, x_k)$ whenever it does not lead to a confusion.

The syntax of TiMo [11] is given in Table 1, where P are processes, L located processes, and N networks. Moreover, for each $id \in Id$ there is a unique definition of the form:

$$id(u_1, \ldots, u_{m_{id}} : X_1^{id}, \ldots, X_{m_{id}}^{id}) = P_{id}\ , \tag{1}$$

where P_{id} is a process expression, the u_i's are distinct variables playing the role of parameters, and the X_i^{id}'s are data types. In Table 1, it is assumed that:

– $a \in Chan$ is a channel;
– $lt \in \mathbb{N} \cup \{\infty\}$ is a deadline, where lt stands for *local time*;
– each v_i in $\boldsymbol{v}$ is an expression built from data values and variables;
– each u_i in $\boldsymbol{u}$ is a variable, and each X_i in $\boldsymbol{X}$ is a data type;
– l is a location or a location variable; and
– Ⓢ is a special symbol used to state that a process is temporarily 'stalled' and will be re-activated after a time progress.

The only variable binding constructor is $a^{\Delta lt}?(\boldsymbol{u}{:}\boldsymbol{X})$ then P else P' which binds the variables $\boldsymbol{u}$ within P (but *not* within P'). We use $fv(P)$ to denote the free variables of a process P (and similarly for networks). For a process definition as in (1), we assume that $fv(P_{id}) \subseteq \{u_1, \ldots, u_{m_{id}}\}$ and so the free variables of P_{id} are parameter bound. Processes are defined up to alpha-conversion, and $\{v/u, \ldots\}P$ is obtained from P by replacing all free occurrences of a variable u by v, etc, possible after alpha-converting P in order to avoid clashes. Moreover, if $\boldsymbol{v}$ and $\boldsymbol{u}$ are tuples of the same length then $\{\boldsymbol{v}/\boldsymbol{u}\}P$ denotes $\{v_1/u_1, v_2/u_2, \ldots, v_k/u_k\}P$.

Intuitively, a process $a^{\Delta lt}!\langle \boldsymbol{v}\rangle$ then P else P' attempts to send a tuple of values $\boldsymbol{v}$ over channel a for lt time units. If successful, it continues as process P; otherwise it continues as process P'. Similarly, $a^{\Delta lt}?(\boldsymbol{u}{:}\boldsymbol{X})$ then P else P' is a process that attempts for lt time units to input a tuple of values of type $\boldsymbol{X}$ and substitute them for the variables $\boldsymbol{u}$. Mobility is implemented by a process go$^{\Delta lt}l$ then P which moves from the current location to the location l within lt time units. Note that since l can be a variable, and so its value is assigned dynamically through the communication with other processes, migration actions support a flexible scheme for the movement of processes from one location to another. Processes are further constructed from the (terminated) process 0 and

Table 1. TiMo syntax

Processes	$P ::= a^{\Delta lt}!\langle \boldsymbol{v}\rangle$ then P else P' $\mid$	(output)
	$a^{\Delta lt}?(\boldsymbol{u}{:}\boldsymbol{X})$ then P else P' $\mid$	(input)
	$\mathsf{go}^{\Delta lt}\ l$ then P $\mid$	(move)
	$P \mid P'$ $\mid$	(parallel)
	0 $\mid$	(termination)
	$id(\boldsymbol{v})$	(recursion)
	$Ⓢ P$	(stalling)
Located processes	$L ::= l[\![P]\!]$	
Networks	$N ::= L \mid L \mid N$	

parallel composition $P|P'$. A located process $l[\![P]\!]$ specifies a process P running at location l, and networks are composed out of located processes. A network N is *well-formed* if the following hold:

– there are no free variables in N;
– there are no occurrences of the special symbol Ⓢ in N;
– assuming that id is as in the recursive equation (1), for every $id(\boldsymbol{v})$ occurring in N or on the right hand side of any recursive equation, the expression v_i is of type corresponding to X_i^{id} (where we use the standard rules of determining the type of an expression).

The set of processes is denoted by $\mathcal{P}$, the set of located processes by $\mathcal{L}$, and the set of networks by $\mathcal{N}$.

By delaying the migration to another location, we can model in a simple way the movement time of processes within the network which is, in general, outside the control of a system designer.

2.2 Semantics

The first component of the operational semantics of TiMo is the structural equivalence $\equiv$ on networks; it is the smallest congruence such that the first three equalities in Table 2 hold. Its role is to rearrange a network in order to apply the action rules which are also given in Table 2. Using the first three equalities in Table 2, one can always transform a given network N into a finite parallel composition of located processes of the form

$$l_1[\![P_1]\!] \mid \ldots \mid l_n[\![P_n]\!]$$

Table 2. TiMo operational semantics

(NComm)	$N \mid N' \equiv N' \mid N$
(NAssoc)	$(N \mid N') \mid N'' \equiv N \mid (N' \mid N'')$
(NSplit)	$l[\![P \mid P']\!] \equiv l[\![P]\!] \mid l[\![P']\!]$
(Move)	$l[\![\mathsf{go}^{\Delta lt} l' \text{ } \mathsf{then} \text{ } P]\!] \xrightarrow{l' @ l} l'[\![Ⓢ P]\!]$
(Com)	$\dfrac{v_1 \in X_1 \ \ldots \ v_k \in X_k}{\begin{array}{c} l[\![a^{\Delta lt}!\langle \boldsymbol{v} \rangle \text{ } \mathsf{then} \text{ } P \text{ } \mathsf{else} \text{ } Q \mid a^{\Delta lt'}?(\boldsymbol{u}{:}\boldsymbol{X}) \text{ } \mathsf{then} \text{ } P' \text{ } \mathsf{else} \text{ } Q']\!] \\ \xrightarrow{a\langle \boldsymbol{v} \rangle @ l} l[\![Ⓢ P \mid Ⓢ\{\boldsymbol{v}/\boldsymbol{u}\}P']\!] \end{array}}$
(Call)	$l[\![id(\boldsymbol{v})]\!] \xrightarrow{id @ l} l[\![Ⓢ\{\boldsymbol{v}/\boldsymbol{u}\}P_{id}]\!]$
(Par)	$\dfrac{N \xrightarrow{\lambda} N'}{N \mid N'' \xrightarrow{\lambda} N' \mid N''}$
(Equiv)	$\dfrac{N \equiv N' \qquad N' \xrightarrow{\lambda} N'' \qquad N'' \equiv N'''}{N \xrightarrow{\lambda} N'''}$
(Time)	$\dfrac{N \not\rightarrow_l}{N \xrightarrow{\surd_l} \phi_l(N)}$

such that no process P_i has the parallel composition operator at its topmost level. Each located process $l_i[\![P_i]\!]$ is called a component of N, and the parallel composition is called a *component decomposition* of the network N. Note that these notions are well defined since component decomposition is unique up to the permutation of the components. This follows from the rule (Call) which treats recursive definitions as function calls which take a unit of time. Another consequence of such a treatment is that it is impossible to execute an infinite sequence of action steps without executing any time actions.

Table 2 introduces two kinds of rules,

$$N \xrightarrow{\lambda} N' \text{ and } N \xrightarrow{\surd_l} N' .$$

The former is an execution of an action λ, and the latter a time step at location l. In the rule (Time), $N \not\rightarrow_l$ means that the rules (Call) and (Com) as well as (Move) with $\Delta lt = \Delta 0$ cannot be applied to N for location l. It can be noticed that in rule (Time) we use negative premises, i.e., an activity is performed in the absence of other actions. This is due to the fact that sequencing the evolution in time units can only be defined using negative premises, as done for sequencing processes in [4,13]. Moreover, $\phi_l(N)$ is obtained by taking the component decomposition of N and simultaneously replacing all components:

$$l[\![a^{\Delta lt}\omega \text{ then } P \text{ else } Q]\!] \quad \text{by} \quad \begin{cases} l[\![Q]\!] & \text{if } lt = 0 \\ l[\![a^{\Delta lt-1}\omega \text{ then } P \text{ else } Q]\!] & \text{otherwise} \end{cases}$$

$$l[\![\mathsf{go}^{\Delta lt} l' \text{ then } P]\!] \quad \text{by} \quad l[\![\mathsf{go}^{\Delta lt-1} l' \text{ then } P]\!]$$

where ω stands for $!\langle \boldsymbol{v}\rangle$ or $?(\boldsymbol{u}{:}\boldsymbol{X})$. After that, all the occurrences of the symbol Ⓢ in N are erased since processes that were unfolded or interacted with other processes or migrated need to be activated (note that the number of the symbols Ⓢ to be erased cannot exceed the number of the components of the network).

The rules of Table 2 express executions of individual actions. A complete computational step is captured by a derivation of the form

$$N \xRightarrow{\Lambda @ l} N' ,$$

where $\Lambda = \{\lambda_1, \ldots, \lambda_m\}$ $(m \geq 0)$ is a finite multiset of actions for some location l (i.e., actions λ_i of the form $l'@l$ or $a\langle v\rangle@l$ or $id@l$) such that

$$N \xrightarrow{\lambda_1} N_1 \ldots N_{m-1} \xrightarrow{\lambda_m} N_m \xrightarrow{\surd_l} N' .$$

That means that a derivation is a condensed representation of a sequence of individual actions followed by a clock tick, all happening at the same location. Intuitively, we capture the cumulative effect of the concurrent execution of the multiset of actions Λ at location l. If there is only a time progression at a location l, we write $N \xRightarrow{\emptyset @ l} N'$.

In terms of executing TiMo specifications on an abstract machine, one can imagine the latter as a device transforming well-formed networks into well-formed networks. At any stage, the machine selects one location l as the active one. Then, it executes all interprocess communications within location l as well as all migrations with expired (zero) timers in a maximally concurrent way. This is followed by the execution of arbitrarily many migrations with unexpired timers at location l. Finally, one decrements all the top-most timers in all the network components at location l which have not yet been involved in the current computational step.

2.3 An Example

The *TravelShop* example discussed in [10] is rather involved, so in this paper we use its simplified version to illustrate the operational semantics of TiMo. In the *UrgentTravel* example a client process attempts to initiate an unspecified *travel* process as soon as it receives a flight offer.

The scenario involves three locations and three processes. The role of each location is as follows: *office* is a location where the *client* process starts its work, and $agency_i$ (for $i = 1, 2$) is a travel agency where the client can find out about the price of tickets. The role of each process is as follows:

– *client* resides in the *office* location, and is determined to pay for a flight as soon as it receives an offer from one of two travel agencies. After sending an email to each agency, it awaits for the quickest response to initiate the *travel* process.
– $agent_i$ (for $i = 1, 2$) resides in the $agency_i$ location, and replies to emails received from clients.

We use timers in order to impose deadlines on the execution of communications and migrations. Each location has its local clock which determines the timing of actions executed at that location. The process specifications that capture the essential features of the above scenario are:

$$\begin{aligned}
agent_i &= a^{\Delta 5}!\langle offer_i\rangle \ \mathsf{then}\ agent_i\ \mathsf{else}\ agent_i\\
client\ &= d^{\Delta 6}?(y)\ \mathsf{then}\ travel(y)\ \mathsf{else}\ 0\\
&\quad \mid\ \mathsf{go}^{\Delta 2} agency_1\ \mathsf{then}\ (a^{\Delta 1}?(x)\ \mathsf{then}\ (\mathsf{go}^{\Delta 2} office\ \mathsf{then}\ d^{\Delta 1}!\langle x\rangle)\ \mathsf{else}\ 0)\\
&\quad \mid\ \mathsf{go}^{\Delta 3} agency_2\ \mathsf{then}\ (a^{\Delta 1}?(x)\ \mathsf{then}\ (\mathsf{go}^{\Delta 3} office\ \mathsf{then}\ d^{\Delta 1}!\langle x\rangle)\ \mathsf{else}\ 0)
\end{aligned}$$

Note that in the above definitions we slightly simplified the notation and used:

- $d^{\Delta t}!\langle x\rangle$ instead of $d^{\Delta t}!\langle x\rangle$ $\mathsf{then}\ 0\ \mathsf{else}\ 0$
- $d^{\Delta 6}?(y)$ instead of $d^{\Delta 6}?(y{:}1..1000)$
- $a^{\Delta 1}?(x)$ instead of $a^{\Delta 1}?(x{:}1..1000)$.

Table 3 shows a typical execution of the following network modelling our scenario:

$$UrgentTravel = office[\![client]\!] \mid agency_1[\![agent_1]\!] \mid agency_2[\![agent_2]\!]$$

3 Timed Bisimulations in TiMo

In what follows, we define various behavioural equivalences for networks of located processes. Similarly as in timed distributed π-calculus [7], we start by extending the standard notion of strong bisimilarity to take into account timed transitions.

Definition 1 (strong timed bisimulation)
Let $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{N}$ be a binary relation on networks of processes.

1. $\mathcal{R}$ *is a* strong timed simulation *(ST simulation) if*

$$(N_1, N_2) \in \mathcal{R} \wedge N_1 \xrightarrow{\psi} N_1' \ \ implies\ \ \exists N_2' \in \mathcal{N} :\ N_2 \xrightarrow{\psi} N_2' \wedge (N_1', N_2') \in \mathcal{R}\ .$$

where ψ is any action allowed by the operational semantics.
2. $\mathcal{R}$ *is a* strong timed bisimulation *(ST bisimulation) if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong timed simulations.*
3. *The* strong timed bisimilarity $\sim$ *is the union of all ST bisimulations.*

Table 3. Applying operational semantics

$UrgentTravel$

$$\xRightarrow{\emptyset @ office} \xRightarrow{\emptyset @ office} \xRightarrow{\emptyset @ office} \xRightarrow{\emptyset @ agency_1} \xRightarrow{\emptyset @ agency_1} \xRightarrow{\emptyset @ agency_2}$$

$$\begin{aligned} & office[\![d^{\Delta 4}?(y) \text{ then } travel(y) \text{ else } 0 \\ & \qquad \mid \mathsf{go}^{\Delta 0} agency_1 \text{ then } (a^{\Delta 1}?(x) \text{ then } (\mathsf{go}^{\Delta 2} office \text{ then } d^{\Delta 1}!\langle x\rangle) \text{ else } 0) \\ & \qquad \mid \mathsf{go}^{\Delta 1} agency_2 \text{ then } (a^{\Delta 1}?(x) \text{ then } (\mathsf{go}^{\Delta 3} office \text{ then } d^{\Delta 1}!\langle x\rangle) \text{ else } 0)]\!] \\ \mid\ & agency_1[\![a^{\Delta 4}!\langle offer_1\rangle \text{ then } agent_1 \text{ else } agent_1]\!] \\ \mid\ & agency_2[\![a^{\Delta 5}!\langle offer_2\rangle \text{ then } agent_2 \text{ else } agent_2]\!] \end{aligned}$$

$$\xRightarrow{\{agency_1 @ office,\ agency_2 @ office\} @ office}$$

$$\xRightarrow{\{a\langle offer_1\rangle @ agency_1\} @ agency_1} \xRightarrow{\{a\langle offer_2\rangle @ agency_2\} @ agency_2}$$

$$\begin{aligned} & office[\![d^{\Delta 3}?(y) \text{ then } travel(y) \text{ else } 0]\!] \\ \mid\ & agency_1[\![agent_1 \mid \mathsf{go}^{\Delta 2} office \text{ then } d^{\Delta 1}!\langle offer_1\rangle]\!] \\ \mid\ & agency_2[\![agent_2 \mid \mathsf{go}^{\Delta 3} office \text{ then } d^{\Delta 1}!\langle offer_2\rangle]\!] \end{aligned}$$

$$\xRightarrow{\{office @ agency_1\} @ agency_1}$$

$$\begin{aligned} & office[\![d^{\Delta 3}?(y) \text{ then } travel(y) \text{ else } 0 \mid d^{\Delta 1}!\langle offer_1\rangle]\!] \\ \mid\ & agency_1[\![a^{\Delta 5}!\langle offer_1\rangle \text{ then } agent_1 \text{ else } agent_1]\!] \\ \mid\ & agency_2[\![agent_2 \mid \mathsf{go}^{\Delta 3} office \text{ then } d^{\Delta 1}!\langle offer_2\rangle]\!] \end{aligned}$$

$$\xRightarrow{\{d\langle offer_1\rangle @ office\} @ office}$$

$$\begin{aligned} & office[\![travel(offer_1) \mid 0]\!] \\ \mid\ & agency_1[\![a^{\Delta 5}!\langle offer_1\rangle \text{ then } agent_1 \text{ else } agent_1]\!] \\ \mid\ & agency_2[\![agent_2 \mid \mathsf{go}^{\Delta 3} office \text{ then } d^{\Delta 1}!\langle offer_2\rangle]\!] \end{aligned}$$

Essentially, the above definition treats timed transitions just as any other transitions, and therefore coincides with the original notion of bisimilarity for labelled transition systems. It is easy to check that $\sim$ is an equivalence relation, and also the largest strong timed bisimulation. From the point of view of the behaviour of TiMo networks, a crucial result is that strong timed bisimularity can be used to compare their evolutions in terms of complete computational steps of well-formed networks.

Theorem 1. *Let N_1 and N_2 be two well-formed networks. Then:*

$$N_1 \sim N_2 \ \wedge\ N_1 \xRightarrow{\Lambda @ l} N_1' \quad \textit{implies} \quad \exists N_2' \in \mathcal{N} :\ N_2 \xRightarrow{\Lambda @ l} N_2' \ \wedge\ N_1' \sim N_2' \ .$$

Together with the fact that, for every well-formed network N, $N \xRightarrow{\Lambda @ l} N'$ implies that N' is also well-formed (see [11]), this means that the strong timed bisimilarity is an adequate tool for comparing the behaviour of (well-formed) networks.

The above definition of equivalence compares the evolution of whole networks, but does not provide means for reasoning about equivalence of compositionally defined networks. Consider, for example, two networks:

$$N_1 = l[\![a^{\Delta lt}!\langle 1\rangle \ \mathsf{then}\ 0\ \mathsf{else}\ 0]\!] \ \text{ and } \ N_2 = l[\![0]\!] \ .$$

Clearly, $N_1 \sim N_2$ as both networks allow only the transition $\xRightarrow{\emptyset @ l}$. However, when we compose them with $N = l[\![a^{\Delta lt}?(u:\mathbb{N})\ \mathsf{then}\ 0\ \mathsf{else}\ 0]\!]$ then:

$$N_1 \mid N \not\sim N_2 \mid N$$

as the first composition can execute transition $\xRightarrow{\{a\langle 1\rangle @ l\} @ l}$ whereas the second one can only execute $\xRightarrow{\emptyset @ l}$.

To be able to reason about networks in a compositional way, one may augment (only for the purpose of dealing with equivalences) the operational semantics of processes with two additional rules relating to communication, which intuitively represent individual evolutions of interacting processes:

$$(\textsc{Snd}) \qquad l[\![a^{\Delta lt}!\langle \boldsymbol{v}\rangle \ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{a!\langle \boldsymbol{v}\rangle @ l} l[\![ⓢP]\!]$$

$$(\textsc{Rcv}) \qquad \frac{v_1 \in X_1 \ \ldots\ v_k \in X_k}{a^{\Delta lt}?(\boldsymbol{u}{:}\boldsymbol{X})\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{a?\langle \boldsymbol{v}\rangle @ l} l[\![\{\boldsymbol{v}/\boldsymbol{u}\}P]\!]}$$

All the previous rules remain unchanged. In particular, $N \not\to_l$ in the rule (Time) still means that the rules (Call) and (Com) as well as (Move) with $\Delta lt = \Delta 0$ cannot be applied to N for location l; in other words the two new rules, (Snd) and (Rcv), are not taken into account. The transitions of the extended operational semantics will be denoted by $\xrightarrow{\psi}_e$ rather than $\xrightarrow{\psi}$.

Definition 2 (strong extended timed bisimulation)

Let $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{N}$ be a binary relation on networks of processes.

1. *$\mathcal{R}$ is a* strong extended timed simulation *(SET simulation) if*

$$(N_1, N_2) \in \mathcal{R} \wedge N_1 \xrightarrow{\psi}_e N_1' \ \textit{implies} \ \exists N_2' \in \mathcal{N} : N_2 \xrightarrow{\psi}_e N_2' \wedge (N_1', N_2') \in \mathcal{R} \ .$$

where ψ is any action allowed by the extended operational semantics.

2. $\mathcal{R}$ *is a* strong extended timed bisimulation *(SET bisimulation) if both* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are strong extended timed simulations.*
3. *The* strong extended timed bisimilarity $\sim^e$ *is the union of all SET bisimulations.*

The strong extended timed bisimilarity is compositional, and it implies strong timed bisimilarity.

Theorem 2. *Let N_1, N_1', N_2 and N_2' be well formed networks. Then:*

$$N_1 \sim^e N_2 \textit{ and } N_1' \sim^e N_2' \quad \textit{implies} \quad N_1 \mid N_1' \sim^e N_2 \mid N_2' \; .$$

Proposition 1. *Let N and N' be well formed networks. Then:*

$$N \sim^e N' \quad \textit{implies} \quad N \sim N' \; .$$

It therefore follows, in the context of Theorem 1, that strong extended timed bisimilarity provides an adequate tool for comparing behaviours of compositionally defined networks considered up to certain time deadline.

4 Bounded Timed Bisimulations in TiMo

The above notion of equivalence takes into account the timed behaviour requiring an exact match of transitions of two networks, for their entire evolution. Sometimes these requirements are too strong. According to [18] where a similar approach is presented, real-time distributed systems usually require a certain behaviour within a given threshold of time units. That is why we will now restrict equivalences up-to some threshold time values specified individually for each location $l \in Loc$, defining *bounded* timed equivalences.

In what follows we assume that $Loc = \{l_1, \ldots, l_n\}$. We then introduce some additional notations and terminology:

– $\mathbb{T} = \{(t_1@l_1, \ldots, t_n@l_n) \mid t_1, \ldots t_n \in \mathbb{N}\}$ comprises tuples in which each location l_i has an associated number of time units in which it will be observed. We use $\widehat{t}$ to denote $(t_1@l_1, \ldots, t_n@l_n)$, and $\widehat{t}_{l_i}$ to denote t_i.
– For every $\widehat{t} = (t_1@l_1, \ldots, t_n@l_n) \in \mathbb{T}$ and $l_i \in Loc$,

$$\widehat{t} \ominus l_i = (t_1@l_1, \ldots, t_{i-1}@l_{i-1}, t_i - 1@l_i, t_{i+1}@l_{i+1}, \ldots, t_n@l_n) \; .$$

Intuitively, $\widehat{t} \ominus l_i$ records that one time unit has passed at location l_i, and the remaining observation time has been updated accordingly.
– Any relation $\mathcal{R} \subseteq \mathcal{N} \times \mathbb{T} \times \mathcal{N}$ is a *timed relation* over networks.
– The *inverse of a timed relation* $\mathcal{R}$ is

$$\mathcal{R}^{-1} = \{(N', \widehat{t}, N) \mid (N, \widehat{t}, N') \in \mathcal{R}\} \; .$$

– If $\mathcal{R}$ is a timed relation and $\widehat{t} \in \mathbb{T}$ then the $\widehat{t}$*-projection* of $\mathcal{R}$ is:

$$\mathcal{R}_{\widehat{t}} = \{(N_1, N_2) \mid (N_1, \widehat{t}, N_2) \in \mathcal{R}\} \; .$$

Definition 3 (strong bounded timed bisimulation)

Let $\mathcal{R} \subseteq \mathcal{N} \times \mathbb{T} \times \mathcal{N}$ be a timed relation over $\mathcal{N}$.

1. *$\mathcal{R}$ is a* strong bounded timed simulation *(SBT simulation) if*

$$\left\{ \begin{array}{c} (N_1, \widehat{t}, N_2) \in \mathcal{R} \\ N_1 \xrightarrow{\surd_l} N_1' \text{ and } \widehat{t}_l > 0 \end{array} \right\} \text{ implies } \exists N_2' \in \mathcal{N} : \left\{ \begin{array}{c} N_2 \xrightarrow{\surd_l} N_2' \\ (N_1', \widehat{t} \ominus l, N_2') \in \mathcal{R} \end{array} \right\}$$

and, for each λ of the form $l'@l$ or $a\langle \boldsymbol{v} \rangle @l$ or $id@l$,

$$\left\{ \begin{array}{c} (N_1, \widehat{t}, N_2) \in \mathcal{R} \\ N_1 \xrightarrow{\lambda} N_1' \text{ and } \widehat{t}_l > 0 \end{array} \right\} \text{ implies } \exists N_2' \in \mathcal{N} : \left\{ \begin{array}{c} N_2 \xrightarrow{\lambda} N_2' \\ (N_1', \widehat{t}, N_2') \in \mathcal{R} \end{array} \right\}$$

2. *$\mathcal{R}$ is a* strong bounded timed bisimulation *(SBT bisimulation) if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong bounded timed simulations.*
3. *The* strong bounded timed bisimilarity *$\simeq$ is the union of all SBT bisimulations.*

One can see that $\simeq$ is the largest SBT bisimulation. Moreover, SBT bisimulations enjoy properties similar to those satisfied by equivalence relations.

Proposition 2. *The inverse, composition and union of SBT bisimulations are SBT bisimulations, where the composition of timed relations $\mathcal{R}$ and $\mathcal{R}'$ comprises all triples $(N, \widehat{t}, N'')$ for which there is $N' \in \mathcal{N}$ satisfying $(N, \widehat{t}, N') \in \mathcal{R}$ and $(N', t, N'') \in \mathcal{R}'$.*

Strong bounded timed bisimilarity is such that being equivalent up-to a certain time bound implies equivalence up-to any smaller time bound.

Proposition 3. *Let $N \simeq_{\widehat{t}} N'$ be two well-formed networks. Then $N \simeq_{\widehat{t'}} N'$, for every $\widehat{t'} \in \mathbb{T}$ satisfying $t_1 \leq t_1', \ldots, t_n \leq t_n'$.*

Finally, we have a crucial result that strong bounded timed bisimularity can be used to compare the complete computational steps of two networks.

Theorem 3. *Let N_1 and N_2 be two well-formed networks. Then:*

$$\left\{ \begin{array}{c} N_1 \simeq_{\widehat{t}} N_2 \\ N_1 \xRightarrow{\Lambda @ l} N_1' \text{ and } \widehat{t}_l > 0 \end{array} \right\} \text{ implies } \exists N_2' \in \mathcal{N} : \left\{ \begin{array}{c} N_2 \xRightarrow{\Lambda @ l} N_2' \\ N_1' \simeq_{\widehat{t} \ominus l} N_2' \end{array} \right\}$$

Similarly as strong time bisimilarity is not preserved by network composition, strong bounded time bisimilarity is not preserved by network composition. However, as in the previous case, one can consider two additional rules, (Snd) and (Rcv), and obtain the extended version of $\simeq_{\widehat{t}}$.

Definition 4 (strong extended bounded timed bisimulation).

Let $\mathcal{R} \subseteq \mathcal{N} \times \mathbb{T} \times \mathcal{N}$ be a timed relation over $\mathcal{N}$.

1. *$\mathcal{R}$ is a* strong extended bounded timed simulation *(SEBT simulation) if*

$$\left\{\begin{matrix} (N_1, \widehat{t}, N_2) \in \mathcal{R} \\ N_1 \xrightarrow{\surd_l}_e N_1' \text{ and } \widehat{t}_l > 0 \end{matrix}\right\} \text{ implies } \exists N_2' \in \mathcal{N} : \left\{\begin{matrix} N_2 \xrightarrow{\surd_l}_e N_2' \\ (N_1', \widehat{t} \ominus l, N_2') \in \mathcal{R} \end{matrix}\right\}$$

and, for each ψ of the form $l'@l$ or $a\langle \boldsymbol{v} \rangle @l$ or $a!\langle \boldsymbol{v} \rangle @l$ or $a?\langle \boldsymbol{v} \rangle @l$ or $id@l$,

$$\left\{\begin{matrix} (N_1, \widehat{t}, N_2) \in \mathcal{R} \\ N_1 \xrightarrow{\psi}_e N_1' \text{ and } \widehat{t}_l > 0 \end{matrix}\right\} \text{ implies } \exists N_2' \in \mathcal{N} : \left\{\begin{matrix} N_2 \xrightarrow{\psi}_e N_2' \\ (N_1', \widehat{t}, N_2') \in \mathcal{R} \end{matrix}\right\}$$

2. *$\mathcal{R}$ is a* strong extended bounded timed bisimulation *(SEBT bisimulation) if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong extended bounded timed simulations.*
3. *The* strong extended bounded timed bisimilarity *is the union $\simeq^e$ of all SEBT bisimulations.*

The strong extended bounded timed bisimilarity is compositional, and it implies strong bounded timed bisimilarity.

Theorem 4. *Let N_1, N_1', N_2 and N_2' be well formed networks and $\widehat{t} \in \mathbb{T}$. Then:*

$$N_1 \simeq^e_{\widehat{t}} N_2 \text{ and } N_1' \simeq^e_{\widehat{t}} N_2' \text{ implies } N_1 \mid N_1' \simeq^e_{\widehat{t}} N_2 \mid N_2' \,.$$

Proposition 4. *Let N_1 and N_2 be well formed networks and $\widehat{t} \in \mathbb{T}$. Then:*

$$N_1 \simeq^e_{\widehat{t}} N_2 \text{ implies } N_1 \simeq_{\widehat{t}} N_2 \,.$$

It therefore follows, in the context of Theorem 3, that strong extended timed bisimilarity provides an adequate tool for comparing behaviours of compositionally defined networks.

5 Relaxing Timed Bisimulations

In what follows we use the 'up-to' technique presented in [19] in the context of bounded timed bisimulations. The standard proof technique to establish that N_1 and N_2 are bisimilar is to find a bisimulation $\mathcal{R}$ s.t.such that $(N_1, N_2) \in \mathcal{R}$ and $\mathcal{R}$ is closed under transitions of the operational semantics; in particular, that the derivatives (N_1', N_2') of (N_1, N_2) are also in $\mathcal{R}$. Sometimes it is difficult to find directly such a relation $\mathcal{R}$. Instead, there is an useful alternative technique, the so-called bisimulation 'up-to' some relation $\mathcal{R}'$: for a relation $\mathcal{R}$, which is not a bisimulation, if $(N_1, N_2) \in \mathcal{R}$, then one requires that the derivatives (N_1', N_2') are in $\mathcal{R}'$. Under certain conditions one can then establish that N_1 and N_2 are bisimilar. For such a technique, a general framework working for untimed operational semantics was presented in [20]. We cannot make a direct use of that framework, but we can adapt it in a straightforward manner to our setting.

We begin by introducing a notion of 'progressing' a timed relation towards another timed relation.

Definition 5 (strong progress)

Let $\mathcal{R}$ and $\mathcal{R}'$ be two timed relations. Then $\mathcal{R}$ strongly progresses *to $\mathcal{R}'$ if*

$$\left\{ \begin{array}{c} (N_1, \widehat{t}, N_2) \in \mathcal{R} \\ N_1 \xrightarrow{\surd_l} N_1' \ \wedge \ \widehat{t}_l > 0 \end{array} \right\} \implies \exists N_2' \in \mathcal{N} : \left\{ \begin{array}{c} N_2 \xrightarrow{\surd_l} N_2' \\ (N_1', \widehat{t} \ominus l, N_2') \in \mathcal{R}' \end{array} \right\}$$

and, for each λ of the form $l'@l$ or $a\langle \boldsymbol{v} \rangle @l$ or $id@l$,

$$\left\{ \begin{array}{c} (N_1, \widehat{t}, N_2) \in \mathcal{R} \\ N_1 \xrightarrow{\lambda} N_1' \ \wedge \ \widehat{t}_l > 0 \end{array} \right\} \implies \exists N_2' \in \mathcal{N} : \left\{ \begin{array}{c} N_2 \xrightarrow{\lambda} N_2' \\ (N_1', \widehat{t}, N_2') \in \mathcal{R}' \end{array} \right\}$$

We denote this by $\mathcal{R} \rightsquigarrow \mathcal{R}'$.

The above definition is similar to that of SBT bisimulation, except that the derivatives $(N_1', \widehat{t}, N_2')$ and $(N_1', \widehat{t} \ominus l, N_2')$ must be in $\mathcal{R}'$ rather than $\mathcal{R}$.

Proposition 5. *If $\mathcal{R} \rightsquigarrow \mathcal{R}'$ and $\mathcal{R}'$ is an SBT bisimulation, then $\mathcal{R}$ is also an SBT bisimulation.*

Therefore, to establish that $N_1 \simeq_{\widehat{t}} N_2$ it is enough to find a relation $\mathcal{R}$ with $(N_1, \widehat{t}, N_2) \in \mathcal{R}$ which strongly progresses to a known SBT bisimulation $\mathcal{R}'$. The choice of $\mathcal{R}'$ depends on the particular equivalence we are trying to establish. One of the most common cases is when $\mathcal{R}' = \simeq$. However, in general we may not have a relation $\mathcal{R}'$ known to be a bisimulation. Nevertheless, we may find that $\mathcal{R}$ progresses to a relation $\mathcal{R}' = \mathcal{F}(\mathcal{R})$ for some mapping $\mathcal{F}$ over relations. The idea is that if $\mathcal{R}$ progresses to $\mathcal{F}(\mathcal{R})$ and $\mathcal{F}$ satisfies certain conditions, then $\mathcal{R}$ is included in $\simeq$. Thus, to establish $N_1 \simeq_{\widehat{t}} N_2$ we need to find such an $\mathcal{F}$ whenever $\mathcal{R}$ contains $(N_1, \widehat{t}, N_2)$.

Suitable mappings $\mathcal{F}$ are characterised in [20] as being *strongly safe* which, in our context, means that for any timed relations $\mathcal{R}$ and $\mathcal{R}'$, if $\mathcal{R} \subseteq \mathcal{R}'$ and $\mathcal{R} \rightsquigarrow \mathcal{R}'$, then $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{R}')$ and $\mathcal{F}(\mathcal{R}) \rightsquigarrow \mathcal{F}(\mathcal{R}')$. More details about the 'up-to' techniques and safe functions can be found in [20].

6 Conclusion

This paper presents an approach in which local clocks operate independently, and so processes at different locations evolve asynchronously. On the other hand, processes operating at the same location evolve synchronously, and temporal evolution of mobile processes is governed by independent local clocks and their migration timeouts. A computational step captures the cumulative effect of the concurrent execution of a group of actions executed at one location.

In process calculi such as distributed π-calculus, timed distributed π-calculus and other formalisms with explicit migration operators, bisimulations are used to compare behaviours of mobile processes evolving in distributed systems with explicit locations. Bisimulations are behavioural equivalences used to study the properties of a concurrent system by verifying its bisimilarity with a system known to enjoy those properties. Moreover, given the model of a system, bisimulations can be used to consider equivalent simplified models.

In this paper, we defined behavioural equivalences between migrating process in distributed systems in terms of local time and locations. In particular, the strong timed bisimilarity can be used to compare the complete computational steps of two networks. Moreover, two networks that are strong bounded timed bisimilar up to certain deadlines $\widehat{t}$ remain equivalent provided that their execution is restricted to the time limit given by $\widehat{t}$. We also defined extended versions of both equivalences which can support compositional reasoning.

Acknowledgement. The work of Bogdan Aman and Gabriel Ciobanu was supported by a grant from the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-II-ID-PCE-2011-3-0919. In addition, the work of Bogdan Aman was supported by POSDRU/89/1.5/S/49944.

References

1. Aman, B., Ciobanu, G.: Timed Mobile Ambients for Network Protocols. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 234–250. Springer, Heidelberg (2008)
2. Baeten, J.C.M., Bergstra, J.A.: Discrete time process algebra: Absolute time, relative time and parametric time. Fundam. Inform. 29(1-2), 51–76 (1997)
3. Berger, M.: Towards Abstractions for Distributed Systems. Ph.D. thesis, Department of Computing, Imperial College (2002)
4. Bloom, B., Istrail, S., Meyer, A.R.: Bisimulation can't be traced. In: POPL, pp. 229–239 (1988)
5. Cardelli, L., Gordon, A.D.: Mobile Ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)
6. Chen, L.: Timed Processes: Models, Axioms and Decidability. Ph.D. thesis, School of Informatics, University of Edinburgh (1993)
7. Ciobanu, G.: Behaviour Equivalences in Timed Distributed π-Calculus. In: Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.) Soft-Ware Intensive Systems. LNCS, vol. 5380, pp. 190–208. Springer, Heidelberg (2008)
8. Ciobanu, G., Juravle, C.: Flexible software architecture and language for mobile agents. Concurrency and Computation: Practice and Experience 24(6), 559–571 (2012)
9. Ciobanu, G., Koutny, M.: Modelling and Verification of Timed Interaction and Migration. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 215–229. Springer, Heidelberg (2008)
10. Ciobanu, G., Koutny, M.: Timed Migration and Interaction with Access Permissions. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 293–307. Springer, Heidelberg (2011)
11. Ciobanu, G., Koutny, M.: Timed mobility in process algebra and petri nets. J. Log. Algebr. Program. 80(7), 377–391 (2011)
12. Ciobanu, G., Prisacariu, C.: Timers for distributed systems. Electr. Notes Theor. Comput. Sci. 164(3), 81–99 (2006)

13. Groote, J.F.: Transition system specifications with negative premises. Theoretical Computer Science 118, 263–299 (1993)
14. Hennessy, M.: A distributed π-calculus. Cambridge University Press (2007)
15. Hennessy, M., Regan, T.: A process algebra for timed systems. Inf. Comput. 117(2), 221–239 (1995)
16. Milner, R.: Communicating and mobile systems - the π-calculus. Cambridge University Press (1999)
17. Nicollin, X., Sifakis, J.: The algebra of timed processes, atp: Theory and application. Inf. Comput. 114(1), 131–178 (1994)
18. Posse, E., Dingel, J.: Theory and Implementation of a Real-Time Extension to the π-Calculus. In: Hatcliff, J., Zucca, E. (eds.) FMOODS/FORTE 2010. LNCS, vol. 6117, pp. 125–139. Springer, Heidelberg (2010)
19. Sangiorgi, D.: A theory of bisimulation for the π-calculus. Acta Inf. 33(1), 69–97 (1996)
20. Sangiorgi, D., Walker, D.: The π-Calculus - a theory of mobile processes. Cambridge University Press (2001)
21. Yi, W.: A Calculus of Real-Time Systems. Ph.D. thesis, Department of Computer Science, Chalmers University of Technology (1991)

Checking Soundness of Business Processes Compositionally Using Symbolic Observation Graphs

Kais Klai[1] and Jörg Desel[2]

[1] LIPN, CNRS UMR 7030, Université Paris 13, France
[2] FernUniversität in Hagen, 58084 Hagen, Germany

Abstract. The Symbolic Observation Graph (SOG) associated with a labelled transition system and a subset of its labels is an efficient BDD-based abstraction representing the behavior of a system. The goal of this paper is to compose SOGs such that the resulting SOG is still small but represents the behavior of the composed business process in an appropriate way. In particular, we would like to deduce the properties of a composed business process by analysing the composition of the SOGs associated with its components. This question was already answered for the deadlock-freeness property in previous work. In this paper, we extend this result to other generic properties: the so-called soundness properties. These properties guarantee the absence of livelocks, deadlocks and other anomalies that can be formulated without domain knowledge. Thus, we show how the SOG can be adapted and used so that the verification of several variants of the soundness property can be performed modularly.

1 Introduction

Behavioral correctness of a process model can be defined in various ways, depending on the properties considered. For different notions of correctness, and for different process modeling languages, there exist a variety of tools to check correctness. The most important challenge with checking correctness is its inherent high complexity; since correctness refers to the model behavior, each straightforward algorithm requires the construction of a behavioral representation of the model, which is often very large or even infinite. If a business process model is obtained by composition of other models, then concurrency between the respective activities leads to the well-known state explosion problem. Our approach to tackle the state explosion problem is to: (1) provide a behavioral model of a single business process model which is of manageable size but contains sufficient information about the process' behavior such that the relevant properties can be checked using this model, and (2) provide an efficient composition operation on this behavioral model such that analysis of this composed model yields results on the composed business process.

As a behavioral model we use the Symbolic Observation Graph (SOG) [7] which is an efficient BDD-based abstraction of the behavior of a system model.

Formally, a SOG can be viewed as a coarse representation of the state graph of a system model. Algorithmically, this state graph does not have to be constructed explicitly because the SOG can directly, and on the fly, be obtained from the original model. In this paper, the example models will be WF-nets [2]. However, since we do not restrict our work to a particular modeling language, we nevertheless start with state-based representations of process models, namely with Labeled Transition Systems (LTS). Recall that this is done only for presentation purposes and does not mean that an LTS has to be constructed in our approach.

In business process modelling, *soundness* represents a relevant property which is frequently studied. There exist various variants of soundness notions that weaken or strengthen the original definition given in [1]. Roughly speaking, soundness requires that every task of a business process model can actually occur and that it is always possible to reach a legal final state. The notion of *relaxed soundness* is introduced in [5]. This notion allows for potential deadlocks and livelocks, however, each task should occur in at least one proper execution (leading to a final state). In [13] the notion of *weak soundness*, allowing for dead transition, is proposed. Finally, *easy soundness* [17] only requires that the final state is reachable from the initial state. Other variants of soundness addressing problems related to multiple instantiation of the workflow model (e.g., k-soundness and generalized soundness [18]) or focusing on termination conditions (e.g., lazy soundness [15]) are not considered in this paper.

We first translate the definition of these variants of the soundness property, originally defined for Petri nets, to the LTS notation. Then, we show how checking these properties can be done on a SOG instead of the underlying LTS. Finally, we establish that, when the components of a composed business process are proved to be sound, how to check using SOGs whether the composition is sound or not. The last task is performed by considering only the collaboration activities of the model components. In other words, what has been already checked locally is not checked again after composition.

The paper is organized as follows: In Section 2, we give definitions and useful notations. In Section 3, we describe an example of an interorganizational workflow to illustrate the presented concepts and to progressively apply our approach. The Symbolic Observation Graph and the preservation results are presented in Section 4. Composition operators are defined in Section 5 while Section 6 is dedicated to discussing related works and to comparing our approach with existing ones. Section 7 concludes the paper and provides some future perspectives.

2 Preliminaries

The technique presented in this paper applies to different languages for business process modeling that can map to Labeled Transition Systems (one prominent example is the language of WF-nets). For the sake of simplicity and generality, we choose to present it directly for Labeled Transition Systems.

Definition 1 (Labeled Transition System). *A* Labeled Transition System *(LTS for short) is a* 5*-tuple* $\langle \Gamma, Act, \rightarrow, I, F \rangle$ *where*

– Γ *is a nonempty finite set of* states
– *Act is a nonempty finite set of* actions
– $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ *is a* transition relation
– $I \subseteq \Gamma$ *is a nonempty set of* initial states
– $F \subseteq \Gamma$ *is a nonempty set of* final states

In this paper, we restrict the set of states Γ to those that are reachable from an initial state in I. Moreover, we assume that final states are terminal, i.e., no final state has a successor. We distinguish observed actions, denoted by the set *Obs*, from unobserved actions, denoted by *UnObs* (with $Obs \cup UnObs = Act$ and $Obs \cap UnObs = \emptyset$). Here, the observed actions are those belonging to the interface (i.e., collaborative actions) while unobserved actions are those performing local activities.

– For $s, s' \in \Gamma$ and $a \in Act$, we denote by $s \xrightarrow{a} s'$ that $(s, a, s') \in \rightarrow$ and by $s \xrightarrow{a}$ that $s \xrightarrow{a} s''$ for some state s''.
– If $\sigma = a_1 a_2 \cdots a_n$ is a sequence of actions, $\overline{\sigma}$ denotes the set of actions occurring in σ, while $|\sigma|$ denotes the length of σ. $s \xrightarrow{\sigma} s'$ denotes that $\exists s_1,\ s_2, \cdots s_{n-1} \in \Gamma$: $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots s_{n-1} \xrightarrow{a_n} s'$ and is called a path.
– For a state s, the set *Enable(s)* denotes the set of actions a such that $s \xrightarrow{a}$. For a set of states S, *Enable(S)* denotes $\bigcup_{s \in S}$*Enable(s).*
– For $s \in (\Gamma \setminus F)$, $s \not\rightarrow$ denotes that s is a dead state, i.e., $Enable(s) = \emptyset$.
– $Sat(s) = \{s' \mid s \xrightarrow{\sigma} s' \wedge \overline{\sigma} \subseteq UnObs\}$ is the set of states that are reachable from a state s by using unobserved actions only. For $S \subseteq \Gamma$, $Sat(S) = \bigcup_{s \in S} Sat(s)$.
– $s \Rightarrow s'$ means that state s' is reachable from state s (possibly through observed actions).
– For $s \in \Gamma$, $s \not\Rightarrow$ denotes that no state of $Sat(s)$ is final or enables an observed action, i.e., $Sat(s) \cap F = \emptyset \wedge Enable(Sat(s)) \cap Obs = \emptyset$. Conversely, $s \Rightarrow$ means that either a final state or a state enabling an observed action is reachable from s.
– A finite path $C = s_1 \xrightarrow{\sigma} s_n$ is said to be a *cycle* if $s_n = s_1$ and $|\sigma| \geq 1$. If moreover $\overline{\sigma} \subseteq UnObs$ then C is said to be a *livelock*. If, in addition, $s_1 \not\Rightarrow$ then C is called a *strong livelock* (a terminal cycle). Otherwise it is called a *weak livelock.*

If $s \not\Rightarrow$, only a dead state or a *strong livelock* are reachable from s. In this paper we assume that a strong livelock behavior is equivalent to a dead state because these two behaviors are not distinguishable. Both will be called deadlock. In the sequel, the set *Dead(S)*, for a given subset of states S, denotes the set of states $s \in S$ satisfying $s \not\Rightarrow$.

Definition 2. *Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I, F \rangle$ be an LTS. Then $\mathcal{T}$ is said to be:*

– *sound iff*
  • $\forall s \in \Gamma\ \exists f \in F$: $s \Rightarrow f$
  • $\forall t \in Act\ \exists s \in \Gamma$: $s \xrightarrow{t}$
– *relaxed sound iff* $\forall t \in Act\ \exists s,\ s' \in \Gamma\ \exists f \in F$: $s \xrightarrow{t} s' \ \wedge\ s' \Rightarrow f$

– *weakly sound iff* $\forall s \in \Gamma\ \exists f \in F : s \Rightarrow f$
– *easily sound iff* $\exists i \in I\ \exists f \in F : i \Rightarrow f$

The soundness property, originally defined in [1], has two requirements. The first one is that it is always possible to reach a final state. If we assume an appropriate notion of fairness, then this requirement implies that a final state is eventually reached from an initial state. If we require termination without such an assumption, all models allowing loops in their execution sequences would be unsound, which is clearly undesirable. Relaxed soundness [5] allows for potential deadlocks and livelocks. However, each action should occur in at least one "good" execution path. Weak soundness [13] allows for dead transitions as long as a final state is reachable from any state. Finally, easy soundness [17] requires that a final state is reachable from some initial state. It is obvious that soundness implies both relaxed and weak soundness, which are incomparable, and that each other soundness notion implies easy soundness.

In the following, we define the synchronized product of two LTSs. The synchronized product of n LTS (for $n > 2$) can be built by iterative multiplication.

Definition 3 (LTS Synchronized Product). *Let* $\mathcal{T}_i = \langle \Gamma_i, Act_i, \rightarrow_i, I_i, F_i \rangle$, $i = 1, 2$ *be two LTSs. The* synchronized product *of* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *is the minimal LTS* $\mathcal{T}_1 \times \mathcal{T}_2 = \langle \Gamma, Act, \rightarrow, I, F \rangle$ *given by:*

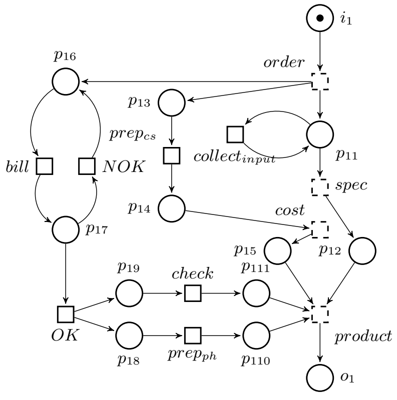
1. $\Gamma \subseteq \Gamma_1 \times \Gamma_2$
2. $Act = Act_1 \cup Act_2$
3. $\rightarrow$ *is the transition relation, defined by:*
$\forall (s_1, s_2) \in \Gamma : (s_1, s_2) \xrightarrow{a} (s_1', s_2') \Leftrightarrow$
$$\begin{cases} s_1 \xrightarrow{a}_1 s_1' \wedge s_2 \xrightarrow{a}_2 s_2' & \text{if } a \in Act_1 \cap Act_2 \\ s_1 \xrightarrow{a}_1 s_1' \wedge s_2 = s_2' & \text{if } a \in Act_1 \setminus Act_2 \\ s_1 = s_1' \wedge s_2 \xrightarrow{a}_2 s_2' & \text{if } a \in Act_2 \setminus Act_1 \end{cases}$$
4. *The set of states* Γ *contains all (and by minimality only) reachable states:*
$\Gamma = \{(s_1, s_2) \in \Gamma_1 \times \Gamma_2 \mid \exists (i_1, i_2) \in I_1 \times I_2\ \exists \sigma \in Act^* : (i_1, i_2) \xrightarrow{\sigma} (s_1, s_2)\}$
5. $I = I_1 \times I_2$
6. $F = (F_1 \times F_2) \cap \Gamma$

Every state of the synchronized product is a pair of states, the first component indicating the respective state of the first LTS, the second component indicating the respective state of the second LTS. Each LTS can still do its private activities autonomously, i.e., only one component of the pair representing a state of the composed LTS is changed by such an action. For common activities both components of the state are changed synchronously.

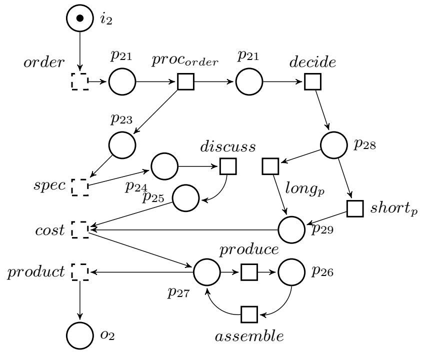
3 Running Example

To introduce the problems tackled in this paper we use an example, taken from [4], of an interorganizational workflow involving two business partners: a contractor and a subcontractor. Figure 1 illustrates the WF-nets associated with these business processes. We choose WF-nets to represent these processes instead of the corresponding LTSs because the LTSs are too large (38 states and 104 edges for the contractor's LTS, 14 states and 22 edges for the subcontractor's LTS). The collaborative tasks are represented by dashed transitions and are the only observed actions. The main scenario of the collaboration between these two partners is the following: First, the contractor sends an order to the subcontractor. Then, the contractor sends a detailed specification to the subcontractor and the subcontractor sends a cost statement to the contractor. Based on the specification, the subcontractor manufactures the desired product and sends it to the contractor. Several transitions (tasks) have been added to the original WF-net of the contractor. They are only of local interest, e.g., between the sending of an order and creation of the specification, the task called $collect_{input}$ may be executed multiple times. Internal transitions were also added to the original WF-net of the subcontractor. Both processes are sound (hence relaxed, weakly and easily sound). The same holds for the process model obtained by composing these two WF-nets (obtained by merging the common transitions) and for the corresponding LTSs.

For both models, the initial marking represents the only initial state, and the only final marking is the one with one token in o_1 (o_2 respectively) and no token elsewhere.



(a) WF-net of a contractor



(b) WF-net of a subcontractor

Fig. 1. The WF-nets of a contractor and of a subcontractor

4 Symbolic Observation Graphs

In this section, we show how *Symbolic Observation Graphs* [7] (*SOGs*) can be used to abstract processes while allowing their analysis with respect to the various soundness notions. The construction of a *SOG* associated with an LTS is guided by a subset of *observed* actions. The *SOG* is defined as a graph where each node is a set of states linked by unobserved actions and each arc is labeled by an observed action. Nodes of the *SOG* are called *aggregates* and may be represented and managed efficiently using decision diagram techniques (e.g., BDDs). In practice, the size of a SOG is proportional to the number of observed actions (see [7,10,9] for experimental results). Thus, by observing only the collaborative actions of a business process, one can hide the internal behavior and hope for a reduced size of the SOG when building and analysing composed business processes, especially when the components are loosely coupled.

Definition 4 (Aggregate). *Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I, F\rangle$ be a Labeled Transition System with $Act = Obs \cup UnObs$. An* aggregate *is a tuple $a = \langle S, d, f\rangle$ defined as follows:*

1. *S is a non-empty subset of Γ satisfying $Sat(S) = S$*
2. *$d \in \{true, false\}$; $d = true$ iff $Dead(S) \neq \emptyset$*
3. *$f \in \{true, false\}$; $f = true$ iff $S \cap F \neq \emptyset$*

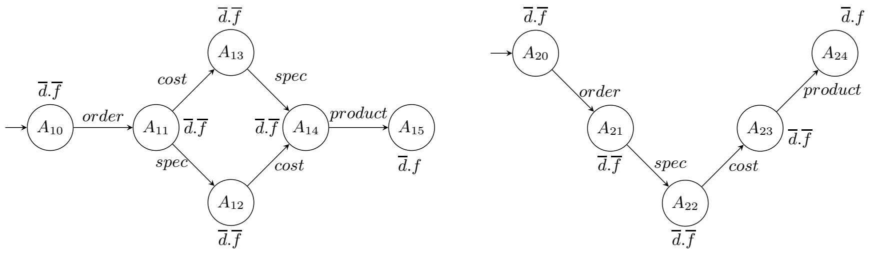
From now on, $a.S$, $a.d$ and $a.f$ denote the corresponding attributes of an aggregate a.

Definition 5 (Symbolic Observation Graph). *A* symbolic observation graph *associated with an LTS $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F\rangle$ is a five-tuple $\langle \mathcal{A}, Act', \rightarrow', I', F'\rangle$ where:*

1. *$\mathcal{A}$ is a finite set of aggregates satisfying:*
   - *there is an aggregate $a_0 \in \mathcal{A}$ with $a_0.S = Sat(I)$*
   - *if, for some $a \in \mathcal{A}$ and $o \in Obs$, the set $Ext(a, o) := \{s' \notin a.S \mid \exists s \in a.S, s \xrightarrow{o} s'\}$ is not empty, then it is a pairwise disjoint union of non-empty sets $S_1 \ldots S_k$, and for $i = 1 \ldots k$, there is an aggregate $a_i \in \mathcal{A}$ with $a_i.S = Sat(S_i)$*
2. *$Act' = Obs$*
3. *$\rightarrow' \subseteq \mathcal{A} \times Act' \times \mathcal{A}$ is the transition relation satisfying:*
   - *if $a \neq a'$ then $(a, o, a') \in \rightarrow'$ iff $a'.S = Sat(S')$ for some $S' \subseteq Ext(a, o)$*
   - *$(a, o, a) \in \rightarrow'$ iff $Sat(\{s' \in \Gamma \mid \exists s \in a.S, s \xrightarrow{o} s'\}) = a.S$*
4. *$I' = \{a_0\}$ (where $a_0.S = Sat(I)$)*
5. *$F' = \{a \in \mathcal{A} \mid a.S \cap F \neq \emptyset\}$ ($=\{a \in \mathcal{A} \mid a.f = true\}$)*

Notice that Definition 5 does not guarantee the uniqueness of a SOG for a given LTS. In fact, it supplies a certain flexibility for its implementation. In particular, the SOG can be nondeterministic even if the original LTS is not. Actually, one can take advantage of such nondeterminism to obtain smaller aggregates. Even if the SOG obtained in this way has more aggregates than a deterministic one, its construction might consume less time and memory.

Definition 6. *Let $\mathcal{G} = \langle \mathcal{A}, Act', \rightarrow', I', F'\rangle$ be a SOG over a set of observed actions Obs, corresponding to an LTS $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I, F\rangle$. Let $Live(a)$, for an aggregate a, be the set of non dead-states i.e., $Live(a) := a.S \setminus Dead(a.S)$, and let $L(a) := \{t \in Enable(Live(a)) \mid Succ(Live(a), t) \cap Live(a) \neq \emptyset\}$, where $Succ(S, t) := \{s' \mid \exists s \in S \colon s \xrightarrow{t} s'\}$.Then $\mathcal{G}$ is said to be:*



(a) SOG of the contractor (b) SOG of the subcontractor

Fig. 2. SOGs of the contractor and of the subcontractor

– *sound iff*
  - $\forall a \in \mathcal{A}$: $a.d = false$ *and* $\exists f \in F'$: $a \Rightarrow f$ *and*
  - $\bigcup_{a \in A} Enable(a.S) = Act$
– *relaxed sound iff* $\forall t \in Act\ \exists a \in \mathcal{A}\ \exists f \in F' : t \in L(a) \wedge a \Rightarrow f$
– *weakly sound iff* $\forall a \in \mathcal{A}$: $a.d = false$ *and* $\exists f \in F'$: $a \Rightarrow f$
– *easily sound iff* $\exists f \in F'$: $a_0 \Rightarrow f$

The soundness notions are extended to SOGs in order to ensure an equivalence between the soundness of a SOG and the soundness of the underlying LTS. The translation is immediate for all the variants except relaxed soundness. In order to check if each action (observed or not) belongs to a proper execution sequence, we exclude all the dead states (see Section 5.3 for an efficient computation of $Dead(a.S)$) from each aggregate and check wether the obtained subset allows to reach a final aggregate. The absence of dead actions is checked in a similar way.

Proposition 1. *Let $\mathcal{G}$ be a SOG over an arbitrary set of observed actions Obs corresponding to an LTS $\mathcal{T}$. Then the following holds:*

1. *$\mathcal{T}$ is sound $\Leftrightarrow$ $\mathcal{G}$ is sound*
2. *$\mathcal{T}$ is relaxed sound $\Leftrightarrow$ $\mathcal{G}$ is relaxed sound*
3. *$\mathcal{T}$ is weakly sound $\Leftrightarrow$ $\mathcal{G}$ is weakly sound*
4. *$\mathcal{T}$ is easily sound $\Leftrightarrow$ $\mathcal{G}$ is easily sound*

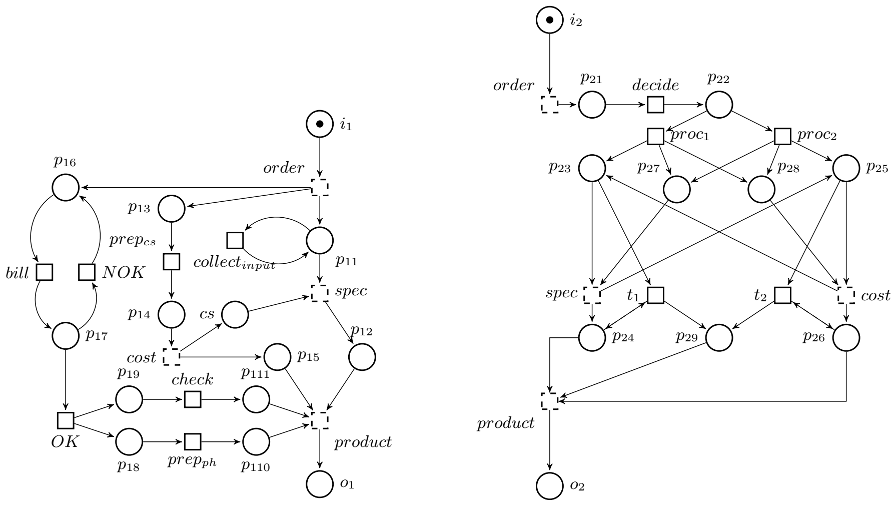
Figure 2 shows two (deterministic) SOGs associated with the WF-nets of the contractor (Figure 2(a)) and the subcontractor (Figure 2(b)) of Figure 1. Each aggregate a is indexed with its attributes $a.d$ and $a.f$. The symbol d (resp. $\overline{d}$) is used when a contains (resp. does not contain) a dead state and the symbol f (resp. $\overline{f}$) is used when a contains (resp. does not contain) a final state. Notice that states of the corresponding LTS are partitioned into aggregates which is not necessary the case in general (i.e., a single state may belong to two (or more) different aggregates).

Note that the corresponding LTSs contain 38 nodes and 104 edges, and 14 nodes and 22 edges, respectively. None of the aggregates of the contractor's (resp. the subcontractor's) SOG contains a deadlock. Both are sound.

5 Composition of SOGs

It is well known that deadlock-freeness is not preserved by composition.

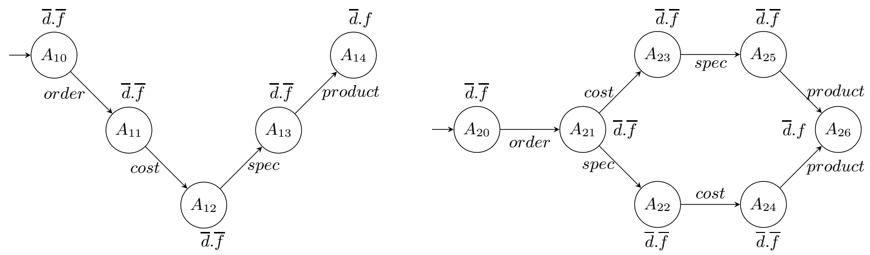
Figure 3(a) presents a WF-net which is almost the same as the WF-net of Figure 1(a). Only the additional place *cs* has been added between transitions *cost* and *spec* to order the corresponding tasks. An alternative WF-net for the subcontractor is represented in Figure 3(b). This workflow contains three new transitions: *decide*, $proc_1$, and $proc_2$. After an order has been received, a decision is made. Based on this decision, one of two possible procedures is executed. In one procedure (transition $proc_1$), the specification is processed before a cost statement is created. In the other procedure (transition $proc_2$), the cost statement is created before the specification is processed. Although both WF-nets are deadlock-free, the synchronization of these models by merging related transitions is not. In fact, the composed process gets stuck as soon as the contractor sends an order to the subcontractor who decides to process it by procedure $proc_1$.



(a) Alternative contractor (b) Alternative subcontractor

Fig. 3. Alternative WF-nets of a contractor and a subcontractor

Instead of analysing the synchronized product of the underlying LTSs (134 nodes and 480 edges), and detecting such an incorrect behavior, we propose in this section to compose the corresponding SOGs (see Figure 4) in such a way that this behavior is detectable. This approach presents several advantages: First, the verification of the composition takes into account the local verification process. We only focus on the common activities between the processes to be composed. The main task at this stage is to check whether, due to the composition, the desirable properties have been violated. Second, such an approach allows to reduce the state space explosion due to the composition. Finally, by abstracting



(a) A SOG of the modified contractor (b) A SOG of the modified subcontractor

Fig. 4. Two SOGs of the new contractor and of the new subcontractor

a business process with a SOG, we hide the local behavior of the process which might represent internal organisation and private information. This allows to respect the privacy feature of the enterprise and to avoid to expose irrelevant or sensitive information.

5.1 Observed Behavior

In the following, we show how, using local information of two aggregates, one can compute the attributes of the aggregate resulting from their synchronisation. Before we define an aggregate a obtained by composition of two aggregates a_1 and a_2, let us define the following particular mapping (called *observed behavior*) applied to states of an LTS $\mathcal{T}$, and extend it progressively to aggregates. It will be established that the *observed behavior* associated with an aggregate is the necessary and sufficient local information to be retained so that soundness properties can be checked on the composition of two process models. For this purpose, and for the remaining part of this paper, we assume the existence of an additional *virtual* observed action *"term"* belonging to *Obs*.

Definition 7 (Observed Behavior Mapping)
Let $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F\rangle$ be an LTS. Let a be an aggregate of a SOG associated with $\mathcal{T}$. The observed behavior is progressively defined by :

1. $\lambda_{\mathcal{T}} : \Gamma \rightarrow 2^{Obs}$
$$\lambda_{\mathcal{T}}(s) = \begin{cases} (Enable(Sat(s)) \cap Obs) \cup \{term\} & \text{if } F \cap Sat(s) \neq \emptyset \\ Enable(Sat(s)) \cap Obs & \text{otherwise} \end{cases}$$
2. $\lambda_{\mathcal{T}} : 2^{\Gamma} \rightarrow 2^{Obs}$
$\lambda_{\mathcal{T}}(S) = \{\lambda_{\mathcal{T}}(s) \mid s \in S\}$
3. $\lambda_a = \{X \in \lambda_{\mathcal{T}}(a.S) \mid \nexists Y \in \lambda_{\mathcal{T}}(a.S) :\ Y \subset (X \setminus \{term\})\}$.

Informally, for each state s of an LTS $\mathcal{T}$, the observed behavior of s, $\lambda_{\mathcal{T}}(s)$, represents the set of observed actions which can be executed from s, possibly via a sequence of unobserved actions. In addition, *term* is a member of $\lambda_{\mathcal{T}}(s)$ if and only if a final state is reachable from s using unobserved actions only. The observed behavior $\lambda_{\mathcal{T}}$ associated with a set of states S is a set of sets of observed actions. This set contains the observed behavior of the states of S. Finally, the observed behavior of an aggregate a, namely λ_a, is the minimal set of subsets (w.r.t. the set inclusion relation) of $\lambda_{\mathcal{T}}(a.S)$. The inclusion relation does not concern the *term* action. For instance, if there exist two states $s, s' \in a.S$ such that $\lambda_{\mathcal{T}}(s) = \emptyset$ and $\lambda_{\mathcal{T}}(s') = \{term\}$, then both sets $\emptyset$ and $\{term\}$ will belong to λ_a. This way we distinguish a dead state from a final state reached in $a.S$.

Table 1. Illustration of the observed behavior function

C_1		SC_1		C_2		SC_2	
a	λ_a	a	λ_a	a	λ_a	a	λ_a
A_{10}	$\{\{order\}\}$	A_{20}	$\{\{order\}\}$	A_{10}	$\{\{order\}\}$	A_{20}	$\{\{order\}\}$
A_{11}	$\{\{spec\},\{cost\}\}$	A_{21}	$\{\{spec\}\}$	A_{11}	$\{\{cost\}\}$	A_{21}	$\{\{spec\},\{cost\}\}$
A_{12}	$\{\{spec\}\}$	A_{22}	$\{\{cost\}\}$	A_{12}	$\{\{spec\}\}$	A_{22}	$\{\{cost\}\}$
A_{13}	$\{\{cost\}\}$	A_{23}	$\{\{product\}\}$	A_{13}	$\{\{product\}\}$	A_{23}	$\{\{spec\}\}$
A_{14}	$\{\{product\}\}$	A_{24}	$\{\{term\}\}$	A_{14}	$\{\{term\}\}$	A_{24}	$\{\{product\}\}$
A_{15}	$\{\{term\}\}$	-	-	-	-	A_{25}	$\{\{product\}\}$
-	-	-	-	-	-	A_{26}	$\{\{term\}\}$

Table 1 illustrates the observed behavior of each aggregate of the SOGs associated with both versions of our running example. C_1 and SC_1 (resp. C_2 and SC_2) stand for the SOGs associated with the contractor and the subcontractor of Figure 2 (resp. Figure 4) respectively.

The observed behavior associated with an aggregate a allows us to get rid of the attributes $a.d$ and $a.f$ which can be directly deduced from λ_a as follows:

Proposition 2. *Let a be an aggregate of a SOG $\mathcal{G}$ (associated with an LTS $\mathcal{T}$).*

1. *$a.d = true$ if and only if $\emptyset \in \lambda_a$*
2. *$a.f = true$ if and only if $\exists O \in \lambda_a : term \in O$*

From now on, an aggregate a is identified by its observed behavior λ_a. In fact, the set of states $a.S$ of an aggregate a has not to be stored explicitly within an aggregate. Once the SOG is built (and the soundness properties checked), it will not play any further role in the composition process.

When composing several processes, each SOG is computed locally by taking into account the observed behavior of each aggregate. The obtained SOGs are then composed leading to a new SOG. The observed behavior of each aggregate of this SOG is deduced from those of the composed aggregates, as follows:

Definition 8. *For $i = 1, 2$, let $\mathcal{G}_i$ be two SOGs corresponding to $\mathcal{T}_i = \langle \Gamma_i, Obs_i \cup UnObs_i, \rightarrow_i, I_i, F_i \rangle$ and let $a_i = \langle \lambda_{a_i} \rangle$ be an aggregate of $\mathcal{G}_i$. The* product aggregate *$a = \langle \lambda_a \rangle = a_1 \times a_2$ is defined by:*

$$\lambda_a = \{(x \cap y) \cup (x \cap (Obs_1 \setminus Obs_2)) \cup (y \cap (Obs_2 \setminus Obs_1)) \mid x \in \lambda_{a_1},\ y \in \lambda_{a_2}\}$$

Note first that the sets of observed actions Obs_1 and Obs_2 are not necessarily identical (but they share at least the virtual action *term*). When we compose a_1 and a_2, if a_1 can progress in $\mathcal{G}_1$ by using locally observed actions (i.e., actions that are observed in $\mathcal{G}_1$ but not shared by $\mathcal{G}_2$), the product aggregate a should be able to do the same. If this is not the case, then a has to have the same behavior as a_1 and a_2 conjointly. In this way, the observed behavior associated with a product aggregate is helpful to deduce whether the involved set of (pairs of) states contains a deadlock. Moreover, once computed, the observed behavior of $a = a_1 \times a_2$ still respects Proposition 2: The product aggregate contains a deadlock iff the corresponding observed behavior contains the empty set, and it is a final aggregate iff *term* belongs to one of its observed behavior's elements. Typically, a composed deadlock is a dead state $\langle s_1, s_2 \rangle$ where the shared observed transitions that are enabled in s_1 are all not enabled in s_2 (or viceversa).

The following definition characterizes the *composed deadlocks*: the deadlocks that are only due to the composition.

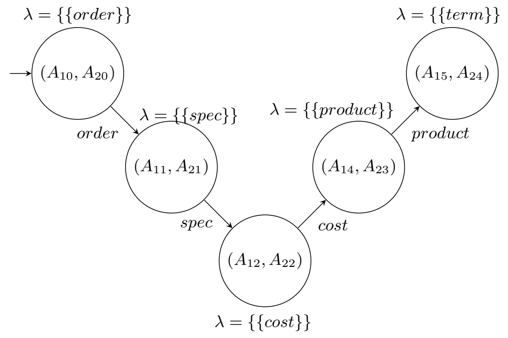
Definition 9. *Let $\mathcal{G}$ be the SOG obtained by synchronizing two SOGs $\mathcal{G}_1$ and $\mathcal{G}_2$. $\mathcal{G}$ is said to be containing a composed deadlock iff it contains an aggregate $a = a_1 \times a_2$ such that $\exists (x, y) \in \lambda_{a_1} \times \lambda_{a_2}$ satisfying:*

1. $x \neq \emptyset \wedge y \neq \emptyset$ *and* $(x \cap y) \cup (x \cap (Obs_1 \setminus Obs_2)) \cup (y \cap (Obs_2 \setminus Obs_1)) = \emptyset$, *or*
2. $x = \emptyset \wedge \emptyset \subset y \subseteq ((Obs_1 \cap Obs_2) \setminus \{term\})$, *or*
3. $\emptyset \subset x \subseteq ((Obs_1 \cap Obs_2) \setminus \{term\}) \wedge y = \emptyset$.

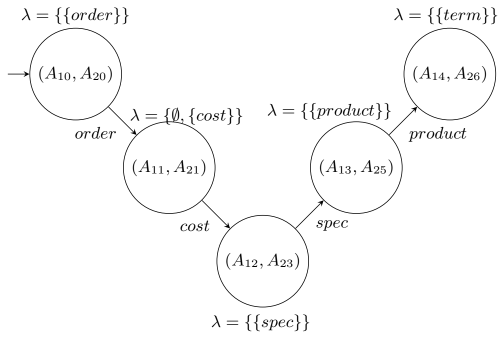
5.2 Synchronous Composition

Given two (or more) LTSs that have been analysed locally and proved to be correct (w.r.t. soundness notions), we would like to reduce the verification of their composition to the verification of the composition of the underlying SOGs. The synchronized product of two SOGs can be defined similarly to the synchronized product of two LTSs (Definition 3). The only difference is that we deal with aggregates (carrying additional information) instead of states. In [11,8] it has been demonstrated that the synchronized product of two SOGs associated with two LTSs is a SOG associated with the synchronized product of these LTSs. Such an approach presents several advantages: First, the verification of the composition takes into account the local verification process. We only focus on the common activities between the processes to be composed. The main task at this stage is to check whether, due to the composition, the desirable properties have been violated. Second, such an approach allows to reduce the state space explosion induced by the concurrency between the activities of the composed components. In fact, these activities are hidden in aggregates of the associated SOGs. Finally, by abstracting a business process with a SOG, we hide the local behavior of the process which would represent internal organisation and private information. This allows to respect the privacy feature of the enterprise and to avoid to expose irrelevant or sensitive information.

Figure 5(a) and Figure 5(b) illustrate the SOGs obtained by synchronizing the SOGs of Figure 2 and Figure 4. The left synchronized SOG inherits the



(a) Synchronized SOG's product



(b) Alternative synchronized SOG's product

Fig. 5. The SOG's synchronized products

same properties of the involved processes. The right synchronized SOG contains a deadlock (aggregate (A_{11}, A_{21})). In fact, $\lambda_{A_{11}} = \{\{p_{cost}\}\}$ and $\lambda_{A_{21}} = \{\{p_{spec}\}, \{p_{cost}\}\}$ which lead to $\lambda_{(A_{11},A_{21})} = \{\emptyset, \{p_{cost}\}\}$, and thus $\{A_{11}, A_{21}\}$ contains a deadlock (a *composed deadlock*).

Proposition 3. *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be SOGs corresponding to the LTSs $\mathcal{T}_1$ and $\mathcal{T}_2$ with respect to observed actions Obs_1 and Obs_2 respectively.*
Let $\mathcal{G} = \langle \mathcal{A}, Obs_1 \cup Obs_2, \rightarrow, I, F \rangle$ be the synchronized product of $\mathcal{G}_1$ and $\mathcal{G}_2$. Then the following holds:

1. *if $\mathcal{G}_1$ and $\mathcal{G}_2$ are sound then $\mathcal{G}$ is sound iff*
   - $\forall a \in \mathcal{A} \colon \emptyset \notin \lambda_a \wedge \exists f \in F \colon a \Rightarrow f$
   - $\forall o \in Obs_1 \cap Obs_2 \; \exists a \in \mathcal{A} \colon a \xrightarrow{o}$
2. *if $\mathcal{G}_1$ and $\mathcal{G}_2$ are relaxed sound then:*
   $\mathcal{G}$ does not contain a composed deadlock $\Rightarrow$ $\mathcal{G}$ is relaxed sound
3. *if $\mathcal{G}_1$ and $\mathcal{G}_2$ are weakly sound then $\mathcal{G}$ is weakly sound iff*
   $\forall a \in \mathcal{A} \colon \emptyset \notin \lambda_a \wedge \exists f \in F \colon a \Rightarrow f$
4. *if $\mathcal{G}_1$ and $\mathcal{G}_2$ are easily sound then $\mathcal{G}$ is easily sound iff $\exists f \in F \colon a_0 \Rightarrow f$*

The soundness notion of a synchronized product of two SOGs involves only the common observed actions. The enabledness of such actions can be checked after composition as well as the deadlock attribute of the composed aggregate. Local infomation has not to be recalculated because it can not be the reason of soundness violation. Hence, the soundness of the synchronized SOG (except the relaxed variant) can be be decided modularly. Concerning, the relaxed soundness, only the absence of *composed deadlock* in the synchronized SOG implies the satisfaction of this property. In fact, the existence of such a deadlock does not allow to know whether such a property still hold or not for the composition.

Corollary 1. *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two LTSs whose synchronized product is $\mathcal{T}$. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be SOGs corresponding to $\mathcal{T}_1$ and $\mathcal{T}_2$ with respect to observed actions Obs_1 and Obs_2 respectively. Let $\mathcal{G}$ be the synchronized product of $\mathcal{G}_1$ and $\mathcal{G}_2$. Then the following holds:*

1. *If $\mathcal{T}_1$ and $\mathcal{T}_2$ are sound then $\mathcal{T}$ is sound iff $\mathcal{G}$ is sound.*
2. *If $\mathcal{T}_1$ and $\mathcal{T}_2$ are relaxed sound then:*
   $\mathcal{G}$ does not contain a composed deadlock $\Rightarrow$ $\mathcal{T}$ is relaxed sound.
3. *If $\mathcal{T}_1$ and $\mathcal{T}_2$ are weakly sound then $\mathcal{T}$ is weakly sound iff $\mathcal{G}$ is weakly sound.*
4. *If $\mathcal{T}_1$ and $\mathcal{T}_2$ are easily sound then $\mathcal{T}$ is easy sound iff $\mathcal{G}$ is easily sound.*

Although in this section we deal with synchronous composition, our technique can also be used for components of a process communicating asynchronously. As long as the whole system is finite, the buffers ensuring the communication between the components together with the associated collaborative actions can be isolated in order to form an intermediate component. The asynchronous composition between two components is thus transformed to a synchronous composition of three components (see. [11,8] for details).

Table 2. Checking Soundness on RG VS SOG

Model	**R. G.**		**SOG**		**S**	**R. S.**	**W. S**	**E. S.**
$Contractor_1$	38	104	6	6	yes	yes	yes	yes
$Contractor_2$	26	66	5	4	yes	yes	yes	yes
$Subontractor_1$	14	22	5	4	yes	yes	yes	yes
$Subontractor_2$	21	22	7	7	yes	yes	yes	yes
$Synchronous_1$	134	480	5	4	yes	yes	yes	yes
$Synchronous_2$	99	320	5	4	no	yes	no	yes
$Asynchronous_1$	248	889	5	4	yes	yes	yes	yes
$Asynchronous_2$	109	373	5	4	no	yes	no	yes

Table 2 summarizes the application of our approach to our running examples. We consider the contractor and subcontractor processes of Figure 1 and their alternatives of Figure 3. Both synchronous and asynchronous compositions are considered. For each obtained model we provide the size (the number of nodes in the first column and the number of edges in the second column) of the corresponding reachability graph (R. G.) and of the SOG. Soundness (S), Relaxed soundness (R. S.), weak soundness (W. S.) and easy soundness (E. S.) are also checked.

5.3 The Observed Behavior Computation Algorithm

A direct implementation of the observed behavior of a given aggregate (following Definition 7) implies to consider each state belonging to the aggregate separately. This would considerably decrease the efficiency of the approach. However, since each aggregate is encoded by a BDD, all the operations manipulating the aggregates should be based on set operations. Therefore, we have implemented an algorithm (see Algorithm 1) for the computation of the observed behavior that is exclusively based on set operations applied to the states of a given aggregate.

The inputs of Algorithm 1 are an aggregate A, a set of observed actions Obs, a set of unobserved actions $UnObs$, and a set of final states F. It computes the observed behavior associated with A (i.e., $A.\lambda$).

We use a map (called R) whose elements are pairs of sets of events and sets of states (line 1). Each element (O, S) satisfies the following: each state of S enables each transition of O. This map is progressively updated so that, at the end of the algorithm, the set composed of its keys form the observed behavior of the aggregagte A (line 18). The first step of the algorithm (lines $2-4$) consists in: (1) checking whether a final state belongs to $A.S$, (2) if it is the case creating a new couple $(\{term\}, S)$ where $term$ is the termination observed action, and S is the set of the immediate predecessors of the final states in $A.S$. The latter task is performed by using the $PreIm()$ function. The second step of the algorithm (lines $5-9$) allows to fill the map R with couples of the form $(\{o\}, S)$ where o is an observed action and S the subset of $A.S$ enabeling o (using function $Enable()$). Once the map R is filled, it is analysed in the third part of the algorithm (lines $10-17$). The idea is to look between elements of R those having the same enabling sets of states (the second component of each couple). For each pair (O, S) and (O', S) in R the first couple is updated by adding O' to O while the second is removed from the map. Indeed, states in S enable both actions in O and actions in O' and should be associated with the set $O \cup O'$.

The final part of the algorithm (lines $19-29$) is dedicated to the analysis of the deadlock states inside the aggregate A. Recall that a dead state is either a (non final) terminal state, or a state belonging to a strong livelock (a terminal cycle). If a deadlock state is found in $A.S$ then the empty set is added to λ. A terminal state is found (lines $19-24$) when the set of states enabling some transition (observed or not) is not equal to the whole set $A.S$. In order to detect strong livelocks (terminal cycles) we iterate on the $PreIm()$ function in order to compute all the states in $A.S$ that possibly lead either to a state in $Enable(A.S, Obs)$ (i.e., a state enabling some observed action), or to a final state. If the result is not equal to $A.S$ then there is a terminal cycle in A and the empty set should belong to $A.\lambda$.

Algorithm 1. Computing the Observed Behavior

```
Require: $Agregate\,A, Obs, UnObs, Set\,of\,state\,F$
Ensure: $A.\lambda$
 1: $Map < Set\,of\,events, Set\,of\,states > R$
 2: if $F \cap A.S \neq \emptyset$ then
 3:     $insert\,(\{term\}, PreIm(F, A.S, UnObs))\,in\,R$
 4: end if
 5: for $o \in Obs$ do
 6:     if $Enable(A.S, o) \neq \emptyset$ then
 7:         $insert\ (\{o\}, Enable(A.S, o))\ in\ R$
 8:     end if
 9: end for
10: for $(O, S) \in R$ do
11:     for $(O', S') \in R$ do
12:         if $S = S'$ then
13:             $(O, S) \leftarrow (O \cup O', S)$
14:             $remove\ (O', S')\ from\ R$
15:         end if
16:     end for
17: end for
18: $\lambda \leftarrow Set\ of\ keys\ of\ R$
19: $Set\,of\,states\,E \leftarrow \emptyset$
20: for $t \in (Obs \cup UnObs)$ do
21:     $E \leftarrow E \cup Enable(S, t)$
22: end for
23: if $E \neq S$ then
24:     $\lambda \leftarrow \lambda \cup \{\emptyset\}$
25: else
26:     if $(PreIm^*(Enable(A.S, Obs) \cup F, UnObs) \neq A.S)$ then
27:         $\lambda \leftarrow \lambda \cup \{\emptyset\}$
28:     end if
29: end if
30: return $\lambda$
```

6 Related Work

The importance of dealing with business processes on the one hand and business process composition on the other hand is reflected in the literature by several publications (e.g., [3,14,12]). To the best of our knowledge, none of the existing approaches combines symbolic abstraction (using BDDs) and modular verification to check the correctness of inter-organisational processes. The originality of our technique is to exploit the efficiency of the SOG's implementation while respecting the privacy of the enterprise, i.e., without exposing irrelevant or sensitive information. Moreover, the SOGs are computed once and locally for each process which reduces the state explosion problem compared to a non-modular approach. Below we discuss some related approaches.

In [16] the authors present various composition alternatives and their ability to preserve relaxed soundness [5]. The aim of this work was to analyze a list of significant composition techniques in terms of WF-nets and to prove that the composition of relaxed sound models is again relaxed sound. Our approach can be applied to any kind of models (not only WF-nets) and allows to check several kinds of soundness including relaxed soundness. In [6], the authors propose an approach for service retrieval based on behavioral specifications. The idea consists of reducing the problem of behavioral matching to a graph matching problem and then adapting existing algorithms for this purpose. The complexity of the graph matchmaking algorithm used is $O(m^2 * n^2)$ in the best case and $O(m^n * n)$ in the worst case where m is the number of nodes of the request graph and n is the number of nodes of the advertised graph [6]. It is obvious that this approach is not suitable for workflow matching and composition when the number of advertised abstractions increases. Another approach for workflow matchmaking was proposed in [14]. It assumes that two workflows match if they are equivalent. To reach this end, the author introduces the notions of communication graph *c-graph* and usability graph (*u-graph*). If the *u-graph* of a workflow is isomorphic to the *c-graph* of another workflow, then the two workflows are considered equivalent. However, the complexity of the *c-graph* construction is exponential in terms of the number of nodes [14].

7 Conclusion

We addressed the problem of checking correctness of inter-organizational business processes compositionally. By correctness we mean soundness with various variants. We established that and how Symbolic Observation Graphs can be extended and efficiently used for that purpose. Moreover, we showed how our approach can be used when the different processes communicate either synchronously or asynchronously.

Our immediate future works follow three directions: First, we are implementing a tool for the abstraction and the verification of inter-organizational business processes. The verification concerns generic properties like deadlock freeness, soundness or specific properties that are expressed by linear-time temporal logics.This helps to check our techniques for concrete applications and makes them available to the scientific community and for practical applications. Second, we plan to extend our approach to deal with resources. Finally, our approach can be used for developing a graph-based registry for abstract process advertisement and discovery.

References

1. 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)
2. van der Aalst, W.M.P.: The application of Petri nets to workflow management. Journal of Circuits, Systems, and Computers 8(1), 21–66 (1998)

3. van der Aalst, W.M.P.: Loosely coupled interorganizational workflows: Modeling and analyzing workflows crossing organizational boundaries. Information and Management 37, 67–75 (2000)
4. van der Aalst, W.M.P.: Inheritance of interorganizational workflows: How to agree to disagree without loosing control? Information Technology and Management 4, 345–389 (2003)
5. Dehnert, J., Rittgen, P.: Relaxed Soundness of Business Processes. In: Dittrich, K.R., Geppert, A., Norrie, M. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 157–170. Springer, Heidelberg (2001)
6. Grigori, D., Corrales, J.C., Bouzeghoub, M.: Behavioral matchmaking for service retrieval. In: ICWS 2006: Proceedings of the IEEE International Conference on Web Services, pp. 145–152. IEEE (2006)
7. Haddad, S., Ilié, J.-M., Klai, K.: Design and Evaluation of a Symbolic and Abstraction-Based Model Checker. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 196–210. Springer, Heidelberg (2004)
8. Klai, K., Ochi, H.: Modular verification of inter-enterprise business processes. In: The Fourth International Conference on Information, Process, and Knowledge Management, eKNOW 2012, pp. 155–161. IEEE (2012)
9. Klai, K., Petrucci, L.: Modular construction of the symbolic observation graph. In: ACSD, pp. 88–97. IEEE (2008)
10. Klai, K., Poitrenaud, D.: MC-SOG: An LTL Model Checker Based on Symbolic Observation Graphs. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 288–306. Springer, Heidelberg (2008)
11. Klai, K., Tata, S., Desel, J.: Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. Data Knowl. Eng. 70(5), 467–482 (2011)
12. Lohmann, N., Wolf, K.: Petrifying operating guidelines for services. In: ACSD, pp. 80–88. IEEE (2009)
13. Martens, A.: On compatibility of web services. Petri Net Newsletter, Special Interest Groups on Petri Nets and Related Systems Models, Gesellschaft fur Informatik e.V. 65, 12–20 (2003)
14. Martens, A.: On Usability of Web Services. In: Calero, C., Daz, O., Piattini, M. (eds.) Web Services Quality Workshop (2003)
15. Puhlmann, F., Weske, M.: Interaction Soundness for Service Orchestrations. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 302–313. Springer, Heidelberg (2006)
16. Siegeris, J., Zimmermann, A.: Workflow Model Compositions Preserving Relaxed Soundness. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 177–192. Springer, Heidelberg (2006)
17. van der Aalst, W.M.P., van Hee, K.M., van der Toorn, R.A.: Component-based software architectures: a framework based on inheritance of behavior. Sci. Comput. Program. 42(2-3), 129–171 (2002)
18. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 337–356. Springer, Heidelberg (2003)

Beyond Lassos: Complete SMT-Based Bounded Model Checking for Timed Automata

Roland Kindermann, Tommi Junttila, and Ilkka Niemelä

Aalto University
Department of Information and Computer Science
P.O. Box 15400, FI-00076 Aalto, Finland
{Roland.Kindermann,Tommi.Junttila,Ilkka.Niemela}@aalto.fi

Abstract. Timed automata (TAs) are a common formalism for modeling timed systems. Bounded model checking (BMC) is a verification method that searches for runs violating a property using a SAT or SMT solver. Previous SMT-based BMC approaches for TAs search for finite counter-examples and infinite lasso-shaped counter-examples. This paper shows that lasso-based BMC cannot detect counter-examples for some linear time specifications expressed, e.g., with LTL or Büchi automata. This paper introduces a new SMT-based BMC approach that can find a counter-example to any non-holding Büchi automaton or LTL specification and also, in theory, prove that a specification holds. Different BMC encodings tailored for the supported features of different SMT solvers are compared experimentally to lasso-based BMC and discretization-based SAT BMC.

1 Introduction

Timed automata, see, e.g., [1–3], are a convenient formalism for describing and model checking finite state systems augmented with real-valued clocks. There are many tools, Uppaal [4] to name just one, for timed automata and model checking algorithms for timed automata have been studied quite a lot during the last two decades. For verification, Uppaal treats the discrete part of a timed automaton's state in an explicit state fashion while using a symbolic representation for the time-related part. Other approaches use decision diagrams for the verification of timed automata [5, 6].

Bounded model checking (BMC) [7] is a symbolic model checking method that has been shown very efficient in bug hunting (i.e., finding counter-examples to specifications) for finite-state systems during the last ten years. Being fully symbolic, it can handle systems with high degree of non-determinism in data and input signals more naturally than explicit-state model checking methods. The basic idea behind BMC is, given a system, a specification, and an integer bound k, to build a propositional logic formula such that the formula is satisfiable if and only if the system has a counter-example of length at most k violating the specification. The bound is incremented until a satisfiable formula is found (implying that the specification does not hold for the system) or a completeness threshold is reached without finding any satisfiable formulas (implying that the specification holds for the system). Infinite runs are handled in BMC by considering finitely representable lasso-shaped infinite runs consisting of a finite prefix followed by a finite loop. In addition to finite state systems, BMC has also been applied to timed automata [8–11]. When a propositional logic encoding is used (as e.g. in [8, 11]), the infinite state space of a timed automaton has to be reduced into a finite one; this can be achieved by using the region abstraction [1], see e.g. [8, 12] for two different propositional logic encodings of regions. A direct benefit of using the region abstraction is that the resulting BMC method can indeed detect whether a propositional ω-regular specification (expressed, e.g., with propositional linear time temporal logic LTL) holds on the system or not by considering lasso-shaped infinite runs only.

Propositional encodings of regions can be rather complicated and large for systems with many clocks with wide ranges. The introduction of Satisfiability Modulo Theories (SMT, see e.g. [13]) solvers with built-in support for reasoning over real and integer arithmetics has made it possible to devise BMC approaches for timed automata *without* using the region abstraction [9, 10]. In these approaches, the transition relation of the automaton is directly expressed as a propositional logic formula augmented with linear arithmetic constraints. In this paper we show that these previous SMT-based BMC approaches for timed automata are actually incomplete in the sense that for some timed automata they cannot find (a representative of) any infinite run despite such runs exist. This is basically caused by the fact that they search for lasso-shaped infinite runs (of the automaton, *not* of its region abstraction) but, unlike in the context of finite state systems, some timed automata have only non-lasso-shaped infinite runs. We propose an alternative *region-based SMT BMC encoding* for timed automata; in contrast to the propositional encoding, only the loop-detection part of the SMT encoding has to deal with regions but the rest of the encoding remains rather simple. We prove that (i) it can find a representative of an infinite run for any a timed automaton having an infinite run, and (ii) there is a completeness threshold, i.e., an integer such that there is an infinite run representative of at most that length unless there is no infinite run. Therefore, the encoding can be used to build an SMT-based BMC approach for model checking propositional LTL specifications on timed automata which is capable of finding counter-examples to all non-holding specifications and, in theory, also of proving that no counter-examples exist. Due to the use of the region abstraction, the formulas in the region-based BMC encoding are more complicated and may contain mixed integer / real expressions that are not supported by all current SMT solvers. We thus provide some alternative encodings and experimentally evaluate the efficiency of these.

We experimentally compare region-based SMT BMC against a traditional lasso-based SMT BMC encoding. We also compare our prototype implementation of SMT BMC methods against the highly optimized SAT BMC engine NuSMV [14] using a region-based propositional logic encoding [12]. The results show that region-based SMT BMC performs very good and is, in fact, sometimes significantly faster as it can find shorter counter-examples than the other methods tested.

2 Timed Automata

We first give basic definitions for timed automata (see e.g. [1–3]). For the sake of simplicity, we use the very basic timed automata defined below in the theoretical parts of the paper. However, in practice (and also in the experimental part of the paper) one usually defines a network of timed automata that can also have (shared and local) finite

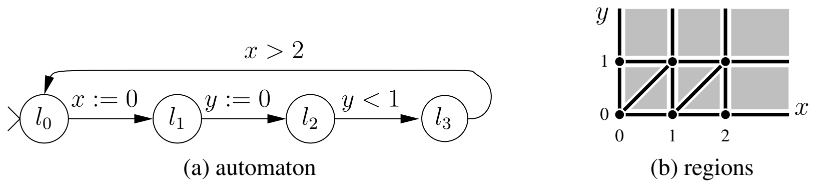


Fig. 1. A timed automaton and its regions

domain non-clock variables manipulated on the edges. The symbolic bounded model checking encodings presented later in the paper can be extended to handle both of these features, see, e.g., [9, 10] for how to handle synchronization in a network of timed automata. Similarly, we do not define any property description language in the theoretical part but consider the reachability problem for timed automata and the non-emptiness problem for timed automata extended with Büchi acceptance conditions (like in [1]). We then later study how bounded model checking can be used to solve these problems. Concerning practical model checking, solving the reachability problem corresponds to finding whether a timed automaton (or a network of such) can reach a bad state. Similarly, bearing in mind that linear-time temporal logic (LTL) properties can be translated into Büchi automata (see e.g. [15]), non-emptiness checking of timed Büchi automata corresponds to checking whether a timed automaton can violate an LTL specification. In the experimental part symbolic encodings for LTL model checking [16] are applied.

Let X be a set of real-valued *clock variables*. Then, a *clock valuation* v is a function $v : X \to \mathbb{R}_{\geq 0}$ and $v+\delta$ for a $\delta \in \mathbb{R}_{\geq 0}$ is the valuation for which $\forall x \in X : (v+\delta)(x) = v(x) + \delta$. The set of *clock constraints* over X, $\mathcal{C}(X)$, is defined by the grammar $C ::= \mathbf{true} \mid x \bowtie n \mid C \wedge C$ where $x \in X$, $\bowtie \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$. A valuation v satisfies a clock constraint C, denoted by $v \models C$, if it evaluates C to true.

A *timed automaton* (TA) is a tuple $\langle L, l_{\text{init}}, X, E, I\rangle$ where

- L is a finite set of *locations*,
- $l_{\text{init}} \in L$ is the *initial location* of the automaton,
- X is a finite set of real-valued *clock variables*,
- $E \subseteq L \times \mathcal{C}(X) \times 2^X \times L$ is a set of edges, each edge $\langle l, g, R, l'\rangle \in E$ specifying a *guard* g and a set R of *clocks to be reset*, and
- $I : L \to \mathcal{C}(X)$ assigns an *invariant* to each location.

As an example, Fig. 1(a) shows a timed automaton (from [1]). It has four locations $l_0, \ldots, l_3$, l_0 being the initial one, and two clocks, x and y. The edge $\langle l_0, \mathbf{true}, \{x\}, l_1\rangle$ from l_0 to l_1 resets the clock x and the edge $\langle l_2, x < 1, \emptyset, l_3\rangle$ has the guard $x < 1$. The invariants of all locations are $\mathbf{true}$.

A *state* of a timed automaton $\mathcal{A} = \langle L, l_{\text{init}}, X, E, I\rangle$ is a pair $\langle l, v\rangle$, where $l \in L$ is a location in $\mathcal{A}$ and v is a clock valuation over X. A state $\langle l, v\rangle$ is (i) *initial* if $l = l_{\text{init}}$ and $v(x) = 0$ for each $x \in X$, and (ii) *valid* if $v \models I(l)$. Let $\langle l, v\rangle$ and $\langle l', w\rangle$ be states of $\mathcal{A}$. There is a time *elapse step* from $\langle l, v\rangle$ to $\langle l', w\rangle$, denoted by $\langle l, v\rangle \overset{\text{e}}{\to} \langle l', w\rangle$, if (i) $l = l'$, (ii) $w = v + \delta$ for some $\delta \in \mathbb{R}_{>0}$, and (iii) $\langle l', w\rangle$ is a valid state. Intuitively, there is a time elapse step from a state to another if the second state can be reached from the first one by letting a certain amount of time pass. There is a *discrete step* from $\langle l, v\rangle$ to $\langle l', w\rangle$, denoted by $\langle l, v\rangle \xrightarrow{\text{d}} \langle l', w\rangle$, if there is an edge $\langle l, g, R, l'\rangle \in E$ such that (i) $v \models g$, (ii) $\langle l', w\rangle$ is a valid, and (iii) $w(x) = 0$ for all $x \in R$ and $w(x) = v(x)$ for all $x \in X \setminus R$. That is, discrete steps can be used to change the current location as long as the guard and the target location invariant are satisfied. A discrete step resets some clocks and leaves the other's values unchanged, i.e., a discrete step does not take any time. For situations in which the type of step between two states is insignificant, we define that $\langle l, v\rangle \rightarrow \langle l', w\rangle$ iff $\langle l, v\rangle \xrightarrow{\text{e}} \langle l', w\rangle$ or $\langle l, v\rangle \xrightarrow{\text{d}} \langle l', w\rangle$.

A *run* of $\mathcal{A}$ is a finite or infinite sequence $\pi = \langle l_0, v_0\rangle\langle l_1, v_1\rangle \ldots$, such that (i) $\langle l_0, v_0\rangle$ is a valid initial state, and (ii) $\langle l_i, v_i\rangle \rightarrow \langle l_{i+1}, v_{i+1}\rangle$ for each consecutive pair of states in the sequence. As an example, the automaton in Fig. 1(a) has the run $\langle l_0, (0, 0)\rangle\langle l_0, (0.7, 0.7)\rangle\langle l_1, (0, 0.7)\rangle\langle l_2, (0, 0)\rangle\langle l_3, (0, 0)\rangle$ where each clock valuation $\{x \mapsto v, y \mapsto w\}$ is abbreviated to (v, w). An infinite run is (i) *non-zeno* if the sum of time passed in time elapse steps in it is infinite, and (ii) *lasso-shaped* if it can be written as $\langle l_0, v_0\rangle \ldots \langle l_{i-1}, v_{i-1}\rangle\big(\langle l_i, v_i\rangle \ldots \langle l_k, v_k\rangle\big)^\omega$ for some $0 \leq i \leq k$. In the context of BMC we sometimes consider k the length of the lasso-shaped run, as it is the length needed to represent the run. The automaton in Fig.1(a) has a lasso-shaped non-zeno run $\langle l_0, (0, 0)\rangle\big(\langle l_0, (2.1, 2.1)\rangle\langle l_1, (0, 2.1)\rangle\langle l_2, (0, 0)\rangle\langle l_3, (0, 0)\rangle\langle l_3, (2.1, 2.1)\rangle\big)^\omega$. While it does not have any zeno runs, the automaton obtained by removing the guard $x > 2$ has the lasso-shaped zeno run $\big(\langle l_0, (0, 0)\rangle\langle l_1, (0, 0)\rangle\langle l_2, (0, 0)\rangle\langle l_3, (0, 0)\rangle\big)^\omega$.

2.1 Model Checking Problems

As said earlier, we study two model checking problems for timed automata. Firstly,

Definition 1 (Reachability Problem). *Given a timed automaton $\mathcal{A}$ and a location l, does $\mathcal{A}$ have a finite run $\langle l_0, v_0\rangle\langle l_1, v_1\rangle \ldots \langle l_k, v_k\rangle$ with $l_k = l$?*

Secondly, we define a *timed Büchi automaton* to be a tuple $\mathcal{B} = \langle L, l_{\text{init}}, X, E, I, F\rangle$ such that (i) $\langle L, l_{\text{init}}, X, E, I\rangle$ is a timed automaton, and (ii) $F \subseteq L$ is the set of *accepting locations*. States, steps, and runs are defined as for timed automata. A run of $\mathcal{B}$ is *accepting* if it is infinite and a location $l \in F$ occurs infinitely many times in it.

Definition 2 (Non-emptiness Problem). *Given a timed Büchi automaton $\mathcal{B}$, does it have an accepting run?*

For example, consider the timed Büchi automaton obtained from the timed automaton in Fig. 1(a) by letting $F = \{l_3\}$. It has a lasso-shaped, non-zeno accepting run $\langle l_0, (0, 0)\rangle\langle l_1, (0, 0)\rangle\big(\langle l_2, (0, 0)\rangle\langle l_3, (0, 0)\rangle\langle l_3, (3, 3)\rangle\langle l_0, (3, 3)\rangle\langle l_1, (0, 3)\rangle\big)^\omega$. Both the reachability and non-emptiness problems are PSPACE-complete [1].

2.2 The Region Abstraction

We will also need the classic concepts of regions and region automata [1] later in the paper. Assume a timed automaton $\mathcal{A} = \langle L, l_{\text{init}}, X, E, I\rangle$. For each clock $x \in X$, let m_x be the largest constant n occurring in any atom of form $x \bowtie n$ on the guards and invariants of the automaton. For each $v \in \mathbb{R}_{\geq 0}$, let $\lfloor v\rfloor \in \mathbb{N}$ be the integral and

$\mathrm{fract}(v) \in [0,1[$ the fractional part of v, i.e., $v = \lfloor v \rfloor + \mathrm{fract}(v)$. Two valuations, v and w, over X are *equivalent*, denoted by $v \sim w$, if all the following conditions hold:

1. For each clock $x \in X$, either $\lfloor v(x) \rfloor = \lfloor w(x) \rfloor$ or $(v(x) > \mathrm{m}_x) \wedge (w(x) > \mathrm{m}_x)$.
2. For all pairs of clocks $x, y \in X$ with $v(x) \leq \mathrm{m}_x$ and $v(y) \leq \mathrm{m}_y$, it holds that $\mathrm{fract}(v(x)) \leq \mathrm{fract}(v(y))$ iff $\mathrm{fract}(w(x)) \leq \mathrm{fract}(w(y))$.
3. For all clocks $x \in X$ with $v(x) \leq \mathrm{m}_x$ it holds $\mathrm{fract}(v(x)) = 0$ iff $\mathrm{fract}(w(x)) = 0$.

A *region* is an equivalence class of valuations induced by the relation $\sim$, and the region of a valuation v is denoted by $[v]$. The set of all regions is denoted by $\mathrm{regions}(\mathcal{A})$ and it contains *at most* $|X|! \cdot 2^{|X|} \cdot \prod_{x \in X}(2\mathrm{m}_x + 2)$ regions [1].

As an example, Fig. 1(b) graphically illustrates the regions of the timed automaton in Fig. 1(a); the 28 regions are the thick black dots and lines as well as the gray areas.

The *region automaton* of a timed automaton $\mathcal{A}$ is the finite state automaton

$$\mathcal{A}^{\mathrm{R}} = \langle Q, q_{\mathrm{init}}, \Delta \rangle,$$

where (i) $Q = L \times \mathrm{regions}(\mathcal{A})$ is the set of states, (ii) $q_{\mathrm{init}} = \langle l_{\mathrm{init}}, [v_0] \rangle$ with $v_0(x) = 0$ for all $x \in X$ is the initial state, and (iii) $\Delta \subseteq Q \times Q$ is the transition relation with $\langle \langle l, r \rangle, \langle l', r' \rangle \rangle \in \Delta$ iff $\exists v, v' : \langle l, v \rangle \rightarrow \langle l', v' \rangle \wedge [v] = r \wedge [v'] = r'$. A run of $\mathcal{A}^{\mathrm{R}}$ is a finite or infinite sequence $q_0 q_1 \ldots$ of states in Q such that (i) $q_0 = q_{\mathrm{init}}$, and (ii) $\langle q_i, q_{i+1} \rangle \in \Delta$ for all consecutive pairs of states in the sequence.

A timed automaton $\mathcal{A}$ and its region automaton $\mathcal{A}^{\mathrm{R}}$ are bisimilar in the sense that

1. $\langle l, v \rangle \rightarrow \langle l', v' \rangle$ implies $\langle \langle l, [v] \rangle, \langle l', [v'] \rangle \rangle \in \Delta$, and
2. $\langle (l, r), (l', r') \rangle \in \Delta$ implies $\forall v : ([v] = r) \Rightarrow \exists v' : ([v'] = r') \wedge \langle l, v \rangle \rightarrow \langle l', v' \rangle$.

Thus $\mathcal{A}$ and $\mathcal{A}^{\mathrm{R}}$ also have corresponding runs: (i) if $\mathcal{A}$ has a run $\langle l_0, v_0 \rangle \langle l_1, v_1 \rangle \ldots$, then $\mathcal{A}^{\mathrm{R}}$ has a run $\langle l_0, [v_0] \rangle \langle l_1, [v_1] \rangle \ldots$, and (ii) if $\mathcal{A}^{\mathrm{R}}$ has a run $\langle l_0, r_0 \rangle \langle l_1, r_1 \rangle \ldots$, then $\mathcal{A}$ has a run $\langle l_0, v_0 \rangle \langle l_1, v_1 \rangle \ldots$ such that $[v_i] = r_i$ for each i. Note that some runs of $\mathcal{A}^{\mathrm{R}}$ may have both corresponding zeno runs and corresponding non-zeno runs in $\mathcal{A}$. We define that a *run in* $\mathcal{A}^{\mathrm{R}}$ *is non-zeno* if it has at least one corresponding non-zeno run in $\mathcal{A}$.

3 Bounded Model Checking for Reachability and Lassos

As explained in the introduction, the idea behind bounded model checking is to construct formulas whose satisfying interpretations correspond to runs having some desired property (e.g., reachability, Büchi acceptance) and *bounded length*. The bound is incremented until a satisfiable formula (and thus a run with the desired property) is found or a completeness threshold is reached (meaning that no such run exists). This section introduces BMC for finite runs and lasso-shaped infinite runs of timed (Büchi) automata. The lasso-based BMC for TAs is very similar to BMC for untimed systems and has been previously described in [9, 10]. Lasso-based BMC is complete for untimed finite state systems but, as will be shown, despite a previous claim not complete for TAs.

Let $\mathcal{A} = \langle L, l_{\mathrm{init}}, X, E, I \rangle$ be a timed automaton (or a timed Büchi automaton $\langle L, l_{\mathrm{init}}, X, E, I, F \rangle$) and let k be the "bound" i.e. the length of the runs currently considered. We first construct a quantifier-free first order formula $|[\mathcal{A}, k]|^{\mathrm{runs}}$ using linear arithmetics over reals whose satisfying interpretations represent $\mathcal{A}$'s runs of length k. For each clock $x \in X$, we introduce $k+1$ "timed copies" $x^{[0]}, x^{[1]}, \ldots, x^{[k]}$ where the variable $x^{[i]}$ gives the value of the clock x at the ith state in the run. If C is a clock constraint, then $C^{[i]}$ is the "timed version" of C obtained by substituting each clock $x \in X$ in it by $x^{[i]}$; e.g. $((x < 3) \wedge (y \geq 2))^{[4]} = (x^{[4]} < 3) \wedge (y^{[4]} \geq 2)$. To represent automaton locations in the run, we use the set $\left\{ at^{[0]}, at^{[1]}, \ldots, at^{[k]} \right\}$ of variables over the domain L. Similarly, to select whether a time elapse or discrete step is taken at the ith step, we use the Boolean variables $elapse^{[0]}, \ldots, elapse^{[k-1]}$, and for the time taken in time elapse steps the real-valued variables $\delta^{[0]}, \ldots, \delta^{[k-1]}$. We now define the formula for runs of length k by

$$|[\mathcal{A}, k]|^{\text{runs}} := |[\mathcal{A}]|^{\text{init}} \wedge |[\mathcal{A}, k]|^{\text{inv}} \wedge |[\mathcal{A}, k]|^{\text{trans}}$$

where $|[\mathcal{A}]|^{\text{init}} := (at^{[0]} = l_{\text{init}}) \wedge \bigwedge_{x \in X}(x^{[0]} = 0)$ ensures that the values of $\left\{ at^{[0]} \right\} \cup \left\{ x^{[0]} \mid x \in X \right\}$ represent the initial state of $\mathcal{A}$, $|[\mathcal{A}, k]|^{\text{inv}} := \bigwedge_{0 \leq i \leq k} \bigwedge_{l \in L}(at^{[i]} = l) \Rightarrow I(l)^{[i]}$ forces all the $k+1$ states to be valid ones, and the formula $|[\mathcal{A}, k]|^{\text{trans}} := \bigwedge_{0 \leq i < k}(elapse^{[i]} \Rightarrow \phi^{[i]}) \wedge (\neg elapse^{[i]} \Rightarrow \psi^{[i]})$ captures the transition relation. The formula $\phi^{[i]} := (\delta^{[i]} > 0) \wedge (at^{[i+1]} = at^{[i]}) \wedge \bigwedge_{x \in X}(x^{[i+1]} = x^{[i]} + \delta^{[i]})$ encodes time elapse steps, while $\psi^{[i]}$ does the same for discrete steps:

$$\psi^{[i]} := (\delta^{[i]} = 0) \wedge \bigvee_{\langle l, g, R, l' \rangle \in E} \Big((at^{[i]} = l) \wedge (at^{[i+1]} = l') \wedge g^{[i]} \wedge \\ \bigwedge_{x \in R}(x^{[i+1]} = 0) \wedge \bigwedge_{x \in X \setminus R}(x^{[i+1]} = x^{[i]}) \Big)$$

The automaton $\mathcal{A}$ has a run $\langle l_0, v_0 \rangle \ldots \langle l_k, v_k \rangle$ iff the formula $|[\mathcal{A}, k]|^{\text{runs}}$ is satisfiable under any interpretation extending $\left\{ at^{[i]} \mapsto l_i, x^{[i]} \mapsto v_i(x) \mid 0 \leq i \leq k, x \in X \right\}$.

BMC for Reachability. Based on the tight correspondence between the runs of $\mathcal{A}$ and satisfying interpretations of $|[\mathcal{A}, k]|^{\text{runs}}$, one can use $|[\mathcal{A}, k]|^{\text{runs}}$ to solve the reachability problem. Given a timed automaton $\mathcal{A}$ and a location l in it, check whether the formula

$$|[\mathcal{A}, l, k]|^{\text{reach}} := |[\mathcal{A}, k]|^{\text{runs}} \wedge \bigvee_{0 \leq i \leq k} (at^{[i]} = l)$$

is satisfiable for some bound $k \in \{0, 1, \ldots\}$. If $|[\mathcal{A}, l, k]|^{\text{reach}}$ indeed is satisfiable, then one can construct a run of $\mathcal{A}$ ending in l from the satisfying interpretation for $|[\mathcal{A}, l, k]|^{\text{reach}}$. We return to the issue of completeness, i.e. detecting the case that l is not reachable, later in this section.

BMC for Lasso-Shaped Infinite Runs. Assume now that $\mathcal{B}$ is a timed Büchi automaton $\langle L, l_{\text{init}}, X, E, I, F \rangle$. We can use the formula $|[\mathcal{B}, k]|^{\text{runs}}$ to define a formula $|[\mathcal{B}, k]|^{\text{lasso}}$ such that the timed Büchi automaton $\mathcal{B}$ has an accepting infinite lasso-shaped run $\langle l_0, v_0 \rangle \ldots \langle l_{i-1}, v_{i-1} \rangle \big(\langle l_i, v_i \rangle \ldots \langle l_k, v_k \rangle \big)^{\omega}$ for some $1 \leq i < k$ iff $|[\mathcal{B}, k]|^{\text{lasso}}$ is satisfiable under an interpretation extending $\left\{ at^{[i]} \mapsto l_i, x^{[i]} \mapsto v_i(x) \mid 0 \leq i \leq k, x \in X \right\}$. To do this, we use an auxiliary set $loop^{[1]}, \ldots, loop^{[k]}$ of Boolean *loop variables* to detect loops in the finite runs represented with $|[\mathcal{B}, k]|^{\text{runs}}$. A variable $loop^{[i]}$ being true means that the $i-1$th and the kth state are the same, meaning that a lasso-shaped run can be obtained by looping back from the kth to the ith state. Furthermore, an auxiliary set $acc^{[1]}, \ldots, acc^{[k]}$ of Boolean variables is used to compute whether an accepting location is visited at the ith or later state in the run. We define

$$|[\mathcal{B}, k]|^{\text{lasso}} := |[\mathcal{B}, k]|^{\text{runs}} \wedge |[\mathcal{B}, k]|^{\text{loop}} \wedge |[\mathcal{B}, k]|^{\text{accept}}$$

where $|[\mathcal{B}, k]|^{\text{loop}} := \bigwedge_{1 \le i \le k} \left(loop^{[i]} \Rightarrow (at^{[i-1]} = at^{[k]}) \wedge \bigwedge_{x \in X} (x^{[i-1]} = x^{[k]}) \right)$ detects the loops in the finite runs, and $|[\mathcal{B}, k]|^{\text{accept}} := (acc^{[k]} \Leftrightarrow \bigvee_{l \in F}(at^{[k]} = l)) \wedge \bigwedge_{1 \le i \le k-1} \left(acc^{[i]} \Leftrightarrow acc^{[i+1]} \vee \bigvee_{l \in F}(at^{[i]} = l) \right) \wedge \left(\bigvee_{1 \le i \le k} (loop^{[i]} \wedge acc^{[i]}) \right)$ forces satisfying interpretations to correspond to accepting runs only: there shall be a loop in the run and an accepting location in the loop.

The encoding can easily be modified to accept only non-zeno runs. A lasso-shaped run is zeno iff it does not contain any time elapse step in its looping part. Thus, non-zenoness can be enforced by requiring the looping part to contain at least one time elapse step. For this purpose k Boolean variables $el^{[0]}, \ldots, el^{[k-1]}$ where $el^{[i]}$ being true for a given i means that there is a time elapse step after the ith state. Looping back to the ith state is allowed only if $el^{[i]}$ is true, leading to the following conjunct:

$$|[\mathcal{B}, k]|^{\text{lnz}} := (el^{[k-1]} \Leftrightarrow elapse^{[k-1]}) \wedge \bigwedge_{1 \le i \le k-2} \left(el^{[i]} \Leftrightarrow (elapse^{[i]} \vee el^{[i+1]}) \right) \wedge \bigwedge_{1 \le i \le k} loop^{[i]} \Rightarrow el^{[i-1]}$$

Note that for a run that loops back from the last to the ith state, the step that loops back from the last to the ith state is of the same type as the step from the $i-1$th to the ith state. Thus, it is sufficient if the step from the $i-1$th to the ith state is a time elapse step in order to have a time elapse step in the looping part of the run, which is the reason why we only require $el^{[i-1]}$ to hold if $loop^{[i]}$ holds and not $el^{[i]}$.

(In)completeness. We now study the completeness of the two BMC encodings given above. As in [7, 9], by completeness we mean the ability to find a run if one exists or to demonstrate that no runs exists if this is the case. To show completeness, a completeness threshold is needed, i.e. an integer bound K such that a run of interest (witnessing reachability or Büchi acceptance) exists if and only if one exists with bound K or less.

Previous Completeness Results. For finite state systems, the simple run-unfolding BMC is complete for reachability problems and lasso-BMC is complete for non-emptiness under Büchi acceptance conditions [7]. When considering timed automata, the reachability BMC encoding given above is complete [9]. This is because (i) a location l is reachable in an automaton $\mathcal{A}$ iff it is reachable in its region automaton $\mathcal{A}^{\text{R}}$ due to bisimilarity (recall Sect. 2.2), (ii) $\mathcal{A}^{\text{R}}$ has at most $|L| \cdot |X|! \cdot 2^{|X|} \cdot \prod_{x \in X}(2\text{m}_x + 2)$ states, which implies that l is reachable with at most $K_{\text{reach}} = |L| \cdot |X|! \cdot 2^{|X|} \cdot \prod_{x \in X}(2\text{m}_x + 2) - 1$ steps in $\mathcal{A}^{\text{R}}$, and (iii) thus l is reachable with at most K_{reach} steps in $\mathcal{A}$. Therefore, l is reachable in $\mathcal{A}$ iff $|[\mathcal{A}, l, K_{\text{reach}}]|^{\text{reach}}$ is satisfiable. Of course, using K_{reach} as a bound is usually infeasible in practice but its existence guarantees the theoretical completeness of the BMC approach.

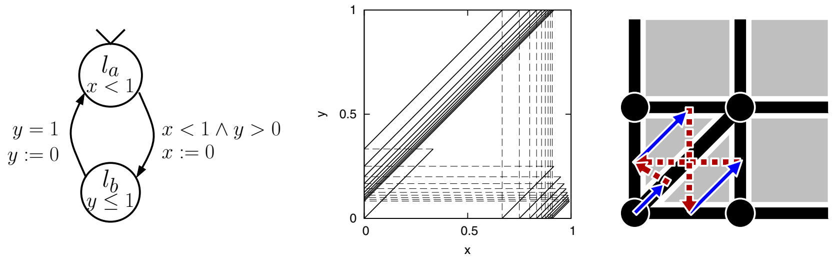
Incompleteness for Büchi Acceptance Conditions on TAs. We now show that, despite a previous claim in [9], lasso-based BMC is not complete for checking non-emptiness of timed Büchi automata or, in fact, for even detecting whether a timed automaton has at least one infinite run. Incompleteness of an encoding can best be shown by giving an automaton for which the encoding can not find a run. For this purpose, we will use automaton in Fig. 2(a) which, as will be demonstrated, does not have any lasso-shaped infinite runs despite having infinite non-zeno runs. Therefore, lasso-based BMC will fail to find any run for the automaton and *is thus incomplete* for (i) detecting whether a timed automaton has at least one infinite run, (ii) deciding non-emptiness of timed Büchi automata, and (iii) model checking propositional LTL on timed automata.

Let us now study the automaton in Fig. 2(a) a bit more closely. It has two locations, l_a and l_b, and two clocks, x and y. For a given infinite run in the automaton, let t_i^{a} be the time spent in l_a the ith time the run visits l_a and t_i^{b} be defined analogously for l_b. The edge from l_a can only be traversed when x is less than one. Furthermore, x is reset when the edge is traversed. Therefore, the time between two subsequent traversals of this edge is strictly less than one time unit: $\forall i \geq 1 : t_i^{\mathrm{b}} + t_{i+1}^{\mathrm{a}} < 1$. Analogously, the time between two subsequent traversals of the edge form l_b to l_a is exactly one time unit: $\forall i \geq 1 : t_i^{\mathrm{a}} + t_i^{\mathrm{b}} = 1$. Combining the two formulas results in $\forall i \geq 1 : t_{i+1}^{\mathrm{a}} < t_i^{\mathrm{a}}$ and $\forall i \geq 1 : t_{i+1}^{\mathrm{b}} > t_i^{\mathrm{b}}$, i.e. in any run the time spent in l_a strictly decreases from any visit to the next and the time spent in l_b strictly increases. Furthermore, the difference between the two clocks in l_b equals the time spent in l_a on the previous visit and vice versa. Consequently, the difference between the clocks strictly increases in l_a and strictly decreases in l_b. Thus, the same location is never reached twice with the same clock difference and therefore, no run can ever visit the same state twice. Hence, the automaton does not have any lasso-shaped run. This, however, does not mean that the automaton does not have any infinite runs at all. A valid infinite run, e.g., stays $\frac{1}{i+2}$ time units in location l_a the ith time it is visited and $1 - \frac{1}{i+2}$ units in location l_b. Figure 2(b) shows the clock valuations on a ten time unit long prefix of this run, while Fig. 2(c) illustrates the clock regions visited by the run. Note that while the run is not lasso-shaped in the space of clock valuations, it indeed is lasso-shaped in the clock regions. Also, the run is non-zeno as the time passing is $\sum_{i=1}^{\infty} \frac{1}{i+2} + 1 - \frac{1}{i+2} = \infty$.

4 Region-Based BMC

As shown above, lasso-based BMC is not complete for checking non-emptiness of timed Büchi automata. This section introduces a region-based BMC approach that fixes this problem. The approach is inspired by the observation that even though the automaton in Fig. 2(a) does not have any lasso-shaped infinite runs, its region automaton does. Based on this observation, our new encoding modifies the lasso-based BMC encoding by not requiring the last state of the run to be exactly the same as a previous state but only in the same region as a previous state. Such a run corresponds to a lasso-shaped run in the region automaton and thus to a set of infinite runs of the TA.

Note that, in order to get runs in which time does not suddenly just stop, it is not sufficient to require that the last state of the run is in the same region as an earlier state. For many clock valuations, it is possible to reach a valuation in the same region by a



(a) The automaton (b) The clock valuations of a prefix of a run (c) The same run in the region abstraction

Fig. 2. A non-empty timed automaton that does not have any lasso-shaped run

sufficiently small time elapse step. Thus, it is often possible to extend a finite run with a short time elapse step to get a run in which the last and second to last state have the same location and clock valuations in the same region. If the only requirement to a run was that the last state is in the same clock region as a previous state, such a run would be accepted. While it is possible to extend such a run to a valid infinite (though zeno) run by adding smaller and smaller time elapse steps, in practice runs of the described type are typically not of interest as they correspond to time not progressing past a certain point. Therefore, we exclude runs of the described type by restricting to non-zeno runs.

In order to check whether two clock valuations are in the same clock region, one needs to split up each clock's value into its integral an fractional parts. Therefore, region-based BMC uses two additional variables, $x_{\text{int}}^{[i]}$ and $x_{\text{fract}}^{[i]}$, for each clock $x \in X$ and each state index i. The integer variable $x_{\text{int}}^{[i]}$ represents the integral part of the value of x while the real-valued variable $x_{\text{fract}}^{[i]}$ represents its fractional part. Given a timed Büchi automaton $\mathcal{B} = \langle L, l_{\text{init}}, X, E, I, F \rangle$, the region-based BMC encoding is

$$|[\mathcal{B}, k]|^{\text{region}} := |[\mathcal{B}, k]|^{\text{runs}} \wedge |[\mathcal{B}, k]|^{\text{accept}} \wedge |[\mathcal{B}, k]|^{\text{close}} \wedge |[\mathcal{B}, k]|^{\text{nz}}$$

where $|[\mathcal{B}, k]|^{\text{runs}}$ and $|[\mathcal{B}, k]|^{\text{accept}}$ are defined as in Sect. 3, $|[\mathcal{B}, k]|^{\text{nz}}$ is used to ensure non-zenoness and is defined later in this section and $|[\mathcal{B}, k]|^{\text{close}}$, detecting whether the clock valuations in the $i-1$th and kth states are in the same region, is defined as

$$|[\mathcal{B}, k]|^{\text{close}} := \bigwedge_{0 \leq i \leq k} \bigwedge_{x \in X} \left(0 \leq x_{\text{fract}}^{[i]} \wedge x_{\text{fract}}^{[i]} < 1 \wedge x^{[i]} = x_{\text{int}}^{[i]} + x_{\text{fract}}^{[i]}\right) \wedge$$
$$\bigwedge_{1 \leq i \leq k} \left(\mathit{loop}^{[i]} \Rightarrow (\mathit{at}^{[i-1]} = \mathit{at}^{[k]} \wedge S_{i,k})\right)$$

with $S_{i,k} := \bigwedge_{x \in X} \Big((x_{\text{int}}^{[i-1]} = x_{\text{int}}^{[k]}) \vee (x_{\text{int}}^{[i-1]} > \mathrm{m}_x \wedge x_{\text{int}}^{[k]} > \mathrm{m}_x)\Big) \wedge \Big(x_{\text{int}}^{[k]} \leq \mathrm{m}_x \Rightarrow \big((x_{\text{fract}}^{[i-1]}{=}0 \Leftrightarrow x_{\text{fract}}^{[k]}{=}0) \wedge \bigwedge_{y \in X \setminus \{x\}} (y_{\text{int}}^{[k]} {\leq} \mathrm{m}_y \Rightarrow (x_{\text{fract}}^{[i-1]} {\leq} y_{\text{fract}}^{[i-1]} \Leftrightarrow x_{\text{fract}}^{[k]} {\leq} y_{\text{fract}}^{[k]}))\big)\Big)$.

The first line in $|[\mathcal{B}, k]|^{\text{close}}$ ensures the integral+fractional decomposition of clock values. Its sub-expression $x^{[i]} = x_{\text{int}}^{[i]} + x_{\text{fract}}^{[i]}$ mixes integer and real variables; such mixed-type expressions are supported, e.g., by the SMT solver Yices [17]. As they are, however, not supported by all SMT solvers, an alternative encoding not requiring them will be introduced in Sect. 5. Analogously to the lasso-based encoding, $\mathit{loop}^{[i]}$ is a Boolean variable indicating that it is possible to loop from the last state in the run to the ith state, or, more precisely, to a state in the same region as the ith state.

4.1 Ensuring Non-zenoness

In order to complete the encoding, a way to ensure non-zenoness of the run is needed. In lasso-based BMC, non-zenoness can be ensured by requiring that the looping part of the run contains at least one time elapse step. For region-based BMC, this approach does not work. Any concrete run corresponding to a lasso-shaped region automaton run having a time elapse step in the looping part is guaranteed to have an infinite number of time elapse steps. The sum of the delays of these steps is, however, not guaranteed to be diverging. Thus, we will instead ensure non-zenoness using the following theorem:

Theorem 1 (Alur and Dill [1]). *An infinite run of a region automaton that has an infinite number of time elapse steps is non-zeno iff each clock $x \in X$ either infinitely often is zero or infinitely often has a value greater than m_x.*[1]

It is straightforward to turn this theorem into a non-zenoness condition for the BMC encoding. Due to the fact that x's value cannot decrease unless x is reset, x is guaranteed to exceed m_x in all states of the looping part if it exceeds m_x in at least one state of the looping part and is never reset inside the looping part. Therefore it is sufficient to require that x is either zero at least once inside the looping part or exceeds its maximum value in the run's last state which is guaranteed to be in its looping part.

Requiring at least one time elapse step in the looping part of the run can be done using the formula $|[\mathcal{B},k]|^{\mathrm{lnz}}$ as for lasso-shaped paths. Furthermore, for any clock $x \in X$, k Boolean variables $ok_x^{[i]}$ are used to ensure that either x is zero at least once in the looping part of the run or exceeds its maximum value in the last state of the run. This results in the following definition of $|[\mathcal{B},k]|^{\mathrm{nz}}$:

$$|[\mathcal{B},k]|^{\mathrm{nz}} := \bigwedge_{x\in X}\Big(ok_x^{[k]} \Leftrightarrow (x^{[k]}{=}0 \vee x^{[k]}{>}\mathrm{m}_x)\Big) \wedge \bigwedge_{1\le i\le k-1}\Big(ok_x^{[i]} \Leftrightarrow (x^{[i]}{=}0 \vee ok_x^{[i+1]})\Big) \wedge$$
$$\bigwedge_{1\le i\le k}\Big(\mathit{loop}^{[i]} \Rightarrow \bigwedge_{x\in X} ok_x^{[i]}\Big) \wedge |[\mathcal{B},k]|^{\mathrm{lnz}}$$

By Theorem 1, any run with infinitely many time elapse steps in the region automaton that does not satisfy the "infinitely often $x = 0 \vee x > \mathrm{m}_x$" condition for a clock $x \in X$ is zeno. Therefore, the bound required to find a run using our encoding is the length of the shortest lasso-shaped non-zeno run in the region automaton.

4.2 Completeness

In the following the completeness of the proposed approach will be shown. More precisely, it will be shown that, given a timed Büchi automaton $\mathcal{B} = \langle L, l_{\text{init}}, X, E, I, F\rangle$,

[1] Note that using the slightly different definition of the region automaton in [1] any infinite run of the region automaton is guaranteed to have an infinite number of elapse steps.

$K_{\text{region}} := (|X| + 3) \cdot |L| \cdot |X|! \cdot 2^{|X|} \cdot \prod_{x \in X}(2\mathrm{m}_x + 2)$ is a completeness threshold, meaning that our approach when used with bound K_{region} can find a non-zeno run for $\mathcal{B}$ if $\mathcal{B}$ has any non-zeno run. To do this, we prove the following theorem.

Theorem 2. *Unless a given timed Büchi automaton $\mathcal{B}$ does not have a single accepting non-zeno run, its region automaton $\mathcal{B}^{\mathrm{R}}$ has an accepting lasso-shaped non-zeno run of length at most K_{region}.*

The theorem will be proven in two parts, each of which is stated as a lemma. The proofs of the lemmas are given in the appendix.

Lemma 1. *If $\mathcal{B}$ has an infinite accepting non-zeno run $\pi = \langle l_0, v_0 \rangle \langle l_1, v_1 \rangle \ldots$, then $\mathcal{B}$'s region automaton $\mathcal{B}^{\mathrm{R}}$ has a* lasso-shaped *accepting non-zeno run $\pi^{\mathrm{R}}_{\text{lasso}}$.*

Lemma 2. *If $\mathcal{B}^{\mathrm{R}}$ has an accepting lasso-shaped non-zeno run, then $\mathcal{B}^{\mathrm{R}}$ has an accepting lasso-shaped non-zeno run of length at most K_{region}.*

Note that the completeness threshold K_{region} should first and foremost be considered a theoretical result, as the given completeness threshold in practice even for small systems is infeasibly high. In order to find more practical completeness thresholds, an approach similar to the ones used for untimed systems in [18] could be used.

5 Alternative Encodings

Avoiding Mixed-Type Expressions and Unbounded Integer Variables. One challenge in the encoding introduced in Sect. 4 is that it uses mixed-type expressions that use both integer and real variables; such are not supported by some SMT solvers. One can, however, modify the encoding to get rid of the mixed-type expressions. To do this, instead of the $x^{[i]}$ clock variables, we use $x^{[i]}_{\text{int}}$ variables for their integral parts and $x^{[i]}_{\text{fract}}$ variables for their fractional parts. Likewise, the difference variables for time elapse steps $\delta^{[i]}$ are each replaced by two variables $\delta^{[i]}_{\text{int}}$ and $\delta^{[i]}_{\text{fract}}$ for their integral and fractional parts. After this, (in)equalities of form $x^{[i]} \bowtie n$ for an $n \in \mathbb{N}$, used e.g. to encode the TA's guards and invariants and to enforce that the initial values of the clocks are zero, are modified to use $x^{[i]}_{\text{int}}$ and $x^{[i]}_{\text{fract}}$ instead of $x^{[i]}$. For instance, $x^{[i]} \leq n$ is replaced with $x^{[i]}_{\text{int}} < n \vee (x^{[i]}_{\text{int}} = n \wedge x^{[i]}_{\text{fract}} = 0)$. Expressions of the form $x^{[i+1]} = x^{[i]} + \delta^{[i]}$ have to be replaced by a case distinction to ensure that the fractional part of $x^{[i+1]}$ is always less than one: each expression of form $x^{[i+1]} = x^{[i]} + \delta^{[i]}$ is replaced with $\big((x^{[i]}_{\text{fract}} + \delta^{[i]}_{\text{fract}} < 1) \Rightarrow (x^{[i+1]}_{\text{int}} = x^{[i]}_{\text{int}} + \delta^{[i]}_{\text{int}} \wedge x^{[i+1]}_{\text{fract}} = x^{[i]}_{\text{fract}} + \delta^{[i]}_{\text{fract}})\big) \wedge \big((x^{[i]}_{\text{fract}} + \delta^{[i]}_{\text{fract}} \geq 1) \Rightarrow (x^{[i+1]}_{\text{int}} = x^{[i]}_{\text{int}} + \delta^{[i]}_{\text{int}} + 1 \wedge x^{[i+1]}_{\text{fract}} = x^{[i]}_{\text{fract}} + \delta^{[i]}_{\text{fract}} - 1)\big)$. When all expressions using the $x^{[i]}$ clock variables are modified as described, the encoding does not contain any mixed-type expressions anymore and can thus be used with a SMT solver not supporting them.

As the exact value of a clock $x \in X$ is irrelevant once it exceeds m_x, the encoding can be further modified to turn the $x^{[i]}_{\text{int}}$ and $\delta^{[i]}_{\text{int}}$ variables into bounded integer variables without affecting the correctness of the approach: the domain of each $x^{[i]}_{\text{int}}$

is $\{0,\ldots,\mathrm{m}_x+1\}$ while $\delta_{\text{int}}^{[i]}$ has the domain $\{0,\ldots,\max_{x\in X}\mathrm{m}_x\}$. A simple case distinction similar to the technique shown for $x^{[i]}+\delta^{[i]}$ expressions is used to set each variable's value to its maximum value whenever it in the unbounded case would exceed its maximum. Restricting to bounded integers is necessary for SMT solvers that do not support unbounded integer variables.

Alternative Non-zenoness Condition. In [19] an alternative way to ensure that all runs of a timed Büchi automaton are non-zeno is proposed. The approach modifies the TA using one additional clock such that accepting states only count as accepting if at least one time unit passed since the last accepting state. While using this approach for BMC would be feasible for Büchi acceptance conditions, it is not clear how it could be extended to other approaches like directly encoding an LTL formula (cf. e.g. [16]) as in such an encoding there is no notion of accepting states. Thus we propose a similar but slightly different approach as an alternative encoding ensuring non-zenoness by requiring that the looping part of the run is at least one time unit long. The resulting non-zenoness condition is simpler and needs less variables than the original one proposed in Sect. 4:

$$|[\mathcal{B},k]|^{\text{nz2}} := (\Sigma^{[k-1]}{=}\delta^{[k-1]})\wedge \bigwedge_{0\le i\le k-2}(\Sigma^{[i]}{=}\Sigma^{[i+1]}{+}\delta^{[i]})\wedge \bigwedge_{1\le i\le k}(\mathit{loop}^{[i]}\Rightarrow \Sigma^{[i]}{\ge}1)$$

Correctness. A lasso-shaped run of the region automaton returns to a previously visited region at the end of the looping part. If the looping part is at least one time unit long, as required by our alternative non-zenoness-encoding, the integral part of the value of any clock $x\in X$ changes somewhere in the looping part. Unless x has exceeded m_x, this implies that a new region is reached. In this case, x has to be reset before the looping part can return to the original region. Hence, each clock x is either reset or exceeds m_x in the looping part, i.e. infinitely often, which according to Theorem 1 implies non-zenoness. Therefore any lasso-shaped run of the region automaton along whose looping part at least one time unit passes is non-zeno and the alternative encoding is correct.

Note that the opposite is not true, i.e. not every lasso-shaped non-zeno run of the region automaton has a looping part along which at least one time unit passes. This implies that using the alternative non-zenoness-encoding sometimes results in needing a higher bound to find a run than would be required with the original encoding.

6 Experiments

The Fischer mutual exclusion protocol and an industrial model with both handmade and random properties were used to compare the different SMT BMC encodings and discretization-based SAT BMC experimentally.

Setup. In the experiments, lasso-based BMC (not requiring non-zenoness, cf. Sect. 3) was compared to region-based BMC. In addition to the basic encoding given in Sect. 4, the two non-mixed type encodings and the alternative non-zenoness condition given in Sect. 5 were used. The basic encoding is referred to as "mixed type, basic nz." while the modifications are referred to as "non-mixed" for the unlimited range integer non-mixed

Table 1. Execution times and maximum bound reached for the Fischer protocol (median over 11 executions, "*to*" means timeout)

	$\neg\mathbf{GF}\mathit{proc1.crit}$										$\neg(\mathbf{GF}\mathit{proc1.crit} \wedge \mathbf{GF}\neg\mathit{proc1.crit})$									
	lasso BMC		mixed type		non-mixed		limited integer		discre-tization		lasso BMC		mixed type		non-mixed		limited integer		discre-tization	
# processes	Time	Bound	Time	Bound	Time	Bound	Time	Bound	Time	Bound	Time	Bound	Time	Bound	Time	Bound	Time	Bound	Time	Bound
2	0.35	9	0.28	**5**	0.2	**5**	0.28	**5**	**0.11**	7	0.33	9	0.31	**6**	0.25	**6**	0.3	**6**	**0.16**	10
3	1.73	12	0.26	**5**	0.28	**5**	0.28	**5**	**0.15**	7	1.41	12	**0.33**	**6**	**0.33**	**6**	0.34	**6**	4.07	13
4	16.94	15	0.35	**5**	0.33	**5**	0.34	**5**	**0.22**	7	14.54	15	**0.41**	**6**	0.43	**6**	0.42	**6**	*to*	14
5	83.46	18	0.5	**5**	0.42	**5**	0.43	**5**	**0.31**	7	66.53	18	0.54	**6**	**0.51**	**6**	0.53	**6**	*to*	13
7	*to*	20	0.99	**5**	0.57	**5**	0.6	**5**	**0.51**	7	*to*	20	1.39	**6**	**0.7**	**6**	0.74	**6**	*to*	13
10	*to*	19	1.53	**5**	**0.9**	**5**	0.97	**5**	0.95	7	*to*	18	*to*	5	**1.13**	**6**	1.22	**6**	*to*	13
20	*to*	15	*to*	4	**2.45**	**5**	2.59	**5**	3.57	7	*to*	18	*to*	5	**3.46**	**6**	3.86	**6**	*to*	12
30	*to*	14	*to*	4	**5.23**	**5**	5.45	**5**	10.04	7	*to*	18	*to*	5	**8.12**	**6**	8.98	**6**	*to*	11

type encoding, "limited integer" for the limited range integer encoding and "alt. nz." for the alternative non-zenoness condition (cf. Sect. 5).

For comparison to the SMT-based BMC variants, discretization-based SAT BMC was applied. That is, the real-time models were transformed into discrete time models [12] and then checked using a discrete time BMC SAT encoding [16] implemented in NuSMV [14] 2.5.4. The used translation algorithm encodes the region abstraction and thus is complete in the same sense as region-based SMT BMC. All BMC approaches were used in an incremental fashion, i.e. successively increasing the bound using an incremental SMT / SAT solver. The real-time models were encoded as symbolic timed transition systems [12], a symbolic representation variant of timed automata. Properties were specified as LTL formulas and encoded using [16].

In our experiments, we focused on comparing different BMC approaches. It should, however, be noted that both benchmarks have been previously studied using different verification methods including the model checker Uppaal [4].

All experiments were conducted on GNU/Linux computers with AMD Opteron 2435 CPUs limited to ten minutes of CPU time and 4 GB of RAM. For the SMT-based BMC variants, Yices [17] 1.0.33 was used.

Fischer Protocol. As a first benchmark, the Fischer mutual exclusion protocol, a standard benchmark for real-time verification, and two non-holding properties ("process one can only finitely often be in the critical section" and "process one can only finitely often enter and exit the critical section") were used. The protocol has been previously studied using Uppaal [20]. Table 1 shows the time needed for finding counter-examples and the maximum bound reached on instances of different sizes. For space restrictions, only results for the basic non-zenoness condition are listed. However, results for the alternative non-zenoness encoding are very similar.

For the first property the discretization-based approach scaled only slightly worse than the non mixed-type region-based encodings. Lasso-based BMC, in contrast, scaled significantly worse, already timing out at size 7. The reason is that the region based BMC variants can find significantly shorter counter-examples. Similarly, the discretization-based method may return significantly longer counter-examples, as it only allows

Table 2. Execution times in seconds for finding counter-examples to the non-holding properties on the industrial benchmark (median over 11 executions, "*to*" means timeout, "nz." abbreviates "non-zenoness condition")

	Entire model						Medium size submodel						Small submodel					
Property	lasso BMC	mixed basic nz.	mixed alt. nz.	non-mixed basic nz.	non-mixed alt. nz.	discreti-zation	lasso BMC	mixed basic nz.	mixed alt. nz.	non-mixed basic nz.	non-mixed alt. nz.	discreti-zation	lasso BMC	mixed basic nz.	mixed alt. nz.	non-mixed basic nz.	non-mixed alt. nz.	discreti-zation
1	**1.0**	43.55	*to*	3.97	*to*	*to*	**0.55**	0.85	*to*	0.73	*to*	*to*	**0.29**	0.69	*to*	0.51	*to*	*to*
2	**2.19**	*to*	*to*	18.11	20.4	*to*	**0.4**	0.43	0.56	0.41	0.47	*to*	**0.21**	0.3	0.37	0.27	0.27	*to*
3	**0.66**	2.12	16.81	2.04	1.78	*to*	**0.32**	0.49	0.81	0.42	0.5	*to*	**0.21**	1.0	0.36	0.28	0.29	*to*
4	**2.16**	*to*	*to*	21.1	19.27	*to*	**0.32**	0.47	0.59	0.45	0.52	*to*	**0.21**	0.45	0.38	0.3	0.37	*to*
5	**1.87**	*to*	*to*	19.74	22.14	*to*	**0.32**	0.63	0.65	0.47	0.54	*to*						
6	**1.8**	*to*	*to*	20.36	30.36	*to*	**0.33**	0.58	0.59	0.44	0.49	*to*						
7	**0.47**	0.66	1.11	0.73	0.8	*to*	**0.27**	0.3	0.32	0.31	0.32	*to*						
8	**0.67**	1.94	4.55	1.26	1.75	*to*	**0.33**	0.47	0.47	0.43	0.44	*to*						
9	**0.57**	0.63	1.07	0.71	0.77	*to*												
10	**1.32**	*to*	*to*	3.41	4.64	*to*												
11	*to*	*to*	*to*	**25.4**	41.12	*to*												
12	*to*	*to*	*to*	**26.15**	26.71	*to*												
13	*to*	*to*	*to*	**27.54**	28.68	*to*												

time elapse steps that either leave the ordering of clocks' fractional parts unchanged or advance to the very next region. This often makes it necessary to break up a single time elapse step in a counter-example into multiple time elapse steps in the discretized model. While not affecting execution times for the first property by much, the longer counter-examples for the discretization-based approach had a huge impact on the second, more complicated property. Another interesting result is that the mixed-type encoding performed significantly worse than the non-mixed type encodings, indicating that avoiding mixed-type expressions can be beneficial even for SMT-solvers that support them.

Industrial Benchmark. As second benchmark, a model of an emergency diesel generator intended for the use in a nuclear power plant was used. The model is fairly large, having 24 clocks and its location being defined by the valuation of 130 finite domain state variables. Additionally, two submodels (7 clocks / 64 state variables and 6 clocks / 36 state variables) which are sufficient for verifying some of the properties were used. The model has previously been studied using different verification methods including Uppaal [12, 21]. Table 2 shows the execution times for non-holding properties. For space restrictions, the results for the limited range integer encoding, which are just slightly worse than the unlimited range integer encoding, are omitted here. The discretization-based approach timed out for all properties on all submodels, clearly indicating that its applicability is restricted to small models. The choice of non-zenoness condition was irrelevant for all except one property. Unlike for the Fischer protocol, lasso-based BMC performed significantly better for some properties while performing significantly worse for others. A likely explanation again is the length of the counter-examples that can be found using the respective variants.

Furthermore, random properties for the industrial model were generated using the following LTL patterns: unconditional fairness ($\mathbf{GF}x$), strong fairness ($\mathbf{GF}x \Rightarrow \mathbf{GF}y$), weak fairness ($\mathbf{FG}x \Rightarrow \mathbf{GF}y$), "leads to" ($\mathbf{G}(x \Rightarrow \mathbf{F}y)$) and $\mathbf{G}x \Rightarrow \mathbf{G}y$, a pattern that

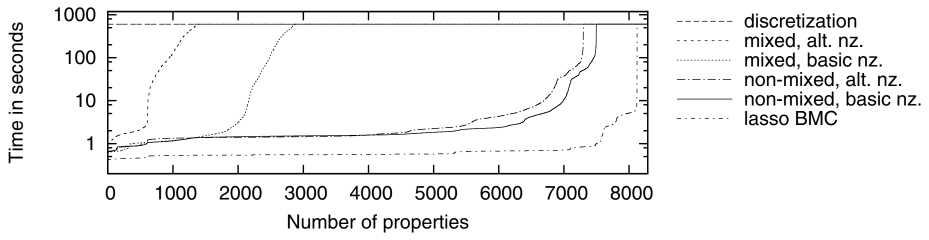


Fig. 3. Execution time by number of properties for random properties on the industrial benchmark. A point (x, y) indicates that for x properties y or less time was needed (each), corresponding to a plot of quantiles.

had been used by the authors of the industrial benchmark. 2000 properties of each type were randomly selected, except for the unconditional fairness pattern for which all 371 generated properties were selected. Figure 3 shows the times required to find counter-examples for all properties for which at least one method found a counter-example. Again, the discretization-based approach timed out for all properties. For random properties, lasso-based BMC was clearly faster than the region-based approaches. A likely explanation is that most random properties have very short counter-examples, meaning that the potentially smaller bound needed by region-based BMC is outweighted by the more complicated transition relation. Furthermore, random properties that involve only non-timing related parts of the system tend to have exceptionally short zeno counter-examples, further favoring lasso-based BMC which does not require non-zenoness.

7 Conclusions

In this paper, we have shown that traditional lasso-based SMT BMC is not complete for model checking linear-time properties on timed automata, introduced region-based SMT BMC to fix this problem, and shown its completeness. Different variations of the approach tailored for supported features of different SMT solvers are given. The variations of region-based SMT BMC have been experimentally compared to each other, to lasso-based SMT BMC and to discretization-based SAT BMC. The experiments indicate that region-based SMT BMC outperforms discretization-based SAT BMC. For hand-made properties, region-based SMT BMC also was more robust than lasso-based SMT BMC. For random properties, however, lasso-based SMT BMC performed better.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Alur, R.: Timed Automata. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999)
3. Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
4. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)

5. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 341–353. Springer, Heidelberg (1999)
6. Beyer, D., Noack, A.: Can Decision Diagrams Overcome State Space Explosion in Real-Time Verification? In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 193–208. Springer, Heidelberg (2003)
7. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
8. Woźna, B., Zbrzezny, A., Penczek, W.: Checking reachability properties for timed automata via SAT. Fundamenta Informatica 55(2), 223–241 (2003)
9. Sorea, M.: Bounded model checking for timed automata. Electronic Notes in Theoretical Computer Science 68(5) (2002)
10. Audemard, G., Cimatti, A., Kornilowicz, A., Sebastiani, R.: Bounded Model Checking for Timed Systems. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 243–259. Springer, Heidelberg (2002)
11. Malinowski, J., Niebert, P.: SAT Based Bounded Model Checking with Partial Order Semantics for Timed Automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 405–419. Springer, Heidelberg (2010)
12. Kindermann, R., Junttila, T., Niemelä, I.: Modeling for symbolic analysis of safety instrumented systems with clocks. In: ACSD 2011, pp. 185–194. IEEE (2011)
13. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, pp. 825–885. IOS Press (2009)
14. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
15. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
16. Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Computer Science 2(5:5), 1–64 (2006)
17. Dutertre, B., de Moura, L.M.: 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)
18. Clarke, E.M., Kroning, D., Ouaknine, J., Strichman, O.: Completeness and Complexity of Bounded Model Checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 85–96. Springer, Heidelberg (2004)
19. Tripakis, S., Yovine, S., Bouajjani, A.: Checking timed büchi automata emptiness efficiently. Formal Methods in System Design 26(3), 267–292 (2005)
20. Larsen, K.G., Pettersson, P., Yi, W.: Model-checking for Real-Time Systems. In: Reichel, H. (ed.) FCT 1995. LNCS, vol. 965, pp. 62–88. Springer, Heidelberg (1995)
21. Lahtinen, J., Björkman, K., Valkonen, J., Frits, J., Niemelä, I.: Analysis of an emergency diesel generator control system by compositional model checking. VTT Working Papers 156, VTT Technical Research Centre of Finland (2010)

A Appendix: Proofs of Lemmas 1 and 2

Lemma 1. *If $\mathcal{B}$ has an infinite accepting non-zeno run $\pi = \langle l_0, v_0\rangle\langle l_1, v_1\rangle \ldots$, then $\mathcal{B}$'s region automaton $\mathcal{B}^{\mathrm{R}}$ has a* lasso-shaped *accepting non-zeno run $\pi^{\mathrm{R}}_{\mathrm{lasso}}$.*

Proof. Let $\pi^{\mathrm{R}} = \langle l_0, [v_0]\rangle\langle l_1, [v_1]\rangle \ldots$ be the region automaton run corresponding to π. In the following we will construct a lasso-shaped run $\pi^{\mathrm{R}}_{\mathrm{lasso}}$ of $\mathcal{B}^{\mathrm{R}}$ from a prefix of π^{R}.

Let $X^{\mathrm{U}} \subseteq X$ be the set of clocks which are reset only finitely many times along π. Then according to Theorem 1, any clock $x \in X^{\mathrm{U}}$ exceeds m_x infinitely often on π^{R}, which combined with the fact that x is only reset finitely often, yields that x exceeds m_x in all states after a given point in π, i.e. $x \leq \mathrm{m}_x$ holds only finitely often on π. Let $s^{\mathrm{R}}_{\mathrm{inf}}$ be an arbitrary state occurring infinitely often in π^{R}. Note that such a state is guaranteed to exist due to the fact that $\mathcal{B}^{\mathrm{R}}$ has only finitely many states. Each $x \in X^{\mathrm{U}}$ is guaranteed to exceed m_x in $s^{\mathrm{R}}_{\mathrm{inf}}$, as $s^{\mathrm{R}}_{\mathrm{inf}}$ occurs infinitely often on π^{R} while $x \leq \mathrm{m}_x$ holds only finitely often. Let i_{inf} the lowest number such that $\langle l_{i_{\mathrm{inf}}}, [v_{i_{\mathrm{inf}}}]\rangle = s^{\mathrm{R}}_{\mathrm{inf}}$. We will turn π^{R} into a lasso-shaped run by looping back from a later occurrence of $s^{\mathrm{R}}_{\mathrm{inf}}$ to its occurrence at index i_{inf}.

Let i_{end} be the lowest index such that $\langle l_{i_{\mathrm{end}}}, [v_{i_{\mathrm{end}}}]\rangle = s^{\mathrm{R}}_{\mathrm{inf}}$ and (i) every clock in $X \setminus X^{\mathrm{U}}$ is reset, (ii) there is a time elapse step and (iii) an accepting state of $\mathcal{B}$ visited between the i_{inf}th and the i_{end}th state of π. As each of the described events occurs infinitely often on π, a corresponding choice of i_{end} is possible. Now let

$$\pi^{\mathrm{R}}_{\mathrm{lasso}} := \langle l_1, [v_1]\rangle \ldots \langle l_{i_{\mathrm{inf}}-1}, [v_{i_{\mathrm{inf}}-1}]\rangle(\langle l_{i_{\mathrm{inf}}}, [v_{i_{\mathrm{inf}}}]\rangle \ldots \langle l_{i_{\mathrm{end}}-1}, [v_{i_{\mathrm{end}}-1}]\rangle)^{\omega}$$

Then $\pi^{\mathrm{R}}_{\mathrm{lasso}}$ is a lasso shaped run of $\mathcal{B}^{\mathrm{R}}$. Furthermore, any clock $x \in X^{\mathrm{U}}$ exceeds its m_x value infinitely often on $\pi^{\mathrm{R}}_{\mathrm{lasso}}$, namely in $\langle l_{i_{\mathrm{inf}}}, [v_{i_{\mathrm{inf}}}]\rangle = s^{\mathrm{R}}_{\mathrm{inf}}$. In addition, due to the way i_{end} was chosen, any clock $X \setminus X^{\mathrm{U}}$ is reset infinitely often, an accepting state visited infinitely often and a time elapse step taken infinitely often on $\pi^{\mathrm{R}}_{\mathrm{lasso}}$. According to Theorem 1, this implies the non-zenoness of $\pi^{\mathrm{R}}_{\mathrm{lasso}}$, which concludes the proof of Lemma 1. □

Lemma 2. *If $\mathcal{B}^{\mathrm{R}}$ has an accepting lasso-shaped non-zeno run, then $\mathcal{B}^{\mathrm{R}}$ has an accepting lasso-shaped non-zeno run of length at most K_{region}.*

Proof. We will prove the lemma by construction of a lasso-shaped non-zeno run of $\mathcal{B}^{\mathrm{R}}$ of length at most K_{region} from the lasso-shaped run $\pi^{\mathrm{R}}_{\mathrm{lasso}}$ constructed in the proof of Lemma 1. Note that we only needed certain parts of the looping part of $\pi^{\mathrm{R}}_{\mathrm{lasso}}$ for showing non-zenoness of the run, namely one time elapse step, one reset step per clock in $X \setminus X^{\mathrm{U}}$, an accepting state and $s^{\mathrm{R}}_{\mathrm{inf}}$. Thus, we can replace every segment between two such relevant steps / states by the shortest path between them without affecting the non-zenoness of the run or the fact that it is accepting. The resulting loop consists of $\left|X \setminus X^{\mathrm{U}}\right|$ reset steps, one time elapse step, one accepting state, $s^{\mathrm{R}}_{\mathrm{inf}}$ and $\left|X \setminus X^{\mathrm{U}}\right| + 3$ shortest paths in between. Any shortest path has length of at most $\left|\mathcal{B}^{\mathrm{R}}\right| - 2$, not counting initial and final state, where $\left|\mathcal{B}^{\mathrm{R}}\right| \leq |L| \cdot |X|! \cdot 2^{|X|} \cdot \prod_{x \in X}(2\mathrm{m}_x + 2)$ is the number of states in $\mathcal{B}^{\mathrm{R}}$. Taking into account that every step consists of two states, this yields a length of at most $(\left|X \setminus X^{\mathrm{U}}\right| + 1) \cdot 2 + 2 + (\left|X \setminus X^{\mathrm{U}}\right| + 3) \cdot (\left|\mathcal{B}^{\mathrm{R}}\right| - 2) < (|X| + 3) \cdot \left|\mathcal{B}^{\mathrm{R}}\right| \leq (|X| + 3) \cdot |L| \cdot |X|! \cdot 2^{|X|} \cdot \prod_{x \in X}(2\mathrm{m}_x + 2) \leq K_{\mathrm{region}}$.

After modifying the looping part of our lasso-shaped run, we can replace the non-looping part with the shortest path from the initial state of $\mathcal{B}^{\mathrm{R}}$ to any state in the looping part. Note that in the resulting run the set of states that occur in the non-looping part is disjunct from the set of states in the looping part. This implies that we can reduce our upper bound to the length of the looping part by $|X| + 3$ for each state in the non-looping part and, ultimately, that K_{region} is an upper bound for the length of the entire resulting lasso-shaped run. □

Conformance Testing of Boolean Programs with Multiple Faults

Pavithra Prabhakar[1,2,⋆] and Mahesh Viswanathan[3]

[1] California Institute of Technology
[2] IMDEA Software Institute
[3] University of Illinois at Urbana-Champaign

Abstract. Conformance testing is the problem of constructing a complete test suite of inputs based on a specification S such that any implementation I (of size less than a given bound) that is not equivalent to S gives a different output on the test suite than S. Typically I and S are assumed to be some type of finite automata. In this paper we consider the problem of constructing test suites for boolean programs (or more precisely modular visibly pushdown automata) that are guaranteed to catch all erroneous implementations that have at least R faults, and pass all correct implementations; if the incorrect implementation has fewer than R faults then the test suite may or may not detect it. We present a randomized algorithm for the construction of such test suites, and prove the near optimality of our test suites by proving lower bounds on the size of test suites.

1 Introduction

Conformance testing is the problem of designing test suites based on a given formal specification S which is typically an automaton (either finite or infinite). In the framework of conformance testing in general, the implementation I being tested, is assumed to have an unknown internal structure, but can be tested by applying a sequence of inputs and observing the outputs it produces. Given an integer bound N, the goal is to construct a test suite T such that if some I of size less than N does not "conform" to S, then there is some input sequence in T on which the outputs of I and S differ. Typically, the notion of conformance is taken to be language equivalence, though weaker notions such as ioco have been explored [13]. Such conformance tests have not only been used to test circuits and protocols [6,10] but have also been used to model check black-box systems [12,5].

Since Moore's seminal work [11] on this problem, many algorithms for solving conformance testing have been proposed; major results are summarized in [6,4,9,10] [1]. All of these algorithms construct test suites when the specification and implementation are assumed to be finite state automata. Broadly, it

⋆ This work was done while the first author was a student at the University of Illinois at Urbana-Champaign.

[1] These references are to algorithms that construct complete test suites, which is the focus of this paper. There has also been a lot of work on constructing incomplete test suites that catch all bugs in the limit.

is understood that the running time of the algorithm and the size of the test suite are polynomial in the size of the specification, when the implementation is assumed to have at most as many states as the specification. When the implementation can have Δ extra states, the running time and the size of the constructed test suite have an exponential dependence on Δ. These bounds are known to be optimal [14].

While finite state models are convenient abstractions in many situations, in order to faithfully model software, it becomes imperative to consider models that explicitly capture recursion. Therefore, Boolean programs, or more precisely *modular VPAs* [7], have been considered, and the results on conformance testing finite state machines have been extended to such recursive models [7]. Modular VPAs are an automata model, inspired by Visibly Pushdown Automata (VPA) [2] and Recursive State Machines (RSMs) [1], that capture sequential recursive programs all of whose data variables are Boolean. Thus, they are pushdown automata whose control states have been partitioned into modules that correspond to functions in a program. The trace of these machines explicitly encodes recursive function calls by the name of the module being called and the associated parameter, as well as returns from such calls. In addition, like in typical programming languages, function calls result in the calling state being pushed onto the call stack; the parameter of the call is not pushed onto the stack but is rather stored in the local state of the called module. These restrictions ensure that modular VPAs have unique minimized (in terms of the number of control states) machines that can be constructed in polynomial time. The minimization procedure is based on a congruence-based characterization of modular VPAs, which can then be exploited to solve the conformance testing problem. Assuming that the specification S is a minimized modular VPA and the number of control states of the implementation I is not more than that of S, there is polynomial time algorithm that constructs a complete test suite. The input sequences in the test suite are presented symbolically using an equation system; when expanded to get an explicit sequence of input symbols, the size can be exponential in the number of control states of S. This exponential dependency cannot be avoided because the shortest path reaching a particular control state in a pushdown system can be exponentially long. When the implementation I has Δ additional control states, the running time of the algorithm, the symbolic representation of the test suite, and the explicit representation of the test suite, are exponentially dependent on Δ.

In this paper we investigate if the exponential dependence on the extra states of the implementation can be avoided if we relax the completeness requirements of the test suite. More precisely, we consider the problem of designing an (R, Δ) conformance test. An (R, Δ) test for a specification S with n control states, is a test suite T such that any implementation I that has at most $n + \Delta$ control states and at least R faults, gives a different output from S on some input in T; here we say that I has at least R faults, if at least R changes to the transition relation of I must be made in order to get a correct implementation. The notion of (R, Δ) conformance tests was first introduced in [8], where such test suites were constructed for specifications and implementations that are finite state machines. In this paper, we continue this line of work, and extend it to the case of recursive software.

In order to explain the challenges and contributions of our work, we recall the main ideas used in conformance testing algorithms. In a minimized specification machine S, any pair of control states p, q can be *distinguished* by a test; in the case of finite state systems it is simply an input sequence that is applied from p and q, and in the case of pushdown systems, it is a pair of input strings — one that sets up a common stack for p and q, and the other on which p and q give different outputs. Moreover, for every control state q, there is an input string, called the *access string* for q, that takes the specification to q. When the implementation does not have extra states, the idea is to check that the implementation state reached on x_q (access string for q) "behaves like" q; in other words, this implementation state gives the same output as q on all the distinguishing tests, and transitions out of the implementation state go to states that behave like the target of transitions out of q. When there are Δ extra states, the test suite must have input sequences that visit the states not reached by the x_q inputs (called "unknown" states of I), and check the transitions out of those.

If I and S are finite state machines, then these unknown states can be reached within Δ transition steps from a "known state". Thus, the idea is to have tests that explore *every* input sequence of length at most Δ from *every* known state, and check that the states reached in I behave the same way as the states reached in S after the same input sequence. Hence, both the size of the test and the time to generate it, depend exponentially on Δ. Moreover, if any of these "walks" of length at most Δ from known states is omitted from the test, then the test suite cannot guarantee to catch every incorrect implementation. When the completeness requirements are relaxed (namely, to provably catch implementations with at least R faults), two factors come into play [8]. First, one can show that a "faulty" state can be reached in a fewer number of steps from a known state, when I has at least R faults; here, by faulty state we mean one for which the error can be observed when the distinguishing tests are applied. Second, many of these short walks from known states lead to faulty states. Thus, if we were to choose (randomly) a few of these short walks from every known state, then the test suite is likely to catch every implementation with at least R faults. These observations were used in [8] to give a randomized algorithm that outputs a small test suite that, with high probability, is likely to be a (R, Δ) conformance test.

When I and S are recursive programs, the situation changes. Since the shortest path reaching a particular control state in a pushdown automaton can be exponentially long, this means that unknown states may be reached only if we take $2^{n+\Delta}$ steps from a known state. Thus, a naïve application of the observations from the finite state case suggests that the dependence of the size of the test suite on Δ would be doubly exponential. However, it was observed in [7] that one doesn't need to consider all input sequences of length $2^{n+\Delta}$ from known states, but rather only certain "special" ones that are described succinctly using equation systems of linear size. This key observation was used to get a test generation algorithm and test suite, whose asymptotic complexity is similar to that for finite state systems.

For (R, Δ) conformance tests and modular VPAs, the ideas from the finite state case do not extend easily. In [8], the proof of the existence of many short walks from known states to faulty states, relied on the existence of large cuts separating known states from faulty states in the implementation. Such large cuts do not seem to exist for modular VPAs. However, using new proof techniques, we show that when I has R faults, many paths described by equations systems lead to faulty states. Thus, if the test generation algorithm randomly picks some of these walks from known states then, with high probability, the resulting test suite will be a (R, Δ) conformance test. Note that, unlike the finite state case, we cannot show that "short walks" are sufficient. Finally, we present lower bounds on the size of (R, Δ) conformance tests for modular VPAs. These lower bounds demonstrate that the test suite constructed by our randomized algorithm is close to optimal.

We conclude this introduction by discussing the practical relevance of our algorithm, and (R, Δ) tests in general. Our algorithm is a randomized algorithm that is highly likely to output a (R, Δ) test. Here the probability of error is over random decisions made by the algorithm, not on a distribution over machines S and I. Thus the error can be reduced to as small a number as desired by increasing the size of the test suite. (R, Δ) tests, though guaranteed to catch implementations with at least R faults, nonetheless, can detect errors in implementations with fewer faults. Thus, (R, Δ) tests can be seen as incomplete tests along a dimension orthogonal to traditional metrics like coverage. Therefore, their importance is derived not so much in the precise way we count faults in an implementation, but rather from the fact that their incompleteness can be mathematically characterized. Hence, (R, Δ) tests should be seen as a hierarchy of test suites of increasing precision, to be chosen from, based on practical time constraints imposed on the testing process by product release times.

2 Modular Visibly Pushdown Automata (*MVPA*)

Boolean programs are essentially programs in any imperative language in which all the variables have a boolean type. In particular, they do not have dynamic memory, and parameters are passed to functions by call-by-value. Formally, they define a Modular Visibly Pushdown Automata, which we present next.

Modular Visibly Pushdown Automata. Let M be a finite set of *modules* and $m_0 \in M$ be the initial module. For each $m \in M$, let P_m be a nonempty finite set of *parameters* and let $P_{m_0} = \{p_0\}$. A *call* c is a pair (m, p) where $m \in M \backslash \{m_0\}$ and $p \in P_m$, and denotes the action of calling a module m with parameter p (we won't allow initial module to be called except at the beginning, and hence (m_0, p_0) will not be a call). Let Σ^{call} denote the set of all calls. Let Σ^{int} be a finite set of internal actions, and let $\Sigma^{ret} = \{r\}$ be the alphabet of returns, containing the unique symbol r. We will assume that the sets Σ^{call}, Σ^{int} and Σ^{ret}

are mutually disjoint. Let $\hat{\Sigma} = (\Sigma^{call}, \Sigma^{int}, \Sigma^{ret})$ and let $\Sigma = \Sigma^{call} \cup \Sigma^{int} \cup \Sigma^{ret}$. We call $\langle M, \{P_m\}_{m\in M}, m_0, \hat{\Sigma}\rangle$ a *signature.*

A *modular visibly pushdown automaton (MVPA)* $\mathcal{A}$ over the signature $\langle M, \{P_m\}_{m\in M}, m_0, \hat{\Sigma}\rangle$ is a tuple $(\{Q_m, \{q_m^p\}_{p\in P_m}, \delta_m\}_{m\in M}, F)$ where for each $m \in M$:

- Q_m is a finite set of states. We assume that for $m \neq m'$, $Q_m \cap Q'_m = \emptyset$. Let $Q = \cup_{m\in M} Q_m$ denote the set of all states.
- For each parameter $p \in P_m$, $q_m^p \in Q_m$ is a state associated with p; we will call this the entry associated with the call (m, p). (Note that we do not insist that q_m^p is different from $q_m^{p'}$, when $p \neq p'$.)
- $\delta_m : Q_m \times (\Sigma^{call} \cup \Sigma^{int} \cup Q) \to Q$ such that:
  - Call transitions:
    for every $q \in Q_m$, $(n, p) \in \Sigma^{call}$, $\delta_m(q, (n, p)) = q_n^p$;
  - Internal transitions:
    for every $q \in Q_m$, $a \in \Sigma^{int}$, $\delta_m(q, a) \in Q_m$;
  - Return transitions:
    for every $q \in Q_m$ and $q' \in Q_{m'}$, $\delta_m(q, q') \in Q_{m'}$;

  The transition function $\delta : Q \times (\Sigma^{call} \cup \Sigma^{int} \cup Q)$ is such that δ restricted to $Q_m \times (\Sigma^{call} \cup \Sigma^{int} \cup Q)$ is δ_m.
- $F \subseteq Q_{m_0}$ is the set of final states.

Notation. We write $q \xrightarrow{a}_{\mathcal{A}} q'$ to denote $\delta(q, a) = q'$ for $a \in (\Sigma^{call} \cup \Sigma^{int} \cup Q)$. We drop the subscript $\mathcal{A}$ when it is clear from the context. Unless stated otherwise, we will always assume the signature to be $Sig = \langle M, \{P_m\}_{m\in M}, m_0, \hat{\Sigma}\rangle$.

We consider *MVPA* with deterministic transitions, since the non-deterministic version is equivalent in expressiveness to the deterministic version [7].

A *MVPA* reads words that are well-matched. A word u is a well matched word if the number of occurrences of the call symbols in it is equal to the number of occurrences of return symbols in it, and the number of occurrences of the call symbols in any prefix of u is greater than or equal to the number of occurrences of return symbols in the prefix. The set of all well-matched words over $\hat{\Sigma}$ is denoted by $WM(\hat{\Sigma})$. From now on, we will use u, u', u_i to denote words and w, w', w_i to denote well-matched words.

A *MVPA* operates by reading a well-matched word, and modifying its state and stack accordingly. It starts in the initial state, $q_{m_0}^{p_0}$, with an empty stack. It reads a symbol of the word and takes one of the transitions out of the current state which matches the symbol. When the automaton reads the symbol r, it takes a return transition. It changes its state to the target state of the transition. When an internal transition is taken, the stack remains unchanged. If it takes a call transition, then it pushes the current state onto the stack. A return transition can be taken only if the transition label and the top of the stack match, in which case the top element of the stack is popped.

Formally, the semantics of *MVPA* is defined in terms of a graph over configurations. A *configuration* is a pair $(q, \sigma) \in Q \times Q^*$, where q denotes the current state of the *MVPA* and σ denotes its stack contents. We assume that the last symbol of σ is the top of the stack. Let *Conf* denote the set of all configurations along with a special configuration c_0. The semantics of a *MVPA* $\mathcal{A}$ is given by a graph (V, E) where the set of vertices V is given by the set of configurations *Conf* and the set of edges $E \subseteq V \times V$ is the smallest set satisfying the following:

– (Initial) The edge $c_0 \xrightarrow{(m_0,p_0)} q_{m_0}^{p_0}$ is in E.
– (Internal) If $(q, \sigma) \in V$, $a \in \Sigma^{int}$ and $\delta(q, a) = q'$, then the edge $(q, \sigma) \xrightarrow{a} (q', \sigma)$ is in E.
– (Call) If $(q, \sigma) \in V$ and $(m, p) \in \Sigma^{call}$, then $(q, \sigma) \xrightarrow{(m,p)} (q_m^p, \sigma q)$ is in E.
– (Return) If $(q, \sigma q') \in V$ and $\delta(q, q') = q''$, then $(q, \sigma q') \xrightarrow{r} (q'', \sigma)$ is in E.

A *run* of $\mathcal{A}$ on a word $u = a_1 \cdots a_n$ is a path in the configuration graph on u, that is, a path $\pi = c_0 c_1 \cdots c_n$ such that $c_i \xrightarrow{a_{i+1}} c_{i+1}$ for all $0 \leq i < n$. Note that such a path is unique. We say that $\mathcal{A}$ *reaches* the state q on u, if there exists a run $\pi = c_0 \cdots c_n$ of $\mathcal{A}$ on u such that $c_n = (q, \sigma)$ for some stack configuration σ. π is an *accepting* run of $\mathcal{A}$ on u if the last configuration c_n of π is (q, σ) for some final state $q \in F$. A word u is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on u. The *language* of $\mathcal{A}$, $L(\mathcal{A})$, is defined as the set of words $u \in \Sigma^*$ accepted by $\mathcal{A}$.

Remark 1. We note that a *MVPA* accepts only well-matched words since the start state $q_{m_0}^{p_0}$ and the final states are all in Q_{m_0}, and every return transition returns to the module of the corresponding call transition.

Example 1. Figure 1 shows an *MVPA* with two modules m_0 and m, with $P_{m_0} = \{p_0\}$ and $P_m = \{p\}$. Here $\Sigma^{int} = \{a, b\}$, $q_{m_0}^{p_0} = q_0$, $q_m^p = q_1$ and $F = q_0$. There are internal transitions within the module and return transitions from module m to module m_0 on states of m_0. There are call transitions from every state to q_1 on (m, p). The language of the *MVPA* in Figure 1 is the set of all well matched words without nested calls such that any non-empty sequence between a call and the following return consists of an alternating sequence of as and bs starting with an a.

Later we will need the notion of a partial homomorphism and homomorphism, which we define below:

Definition 1. *Given MVPA* $\mathcal{A} = (\{Q_m, \{q_m^p\}_{p \in P_m}, \delta_m\}_{m \in M}, F)$ *and* $\mathcal{A}' = (\{Q'_m, \{q'^p_m\}_{p \in P_m}, \delta'_m\}_{m \in M}, F')$ *over the signature Sig with* $Q = \cup_{m \in M} Q_m$ *and* $Q' = \cup_{m \in M} Q'_m$*, a function* $h : Q'' \to Q$*, where* Q'' *is a subset of* Q' *containing* $q'^{p_0}_{m_0}$*, is called a* partial homomorphism *from* $\mathcal{A}'$ *to* $\mathcal{A}$ *if:*

*B*1 $h(q'^{p_0}_{m_0}) = q_{m_0}^{p_0}$.
*B*2 *For every* $q' \in Q''$, $q' \in F'$ *iff* $h(q') \in F$.
*B*3 *For every* $q' \in Q''$ *and* $a \in \Sigma^{call} \cup \Sigma^{int} \cup Q''$*, if* $\delta'(q', a) \in Q''$*, then* $h(\delta'(q', a)) = \delta(h(q'), a')$*, where* $a' = a$ *if* $a \in \Sigma^{call} \cup \Sigma^{int}$ *and* $a' = h(a)$ *otherwise.*

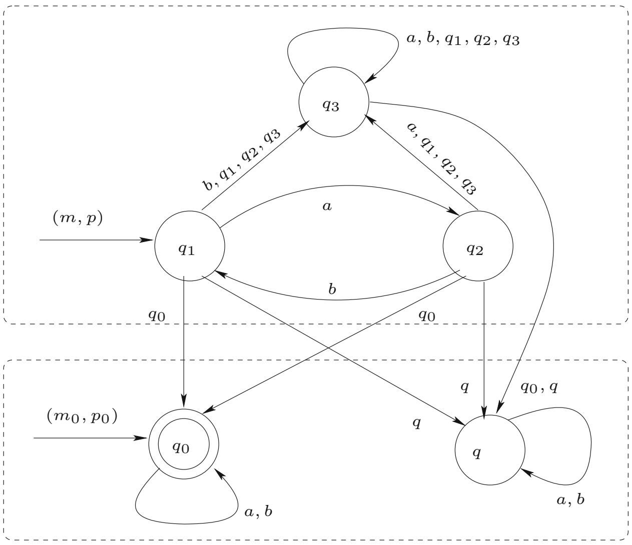


Fig. 1. Example *MVPA*: There are call transitions from every state to q_1 on (m, p) which is omitted from the figure

In addition, if the following condition is satisfied, then we call h a *homomorphism.*

$B4$ For every $q' \in Q''$ and $a \in \Sigma^{call} \cup \Sigma^{int} \cup Q''$, $\delta'(q', a) \in Q''$.

Proposition 1. *If there is a homomorphism from $\mathcal{A}'$ to $\mathcal{A}$, then $\mathcal{A}'$ and $\mathcal{A}$ are* equivalent*, that is, $L(\mathcal{A}') = L(\mathcal{A})$.*

Next we state a result from [7] on the minimization of *MVPA*. Since the size of the *MVPA*, which is the total number of bits required to describe it, is polynomial in the number of states of the automaton, we will consider the size of the automaton to be the number of states.

Theorem 1 ([7]). *Given a MVPA $\mathcal{A}$ over Sig, there exists a unique minimal MVPA $\mathcal{A}_{min}$ over Sig such that $L(\mathcal{A}_{min}) = L(\mathcal{A})$.*

The construction of the automaton $\mathcal{A}_{min}$ is based on the following congruence relation. For each $m \in M$, the equivalence relation $\sim_m$ on $P_m \times WM(\hat{\Sigma})$ which depends on $L = L(\mathcal{A})$ (and not on $\mathcal{A}$) is defined as follows: $(p_1, w_1) \sim_m (p_2, w_2)$ iff $\forall u, v \in \Sigma^*$, $u(m, p_1)w_1v \in L \Leftrightarrow u(m, p_2)w_2v \in L$. The states of $\mathcal{A}_{min}$ in the module m are the equivalence classes of $\sim_m$, and a state $[(p, w)]$ can be reached by the string $(m_0, p_0)(m, p)w$. From the definition of $\sim_m$, it is clear that given two distinct states $[(p_1, w_1)]$ and $[(p_2, w_2)]$ of module m, there exist $u, v \in \Sigma^*$ such that exactly one of $u(m, p_1)w_1v$ and $u(m, p_2)w_2v$ is in L. We call (u, v) a distinguishing pair and $(m, p_1)w_1$ an access string. We will need these notions later, hence we will define these next.

Informally an access string is a word which can be used to reach a certain state from an entry state of its module. An *access string* for a state q of module m is a string of the form $(m,p)w$, where $p \in P_m$ and $w \in WM(\hat{\Sigma})$, such that $(m_0,p_0)(m,p)w$ reaches q. We call a state *accessible* if there is an access string x such that $(m_0,p_0)x$ reaches it. A *complete set of access strings* is a set containing an access string for every accessible state of the automaton. Given a *MVPA* $\mathcal{A}$ and an access string x, we denote the state reached in $\mathcal{A}$ on $(m_0,p_0)x$ by $state_{\mathcal{A}}(x)$.

For distinct states q_1,q_2 in module m of $\mathcal{A}$, a pair of strings (u,v) is a *distinguishing test* for $\{q_1,q_2\}$ if for all access strings $(m,p_1)w_1$ and $(m,p_2)w_2$ of q_1 and q_2 respectively, exactly one of $u(m,p_1)w_1v$ and $u(m,p_2)w_2v$ is in $L(\mathcal{A})$. We also say that (u,v) distinguishes q_1 and q_2. D is a *complete set of distinguishing* tests if for every module m and distinct states q_1,q_2 in module m of $\mathcal{A}$, there is a distinguishing test $(u,v) \in D$ for $\{q_1,q_2\}$. Observe that a complete set of distinguishing tests always exists for a minimal *MVPA*.

For the *MVPA* in Figure 1, an access string for q_2 is $(m,p)a$, and an access string for q is $a(m,p)br$. A distinguishing test for the states q_1,q_2 is $u=(m_0,p_0)$ and $v=ar$, since for any access string x for q_1, uxv belongs to the language of the *MVPA* and for any access string y for q_2, uyv does not belong to the language.

Let $\mathcal{A}$ be a minimal *MVPA* with n states and $\{x_1,\cdots,x_n\}$ be a complete set of access strings for $\mathcal{A}$ (every state of a minimal *MVPA* is accessible). Let $\Omega = \Sigma \cup \{x_i\}_{i=1}^n$. We recall the following facts about distinguishing tests from [7].

Lemma 1 ([7]). *A complete set of distinguishing tests D for $\mathcal{A}$ can be constructed in time $O(n^5)$. Further, D can be represented as $\binom{n}{2}$ strings in Ω^*, each of length $O(n^2)$.*

3 Conformance Testing

In this section we define the problem of conformance testing *MVPA* and prove some preliminary lemmas.

By "conformance", we mean language equivalence. Given a *specification machine* $\mathcal{S}$ and a "black-box" implementation machine $\mathcal{I}$ that are both deterministic complete modular *MVPA* over signature *Sig*, we want to test if $\mathcal{I}$ is equivalent to $\mathcal{S}$, i.e., whether $L(\mathcal{I}) = L(\mathcal{S})$. We make the following assumptions:

1. $\mathcal{S}$ is minimized and has n states;
2. $\mathcal{I}$ has at most $N = n + \Delta$ states;

Note that assumption 1 is not a restriction since the details of $\mathcal{S}$ are known and hence can be minimized. Assumption 2 is necessary to guarantee that every state of the implementation is explored. Hence, whenever we refer to a specification machine we assume it is minimized. Also, all the automata we refer to from now on are *MVPA*.

A *sample* is a set of well-matched words. Let the length of a sample be the sum of the lengths of the words in the test. A sample T *distinguishes* $\mathcal{I}$ from $\mathcal{S}$, if there is a word $w \in T$ such that w is accepted by exactly one of $\mathcal{S}$ and $\mathcal{I}$. Given a specification machine $\mathcal{S}$ with n states, a Δ-*conformance test* is a sample T of well-matched words that distinguishes every incorrect implementation machine $\mathcal{I}$, that is, $\mathcal{I}$ such that $L(\mathcal{S}) \neq L(\mathcal{I})$, with at most $n + \Delta$ states from $\mathcal{S}$. Given $\mathcal{S}$ and Δ, a *conformance testing algorithm* outputs a Δ-conformance test.

Let us fix a "black box" implementation machine $\mathcal{I}$ with at most $n + \Delta$ states and a specification machine $\mathcal{S}$ with n states such that $\mathcal{I}$ is not equivalent to $\mathcal{S}$. We first focus on the problem of finding a sample T which distinguishes $\mathcal{I}$ from $\mathcal{S}$. Let Q be the accessible states of $\mathcal{I}$, and $\hat{Q}$ the accessible states of $\mathcal{S}$. Let D be a complete set of distinguishing tests for $\mathcal{S}$. Let $(u_{\hat{q_1}\hat{q_2}}, v_{\hat{q_1}\hat{q_2}}) \in D$ be a distinguishing test for $\{\hat{q}_1, \hat{q}_2\}$ and $D_{\hat{q_1}} = \bigcup_{\hat{q_2}} \{(u_{\hat{q_1}\hat{q_2}}, v_{\hat{q_1}\hat{q_2}})\}$.

We find a sample T such that if T does not distinguish an implementation from $\mathcal{S}$ then there exists a homomorphism from the implementation to $\mathcal{S}$. If T does not distinguish $\mathcal{I}$ from $\mathcal{S}$, then Proposition 1 would imply that $L(\mathcal{I}) = L(\mathcal{S})$, contradicting the assumption on $\mathcal{I}$ and $\mathcal{S}$.

We find a set of access strings for the states of $\mathcal{I}$. We then check that the states reached by these strings in $\mathcal{I}$ and $\mathcal{S}$ are indistinguishable with respect to the distinguishing tests. In order to verify that the transitions in $\mathcal{I}$ are correct, we check that the states reached by taking the transitions in both $\mathcal{I}$ and $\mathcal{S}$ are indistinguishable, that is, to verify that a transition from a state q on a symbol a in $\mathcal{I}$ is correct, we check that the states reached in $\mathcal{I}$ and $\mathcal{S}$ on reading ya are indistinguishable, where y is an access string for q in $\mathcal{I}$, and so on.

Let Y be an arbitrary set of access strings for $\mathcal{I}$, and let $Q' = \{state_{\mathcal{I}}(y) \mid y \in Y\}$. For each $q \in Q'$, fix an access string $y_q \in Y$ for q. Let us define a function $h_Y : Q' \to \hat{Q}$ as follows. $h_Y(q) = state_{\mathcal{S}}(y_q)$. We give a characterization of when h_Y is a partial homomorphism and when h_Y is a homomorphism by describing a set of tests.

Definition 2. *A set of access strings Y is called* safe *if it contains $\{\epsilon\} \cup \{(m, p) \mid m \in M \backslash \{m_0\}, p \in P_m\}$ as a subset and satisfies the following conditions:*

$C1$ *For each $y \in Y$, $(m_0, p_0)y \in L(\mathcal{I})$ iff $(m_0, p_0)y \in L(\mathcal{S})$.*

$C2$ *For each $y \in Y$, for each $(u, v) \in D_{state_{\mathcal{S}}(y)}$, $uyv \in L(\mathcal{I})$ iff $uyv \in L(\mathcal{S})$.*

$C3$ *For each $y \in Y$ and $a \in \Sigma^{int}$, for each $(u, v) \in D_{state_{\mathcal{S}}(ya)}$, $uyav \in L(\mathcal{I})$ iff $uyav \in L(\mathcal{S})$.*

$C4$ *For each $y_1, y_2 \in Y$, and for each $(u, v) \in D_{state_{\mathcal{S}}(y_2 y_1 r)}$, $uy_2y_1rv \in L(\mathcal{I})$ iff $uy_2y_1rv \in L(\mathcal{S})$.*

Informally, a safe set of access strings corresponds to a set of states such that the transitions out of these states do not contain any "bad" transitions. Condition $C1$ verifies that an access string reaches a final state of $\mathcal{I}$ iff it reaches a final state of $\mathcal{S}$. Condition $C2$ ensures that, the states reached by a string of the set in the specification and implementation behave similarly. Condition $C3$ ensures that a transition labelled by an internal symbol is not "bad", that is, the states reached in $\mathcal{I}$ and $\mathcal{S}$ after reading the symbol a from states reached by the same access string exhibit similar behavior. Similarly $C4$ ensures that the return transitions are not "bad". The next lemma states that if Y is a safe set then it defines a partial homomorphism.

Lemma 2. *If Y is safe then h_Y is a partial homomorphism.*

Corollary 1. *If Y is a safe and complete set of access strings for $\mathcal{I}$, then h_Y is a homomorphism.*

If we are given a complete set of access strings Y for $\mathcal{I}$ which contains $\{\epsilon\} \cup \{(m,p) \,|\, m \in M \backslash \{m_0\}, p \in P_m\}$, then we can use the above characterization to obtain a sample T_Y which distinguishes $\mathcal{I}$ from $\mathcal{S}$. T_Y is the union of the following sets:

- $T_0 = \{(m_0, p_0)y \,|\, y \in Y\}$.
- $T_1 = \{uyv \,|\, y \in Y, (u,v) \in D_{state_{\mathcal{S}}(y)}\}$.
- $T_2 = \{uyav \,|\, y \in Y, (u,v) \in D_{state_{\mathcal{S}}(ya)}\}$.
- $T_3 = \{uy'yrv \,|\, y, y' \in Y, (u,v) \in D_{state_{\mathcal{S}}(y'y)}\}$.

However, we cannot compute the set of access strings directly, since we do not have knowledge about the internal structure of $\mathcal{I}$. We use the following result from [7]. Let us fix a complete set of access strings $\{x_1, \cdots, x_n\}$ for $\mathcal{S}$ which contains $\{\epsilon\} \cup \{(m,p) \,|\, m \in M \backslash \{m_0\}, p \in P_m\}$. Let us denote by $x_{\hat{q}}$ the access string for the state $\hat{q} \in \hat{Q}$ in the above set.

Lemma 3 ([7]). *For each $\hat{q} \in \hat{Q}$ and $(u,v) \in D_{\hat{q}}$, let $ux_{\hat{q}}v \in L(\mathcal{I})$ iff $ux_{\hat{q}}v \in L(\mathcal{S})$. Then there exist access strings for the states of $\mathcal{I}$, $y_1, \cdots, y_N$, where $y_i = x_i$ for $1 \leq i \leq n$ and for each $n < i \leq N$, $y_i = y_j a$ or $y_i = y_j y_k r$ for some $a \in \Sigma^{int}$ and $j, k < i$.*

The premise of the above lemma ensures that distinct x_i access distinct states of $\mathcal{I}$. The above lemma states that a complete set of access strings of $\mathcal{I}$ can be represented as a system of $N - n$ equations of the form $y_i = y_j a$ or $y_i = y_j y_k r$. Since $\mathcal{I}$ is given as a "black box", in order to obtain a sample distinguishing $\mathcal{I}$ from $\mathcal{S}$, we need to consider all the $(N|\Sigma| + N^2)^{N-n}$ systems of equations. We denote the set of all systems of equations as Γ.

Definition 3. *Given a complete set of access strings X for the specification $\mathcal{S}$, we denote by $\Gamma(X, \Delta)$, the set of all systems of equations of the form $y_i = ua$ or $y_i = uvr$ for $1 \leq i \leq N - n$ where each of u, v is either an element of X or is y_j for some $j < i$. Given an element $\gamma \in \Gamma(X, \Delta)$, we denote by Y_γ, the set of access strings generated by γ, that is, the elements of X and the word assignments for y_i, $1 \leq i \leq N - n$ which satisfy the equations in γ.*

Next we present the algorithm given in Algorithm 1.1, which takes as input the specification $\mathcal{S}$, a complete set of access strings $X = \{x_1, \cdots, x_n\}$ for $\mathcal{S}$, a complete set of distinguishing tests D for $\mathcal{S}$ and the "black box" implementation

Algorithm 1.1

```
1 Input: $(\mathcal{S}, X, D, \mathcal{I})$
2 Output: Sample $T$
3 $T \leftarrow \emptyset$
4 for every $\gamma \in \Gamma(X, \Delta)$ do
5     $T \leftarrow T \cup T_{Y_\gamma}$
6 end for
```

Algorithm 1.2

```
1 Input: $(\mathcal{S}, X, D, \mathcal{I})$
2 Output: Sample $T$
3 $T \leftarrow \emptyset$
4 for $l = 1, \cdots, m$ do
5     $\gamma \leftarrow Rand(\Gamma(X, \Delta))$
6     $T \leftarrow T \cup T_{Y_\gamma}$
7 end for
```

$\mathcal{I}$, and outputs a sample T which distinguishes $\mathcal{I}$ from $\mathcal{S}$ if $\mathcal{I}$ is an incorrect implementation.

Observe that if $\mathcal{I}$ is equivalent to $\mathcal{S}$, then no sample can distinguish the two. On the other hand, it $\mathcal{I}$ and $\mathcal{S}$ are not equivalent, then the output of Algorithm 1.1, namely T, distinguishes $\mathcal{I}$ from $\mathcal{S}$. In fact T distinguishes any $\mathcal{I}$ which is not equivalent to $\mathcal{S}$, hence the algorithm outputs a Δ-conformance test.

Theorem 2 ([7]). *The length of the Δ-conformance test output by Algorithm 1.1 is $O((a2^{\Delta} + b)(n + \Delta)dz((n + \Delta)d)^{\Delta})$, where a is the maximum length of the strings in X which is $O(2^n)$, b the maximum length of $|u| + |v|$ for any pair $(u, v) \in D$ which is $O(2^n)$, $z = \max_{\hat{q} \in \hat{Q}} |D_{\hat{q}}|$ which is $O(n)$, and $d = (n + \Delta + |\Sigma^{int}|)$.*

For the sake of illustration, we will give a conformance test for the example in Figure 1 for the case with no extra states. That is, let the specification $\mathcal{S}$ be the *MVPA* of Figure 1. Note that it is a minimal *MVPA* (we will exhibit a complete set of distinguishing tests). We give a conformance test which distinguishes every *MVPA* with at most 5 states which is not equivalent to $\mathcal{S}$. First let us fix access strings for every state of $\mathcal{S}$. Let $x_{q_0} = \epsilon$, $x_q = (m, p)br$, $x_{q_1} = (m, p)$, $x_{q_2} = (m, p)a$ and $x_{q_3} = (m, p)b$. Here y_s is an access string for state s, that is, $(m_0, p_0)y_s$ reaches state s. Next let us define a complete set of distinguishing tests. $((m_0, p_0), \epsilon)$ is a distinguishing pair for $\{q_0, q\}$. $((m_0, p_0), ar)$ is a distinguishing pair for $\{q_1, q_2\}$ and $\{q_1, q_3\}$, and $((m_0, p_0), r)$ is a distinguishing pair for $\{q_2, q_3\}$. Since $\Delta = 0$, Γ is a singleton set $\{\gamma\}$ and the corresponding set $Y_\gamma = \{x_{q_0}, x_q, x_{q_1}, x_{q_2}, x_{q_3}\}$. The test T is simply T_{Y_γ} which is the union of the following sets: (The substrings in bold font correspond to the part of the string which comes from the set Y.)

- $T_0 = \{(m_0, p_0), (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{br}, (m_0, p_0)(\mathbf{m}, \mathbf{p}),$
  $(m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{a}, (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{b}\}$.
- $T_1 = \{(m_0, p_0), (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{br}, (m_0, p_0)(\mathbf{m}, \mathbf{p})ar,$
  $(m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{a}ar, (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{b}ar, (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{a}r,$
  $(m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{b}r\}$.
- $T_2 = \{(m_0, p_0)\mathbf{a}, (m_0, p_0)\mathbf{b}, (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{bra},$
  $(m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{brb}, (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{a}ar,$
  $(m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{a}r,\ (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{b}ar, (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{b}r,$
  $(m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{aa}ar, (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{aa}r,$

$(m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{ab}ar, (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{ba}ar,$
$(m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{ba}r, (m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{bb}ar,$
$(m_0, p_0)(\mathbf{m}, \mathbf{p})\mathbf{bb}r\}.$

- T_3 has more than 25 strings, hence in interest of space, we will explain how some of the elements of T_3 are constructed. Choose any two strings from Y, say x_{q_1} and x_{q_2}. The state reached on $x_{q_1}x_{q_2}r = (m, p)(m, p)ar$ is q_3. Add the strings $ux_{q_1}x_{q_2}rv$ for every (u, v) which distinguishes q_3 from the other states in the module m, namely, T_3 contains $(m_0, p_0)(\mathbf{m}, \mathbf{p})(\mathbf{m}, \mathbf{p})\mathbf{ar}ar$ and $(m_0, p_0)(\mathbf{m}, \mathbf{p})(\mathbf{m}, \mathbf{p})\mathbf{ar}r$.

4 (R, Δ)-Conformance Testing

In this section, we consider a relaxed version of the conformance testing problem in which a test is required to distinguish only those implementations which are at a certain distance from the specification. We define the distance of an implementation from a specification to be the number of transitions that need to be changed in the implementation so as to make it equivalent to the specification. The formal definition is given below:

Definition 4. *A MVPA $\mathcal{I}$ is at distance R from $\mathcal{S}$ if the smallest set $D \subseteq Q' \times (\Sigma^{call} \cup \Sigma^{int} \cup Q')$ such that there exists a MVPA $\mathcal{J} = (\{Q''_m, \{q''^p_m\}_{p \in P_m}, \delta''_m\}_{m \in M}, F'')$ over Sig satisfying the following conditions has size R.*

- *For each m, $Q''_m = Q'_m$.*
- *$F'' = F'$.*
- *δ'' and δ' differ only on the set D.*

Given a specification *MVPA* $\mathcal{S}$ of size n, an (R, Δ)-*conformance test* is a sample T of well-matched words which distinguishes $\mathcal{I}$ from $\mathcal{S}$, for every $\mathcal{I}$ of size at most $n + \Delta$ which is at distance at least R from $\mathcal{S}$. The test may or may not distinguish implementations whose distance from $\mathcal{S}$ is less than R. An (R, Δ)-*conformance testing algorithm* is an algorithm that takes $\mathcal{S}$, R and Δ as input and outputs an (R, Δ)-conformance test for $\mathcal{S}$. Note that a $(1, \Delta)$-conformance test is the same as a Δ-conformance test.

Next we present a randomized algorithm that outputs an (R, Δ)-test with high probability. We first present a randomized algorithm which distinguishes a particular "black box" implementation $\mathcal{I}$ from a specification $\mathcal{S}$. We use the notation $i \leftarrow \mathit{Rand}(I)$ to denote that i is chosen uniformly at random from the set I.

The randomized algorithm is based on the intuition that if R is large, then there is a large number of equations, that is, a large subset of Γ, such that the sample corresponding to the access strings generated by these equations distinguishes a particular $\mathcal{I}$ from $\mathcal{S}$. Hence if we choose a sample randomly from Γ, then we catch a buggy implementation with some positive probability. In order to obtain a constant probability, the above step is repeated a certain number of times to boost the probability. The algorithm is given in Algorithm 1.2.

For analyzing the algorithm, we need the following lemma relating the distance R between $\mathcal{S}$ and $\mathcal{I}$ to the number of transitions out of a safe set of states. Let $E[Q'']$ denote the set of edges going out of Q'', that is, an edge (q, a) is in $E[Q'']$ if $q \in Q''$, $a \in \Sigma^{int} \cup Q''$ and $\delta(q, a) \notin Q''$.

Lemma 4. *Let $\mathcal{I}$ be at distance at least R from $\mathcal{S}$. Let $Y \supseteq X$ be a set of access strings for $\mathcal{I}$ which is safe. Let Q'' be the states of $\mathcal{I}$ accessed by Y. Then the size of the set $E[Q'']$ is at least R.*

Note that when $L(\mathcal{I}) = L(S)$, no test can distinguish them. So it remains to analyse the probability that the sample output by the algorithm distinguishes $\mathcal{I}$ from $\mathcal{S}$, under the assumption that $\mathcal{I}$ is at distance at least R from $\mathcal{S}$. Let $d = n + \Delta + |\Sigma^{int}|$.

Lemma 5. *Let $\mathcal{I}$ be at distance greater than or equal to $R > 0$ from $\mathcal{S}$. Let $uxv \in L(\mathcal{I})$ iff $uxv \in L(\mathcal{S})$, for every $x \in X$ and $(u, v) \in D_{states_{\mathcal{S}}(x)}$. Probability that for a γ chosen uniformly at random from Γ, Y_γ is an unsafe set, is at least $(\frac{R}{(n+\Delta)d})^{\Delta}$.*

Let $P = (\frac{R}{(n+\Delta)d})^{\Delta}$, where $d = (n + \Delta + |\Sigma^{int}|)$. The next theorem gives the probability that Algorithm 1.2 distinguishes $\mathcal{I}$ from $\mathcal{S}$.

Theorem 3. *Let $\mathcal{I}$ be at distance at least R from $\mathcal{S}$. For any $\epsilon > 0$, the output of Algorithm 1.2 distinguishes $\mathcal{I}$ from $\mathcal{S}$ with probability at least $1 - \epsilon$ after $k = \frac{1}{P}\log(\frac{1}{\epsilon})$ iterations. The length of the sample output by the algorithm is $O((a2^{\Delta} + b)(n + \Delta)dz(\frac{(n+\Delta)d}{R})^{\Delta})$, where a is the maximum length of the strings in X which is $O(2^n)$, b the maximum length of $|u| + |v|$ for any pair $(u, v) \in D$ which is $O(2^n)$, $z = \max_{\hat{q}\in\hat{Q}} |D_{\hat{q}}|$ which is $O(n)$.*

Note that given any $\mathcal{I}$, the output of the algorithm distinguishes $\mathcal{I}$ from $\mathcal{S}$ with high probability. However, it does not guarantee that the output of the algorithm distinguishes every $\mathcal{I}$ from $\mathcal{S}$ with high probability. Next we modify the algorithm by increasing the number of iterations k so that the output of the algorithm distinguishes every $\mathcal{I}$ from $\mathcal{S}$ or in other words is a conformance test. Note that in the case of a deterministic conformance testing algorithm the two are the same, that is, if the output of the algorithm distinguishes $\mathcal{I}$ from $\mathcal{S}$ where $\mathcal{I}$ is unknown, then it is a conformance test, i.e., it distinguishes every $\mathcal{I}$ from $\mathcal{S}$.

Let α be the number of faulty implementations, that is, $\mathcal{I}$ with at most $n+\Delta$ states at distance at least R from $\mathcal{S}$. α is upper bounded by $(n+\Delta)^{(n+\Delta)d}$, total number of implementation machines with at most $n+\Delta$ states. Set $k = \frac{1}{P}\log(\frac{\alpha}{\epsilon})$. Then the output of the algorithm distinguishes a particular $\mathcal{I}$ with probability $1 - \epsilon/\alpha$. So the probability that the output of the algorithm distinguishes every $\mathcal{I}$ is at least $1 - \epsilon$.

Theorem 4. *For any $\epsilon > 0$, the output of Algorithm 1.2 is an (R, Δ)-conformance test with probability at least $1 - \epsilon$, when $k = \frac{1}{P}\log(\frac{\alpha}{\epsilon})$. The length of the (R, Δ)-conformance test output by the algorithm is $O((a2^{\Delta} + b)(n + \Delta)^2 d^2 z \log(n +$*

$\Delta)(\frac{(n+\Delta)d}{R})^{\Delta})$, *where a is the maximum length of the strings in X which is $O(2^n)$, b the maximum length of $|u|+|v|$ for any pair $(u,v) \in D$ which is $O(2^n)$, $z = \max_{\hat{q}\in\hat{Q}} |D_{\hat{q}}|$ which is $O(n)$.*

5 Lower Bounds for Conformance Testing

We will first define the specification and implementation machines involved in the proof of the lower bound that we wish to establish, and prove some properties about these automata.

5.1 Specification *MVPA*

Given an $n > 1$ and Σ^{int} containing a, we define an $(n+4)$ state *MVPA* $\mathcal{S}(n, \Sigma^{int})$ over the signature $\langle M, \{P_m\}_{m\in M}, m_0, \hat{\Sigma}\rangle$, where $M = \{m_0, m_1\}$, $P_{m_0} = \{p_0\}, P_{m_1} = \{p_1\}$, and $\hat{\Sigma} = \{(m_0,p_0),(m_1,p_1),r\} \cup \Sigma^{int}$. Figure 2 gives a diagram of $\mathcal{S}(n,\Sigma^{int})$, all transitions not shown are assumed to go to a fail state in the corresponding module. When n and Σ^{int} is clear from the context, we refer to $\mathcal{S}(n,\Sigma^{int})$ as just $\mathcal{S}$.

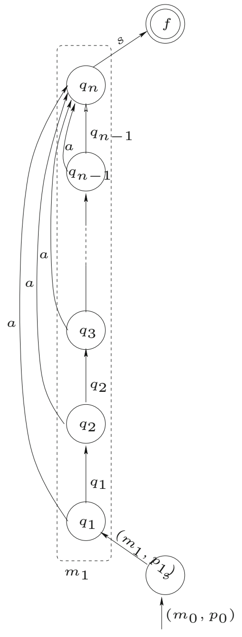


Fig. 2. Specification *MVPA* $\mathcal{S}(n,\Sigma^{int})$

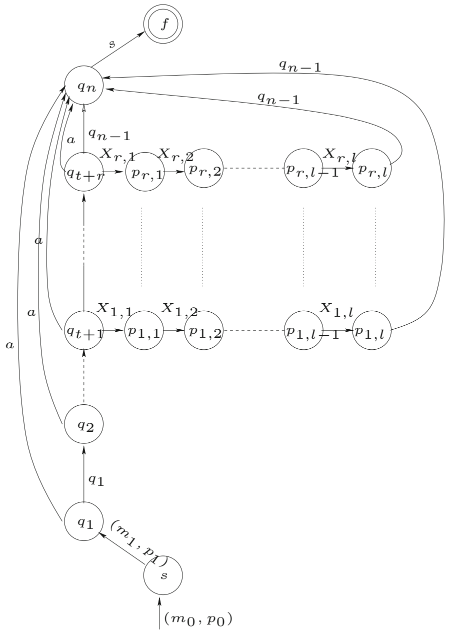


Fig. 3. Implementation *MVPA* $\mathcal{I}(X)$

$\mathcal{S} = (\{Q_m, \{q_m^p\}_{p\in P_m}, \delta_m\}_{m\in M}, F)$ where $Q_{m_0} = \{s,f,d\}$ and $Q_{m_1} = \{q_1,\cdots,$ $q_n, d'\}$. From the start state s, there is a transition on call (m_1,p_1) to q_1 in the module m_1, and a return transition from q_n in module m_1 to the only accepting state f in module m_0. All the other transition on s and f go to the state d (for dead state). Inside module m_1, there is a return transition $(q_i, q_i, q_{i+1}) \in \delta^{ret}$ for every $1 \leq i < n$. Note that all call transitions on (m_1, p_1) go to q_1. The rest of the transitions in module m_1 go to the dead state d'. From now on we will refer to (m_1, p_1) as c.

Proposition 2. *$L(\mathcal{S}(n, \Sigma^{int})) = \{(m_0, p_0)cw_iar \mid 1 \leq i < n\} \cup \{(m_0, p_0)cw_nr\}$, where w_i is defined inductively as:*

- *$w_1 = \epsilon$, and*
- *$w_i = w_{i-1}cw_{i-1}r$, for $i > 1$.*

5.2 Lower Bound for the (R, Δ)-Conformance Test

In this section, we define the implementation machines and prove the lower bound on the length of the conformance test.

We define a class of implementation machines with $n+4+\Delta$ states which are at distance R from $\mathcal{S}(n, \Sigma^{int})$. Hence these machines take as parameters n, Δ, R and Σ^{int}. Let us fix these parameters for this section. Let us also assume that $\Delta = lR$, where $l \in \mathbb{N}$. We define a template for the implementation machine, which when initialized by appropriate values gives us a class of *MVPA*s. Let $X = \{X_{i,j}\}_{i \in [R], j \in [l]}$ be a set of variables, which are labels of the transitions in the implementation machines we define.

We now define $\mathcal{I}(X)$. $\mathcal{I}(X)$ has the same signature as $\mathcal{S}$ and has all the states of $\mathcal{S}$. In addition, in the module m_1, it has Δ extra states. All the transitions consisting of states common to $\mathcal{S}$ and $\mathcal{I}$ are the same except for those going to the dead states. For each state q_{t+i} where $t = n - (R+1)$ and $i \in [R]$, there is a sequence of l states $p_{i,1}, p_{i,2}, \cdots, p_{i,l}$ such that there is a transition from q_{t+i} to $p_{i,1}$ labelled $X_{i,1}$ and for each $j \in [l-1]$, there is a transition from $p_{i,j}$ to $p_{i,j+1}$ on $X_{i,j+1}$. Finally there is a transition from $p_{i,l}$ to q_n on q_{n-1}. This automaton is shown in Figure 3.

A valuation V for X assigns to every $X_{i,j}$ in X, a symbol from the alphabet of the automaton. A valuation V is *valid* if it satisfies the following constraints:

- If j is even, then $V(X_{i,j}) = p_{i,j-1}$ for every i.
- If j is odd, then there is some $a \in \Sigma^{int}$ such that for every i, $V(X_{i,j}) = a$, or there is some $k \geq 1$ such that $n - kR \in \{\lfloor \frac{n-1}{2} \rfloor, \cdots, n-1\}$ such that for every i, $V(X_{i,j}) = n - kR + (i-1)$, or there is an $\hat{j} < j$ such that for every i, $V(X_{i,j}) = p_{i,\hat{j}}$. Also $X_{i,1} \notin \{a, q_{t+i}\}$.

By $\mathcal{I}(V)$ we mean the implementation machine with the variables in X replaced by the corresponding symbol from the alphabet. We will assume from now on that V is valid.

Next we will prove some properties about $\mathcal{I}$.

Proposition 3. *$\mathcal{I}(V)$ is at distance at least R from $\mathcal{S}$.*

Language of $\mathcal{I}(V)$. Language of $\mathcal{I}(V)$ consists of the words from the language of $\mathcal{S}$ and R new words $u_1, \cdots, u_R$ defined as follows. Let $w_1, \cdots, w_n$ be the unique well-matched words which reach q_i from q_1 given in Proposition 2. Then for every $i \in [R]$ and $j \in [l]$, we define a word $u_{i,j}$ inductively as follows.

- Case $j = 1$:
  - If $V(X_{i,j}) \in \Sigma^{int}$, then $u_{i,j} = w_i V(X_{i,j})$.
  - If $V(X_{i,j}) = q_k$, then $u_{i,j} = w_k c w_i r$.
- Case $j > 1$:
  - If $V(X_{i,j}) \in \Sigma^{int}$, then $u_{i,j} = u_{i,j-1} V(X_{i,j})$.
  - If $V(X_{i,j}) = q_k$, then $u_{i,j} = w_k c u_{i,j-1} r$.
  - If $V(X_{i,j}) = p_{i,\hat{j}}$ for some $\hat{j} < j$, then $u_{i,j} = u_{i,\hat{j}} c u_{i,j-1} r$.

Now we set u_i to be $(m_0, p_0) c w_{n-1} c u_{i,l} r r$.

Proposition 4. *$L(\mathcal{I}(V))$ is a union of $L(\mathcal{S})$ and $\{u_1, \cdots, u_R\}$.*

Proposition 5. *If V_1 and V_2 are two different valid valuations, then $L(\mathcal{I}(V_1)) \cap L(\mathcal{I}(V_2)) = L(\mathcal{S})$. Also no string in $L(\mathcal{I}(V_1)) \backslash L(\mathcal{S})$ is a prefix of a string in $L(\mathcal{I}(V_2)) \backslash L(\mathcal{S})$.*

Proposition 6. $|u_i| \geq 2^{n-R+\frac{\Delta}{2R}-5}$.

Proposition 7. *The number of distinct valid valuations is at least*

$$\prod_{i=1}^{\lfloor \frac{\Delta}{2R} \rfloor - 1} (\lfloor \frac{n-1}{2R} \rfloor + |\Sigma^{int}| + i)$$

Using the above facts, we obtain the following theorem:

Theorem 5. *For every n, Δ, Σ^{int} and $R < n$, there is a specification MVPA $\mathcal{S}$ of size n such that any (R, Δ)-conformance test has at least $\prod_{i=1}^{\lfloor \frac{\Delta}{2R} \rfloor - 1} (\lfloor \frac{n-5}{2R} \rfloor + |\Sigma^{int}| + i)$ strings each of length at least $2^{n-R+\frac{\Delta}{2R}-9}$. Hence the length of the (R, Δ)-conformance test is at least*

$$2^{n-R+\frac{\Delta}{2R}-9} \prod_{i=1}^{\lfloor \frac{\Delta}{2R} \rfloor - 1} (\lfloor \frac{n-5}{2R} \rfloor + |\Sigma^{int}| + i).$$

Discussion. The lower bound as given by Theorem 5 on the size of the (R, Δ)-conformance test is $\Omega((2^{n+\frac{\Delta}{R}-R})(\frac{n}{R} + |\Sigma^{int}| + \frac{\Delta}{R})^{\frac{\Delta}{R}})$ and the upper bound as given by Theorem 4 is $O((2^{n+\Delta})(\frac{n+\Delta+|\Sigma^{int}|}{R})^{\Delta})$. Note that when R is $O(1)$, the upper and the lower bounds match.

6 Conclusions

We investigated the problem of constructing (R, Δ) conformance tests for modular VPAs. We presented a randomized algorithm for constructing such tests, that outputs a test suite which is an (R, Δ) conformance test with high probability. We also presented lower bound proofs that demonstrate that our algorithm is close to optimal. One interesting open problem is to tighten the gap between the lower bound and the upper bound. Another line research would be to explore the connections between (R, Δ) tests and learning [3] and model checking [12,5].

References

1. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of Recursive State Machines. ACM Transactions on Programming Languages and Systems 27(4), 786–818 (2005)
2. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the ACM Symposium on Theory of Computation, pp. 202–211 (2004)
3. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the Correspondence Between Conformance Testing and Regular Inference. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 175–189. Springer, Heidelberg (2005)
4. Friedman, A., Menon, P.: Fault Detection in Digital Circuits. Prentice Hall (1971)
5. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. Logic Journal of the IGPL 14(5), 729–744 (2006)
6. Kohavi, Z.: Switching and Finite Automata Theory. McGraw Hill (1978)
7. Kumar, V., Madhusudan, P., Viswanathan, M.: Minimization, Learning, and Conformance Testing of Boolean Programs. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 203–217. Springer, Heidelberg (2006)
8. Kumar, V., Viswanathan, M.: Conformance testing in the presence of multiple faults. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, pp. 1136–1145 (2005)
9. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines; A survey. Proceedings of the IEEE 84, 1090–1126 (1996)
10. Linn, R., Üyar, M. (eds.): Conformance Testing methodologies and architechtures for OSI protocols. IEEE Computer Society Press (1995)
11. Moore, E.F.: Gedanken-experiments on sequential machines. Automata Studies, Annals of Mathematics Studies 34, 129–153 (1956)
12. Peled, D., Vardi, M., Yannakakis, M.: Black Box Checking. Journal of Automata, Languages, and Combinatorics 7(2), 225–246 (2002)
13. Tretmans, J.: A formal approach to conformance testing. In: Protocol Test Systems. IFIP Transactions, vol. C-19, pp. 257–276 (1994)
14. Vasilevskii, M.P.: Fault diagnosis of automata. Kibernetika 4, 98–108 (1973)

Knowledge-Based Distributed Conflict Resolution for Multiparty Interactions and Priorities[⋆]

Saddek Bensalem, Marius Bozga, Jean Quilbeuf, and Joseph Sifakis

UJF-Grenoble 1 / CNRS VERIMAG UMR 5104, Grenoble, F-38041, France

Abstract. Distributed decentralized implementation of systems of communicating processes raises non-trivial problems. Correct execution of multiparty interactions, subject to priority rules, requires sophisticated mechanisms for runtime conflict detection and resolution. We propose a method for detection of false conflicts which combines partial observation of the system's state and apriori knowledge extracted from invariants. We propose heuristics for determining optimal sets of observations leading to implementations with particular guarantees. We provide preliminary experimental results on an implementation of the method in the BIP framework.

Keywords: Distributed System, Priorities, Knowledge, Partial Observation, Multiparty Interactions.

1 Introduction

Systems of communicating processes are a very common model for concurrent systems. Processes have their own data space and can interact by executing interactions, which are atomic synchronization operations involving a simultaneous state change of the set of the processes involved. The meaning of interactions can be specified compositionally by using operational semantics. Specifications are given in the form of rules. The premises include facts about the capabilities of individual processes to execute an action. The conclusion describes interactions, that is, transitions of the system obtained as the composition of actions executed by individual processes. In addition to interactions, operational semantics rules can be used to express priorities between interactions. These are parameterized by a priority order between interactions. They express the fact that some interaction may be executed only if interactions of higher priority are disabled. Priorities are instrumental for specifying scheduling policies [1].

The distributed implementation of systems of communicating processes raises several non trivial problems. It should be consistent with the operational semantics of multiparty interaction which assumes knowledge of the global system

⋆ The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 248776 (PRO3D) and no 257414 (ASCENS) and from ARTEMIS JU grant agreement ARTEMIS-2009-1-100230 (SMECY).

states. Furthermore, multiparty interactions should be replaced in distributed implementations by protocols based on asynchronous message passing.

The BIP component framework [2] allows the construction of composite components from a set of atomic components by using layered parameterized composition of two types of operators: 1) operators parameterized by a set of interactions which are sets of ports of the atomic components that must synchronize; 2) operators parameterized by priorities between interactions. We proposed a distributed implementation method involving a set of transformations from the initial global state model with multiparty interactions to a distributed model that can be directly implemented [3,4]. This method has been extended to handle priorities [5]. The target model consists of components representing processes and interactions representing asynchronous message passing. Correct coordination is achieved through additional components implementing conflict resolution protocols that resolve two types of potential conflicts:

1. The first type of conflicts is symmetric conflicts between interactions. Such conflicts arise when two interactions a and b involve a common component. Since execution of interactions is atomic, execution of interaction a requires ensuring that b will not take place concurrently. Thus execution of a requires permission from some conflict resolution protocol.
2. The second type of conflicts is asymmetric conflicts between interactions related by priorities. Execution of interaction a dominated by interaction b is allowed only if some conflict resolution protocol ensures that b is not enabled.

Conflict resolution protocols are solicited for all potential statically computed conflicts according to a structural analysis of a BIP composite component. This may lead to huge implementation overhead for systems with large numbers of potentially conflicting interactions. Is it possible to reduce this overhead based on a priori knowledge of the system's dynamics and decide that some potential conflicts are not real conflicts?

We denote by $a\#b$ the fact that there is a potential conflict between interactions a and b. A false conflict state for interaction a is a state where a is enabled and all other interactions b such that $a\#b$ are disabled. In such a state, interaction a can be executed independently without arbitration by the conflict resolution protocol. States where a is in a false conflict are characterized by the predicate $FC_a = EN_a \wedge \bigwedge_{a\#b} \neg EN_b$ where EN_x is the state predicate characterizing all the states from which interaction x can be executed.

The aim of the paper is to study whether partial knowledge of the system's state is sufficient for deciding when a potential conflict is a false conflict. In that case, execution of interactions can directly take place, without arbitration and thus, reduce communication with the conflict resolution protocol for a more efficient implementation. The concept of knowledge [6] has been extensively studied for distributed systems in particular with respect to their ability to execute actions [7]. Distributed Knowledge [8] allows a set of components to "know" that an interaction is in a false conflict. We assume that each interaction a observes the states of a set of components L_a. The knowledge predicate denoted

$K_{L_a}FC_a$ characterizes the states where observing only components in L_a is sufficient to ensure that FC_a holds. In other words, it characterizes states where the distributed knowledge of the set of components L_a allows detection of false conflicts for a. We propose conditions for basic and complete implementation, respectively. In a basic implementation, it is possible to detect for each state at least one amongst the false conflicts of the global state model. In a complete implementation all false conflicts of the global state model are detected.

An interesting problem is to minimize the number of observed components, while achieving either basic or complete detection of false conflicts. To this end, we propose heuristics based on simulated annealing strategy [9].

A predicate φ is known to be true for a partial state observed on components L, that is $K_L\varphi$, if it holds in all reachable global states extending this partial state. However, computing the reachable states of a model is not always tractable. Therefore, we use invariants that over-approximate the set of reachable states. Depending on the invariant used, we obtain different results for the minimization heuristics and different performance for the implementation.

The paper is structured as follows: Section 2 provides a formal definition of the BIP global state semantics. In Section 3, we propose a definition of knowledge in the BIP context and we use it to formalize false conflict detection, and detection levels. We provide in Section 4, heuristics based on a simulated annealing strategy to minimize the number of observed components while ensuring a given detection level. In Section 5, we apply false conflict detection to implementation of priorities. We show results about both heuristics from Section 4 and an actual distributed implementation based on [3,4] that uses false conflicts to implement priority resolution. Finally, we present related work in Section 6, concluding remarks and future work in Section 7.

2 The BIP Framework

In this section, we present BIP[2], a component framework for building systems from a set of atomic components by using two types of composition operators: Interaction and Priority.

Atomic Components. An *atomic component* B is a labelled transition system represented by a tuple (Q, P, T) where Q is a set of *control locations* or *states*, P is a set of *communication ports* and $T \subseteq Q \times P \times Q$ is a set of *transitions*.

Interactions. In order to compose a set of n atomic components $\{B_i = (Q_i, P_i, T_i)\}_{i=1}^n$, we assume that their respective sets of control locations and ports are pairwise disjoint; i.e., for any two $i \neq j$ in $\{1..n\}$, we require that $Q_i \cap Q_j = \emptyset$ and $P_i \cap P_j = \emptyset$. We define the global set $P \overset{def}{=} \bigcup_{i=1}^n P_i$ of ports. An *interaction* a is a set of ports such that a contains at most one port from each atomic component. We denote $a = \{p_i\}_{i \in I}$ with $I \subseteq \{1..n\}$ and $p_i \in P_i$. If a is an interaction, we denote by *support*(a) the set of atomic components that participate in a. This notation is extended to sets of interactions γ, that is, *support*$(\gamma) \overset{def}{=} \bigcup_{a \in \gamma}$ *support*(a).

Priorities. Given a set γ of interactions, we define a priority as a strict partial order $\pi \subseteq \gamma \times \gamma$. We write $a\pi b$ for $(a, b) \in \pi$, to express the fact that a has lower priority than b.

Composite Components. A *composite component* $\pi\gamma(B_1, \ldots, B_n)$ (or simply *component*) is defined by a set of atomic components $\{B_i = (Q_i, P_i, T_i)\}_{i=1}^n$ composed by a set of interactions γ and a priority $\pi \subseteq \gamma \times \gamma$. If π is the empty relation, then we may omit π and simply write $\gamma(B_1, \cdots, B_n)$. A global state q of $\pi\gamma(B_1, \cdots, B_n)$ is defined by a tuple of control locations $q = (q_1, \cdots, q_n)$. The behavior of $\pi\gamma(B_1, \cdots, B_n)$ is a labelled transition system $(Q, \gamma, \rightarrow_{\pi\gamma})$, where $Q = \bigotimes_{i=1}^n Q_i$ and $\rightarrow_\gamma, \rightarrow_{\pi\gamma}$ are the least set of transitions satisfying the rules:

$$\frac{\begin{array}{c} a = \{p_i\}_{i\in I} \in \gamma \\ \forall i \in I.\ (q_i, p_i, q_i') \in T_i \\ \forall i \notin I.\ q_i = q_i' \end{array}}{(q_1, \ldots, q_n) \xrightarrow{a}_\gamma (q_1', \ldots, q_n')}\ [\textsc{inter}] \qquad \frac{\begin{array}{c} q \xrightarrow{a}_\gamma q' \\ \forall a' \in \gamma.\ a\pi a' \implies q \overset{a'}{\nrightarrow}_\gamma \end{array}}{q \xrightarrow{a}_{\pi\gamma} q'}\ [\textsc{prio}]$$

Intuitively, transitions $\rightarrow_\gamma$ defined by rule [INTER] express the behaviour of the component without considering priorities. A component can execute an interaction $a \in \gamma$ iff for each port $p_i \in a$, the corresponding atomic component B_i can execute a transition labelled by p_i. If this happens, a is said to be *enabled*. Execution of a modifies atomically the state of all interacting atomic components whereas all others stay unchanged. The behavior of the component is defined by transitions $\rightarrow_{\pi\gamma}$ defined by rule [PRIO]. This rule restricts execution to interactions which are maximal with respect to the priority order. An enabled interaction a can execute only if no other one a' with higher priority is enabled.

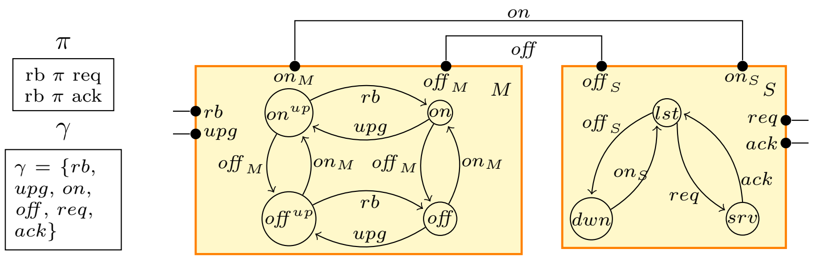


Fig. 1. An example of BIP component. Initial state is $(\mathit{off}, \mathit{dwn})$.

Example 1. A BIP component is depicted in Figure 1 using a graphical notation. It consists of two atomic components named M and S. Component S is a server, that may receive requests (*req*) and acknowledge them (*ack*). Component M is a manager that may perform upgrades (*upg*) and needs to reboot (*rb*) the server for the upgrade to be done. Interactions are represented using lines connecting the interacting ports. There are 4 unary interactions and 2 binary interactions. The component goes up through the interaction *on* and down through *off*, which are both binary interactions. Priorities *rb* π *req* and *rb* π *ack* are used to prevent a reboot whenever a request or an acknowledgement are possible.

Invariants and Reachable States. Let $B = \pi\gamma(B_1, \cdots, B_n)$ be a component. We say that the state q is reachable from a fixed, initial state q_0 if there exist a sequence of interactions $a_1, \cdots, a_k \in \gamma$ and states $q_1, \cdots, q_k$ such that $q_0 \xrightarrow{a_1}_{\pi\gamma} q_1 \xrightarrow{a_2}_{\pi\gamma} \cdots \xrightarrow{a_k}_{\pi\gamma} q_k = q$. We denote by $\mathfrak{R}(B)$ the set of reachable states of B.

An invariant of B is a state predicate $\mathfrak{I}(q)$ satisfied by all its reachable states, that is the characteristic set of $\mathfrak{I}$ contains the set of the reachable states. For a control location $q_i \in Q_i$, we define the predicate $at(q_i)$ which is true (or equal to 1) when the atomic component B_i is at control location q_i. We are interested in two types of invariants that can be generated automatically [10], respectively:

- A *boolean invariant* is a conjunction of boolean constraints of the form $\bigvee_{j \in J} at(q_j)$. For the example of Figure 1, $at(on^{up}) \vee at(on) \vee at(dwn)$ is a boolean invariant. It characterizes a set of control locations such that at each global state, at least one location of the set is active. Such constraints are obtained using methods described in [10].
- A *linear invariant* is a conjunction of linear constraints of the form $\sum_{j \in J} k_j\, at(q_j) = k_0$, where each k_j and k_0 are integer constants. For the example of Figure 1, $at(on^{up}) + at(on) + at(dwn) = 1$ is a linear invariant. Linear invariants are obtained using algebraic methods as described in [11].

The two above categories of invariants are particularly useful for several reasons. First, they provide good approximations for the enabling/disabling conditions of interactions. This has been empirically demonstrated by the successful application of such invariants for checking deadlock-freedom of component-based systems in BIP [10,12]. Second, the methods for computing these invariants are tractable and scale for large systems. Their computation is based on the (interaction) structure of the system and can be done incrementally [13]. In particular, it does not involve fixpoints and avoids state space exploration.

3 Knowledge-Based Detection of False Conflicts

We propose a knowledge-based method for detecting *false conflicts*, that is states where an interaction is enabled and all conflicting interactions are disabled. The enabled interaction can be safely executed without any arbitration. In this section, we consider a component $B = \pi\gamma(B_1, \ldots, B_n)$ and an invariant $\mathfrak{I}$ of B.

3.1 Knowledge and Indistinguishability

The knowledge of a set of atomic components $L \subseteq \{B_1, \cdots, B_n\}$ is the set of the facts that are true by observing the states of these components. The subset L induces an equivalence relation on the global states satisfying $\mathfrak{I}$.

Definition 1 (Indistinguishability Equivalence for L). *Given L, we define the indistinguishability equivalence $\sim_L$ on global states satisfying $\mathfrak{I}$ as $q \sim_L q'$ iff $\forall B_j \in L.\ q_j = q'_j$.*

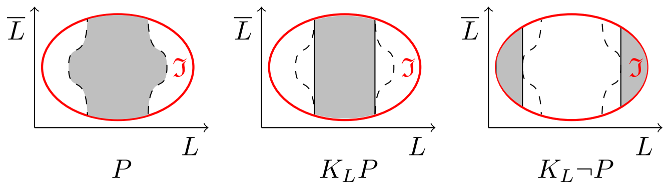


Fig. 2. Knowledge-based approximation of P for observation L, using invariant $\mathfrak{I}$

Intuitively, two states are indistinguishable for L if their restrictions to the states of atomic components of L are identical. The equivalence classes of this relation correspond to sets of global states that can be distinguished by knowing only local states of atomic components satisfying L. Given an invariant $\mathfrak{I}$ and an arbitrary state predicate $P \implies \mathfrak{I}$, we define the predicate "L knows P" as $K_LP(q) = \mathfrak{I}(q) \wedge (\forall q' \; \mathfrak{I}(q') \wedge q' \sim_L q \implies P(q'))$.

Figure 2 illustrates K_LP with respect to P and $\mathfrak{I}$. Each global state within $\mathfrak{I}$ is a point characterized by two coordinates: the projections of this state on the states of L and the states of its complement $\overline{L} = \{B_1, \ldots B_n\} \setminus L$. On the left, the gray region represents the characteristic set of P. In the middle, the gray region represents the characteristic set of "L knows P" that is the set of the global states where observation of their projection on the state space of L suffices to assert "P is true". On the right, the gray region represents the set of the states where "L knows not P" that is the set of the global states where observation on L suffices to assert "P is false".

3.2 Conflict-Free Semantics

For an interaction a, we denote by EN_a the predicate that characterizes the set of global states of $\mathfrak{I}$ where a is enabled. Formally, if $a = \{p_i\}_{i \in I}$ we define $EN_a \overset{def}{=} \bigwedge_{p_i \in a} EN^i_{p_i} \wedge \mathfrak{I}$. By $EN^i_{p_i}$ we denoted the local enabling condition of the port p_i in atomic component $B_i = (Q_i, P_i, T_i)$ that is $EN^i_{p_i} \overset{def}{=} \bigvee_{(q_i, p_i, -) \in T_i} at(q_i)$.

As pointed out in the introduction, we denote by $\#$ a conflict relation between interactions. False conflicts for a correspond to the states where the predicate $FC_a = EN_a \wedge \bigwedge_{a \# b} \neg EN_b$ holds, that is states where a is enabled and all interactions conflicting with a are disabled. We consider executions of the component where only non-conflicting interactions are allowed.

Definition 2 (Conflict-Free Semantics). *Given a component $B = \pi\gamma(B_1,$ $\ldots, B_n)$ and a conflict relation $\#$, we define the* conflict-free semantics *of B as a transition system $(Q, \gamma, \rightarrow_{FC})$, where $\rightarrow_{FC}$ is the least set of transitions satisfying:*

$$\frac{a \in \gamma \qquad FC_a(q) \qquad q \xrightarrow{a}_{\pi\gamma} q'}{q \xrightarrow{a}_{FC} q'}$$

The conflict-free semantics $\rightarrow_{FC}$ is clearly included in the original semantics $\rightarrow_{\pi\gamma}$ of the component. The interest of this semantics is that it captures the set of executions that can be realized without any conflict resolution mechanism. Note that if we consider the priority conflict relation (i.e. we take $\# = \pi$), then $FC_a(q)$ is true only when $q \xrightarrow{a}_{\pi\gamma}$ and $q \overset{b}{\nrightarrow}_{\pi\gamma}$ for all b with higher priority than a. Thus in this particular case $\rightarrow_{FC} = \rightarrow_{\pi\gamma}$. The above semantic rule assumes knowledge of the global state.

3.3 Observational Conflict-Free Semantics

We now propose to restrict the execution semantics presented above by using only a partial observation of the global state. We allow for each interaction a to "observe" a set of atomic components L_a including the atomic components involved in a.

Definition 3 (Observation). *Given an interaction a, an observation is a set of atomic components L_a such that $support(a) \subseteq L_a$.*

Knowledge defines a natural way to describe the false conflicts that can be detected based on an observation. That is, $K_{L_a} FC_a$ characterizes the set of the states where the observation L_a detects that a is in false conflict.

Proposition 1 (Monotoncity). *The predicate $K_{L_a} FC_a$ is monotonic with respect to L_a, i.e. $L'_a \subseteq L_a$ implies $K_{L'_a} FC_a \implies K_{L_a} FC_a$.*

Proof. First, remark that if $L'_a \subseteq L_a$, then $\{q' | q' \sim_{L_a} q\} \subseteq \{q' | q' \sim_{L'_a} q\}$. Then, $K_{L'_a} FC_a(q)$ implies that $\forall q', q' \sim_{L'_a} q,\ FC_a(q')$ and by the above remark $\forall q', q' \sim_{L_a} q,\ FC_a(q')$, that is, $K_{L_a} FC_a(q)$. □

Notice that observing the whole system, that is for $L_a = \{B_1, \ldots B_n\}$, then it is possible to detect all false conflicts, *i.e.* $K_{\{B_1,\ldots B_n\}} FC_a = FC_a$.

We define a new semantics that allows only interactions detected to be in a false conflict. In such a semantics, observation is used to decide whether a conflict-free move is allowed or not.

Definition 4 (Observational Conflict-Free Semantics). *Given a component $B = \pi\gamma(B_1, \cdots, B_n)$, a conflict relation $\#$, and a set of observations $\{L_a\}_{a \in \gamma}$, we define* observational conflict-free semantics *of B as a transition system $(Q, \gamma, \rightsquigarrow)$, where $\rightsquigarrow$ is the least set of transitions satisfying:*

$$\frac{a \in \gamma \qquad K_{L_a} FC_a(q) \qquad q \xrightarrow{a}_{\pi\gamma} q'}{q \overset{a}{\rightsquigarrow} q'}$$

Again, we clearly have $\rightsquigarrow$ included in $\rightarrow_{\pi\gamma}$. However, depending on the observed atomic components, this semantics may not completely implement $\rightarrow_{FC}$. We define two criteria characterizing different levels of false conflict detection, namely basic and complete.

Definition 5 (Detection Level). *A set of observations* $\{L_a\}_{a\in\gamma}$ *is* basic *iff* $\bigvee_{a\in\gamma} K_{L_a}\ FC_a = \bigvee_{a\in\gamma} FC_a$. *A set of observations is* complete *iff for each interaction* $a \in \gamma$: $K_{L_a} FC_a = FC_a$

Theorem [1] below, relates the detection levels and observational conflict-free semantics. Baseness ensures that observational conflicts-free semantics does not introduce deadlocks. Completeness ensures that observational conflict-free semantics corresponds exactly to the conflict-free semantics. This is particularly interesting for the case of priority conflict where the conflict-free semantics is the same as the original semantics of the component.

Theorem 1. *Let* $\pi\gamma(B_1, \cdots, B_n)$ *be a component,* $\#$ *a conflict relation and* $\{L_a\}_{a\in\gamma}$ *a set of observations. Then,* $\rightsquigarrow \subseteq \rightarrow_{FC}$ *and:*

1. *If* $\{L_a\}_{a\in\gamma}$ *is basic, then* $q \in Q$ *is a deadlock for* $\rightsquigarrow$ *only if* q *is a deadlock for* $\rightarrow_{FC}$.
2. *If* $\{L_a\}_{a\in\gamma}$ *is complete, then* $\rightsquigarrow = \rightarrow_{FC}$.

Proof. Since $K_{L_a} FC_a \implies K_{\{B_1,\cdots,B_n\}} FC_a$ (proposition [1]) we have $\rightsquigarrow \subseteq \rightarrow_{FC}$.

1. By contraposition, let $q \in Q$ be a deadlock-free state for $\rightarrow_{FC}$, *i.e.* such that $\exists a \in \gamma$, $FC_a(q)$. Baseness ensures that $\bigvee_{a\in\gamma} K_{L_a} FC_a$ holds and thus $\exists b \in \gamma$ such that $K_{L_b} FC_b(q)$. Thus $q \overset{b}{\rightsquigarrow}$ and q is a deadlock-free state for $\rightsquigarrow$.

2. Assume that $q \xrightarrow{a}_{FC} q'$. Then $FC_a(q)$ and completeness ensures $K_{L_a} FC_a(q)$. Thus $q \overset{a}{\rightsquigarrow} q'$. □

These results characterize to what extent conflict-free semantics, can be captured through partial observation. They can be used in a distributed implementation where the process responsible for executing interaction a can only see the states of atomic components in L_a. However, adding observation increases communication between processes, which may slow down execution. Therefore, we propose in the next section heuristics to minimize the number of observed atomic components, yet ensuring the required detection level.

4 Heuristics for Minimizing Observation

Given a BIP component and a conflict relation, we want to minimize the number of observed atomic components while ensuring either baseness or completeness. For practical reasons, we consider that a single process may coordinate the execution of several interactions. We call such a process an *engine*. All interactions managed by the same engine share a common set of observed atomic components. We consider that the mapping of interactions γ into engines is defined by an arbitrary, fixed partition of γ as $\bigcup_{1\leq j\leq m} \gamma_j$. An engine E_j is used to execute interactions in γ_j while observing a set of atomic components L_j. With these notations, minimizing observation means minimizing the sum $\sum_{j=1}^{m} |L_j|$.

We propose a solution to the minimizing observation problem based on simulated annealing [9]. A pseudo-code for the heuristic is shown in Algorithm [1]. This

Algorithm 1. Pseudo-code of Simulated Annealing

```
Input: An initial solution $init$, a cost function, an alter function.
Output: A solution with a minimized cost.
 1: $sol := init$
 2: $T := T_{max}$
 3: while $T > T_{min}$ do
 4:     $sol' :=$ alter($sol$)
 5:     $\Delta :=$ cost($sol'$) - cost($sol$)
 6:     if $\Delta < 0$ or random() $< e^{\frac{-\Delta}{T}}$ then
 7:         $sol := sol'$
 8:     end if
 9:     $T := 0.99 \times T$
10: end while
11: return $sol$
```

heuristic allows on to search for optimal solutions to arbitrary cost optimization problems. The search through the solution space is controlled by a *temperature* parameter T. At every iteration, temperature decreases slowly (line 9) and the current solution moves into a new, nearby solution still ensuring either baseness or completeness (line 4). If the new solution is better (i.e. observes fewer components), then it becomes the current solution. Otherwise, it may be accepted with a probability that decreases when (1) the temperature decreases or (2) the extra cost of the new solution increases (line 6). The idea is to temporarily allow a bad solution whose neighbors may be better than the current one. By the end of the process, the temperature is low, which prevents bad solutions from being accepted. Now, we provide initial solutions *init* as well as `alter` and `cost` functions that are used to ensure either completeness or baseness.

Ensuring Completeness. According to Definition 5, checking for completeness is performed interaction by interaction, Therefore, minimizing observation can be carried out independently for each engine. Given the set of interactions γ_j we are seeking for a minimal set of atomic components L_j, whose observation ensures complete detection of false conflicts for all interactions in γ_j.

Algorithm 2. Function `alter` for ensuring complete detection of false conflicts

```
Input: A BIP component $B$, a subset of interactions $\gamma_j$, a conflict relation $\#$ and a solution $L_j$.
Output: A solution $L'_j$ that is a neighbor of $L_j$.
 1: $L'_j := L_j$
 2: choose $B_i$ in $L'_j \setminus support(\gamma_j)$
 3: $L'_j := L'_j \setminus \{B_i\}$                          //perturbation
 4: while not complete($L'_j, \gamma_j$) do
 5:     choose $B_i$ in $\{B_1, \ldots, B_n\} \setminus L'_j$
 6:     $L'_j := L'_j \cup \{B_i\}$                           //completion
 7: end while
 8: choose $B'_i$ in $L'_j \setminus support(\gamma_j)$
 9: while complete($L'_j \setminus \{B'_i\}, \gamma_j$) do
10:     $L'_j := L'_j \setminus \{B'_i\}$                     //reduction
11:     choose $B'_i$ in $L'_j \setminus support(\gamma_j)$
12: end while
13: return $L'_j$
```

The initial solution is obtained by taking the set of atomic components involved in interactions conflicting with those of γ_j, that is $init_j = \bigcup_{a \in \gamma_j} (support(a) \cup \bigcup_{a \# b} support(b))$. At each iteration of the simulated annealing, a new solution is computed using the `alter` function shown in Algorithm 2. First, one atomic component is removed from the solution (perturbation), possibly breaking completeness. Then, new atomic components are added randomly until the solution ensures complete detection again (completion). Finally, atomic components are removed randomly (reduction).

After completion and during reduction steps, completeness is ensured by checking the condition $\texttt{complete}(L_j, \gamma_j) \equiv \bigwedge_{a \in \gamma_j} \left(FC_a = K_{L_j} FC_a\right)$. On termination, this ensures that the solution returned by the heuristic is complete.

The cost of the solution is obtained by counting the number of atomic components additionally observed by each engine. That is, for an engine E_j the cost of solution L_j is $\texttt{cost}(L_j) = |L_j \setminus support(\gamma_j)|$.

Ensuring Baseness. Baseness is achieved if for every state which contains false conflicts, at least one engine detects one of them. Baseness is a global property that can be ensured by cooperation between engines. On one hand, allowing an engine E_j to observe additional atomic components may extend the set of false conflicts detected by E_j. On the other hand, reducing observation of E_j, while restricting the set of false conflicts detected, might not necessarily break the baseness. Therefore, the solution $L_1, \cdots, L_m$ to the minimizing observation ensuring baseness cannot be built independently for each engine. Given a partition $\gamma_1, \cdots, \gamma_m$ of the interactions, we build a tuple of sets of atomic components $L = (L_1, \cdots, L_m)$ ensuring baseness.

Algorithm 3. Function `alter` for ensuring basic detection of false conflicts

```
Input: A BIP component $B$, a partition of interactions $\gamma = \bigcup_{1 \le j \le m} \gamma_j$, a conflict relation $\#$ and
    a solution $L = (L_1, \ldots, L_m)$,
Output: A solution $L'$ that is a neighbor of $L$.
 1: $L' := L$
 2: choose $k$ in $[\![1, m]\!]$ and $B_i$ in $L'_k \setminus support(\gamma_k)$
 3: $L'_k := L'_k \setminus \{B_i\}$                                   //perturbation
 4: while not basic($L', \{\gamma_1, \ldots, \gamma_m\}$) do
 5:     choose $k$ in $[\![1, m]\!]$ and $B_i$ in $\{B_1, \ldots, B_n\} \setminus L'_k$
 6:     $L'_k := L'_k \cup \{B_i\}$                                    //completion
 7: end while
 8: choose $k$ in $[\![1, m]\!]$ and $B'_i$ in $L'_k \setminus support(\gamma_k)$
 9: while basic($(L'_1, \ldots, L'_k \setminus \{B'_i\}, \ldots, L'_m), \{\gamma_1, \ldots, \gamma_m\}$) do
10:     $L'_k := L'_k \setminus \{B'_i\}$                              //reduction
11:     choose $k$ in $[\![1, m]\!]$ and $B'_i$ in $L'_k \setminus support(\gamma_k)$
12: end while
13: return $L'$
```

The initial solution assumes that each engine E_j observes all atomic components involved in interactions conflicting with those of γ_j, that is $init = (init_1, \ldots, init_m)$. As for completeness, the `alter` function for baseness presented in Algorithm 3 computes a new solution based on the same three steps (perturbation,completion,reduction) being performed on a tuple of sets of observed atomic components, instead of a single set.

After completion and during reduction steps, baseness is ensured by the condition `basic` $(L, \{\gamma_1, \ldots, \gamma_m\}) \equiv \left(\bigvee_{a\in\gamma} FC_a = \bigvee_{j=1}^{m} \bigvee_{a\in\gamma_j} K_{L_j} FC_a\right)$. This guarantees that the returned solution is basic.

Here the cost of the solution is the sum of the number of additional atomic components observed by each engine. Thus, we define the `cost` function as `cost`$(L) = \sum_{j=1}^{m} |L_j \setminus \mathit{support}(\gamma_j)|$.

5 Implementation and Experiments

In this section, we provide experiments related to detection of false priority conflicts. We apply our simulated annealing heuristics to compute minimal basic or complete solutions for two examples. Then we provide performance gains for the corresponding distributed implementations.

5.1 Dining Philosophers

We consider a variation of the dining philosophers problem, denoted by Philo$N$ where N is the number of philosophers. A fragment of this composite component is presented in Figure 3. In this component, an "eat" interaction eat_i involves a philosopher and the two adjacent forks. After eating, philosopher P_i cleans the forks one by one ($\mathit{cleanleft}_i$ then $\mathit{cleanright}_i$). We consider that each eat_i interaction has higher priority than any $\mathit{cleanleft}_j$ or $\mathit{cleanright}_j$ interaction. We evaluate two different partitions. In Partition 1, there is one engine E_i for every eat_i interaction and one engine C_i for every pair $\mathit{cleanright}_{i-1}$, $\mathit{cleanleft}_i$. Only the latter deals with low priority interactions and therefore may need to observe additional atomic components. Partition 2 is coarser. One engine E_i manages all interactions involving philosopher P_{2i} or P_{2i+1}, for $0 \le i < \lceil N/2 \rceil$. Thus, for N philosophers, there are $\lceil N/2 \rceil$ engines.

Computing complete solutions is done independently for each engine. Table 1 shows results of engine C_0 for Partition 1 and engine E_0 for Partition 2. The total

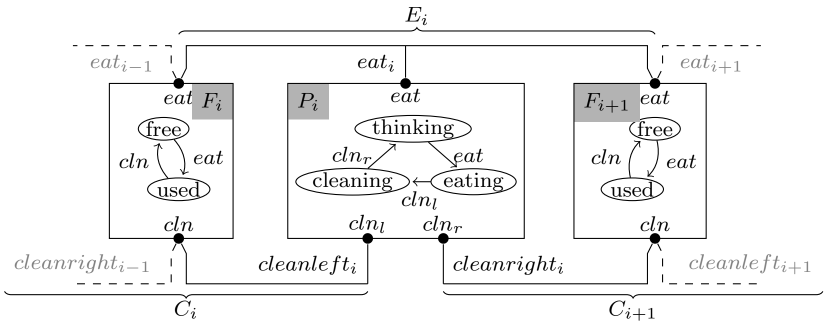


Fig. 3. Fragment of the dining philosopher component. Braces illustrates Partition 1.

Table 1. Minimal observation for completeness

Eng.	**Comp./Part.**	*Size*	*true*	*BI*	*LI*	*optimal*
C_0	Philo3 / 1	6	3	3	1	1
	Philo4 / 1	8	5	5	2	2
	Philo5 / 1	10	7	7	3	3
	Philo10 / 1	20	17	17	8	8
	Philo20 / 1	40	37	37	18	18
	Philo100 / 1	200	197	197	108	98
E_0	Philo3 / 2	6	1	1	0	0
	Philo4 / 2	8	3	3	1	1
	Philo5 / 2	10	5	5	2	2
	Philo10 / 2	20	15	15	7	7
	Philo20 / 2	40	35	35	18	17
	Philo100 / 2	200	195	195	106	97

Table 2. Minimal observation for baseness

Comp./Part.	*Size*	*true*	*BI*	*LI*
Philo3 / 1	6	9	9	0
Philo4 / 1	8	20	20	4
Philo5 / 1	10	35	35	6
Philo10 / 1	20	170	170	23
Philo3 / 2	6	4	4	0
Philo4 / 2	8	6	6	1
Philo5 / 2	10	17	17	3
Philo10 / 2	20	75	75	14

number of atomic components in the composite component is indicated in Column *Size*. Columns *true*, *BI* and *LI* provide the cost of the solutions obtained when using respectively *true*, the boolean invariant and the linear invariant as invariant $\mathfrak{I}$. The column *optimal* indicates the cost of an optimal solution.

$$
\begin{aligned}
& \forall i \in \{0,1,2\}\ (at(F_i.free) \vee at(F_i.used)) && (1)\\
\wedge\ & \forall i \in \{0,1,2\}\ (at(P_i.thinking) \vee at(P_i.eating) \vee at(P_i.cleaning)) && (2)\\
\wedge\ & (at(P_1.eating) \vee at(P_0.eating) \vee at(P_0.cleaning) \vee at(F_1.free)) && (3)\\
\wedge\ & (at(P_2.eating) \vee at(P_1.eating) \vee at(P_1.cleaning) \vee at(F_2.free)) && (4)\\
\wedge\ & (at(P_0.thinking) \vee at(F_0.used) \vee at(P_0.cleaning) \vee at(P_2.thinking)) && (5)\\
\wedge\ & (at(P_0.thinking) \vee at(F_1.used) \vee at(P_1.cleaning) \vee at(P_1.thinking)) && (6)\\
\wedge\ & (at(P_2.cleaning) \vee at(F_0.free) \vee at(P_2.eating) \vee at(P_0.eating)) && (7)\\
\wedge\ & (at(F_1.free) \vee at(F_2.free) \vee at(F_0.free) \\
& \vee at(P_1.eating) \vee at(P_2.eating) \vee at(P_0.eating)) && (8)\\
\wedge\ & (at(F_2.used) \vee at(P_2.cleaning) \vee at(P_1.thinking) \vee at(P_2.thinking)) && (9)\\
\wedge\ & (at(F_2.used) \vee at(P_2.cleaning) \vee at(P_1.thinking) \vee at(F_0.free) \vee at(P_0.eating)) && (10)\\
\wedge\ & (at(F_1.free) \vee at(P_1.eating) \vee at(F_0.used) \vee at(P_0.cleaning) \vee at(P_2.thinking)) && (11)\\
\wedge\ & (at(P_0.thinking) \vee at(F_2.free) \vee at(F_1.used) \vee at(P_2.eating) \vee at(P_1.cleaning)) && (12)
\end{aligned}
$$

Fig. 4. Boolean invariant for the Dining Philosophers example with $N = 3$

Here, the linear invariant gives better results than the boolean invariant, which does not give enough information about the system to reduce observation comparatively to the *true* invariant. For $N = 3$, we provide the boolean and linear invariants respectively in Figures 4 and 5. In this case, the linear constraint (15) in linear invariant ensures that interaction $cleanleft_0$ and interaction eat_1 cannot be enabled concurrently, otherwise, control locations $P_0.eating$ and $F_1.free$ would be active and the sum in constraint (15) would be equal to 2. Thus, the

$$
\begin{array}{lr}
 & (at(P_0.thinking) + at(P_0.eating) + at(P_0.cleaning) = 1) \quad (13)\\
\wedge & \forall i \in \{0,1,2\} \quad (at(F_i.free) + at(F_i.used) = 1) \quad (14)\\
\wedge & (at(P_1.eating) + at(P_0.eating) + at(P_0.cleaning) + at(F_1.free) = 1) \quad (15)\\
\wedge & (at(P_1.thinking) + at(P_0.thinking) + at(F_1.used) + at(P_1.cleaning) = 2) \quad (16)\\
\wedge & (at(P_2.eating) + at(P_0.thinking) + at(F_1.used) + at(P_1.cleaning) + at(F_2.free) = 1) \quad (17)\\
\wedge & (at(P_2.cleaning) + 2 * at(P_0.eating) + at(P_0.cleaning) - at(F_1.used) \\
 & - at(P_1.cleaning) + at(F_2.used) - at(F_0.used) = 0) \quad (18)\\
\wedge & (at(P_2.thinking) - at(P_0.eating) + at(F_0.used) = 1) \quad (19)
\end{array}
$$

Fig. 5. Linear invariant for the Dining Philosophers example with $N = 3$

priority $cleanleft_0 \, \pi \, eat_1$ never forbids execution of $cleanleft_0$. A related boolean constraint, that is constraint (3) of boolean invariant guarantees that at least one of these locations is active. However, this constraint is not strong enough to discard the case where two of them are active.

The results for computing basic solutions are presented in Table 2. The column *Size* contains the total number of atomic components in the composite component. The columns *true*, *BI* and *LI* contains respectively the cost of the solutions obtained when using respectively *true*, the boolean invariant and the linear invariant. For Philo3, baseness is achieved when each engine observes only the components involved in the interactions it handles (i.e. no additional atomic component), therefore the cost is 0.

Performance Evaluation. We used the tool-chain described in [3,4] to generate automatically distributed code from the component. The generated code consists of a set of C++ programs communicating through Unix sockets. We generate one program for each atomic component, one program for each engine and one program for conflict resolution between engines (CRP). We executed these programs in a distributed setting (on a UltraSparcT1 with 24 parallel threads) during 60 seconds and counted the number of "eat" interactions.

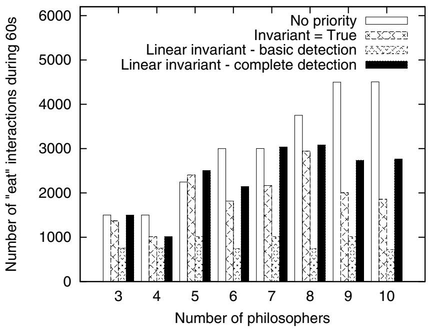


Fig. 6. Performance for different detection levels, using Partition 1

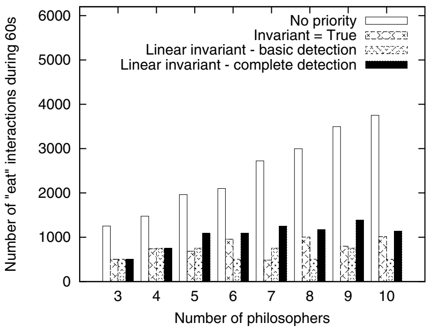


Fig. 7. Performance for different detection levels, using Partition 2

Performance for Partition 1 (resp. Partition 2) is depicted in Figure 6 (resp. 7). We do not show performance for the boolean invariant because it falls back to observing all components, as for the *true* invariant. Since Partition 2 is coarser than Partition 1, it allows less parallelism as shown by comparing performance of execution without priority. Priority limits the number of executions as it enforces a particular scheduling policy and reduces parallelism. For both partitions, the fastest prioritized implementation is the complete one obtained by using the linear invariant. When we observe all involved atomic components (i.e. the invariant is *true*), performance is worse because the lack of knowledge about the reachable states entails more synchronization overhead. Finally, basic solutions are slow because, while restricting the communication, they also restrict the parallelism.

5.2 Jukebox

The second example is a jukebox depicted in Figure 8. It represents a system, where a set of readers $R_1 \dots R_4$ access data located on disks D_1, D_2, D_3. Readers may need to access any disk. Access to disks is managed by jukeboxes J_1, J_2 that can load any disk to make it available to the connected readers. Interactions $load_{i,k}$ and $unload_{i,k}$ allows to load and unload the disk D_i in the jukebox J_k. Each reader R_j is connected to a jukebox through the $read_j$ interaction. Once a jukebox has loaded a disk, it can either take part in a "read" or "unload" interaction. Each jukebox repeatedly loads all 3 disks in a random order.

If unload interactions are always chosen immediately after a disk is loaded, then readers may never be able to read data. Therefore, we add the priority $unload_{i,k} \, \pi \, read_j$, for all i, j, k. This ensures that "read" interactions will take place before corresponding disks are unloaded. Furthermore, we assume that readers connected to J_1 need more often disk 1 and that readers connected to J_2 need more often disk 2. Therefore, loading these disks in the corresponding jukeboxes is assigned higher priority: $load_{i,1} \, \pi \, load_{1,1}$ for $i \in \{2, 3\}$ and $load_{i,2} \, \pi \, load_{2,2}$ for $i \in \{1, 3\}$.

Table 3. Minimal observation cost to ensure baseness or completeness

Interaction	*true*	BI(basic)	BI(complete)	LI(basic)	LI(complete)
$unload_{i,k}$	5	3($k = 1$) or 5($k = 2$)	5	2	2
$load_{i,k}$	1	0	1	0	1

We use a partition assigning one engine per interaction. Results of the simulated annealing heuristic are presented in Table 3. Engines handling a "read" interaction do not need to observe additional atomic components since there is no interaction with higher priority. The boolean invariant allows removing some observed atomic components, in the basic solution. As for Philo$N$ components, the linear invariant is stronger than the boolean invariant. Therefore, the same level of detection is achieved with less observed atomic components.

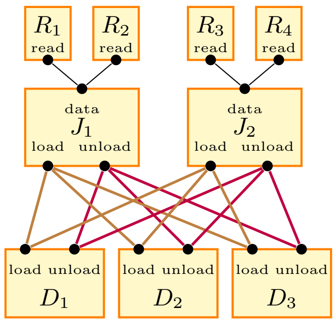


Fig. 8. Jukebox composite component

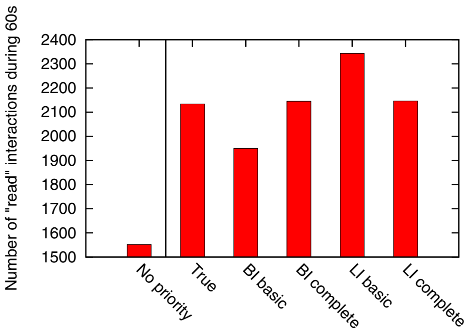


Fig. 9. Performance of the jukebox component for unprioritized and prioritized executions with different invariants/detection levels

Performance Evaluation. We generate C++ code, as for the previous example. We count the number of "read" interactions that take place during 60 seconds of execution, for different settings. In Figure 9, we provide results for a version of the component without priorities, as well as results for the prioritized component for the trivial invariant *true*, the boolean invariant (BI) or the linear invariant (LI). For boolean and linear invariants, we provide performance for both complete and basic implementations.

Notice that adding priority increases the number of "read" interactions executed in 60 seconds. This is due to the fact that a disk is unloaded only if no read is possible, that is only when unload is necessary to progress. Solutions obtained for the boolean invariant require more observation than the ones obtained for the linear invariant, therefore corresponding implementations are slower. More interesting, the best performance is obtained for basic solutions. In that particular case, fewer atomic components are observed which allows more parallelism in the composite component. This parallelism compensates the fact that the detection of false conflicts is not complete.

6 Related Work

Distributed resource conflict resolution boils down to solving the *committee coordination problem* [14], where a set of professors organize themselves in different committees, a meeting requires the presence of all professors to take place and two committees that have a professor in common cannot meet simultaneously. Different solutions have been provided, using managers [14,15,16], a circulating token [17], or a randomized algorithm without managers [18]. Solutions using managers typically rely on a conflict resolution protocol, such as a solution to the dining philosophers problem [19].

Similarly, implementation of priorities needs resolution of asymmetric conflicts. This can be achieved by direct observation as in [20] or [5] where managers observe higher priority interactions to ensure their disabledness. Knowledge is often used to drive action execution in distributed systems. Halpern and Moses [8] defined a logic to reason about the knowledge of system processes. Knowledge is used to control distributed discrete event systems [21] and build distributed controllers for executing multiparty interactions with priorities [22].

In most papers, computing knowledge requires exact computation of reachable states [21,22,8]. Our method overcomes this difficulty by using invariants which are over-approximations of the reachability set. Another common assumption is that the partial state observed by a manager is limited to a neighborhood determined by the architecture of the system [22]. We propose a framework where observation can be adjusted for achieving a certain detection level.

7 Conclusion

Implementing multiparty interactions scheduled by using priorities requires efficient conflict resolution techniques. Most implementations do not distinguish between real and false conflicts to reduce overhead due to conflict resolution.

Dynamic knowledge-based computation of false conflicts based on invariants allows more efficient implementations. We provided simple criteria to define the correctness of the obtained implementation. Baseness ensures preservation of deadlock-freedom and completeness ensures equivalence with the fully-observed model. Finally, the proposed heuristics allow minimization of the number of components to observe and enhanced performance. Heuristics have been applied to non-trivial examples, where the optimal is known, and gave satisfactory results. These have been used for distributed implementation. Experiments show significant performance improvement. However, depending on the model, best performance is achieved either for basic or complete observation.

Future work includes several directions. First, we plan to study in depth how choices of detection levels affect performance of the obtained implementation. We can also consider intermediate levels between basic and complete observation. Such intermediate levels could, for instance, ensure complete detection of false conflicts for some interactions and avoid introduction of deadlocks for the others.

Another improvement is to use static analysis techniques in order to take into account parallelism. These techniques allow automatically computing a partition of the interactions that does not reduce the degree of parallelism by grouping possibly concurrent interactions. The allowed degree of parallelism can also be used to measure the utility of an additional observation, i.e. how observing an additional component can increase parallelism in the obtained implementation.

References

1. Gößler, G., Sifakis, J.: Priority Systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 314–329. Springer, Heidelberg (2004)

2. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Software Engineering and Formal Methods (SEFM), pp. 3–12 (2006)
3. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From high-level component-based models to distributed implementations. In: EMSOFT (2010)
4. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distributed Computing, 1–27, http://dx.doi.org/10.1007/s00446-012-0168-6
5. Bonakdarpour, B., Bozga, M., Quilbeuf, J.: Automated distributed implementation of component-based models with priorities. In: EMSOFT, pp. 59–68 (2011)
6. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press (1995)
7. Halpern, J.Y., Fagin, R.: Modelling knowledge and action in distributed systems. Distributed Computing 3, 159–179 (1988)
8. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM 37, 549–587 (1990)
9. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
10. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.H.: Compositional Verification for Component-Based Systems and Application. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 64–79. Springer, Heidelberg (2008)
11. Krckeberg, F., Jaxy, M.: Mathematical Methods for Calculating Invariants in Petri Nets. In: Rozenberg, G. (ed.) APN 1987. LNCS, vol. 266, pp. 104–131. Springer, Heidelberg (1987)
12. Bensalem, S., Bozga, M., Nguyen, T.H., Sifakis, J.: D-Finder: A Tool for Compositional Deadlock Detection and Verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009)
13. Bensalem, S., Bozga, M., Legay, A., Nguyen, T.H., Sifakis, J., Yan, R.: Incremental component-based construction and verification using invariants. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 256–257 (October 2010)
14. Chandy, K.M., Misra, J.: Parallel program design: a foundation. Addison-Wesley Longman Publishing Co., Inc., Boston (1988)
15. Bagrodia, R.: Process synchronization: Design and performance evaluation of distributed algorithms. IEEE Transactions on Software Engineering (TSE) 15(9), 1053–1065 (1989)
16. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency and Computation: Practice and Experience 16(12), 1173–1206 (2004)
17. Kumar, D.: An implementation of n-party synchronization using tokens. In: ICDCS, pp. 320–327 (1990)
18. Joung, Y.J., Smolka, S.A.: Strong interaction fairness via randomization. IEEE Trans. Parallel Distrib. Syst. 9(2), 137–149 (1998)
19. Chandy, K.M., Misra, J.: The drinking philosophers problem. ACM Transactions on Programming Languages and Systems (TOPLAS) 6(4), 632–646 (1984)
20. Ben-Hafaiedh, I., Graf, S., Quinton, S.: Building distributed controllers for systems with priorities. Journal of Logic and Algebraic Programming 80, 194–218 (2011)
21. Ricker, S., Rudie, K.: Know means no: Incorporating knowledge into discrete-event control systems. IEEE Transactions on Automatic Control 45(9), 1656–1668 (2000)
22. 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)

Modelling Probabilistic Wireless Networks
(Extended Abstract)

Andrea Cerone and Matthew Hennessy

Department of Statistics and Computer Science
Trinity College Dublin
{ceronea,Matthew.Hennessy}@scss.tcd.ie

Abstract. We propose a process calculus to model distributed wireless networks. The calculus focuses on high-level behaviour, emphasising local broadcast communication and probabilistic behaviour.

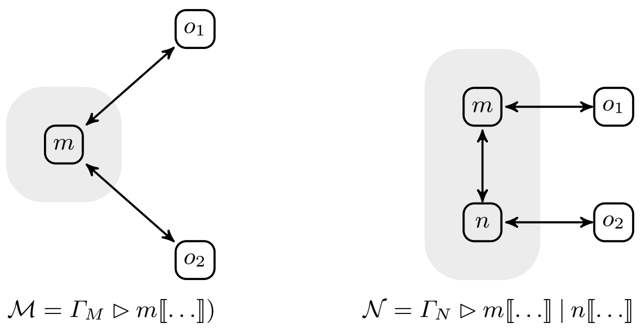
Our formulation of such systems emphasises their *interfaces*, through which their behaviour can be observed and tested, although this complicates their contextual analysis. Nevertheless we propose a novel operator with which networks can be decomposed into components. Using this operator we define probabilistic generalisations of the well-known may-testing and must-testing preorders.

We define an extensional probabilistic labelled transition system in which actions represent particular interactions networks support via their interfaces. We show that novel variations on probabilistic simulations support compositional reasoning for these networks which are sound with respect to the testing preorders. Finally, and rather surprisingly, we show that these simulations turn out not to be complete.

1 Introduction

There is growing interest in the development of formal methods for the analysis of wireless systems, and a number of process calculi have been suggested for describing and analysing their behaviour, [10,11,13]. Our proposal focuses on descriptions at a high-level of abstraction, where for example network nodes use protocols at the *MAC level* [7] to implement reliable communication between nodes; thus we are abstracting from collision prone behaviour. For us a wireless system will take the form $\mathcal{M} = \Gamma \triangleright M$ where Γ describes the network topology, a connected undirected graph of station nodes; some nodes will contain running code, while others will be in the system interface, $\mathsf{Int}(\mathcal{M})$, through which the system may be tested, or indeed composed with peers to form larger systems. The running code at individual stations is described in the component M, using essentially a broadcast version of CCS, [12,15].

However the range of a broadcast from a given station node is determined by the underlying connectivity graph Γ. Further, we allow the code at stations to behave probabilistically. We also assume that a fixed number of communication channels are available to stations to broadcast to their neighbours; it is well-known that multiple access techniques such as *TDMA* and *FDMA* [17] can



o_1
m
o_2
$\mathcal{M} = \Gamma_M \triangleright m[\![\ldots]\!])$
m
o_1
n
o_2
$\mathcal{N} = \Gamma_N \triangleright m[\![\ldots]\!] \,|\, n[\![\ldots]\!]$

Fig. 1. Example networks

be used to implement such virtual channels. In the literature other calculi for modelling wireless systems have been proposed; in particular, our calculus has been inspired by [10,11,8,13]. Recently, there has been also a growing interest in modeling networks with probabilistic behaviour [9,5,6].

Two example systems in our calculus are given in Figure 1; here and henceforth we use shading to denote nodes running code in a network, with the remaining being in the interface. Our goal is to develop behavioural theories for such systems, and associated proof technologies.

Using standard process-calculi techniques we can give an intensional semantics to the set of such (well-formed) networks Nets thereby endowing it with the structure of a probabilistic labelled transition system [16], a pLTS; see Section 3. This is a significant step towards our goal as in [3] behavioural testing preorders have been defined for arbitrary pLTSs, and forms of (probabilistic) simulations have been shown to be both sound and complete with respect to them. However significant problems arise when trying to adapt this approach to Nets.

In [3] a total binary operator $|$ is assumed to exist for arbitrary systems; a system $\mathcal{S}$ is tested by running the combined system $\mathcal{S} \,|\, \mathcal{T}$ and observing the effect on the testing system $\mathcal{T}$; indeed all standard process calculi come equipped with such an operator. However there is no definitive manner in which arbitrary pairs of wireless systems from Nets can be combined, so as to maintain consistency. For example both $\mathcal{S}$ and $\mathcal{T}$ might expect to run their own code at a particular station they have in common; or in the combined system natural well-formedness conditions might be violated. Our intention with such an operator is to implement *black-box testing* of $\mathcal{S}$ by $\mathcal{T}$; so for example $\mathcal{T}$ should have no access to, or indeed knowledge of, the internal stations in $\mathcal{S}$. Instead interaction between $\mathcal{T}$ and the system $\mathcal{S}$ will be restricted to what we will call *the interface* of $\mathcal{S}$.

With this in mind, in Section 4 we propose a novel asymmetric combinator for (well-formed) wireless networks, $\mathcal{S} \not\Vdash\!\!\!> \mathcal{T}$; this in turn leads to formulations of the standard testing pre-orders to Nets, $\mathcal{S}_1 \sqsubseteq_{may} \mathcal{S}_2$ and $\mathcal{S}_1 \sqsubseteq_{must} \mathcal{S}_2$. The asymmetry is necessary; in Theorem 2 we show that if any *reasonable* symmetric combinator were used then the resulting pre-orders would be degenerate.

We then give, in Section 5, an extensional pLTS in which the probabilistic simulations are sound with respect to $\mathcal{S}_1 \sqsubseteq_{may} \mathcal{S}_2$. Here the extensional actions are defined in terms of behaviour, broadcasts and reception of values, which can be detected at the interface of systems, $\mathsf{Int}(\mathcal{S})$. However again this is not simply a straightforward application of the simulations from [3]. A problem arises because in certain situations the broadcast of a message to a set of interface nodes, as might happen in $\mathcal{M}$ of Figure 1 from m to the pair $\{o_1, o_2\}$, can be simulated by a multicast of copies of the message through a series of nodes. In $\mathcal{N}$ from Figure 1 this might happen by a broadcast from m, with reaches the interface node o_1 and the internal node n, followed by a broadcast by n to the second interface node o_2.

We also provide a variation on simulations, in the same extensional pLTS, which are sound with respect to $\mathcal{S}_1 \sqsubseteq_{must} \mathcal{S}_2$. Again results in [3] can not be relied upon. Instead we define a novel notion of *deadlock simulation* for this purpose.

We already know that our notion of simulations are not complete for wireless systems; an example is given at the end of Section 5. Nevertheless we believe that they are powerful enough to treat non-trivial case studies. In Section 6 we provide one example which shows the way in which they can be used. In future we intend to evaluate more fully our proposed methodology.

In this extended abstract we omit all proofs, and some technical definitions are also elided. Full details are available in the accompanying technical report [1], together with some more illustrative examples.

2 Background

Recall that a probability distribution Δ over a set S is a function $\Delta : S \to [0,1]$ such that $\sum_{s\in S} \Delta(s) = 1$. Given a set S, we use $\mathcal{D}(S)$ to denote the set of probability distributions over S.

Definition 1. A *probabilistic labelled transition system* (pLTS) is a 4-tuple $\langle S, \mathsf{Act}_\tau, \to, \omega\rangle$, where

(i) S is a set of states,
(ii) Act_τ is a set of transition labels with a distinguished label τ,
(iii) the relation $\to$ is a subset of $S \times \mathsf{Act}_\tau \times \mathcal{D}(S)$,
(iv) $\omega : S \mapsto \{$ true , false $\}$ is a (success) predicate over the states S.

As usual, we will write $s \xrightarrow{\mu} \Delta$ in lieu of $(s, \alpha, \Delta) \in \longrightarrow$. It is finitary if S is finite and for every $s \in S$, the set $\{\, \Delta \mid s \xrightarrow{\mu} \Delta$ for some $\mu \in \mathsf{Act}_\tau \,\}$ is finite. □

This definition of pLTS is slightly different from that provided in [3], for we have introduced a success predicate ω over states, which will be used when testing processes.

We use standard notation, borrowed from [3], for distributions and operations on them. We use Δ, Θ to range over probability distributions; $\lceil \Delta \rceil$ represents the *support* of Δ, that is all states such that $\Delta(s) > 0$ while $\overline{s}$ denotes the one point distribution for an $s \in S$. We will also have a minor need for *sub-distributions*, with $\mathcal{D}_{sub}(S)$ representing the sub-distributions over S; for $\Delta \in \mathcal{D}_{sub}(S)$, the quantity $\sum_{s \in S} \Delta(s)$, called the size of the sub-distribution, may be strictly less then 1.

Definition 2 (Lifted Relations). *Let $\mathcal{R} \subseteq S \times \mathcal{D}_{sub}(S)$ be a relation from states to subdistributions. Then $\overline{R} \subseteq \mathcal{D}_{sub}(S) \times \mathcal{D}_{sub}(S)$ is the smallest relation which satisfies*

– *$s \,\mathcal{R}\, \Delta$ implies $\overline{s}\, \overline{\mathcal{R}}\, \Delta$*
– *If I is a finite index set and $\Delta_i\ \overline{\mathcal{R}}\ \Theta_i$ for each $i \in I$ then $(\sum_{i \in I} p_i \cdot \Delta_i)\ \mathcal{R}\ (\sum_{i \in I} p_i \cdot \Theta_i)$ whenever $\sum_{i \in I} p_i \leq 1$.* $\square$

Lifting of relations can also be defined for full distributions, by simply requiring $\sum_{i \in I} p_i = 1$ in the last constraint of the definition above.

In a pLTS $\langle S, \mathsf{Act}_\tau, \rightarrow, \omega\rangle$, each transition relation $\xrightarrow{\mu} \subseteq S \times \mathcal{D}(S)$ can be lifted to $\overline{(\xrightarrow{\mu})} \subseteq \mathcal{D}(S) \times \mathcal{D}(S)$. With an abuse of notation, the latter is still denoted as $\xrightarrow{\mu}$.

Lifted transition relations allow us to reason about the behaviour of pLTSs in terms of sequences of transitions. We also need to formalise internal computations, indefinite sequences of τ actions. We employ the infinitary version $\Delta \Longrightarrow \Delta'$ from [3], in which states from the support of Δ may at any point decide to stop performing τ actions; in general Δ' may turn out to be a sub-distribution, rather than a distribution; this is because part of a distribution may never stop performing τ-actions. A minor variation, $\Delta \Longrightarrow\!\!\succ \Delta'$, insists that states must continue performing τ actions so long as they are able; intuitively $\Delta \Longrightarrow\!\!\succ \Delta'$ may be viewed as a probabilistic version of a maximal computation from Δ. The formal definitions are relegated to the appendix; note the presence of the success predicate ω in a pLTS means that our formulation is a slight generalisation from that in [3].

3 Networks and Their Computations

As explained in the Introduction a wireless system is represented by a pair $\Gamma \triangleright M$ where Γ is an undirected graph representing the connectivity in the underlying network between the wireless stations and M the code running in the individual stations. The language for code is given in Figure 2 and uses standard syntax from process calculi. Basically a system consists of a collection of named nodes at each of which there is some running code, $n[\![s]\!]$. The set of nodes appearing in a system M is denoted by nodes(M).

The process calculus operators $c!\langle v\rangle .p$, $c?(x) .p$ will represent the broadcast and reception of values respectively; the latter is a binding operator for the variable x, and the standard notions of free occurrences of a variable as well as closed system terms arise. As usual, given a list of variables $\tilde{x}$ and a list of closed values $\tilde{v}$ of the same length, we use the notation $p\{\tilde{v}/\tilde{x}\}$ to denote process p where

$M,\ N ::=$		**Systems**
	$n[\![s]\!]$	Nodes
	$M \mid N$	Composition
	$\mathbf{0}$	Identity

$p,\ q ::=$		(probabilistic) Processes
	s	
	$p\ {}_{p}\!\oplus\ q$	probabilistic choice

$s,\ t\ ::=$		States
	$c!\langle e\rangle\,.p$	broadcast
	$c?(x)\,.p$	receive
	$\omega.\,\mathbf{0}$	test
	$s + t$	choice
	$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t$	branch
	$\tau.p$	internal activities
	$A(\tilde{x})$	calls
	$\mathbf{0}$	terminate

Fig. 2. Syntax

the free occurrences of a variable x appearing in $\tilde{x}$ is replaced with the respective closed value v appearing in $\tilde{v}$. We also assume a set of process definitions of the form $A(\tilde{x}) \Leftarrow p$, meaning that the definition $A(\tilde{v})$ can be unfolded in $p\{\tilde{v}/\tilde{x}\}$.

The effect of a broadcasts is determined by the underlying network Γ. For example if a value is broadcast from the station n then it can only be received at stations m connected to n in Γ; that is those m such that $(n, m) \in \Gamma_E$ where Γ_E is the set of edges of Γ. For this, and similar concepts, we tend to use more graphic notation such as $\Gamma \vdash n \leftrightarrow m$.

We only consider the sublanguage of well-formed terms, in which each node name has at most one occurrence, and we use sSys to denote the set of all well-formed terms which are closed, meaning that they have no free occurrences of a free variables. Nodes appearing in nodes(M) in a network $\Gamma \triangleright M$ are called internal, in contrast with nodes in $\Gamma_V \setminus$ nodes(M) which are called external. The set $\Gamma_V \setminus$ nodes(M) is also called the interface of the network, denoted as $\mathsf{Int}(\Gamma \triangleright M)$. A network $\Gamma \triangleright M$ is well-formed if:

(i) $M \in \mathsf{sSys}$
(ii) nodes$(M) \subseteq \Gamma_V$, where Γ_V denotes the set of nodes in Γ
(iii) whenever $k \in \mathsf{Int}(\Gamma \triangleright M)$, there exists some $m \in$ nodes(M) such that $\Gamma \vdash k \leftrightarrow m$
(iv) whenever $k_1, k_2 \in \mathsf{Int}(\Gamma \triangleright M)$, $\Gamma \vdash k_1 \not\leftrightarrow k_2$.

Most of these conditions are natural; in particular, requirements (iii) and (iv) establish that internal nodes (that is, nodes running code) in a network have knowledge of the nodes in the external environment to which they are connected, but they have no information about how these nodes are interconnected. Requirement (iv) is also necessary for the soundness of our proof methodologies. We use Nets to denote the set of well-formed networks, in the sequel ranged over by $\mathcal{M}, \mathcal{N}, \ldots$. We will also use some obvious notation, such as nodes$(\mathcal{M})$ to denote the set of nodes running code in $\mathcal{M}$.

Example 1. Consider $\mathcal{M} = \Gamma_M \triangleright M$ described in Figure 1, where M denotes the code $m[\![\tau.(c!\langle v\rangle\,.\,\mathbf{0}\,{}_{0.81}\!\oplus\,\mathbf{0})]\!]$. Intuitively in this network, the station m, after performing some internal computation can broadcast a value v along channel c with probability 0.81. This message can be detected by the interface nodes o_1 and o_2.

Consider now network $\mathcal{N} = \Gamma_N \triangleright N$, in the same figure, where N denotes $m[\![\tau.(c!\langle v\rangle\,{}_{0.9}\!\oplus\,\mathbf{0})]\!] \mid n[\![P]\!]$, and $P \Leftarrow c?(x)\,.(c!\langle x\rangle\,{}_{0.9}\!\oplus\,\mathbf{0}) + c?(x)\,.P$. Here station m broadcasts the value v along channel c with probability 0.9; this message can be detected by the interface node o_1 and the internal station n. Station n, upon receiving the message, decides to forward it with probability 0.9; since the nodes in the range of n are m and o_2, these are the nodes which will detect the value broadcast by n. Therefore, the probability of the original broadcast performed by station m reaching both the interface nodes o_1 and o_2 is 0.81, the same as in the network $\mathcal{M}$.[1] Note also that node n can non-deterministically decide to ignore broadcasts along channel c which can be received either by node m or by the interface node o_2. This ensures that the network $\mathcal{N}$ has a computation in which its behaviour is not affected by the external nodes o_1, o_2. Informally speaking, the network $\mathcal{N}$ is more reliable than the network $\mathcal{M}$, when the latter is viewed optimistically. □

Judgements in the formal intensional semantics of networks take the form $\Gamma \triangleright M \xrightarrow{\mu} \Delta$, where Δ is a distribution over sSys and μ can take one of the forms: $n.\tau$ internal computation at node n, $c.n?v$ reception from node n of value v or $n.c!v$, transmission from node n.

The rules for inferring judgements are given in Figure 3 and they rely on a pre-semantics for the states s from Figure 4. These in turn take the form $s \xrightarrow{\mu} p$, where s is a closed state, p is a process and μ is one of the forms $c!v$, $c?v, \tau$ or ω. The deductive rules for inferring these judgements are given in Figure 4 and should be self-explanatory. The main rules in Figure 3 also use some standard notation from [3] for interpreting processes p from Figure 2 as distributions over states, $[\![p]\!]$; this has the obvious definition, namely $[\![s]\!] = \overline{s}$ and $[\![p_1\,{}_p\!\oplus\,p_2]\!] = p\cdot[\![p_1]\!] + (1-p)\cdot[\![p_2]\!]$.

Rule (B-BROAD) models the evolution of a node n which broadcasts value v along channel c. Here the term $n[\![\Delta]\!]$ represents a distribution over sSys, obtained by extending the function $n[\![\cdot]\!]$ in the standard way to distributions. This technique is also used in subsequent rules, for example extending the operator $\mid$ from one on system terms to distributions over system terms.

[1] Also, the probability of message v being only by node o_1 in network $\mathcal{N}$ is 0.9 and 0.81 in network $\mathcal{M}$.

(B-BROAD)
$$\frac{s \xrightarrow{c!v} p}{\Gamma \triangleright n[\![s]\!] \xrightarrow{c.n!v} n[\![\Delta]\!]} [\![p]\!] = \Delta$$

(B-REC)
$$\frac{s \xrightarrow{c?v} p}{\Gamma \triangleright n[\![s]\!] \xrightarrow{c.m?v} n[\![\Delta]\!]} \; [\![p]\!] = \Delta, \Gamma \vdash n \leftrightarrow m$$

(B-DEAF)
$$\frac{s \not\xrightarrow{c?v}}{\Gamma \triangleright n[\![s]\!] \xrightarrow{c.m?v} \overline{n[\![s]\!]}} \; \Gamma \vdash m \leftrightarrow n$$

(B-DISC)
$$\frac{}{\Gamma \triangleright n[\![s]\!] \xrightarrow{c.m?v} \overline{n[\![s]\!]}} \; \Gamma \vdash n \not\leftrightarrow m$$

(B-$\mathbf{0}$)
$$\frac{}{\mathbf{0} \xrightarrow{c.m?v} \overline{\mathbf{0}}}$$

(B-τ)
$$\frac{s \xrightarrow{\tau} p}{\Gamma \triangleright n[\![s]\!] \xrightarrow{n.\tau} n[\![\Delta]\!]} [\![p]\!] = \Delta$$

(B-τ.PROP)
$$\frac{\Gamma \triangleright M \xrightarrow{n.\tau} \Delta}{\Gamma \triangleright M \mid N \xrightarrow{n.\tau} \Delta \mid \overline{N}}$$

(B-PROP)
$$\frac{\Gamma \triangleright M \xrightarrow{c.m?v} \Delta, \; \Gamma \triangleright N \xrightarrow{c.m?v} \Theta}{\Gamma \triangleright M \mid N \xrightarrow{c.m?v} \Delta \mid \Theta}$$

(B-SYNC)
$$\frac{\Gamma \triangleright M \xrightarrow{c.m!v} \Delta, \; \Gamma \triangleright N \xrightarrow{c.m?v} \Theta}{\Gamma \triangleright M \mid N \xrightarrow{c.m!v} \Delta \mid \Theta}$$

Fig. 3. Intensional semantics of networks

Rules (B-REC), (B-DEAF) and (B-DISC) express how a node n reacts when a message is broadcast by a sender node m; if the former is in the range of transmission of the sender, and it is waiting to receive a value along the same channel used by the sender to broadcast, then it will receive the message correctly. In all the other cases the behaviour of node n is not affected by the broadcast performed by m.

The rules (B-τ) and (B-τ.PROP) model internal activities performed by some node of a system term. Finally, rules (B-SYNC) and (B-PROP) describe how communication between nodes of a network is handled; these rules have been defined to model broadcast communication. See [1] for more discussion and some sanity checks on the rules. For example one can show that if $\Gamma \triangleright M \xrightarrow{\mu} \Delta$ can be inferred from the rules then every N in the support of Δ has exactly the same set of node station names as M.

4 Testing Networks

As discussed in the Introduction, in order to test networks we need to be able to compose the network to be tested, say $\mathcal{M}$, with the network performing the test, say $\mathcal{N}$. A natural definition would be to define

$$(\Gamma_M \triangleright M) \mid\mid\mid (\Gamma_N \triangleright N) = (\Gamma_M \cup \Gamma_N) \triangleright (M \mid N) \tag{1}$$

where the combined connectivity graph $\Gamma_M \cup \Gamma_N$ is obtained set theoretically, by the point-wise union of the individual node sets and edge sets. However in

(s-Snd)
$$\frac{}{c!\langle e\rangle.p \xrightarrow{c!v} p}\ \mathrm{val}(e) = v$$

(s-ω)
$$\frac{}{\omega.\mathbf{0} \xrightarrow{\omega} \mathbf{0}}$$

(s-Rcv)
$$\frac{}{c?(x).p \xrightarrow{c?v} p\{v/x\}}$$

(s-τ)
$$\frac{}{\tau.p \xrightarrow{\tau} p}$$

(s-SumL)
$$\frac{s \xrightarrow{\alpha} p}{s + t \xrightarrow{\alpha} p}$$

(s-SumR)
$$\frac{t \xrightarrow{\alpha} p}{s + t \xrightarrow{\alpha} p}$$

(s-then)
$$\frac{s \xrightarrow{\alpha} p}{\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t \xrightarrow{\alpha} p}\mathrm{val}(b) = \mathrm{true}$$

(s-else)
$$\frac{t \xrightarrow{\alpha} p}{\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t \xrightarrow{\alpha} p}\mathrm{val}(b) = \mathrm{false}$$

(s-Unfold)
$$\frac{A(\tilde{x}) \Leftarrow p}{A\langle\widetilde{e}\rangle \xrightarrow{\tau} p\{\widetilde{e}/\widetilde{x}\}}$$

Fig. 4. Pre-semantics of states

general this will lead to ill-defined networks. Therefore we have to be satisfied by partial composition operators; moreover we should only use a composition operators which reflect in some way the practical manner in which the tester $\mathcal{N}$ can realistically interact with the testee $\mathcal{M}$.

Definition 3 (Network Extension). *The operator $\mathrel{\|\!\triangleright}$ is the partial operator between pairs of networks defined by letting $(\Gamma_M \triangleright M) \mathrel{\|\!\triangleright} (\Gamma_N \triangleright N) = (\Gamma_M \cup \Gamma_N) \triangleright (M \mid N)$ if nodes$(M) \cap (\Gamma_N)_V = \emptyset$, undefined otherwise.* □

This operator is associative but in general not symmetric. In $\mathcal{M} \mathrel{\|\!\triangleright} \mathcal{N}$ the system $\mathcal{N}$ is only allowed to place code at interface of $\mathcal{M}$. In particular it has no access to the internal nodes of $\mathcal{M}$; on the other hand $\mathcal{M}$ can place no code at any node in $\mathcal{N}$. These restrictions are natural if we view $\mathcal{N}$ as testing $\mathcal{M}$ in a black-box manner.

Proposition 1 (Interface Preservation). *If $\mathsf{Int}(\mathcal{M}) = \mathsf{Int}(\mathcal{N})$, and $\mathcal{L}$ is a network such that both $\mathcal{M} \mathrel{\|\!\triangleright} \mathcal{L}$ and $\mathcal{N} \mathrel{\|\!\triangleright} \mathcal{L}$ are defined then $\mathsf{Int}(\mathcal{M} \mathrel{\|\!\triangleright} \mathcal{L}) = \mathsf{Int}(\mathcal{N} \mathrel{\|\!\triangleright} \mathcal{L})$.* □

The operator $\mathrel{\|\!\triangleright}$ is also a universal constructor for (well-formed) networks:

Proposition 2. *Every well-formed network $\Gamma \triangleright M$ such that M is different from $\mathbf{0}$ can be written[2] in the form $\Gamma \triangleright M = (\Gamma' \triangleright M') \mathrel{\|\!\triangleright} (\Gamma'' \triangleright n[\![s]\!])$.* □

[2] Modulo a simple structural equivalence.

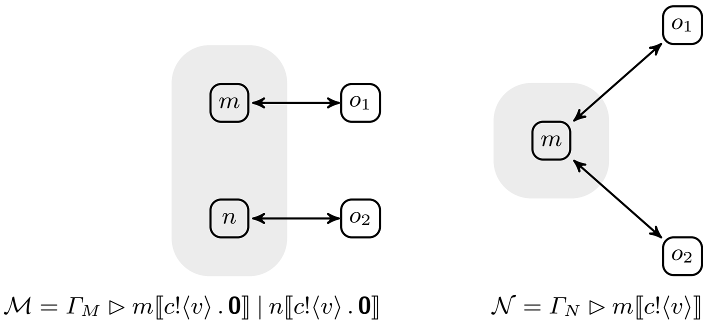

m
n
o_1
o_2
m
o_1
o_2
$\mathcal{M} = \Gamma_M \triangleright m[\![c!\langle v\rangle\,.\,\mathbf{0}]\!] \mid n[\![c!\langle v\rangle\,.\,\mathbf{0}]\!]$
$\mathcal{N} = \Gamma_N \triangleright m[\![c!\langle v\rangle]\!]$

Fig. 5. Broadcast vs. Multicast

We test network $\mathcal{M}$ by considering maximal computations of the composite systems $\mathcal{M} \not\models \mathcal{T}$, where $\mathcal{T}$ is a system designed to elicit certain behaviour from $\mathcal{M}$. This composite testing harness should run in isolation from its environment, for example by ignoring possible broadcasts either $\mathcal{M}$ or $\mathcal{T}$ might receive at their interfaces. So we define a reduction relation $\twoheadrightarrow$ for networks, by letting $(\Gamma \triangleright M) \twoheadrightarrow \Delta$ whenever $(\Gamma \triangleright M) \xrightarrow{m.\tau} \Delta$ or $\Gamma \triangleright M \xrightarrow{c.m!v} \Delta$.

We also define a success predicate $\omega(\cdot)$ for networks by letting $\omega(\mathcal{M}) = \text{true}$ whenever $\mathcal{M} = \Gamma \triangleright (n[\![s]\!] \mid M)$ for some state s such that $s\xrightarrow{\omega}$. Thus we have defined a particular pLTS, as in Definition 1, with a unique transition action $\xrightarrow{\tau}$, which we take to be $\twoheadrightarrow$; such simple pLTSs we refer to as *testing structures*, TSs.

Definition 4 (Tabulating Results). *The* value *of a sub-distribution in a TS is given by the function $\mathcal{V} : \mathcal{D}_{sub}(S) \to [0,1]$, defined by $\mathcal{V}(\Delta) = \sum\{\, \Delta(s) \mid \omega(s) = \mathsf{true}\,\}$. Then the set of possible results from a state s is given by $\mathcal{R}(s) = \{\, \mathcal{V}(\Delta') \mid \Delta \Longrightarrow\!\!\succ \Delta' \,\}$. Recall from page 138 that $\overline{s} \Longrightarrow\!\!\succ \Delta'$ represents a (probabilistic) maximal computation from s.* □

Definition 5 (Testing Networks). *We write*

(i) $\mathcal{M} \sqsubseteq_{may} \mathcal{N}$ if for every system $\mathcal{T}$ such that both $\mathcal{M} \not\models \mathcal{T}$ and $\mathcal{N} \not\models \mathcal{T}$ are well-defined, and every outcome $p \in \mathcal{R}(\mathcal{M} \not\models \mathcal{T})$ there exists $p' \in \mathcal{R}(\mathcal{N} \not\models \mathcal{T})$ such that $p \leq p'$.

(ii) $\mathcal{M} \sqsubseteq_{must} \mathcal{N}$ if for every $\mathcal{T}$ such that both $\mathcal{M} \not\models \mathcal{T}$ and $\mathcal{N} \not\models \mathcal{T}$ are well-defined and for every $p' \in \mathcal{R}(\mathcal{M} \not\models \mathcal{T})$ there exists $p \in \mathcal{R}(\mathcal{N} \not\models \mathcal{T})$ such that $p \leq p'$. □

Example 2 (Broadcast vs Multicast). Consider the networks $\mathcal{M}$ and $\mathcal{N}$ in Figure 5. Intuitively in $\mathcal{N}$ the value v is (simultaneously) broadcast to both nodes o_1 and o_2 while in $\mathcal{M}$ there is a multicast. More specifically o_1 receives v from mode m while in an independent broadcast o_2 receives it from n.

This difference in behaviour can be detected by testing network

$$\mathcal{T} = \Gamma_T \triangleright o_1[\![c?(x)\,.c!\langle 0\rangle\,.\,\mathbf{0}]\!] \mid o_2[\![c?(x)\,.c?(y)\,.\mathsf{if}\ y = 0\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \omega]\!]$$

assuming v is different than 0; here we assume Γ_T is the simple network which connects o_1 with o_2. Both $\mathcal{M} \Vdash \mathcal{T}$ and $\mathcal{N} \Vdash \mathcal{T}$ are well-formed and note that they are both non-probabilistic.

Because $\mathcal{N}$ simultaneously broadcasts to o_1 and o_2 the second value received by o_2 is always 0 and therefore the test never succeeds; $\mathcal{V}(\mathcal{N} \Vdash \mathcal{T}) = \{0\}$. On the other-hand there is a possibility for the test succeeding when applied to $\mathcal{M}$, $1 \in \mathcal{V}(\mathcal{M} \Vdash \mathcal{T})$. This is because in $\mathcal{M}$ node m might first transmit v to o_1 after which n transmits 0 to o_2; now node n might transmit the value v to o_2 and assuming it is different than 0 we reach a success state. It follows that $\mathcal{M} \not\sqsubseteq_{may} \mathcal{N}$.

One might also think it possible to use the difference between broadcast and multicast to design a test which $\mathcal{N}$ passes and $\mathcal{M}$ does not. However this is not possible, and in Example 3 we show that $\mathcal{N} \sqsubseteq_{may} \mathcal{M}$; that is multicast can be implemented by broadcast. □

Theorem 1 (Compositionality). *Suppose* $\mathsf{Int}(\mathcal{M}_1) = \mathsf{Int}(\mathcal{M}_2)$*. Then* $\mathcal{M}_1 \sqsubseteq_{may} \mathcal{M}_2$ *implies* $\mathcal{M}_1 \Vdash \mathcal{N} \sqsubseteq_{may} \mathcal{M}_1 \Vdash \mathcal{N}$*, whenever the composite networks are well-defined. The same result holds for* $\sqsubseteq_{must}$*.* □

The testing preorders over networks can be defined using any (partial) binary constructor $\|$ over networks, although we would only want to use constructors which are in some sense *reasonable*, which we define as follows.

Let us say that the partial constructor is a *merge* operator if whenever it is defined the result coincides with the definition in (1) above. We say it is *invariant under renaming*, if whenever $\mathcal{M} \| \mathcal{N}$ is well-defined then so is $\mathcal{M}\sigma \| \mathcal{N}$, where σ is an aribitrary renaming of nodes which leaves both $\mathsf{Int}(\mathcal{M})$ and nodes($\mathcal{N}$) unchanged, and whose range does not intersect nodes($\mathcal{N}$); intuitively this means that the well-definedness of the composite $\mathcal{M} \| \mathcal{N}$ is not affected by a re-organisation of the internal nodes of $\mathcal{M}$. Then we say $\|$ is *reasonable* if it is a *merge* operator, it *preserves interfaces*, and is *invariant under renaming*; note that $\Vdash$ is reasonable. Let us denote the resulting testing pre-orders by $\sqsubseteq^{alt}_{may}$, $\sqsubseteq^{alt}_{must}$ respectively.

Theorem 2. *If the constructor* $\|$ *is reasonable and symmetric then the resulting testing preorders are degenerate; that is* $\mathcal{M}_1 \sqsubseteq^{alt}_{may} \mathcal{M}_2$ *and* $\mathcal{M}_1 \sqsubseteq^{alt}_{must} \mathcal{M}_2$*, for all networks* $\mathcal{M}_1$, $\mathcal{M}_2$ *such that* $\mathsf{Int}(\mathcal{M}_1) = \mathsf{Int}(\mathcal{M}_2)$. □

5 Proof Techniques for the Testing Preorders

Motivated by [3], our intention is to define simulations over a pLTS to provide reasonable proof techniques for inferring $\mathcal{M} \sqsubseteq_{may} \mathcal{N}$ and $\mathcal{M} \sqsubseteq_{must} \mathcal{N}$. The pLTS induced by the intensional semantics in Figure 3 is much too coarse for this purpose. Instead we need to define *extensional* actions, which capture more closely the manner in which the behaviour of wireless systems can be detected at their interfaces. The following remarks are relevant.

(i) A node m which receives a value v has no information about the name of the node, internal or external, which is responsible for the broadcast; it can only check the content of the value.

(b-Tau)
$$\frac{\Gamma \triangleright M \xrightarrow{m.\tau} \Delta}{\Gamma \triangleright M \xrightarrow{\tau} \Gamma \triangleright \Delta} \quad \begin{array}{l} [\![p]\!] = \Delta \\ \omega(\Gamma \triangleright M) = \mathit{false} \end{array}$$

(b-In)
$$\frac{\Gamma \triangleright M \xrightarrow{c.m?v} \Delta}{\Gamma \triangleright M \xrightarrow{c.m?v} \Gamma \triangleright \Delta} \quad \begin{array}{l} m \in \mathsf{Int}(\Gamma \triangleright M) \\ \omega(\Gamma \triangleright M) = \mathit{false} \end{array}$$

(b-Shh)
$$\frac{\Gamma \triangleright M \xrightarrow{c.m!v} \Delta}{\Gamma \triangleright M \xrightarrow{\tau} \Gamma \triangleright \Delta} \quad \begin{array}{l} \{\, m \in \mathsf{Int}(\Gamma_M \triangleright M) \mid \Gamma \vdash m \leftrightarrow n \,\} = \emptyset \\ \omega(\Gamma \triangleright M) = \mathit{false} \end{array}$$

(b-Out)
$$\frac{\Gamma \triangleright M \xrightarrow{c.m!v} \Delta}{\Gamma \triangleright n[\![s]\!] \xrightarrow{c!v \triangleright \eta} \Gamma \triangleright \Delta} \quad \begin{array}{l} \eta := \{\, m \in \mathsf{Int}(\Gamma_M \triangleright M) \mid \Gamma \vdash m \leftrightarrow n \,\} \\ \eta \neq \emptyset \\ \omega(\Gamma \triangleright M) = \mathit{false} \end{array}$$

Fig. 6. Extensional semantics of networks

(ii) On the other hand, the set of nodes in the interface of a network $\mathcal{M}$ which are affected by a broadcast performed by a node $m \in \text{nodes}(\mathcal{M})$ is relevant; these are the only nodes at the external environment which can detect the broadcast.

(iii) As a consequence, if a broadcast originated by a node in $\mathcal{M}$ does not affect any node in its interface, then this activity cannot be observed by the external environment of $\mathcal{M}$.

(iv) The effect on a network $\mathcal{M}$ by external activity can be captured adequately by broadcasts fired from nodes in the interface of $\mathcal{M}$.

(v) Since we are not interested in the behaviour of a network after it has reached a successful configuration, we require that extensional transitions can be performed only by non-successful network.

These observations motivate the definition of external actions in Figure 6. Note that these actions endow a network with the structure of a pLTS; we say that a network is *finitary* if so is the pLTS it generates via the transitions defined in Figure 6. Henceforth in this Section we always assume that a network is finitary. The extension to weak actions is also non-standard:

Definition 6 (Weak Extensional Action). *Let Δ, Θ be network sub-distributions over* Nets. *We say that*

(i) $\Delta \overset{\tau}{\Longrightarrow} \Theta$ *if* $\Delta \Longrightarrow \Theta$ *in the pLTS induced by the extensional transitions of Figure 6.*

(ii) $\Delta \overset{c.m?v}{\Longrightarrow} \Theta$ *if* $\mathcal{M} \overset{\tau}{\Longrightarrow} \xrightarrow{c.m?v} \overset{\tau}{\Longrightarrow} \Theta$

(iii) $\Delta \overset{c!v \triangleright \eta}{\Longrightarrow} \Theta$ *if either* $\Delta \overset{\tau}{\Longrightarrow} \xrightarrow{c!v \triangleright \eta} \overset{\tau}{\Longrightarrow} \Theta$ *or* $\Delta \overset{c!v \triangleright \eta_1}{\Longrightarrow} \overset{c!v \triangleright \eta_2}{\Longrightarrow} \Theta$, *where* η_1, η_2 *are two non-empty sets of nodes which constitute a partition of* η. □

The non-standard *(iii)* is motivated in Example 3.

These weak actions endow the set of networks Nets with the structure of another pLTS, called the *extensional pLTS* and denoted by $\mathsf{pLTS}_{\mathsf{Nets}}$. It is in this pLTS that we give our definitions of simulations.

The first one is based on the simulation preorder from [3]; for reasons best explained there it is defined as a relation from states to distributions, rather than the more standard states to states.

Definition 7 (Simulation Preorder). In $\mathsf{pLTS}_{\mathsf{Nets}}$ we let $\lhd_{sim}$ denote the largest relation in $\mathsf{Nets} \times \mathcal{D}(\mathsf{Nets})$ such that if $s \lhd_{sim} \Theta$ then:

– if $\omega(s) =$ true, then $\Theta \overset{\tau}{\Longrightarrow} \Theta'$ such that for every $t \in \lceil \Theta' \rceil, \omega(t) =$ true
– otherwise, whenever $\overline{s} \overset{\mu}{\Longrightarrow} \Delta'$, for $\mu \in \mathsf{Act}_\tau$, then there is a $\Theta' \in \mathcal{D}(S)$ with $\Theta \overset{\mu}{\Longrightarrow} \Theta'$ and $\Delta' \; \overline{\lhd_{sim}} \; \Theta'$. □

Our second proof technique is a variation on the *failure simulation preorder* of [3]. Unlike in that more general framework we have no need of acceptance sets. Instead it is sufficient to consider the ability of systems to *deadlock*. See [4] for details. We say that a network $\mathcal{M}$ is *deadlocked*, denoted $\mathcal{M} \not\rightarrow$ whenever $\omega(\mathcal{M}) =$ false and $\mathcal{M} \overset{\tau}{\not\longrightarrow}, \mathcal{M} \overset{c!v \rhd \eta}{\not\longrightarrow}$ for any c, v, η. A sub-distribution Δ over $\mathcal{D}_{sub}(\mathsf{Nets})$ is *deadlocked* if any network in its support is deadlocked.

For reasons explained in [3] it is more straightforward to express this form of simulation as a relation from sub-distributions to sub-distributions.

Definition 8 (Deadlock Simulations). *In* $\mathsf{pLTS}_{\mathsf{Nets}}$ *we let* $\sqsupseteq_{DS}$ *denote the largest relation in* $\mathcal{D}_{sub}(\mathsf{Nets}) \times \mathcal{D}_{sub}(\mathsf{Nets})$ *such that if* $\Delta \sqsupseteq_{DS} \Theta$ *then:*

– *whenever* $\Delta \overset{\mu}{\Longrightarrow} \sum_{i \in I}(p_i \cdot \Delta'_i)$, *where* I *is an index set such that* $\sum_{i \in I} p_i \leq 1$, *then there are* $\Theta'_i \in \mathcal{D}_{sub}(\mathsf{Nets})$ *such that* $\Theta \overset{\mu}{\longrightarrow} \sum_{i \in I}(p_i \cdot \Theta'_i)$ *and, for any* $i \in I$, $\Delta'_i \sqsupseteq_{DS} \Theta'_i$
– *whenever* $\Delta \Longrightarrow \not\rightarrow$ *then* $\Theta \Longrightarrow \not\rightarrow$ *.* □

Theorem 3 (Proof Methods for the Testing Preorders). *Let* $\mathcal{M}, \mathcal{N}$ *be two networks such that* $\mathsf{Int}(\mathcal{M}) = \mathsf{Int}(\mathcal{N})$*. Then*

– *if* $\mathcal{M} \lhd_{sim} \overline{\mathcal{N}}$ *then* $\mathcal{M} \sqsubseteq_{may} \mathcal{N}$
– *if* $\overline{\mathcal{M}} \sqsupseteq_{DS} \overline{\mathcal{N}}$ *then* $\mathcal{M} \sqsubseteq_{must} \mathcal{N}$*.* □

Thus in order to relate two wireless systems it is sufficient to exhibit an appropriate simulation relation.

Example 3. Consider again the networks $\mathcal{M}$ and $\mathcal{N}$ in Figure 5. It is easy to show that both of them can perform the weak extensional action $\overset{c!v \rhd \{o_1, o_2\}}{\Longrightarrow}$. However, the inference of this action is different for the individual networks; while in network $\mathcal{N}$ it is implied by the execution of a single broadcast action, detected by both nodes o_1 and o_2 simultaneously, in $\mathcal{M}$ this is implied by a sequence of weak extensional actions $\mathcal{M} \overset{c!v \rhd \{o_1\}}{\Longrightarrow} \overset{c!v \rhd \{o_2\}}{\Longrightarrow}$.

It is therefore possible to exhibit a simulation between $\mathcal{N}$ and $\mathcal{M}$, thus showing that $\mathcal{N} \lhd_{sim} \overline{\mathcal{M}}$; By Theorem 3 it follows that $\mathcal{N} \sqsubseteq_{may} \mathcal{M}$. Similarly, it is possible to prove that $\overline{\mathcal{M}} \sqsupseteq_{DS} \overline{\mathcal{N}}$, and therefore $\mathcal{M} \sqsubseteq_{must} \mathcal{N}$.

Now suppose that we employed a standard definition of weak extensional actions, and that the simulation preorder had been defined according to this

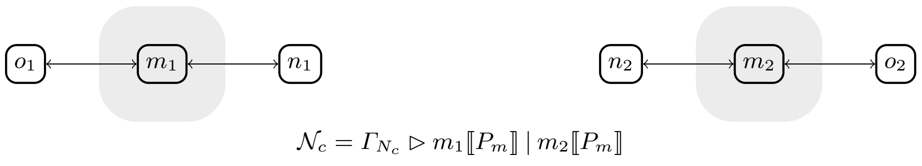
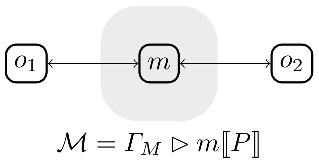

o_1
m_1
n_1
n_2
m_2
o_2
$\mathcal{N}_c = \Gamma_{N_c} \triangleright m_1[\![P_m]\!] \mid m_2[\![P_m]\!]$

Fig. 7. The network $\mathcal{N}_c$

notion. In this case it would not be possible to exhibit a simulation between $\mathcal{N}$ and $\mathcal{M}$, and thus it would not be possible to prove that $\mathcal{N} \sqsubseteq_{may} \mathcal{M}$. The same applies for the $\sqsubseteq_{must}$ testing preorder and deadlock simulations. $\square$

Although simulations provide a sound proof technique for $\sqsubseteq_{may}$ and $\sqsubseteq_{must}$, in general they are not complete. For example one can show, referring to Example 1, that $\mathcal{M} \sqsubseteq_{may} \mathcal{N}$ but $\mathcal{M} \not\triangleleft_{sim} \overline{\mathcal{N}}$. It remains to be seen if our notions of simulation can be further adapted so as to provide complete proof methodologies.

6 An Application: Probabilistic Routing



Sequential routing means that a network can only route one message per time; if a message is received by the network while another one is being routed, then it is ignored. A simple specification (or model) network $\mathcal{M}$ that can be used for such routing in networks is depicted on the left, where $P \Leftarrow c?(x)\,.c!\langle x\rangle\,.P$. It is trivial to see that, whenever node m in $\mathcal{M}$ receives a message from o_1, it will be forwarded to nodes o_1 and o_2; that is, the message has been routed from the external node o_1 to the external node o_2. The model also routes messages from o_2 to o_1 in a symmetric fashion. We provide a possible implementation of this specification as a probabilistic (and nondeterministic) network $\mathcal{N}$ such that $\mathcal{M} \sqsubseteq_{may} \mathcal{N}$; this means that $\mathcal{N}$ will include all the possible behaviour of $\mathcal{M}$, such as the sequential routing between the interface nodes o_1, o_2, but may also have additional behaviour.

In fact we design an entire class of networks $\mathcal{N}$ with this property. Each will have the structure $\mathcal{N} = \mathcal{N}_c \not\Vdash \mathcal{C}$, where $\mathcal{N}_c$ is the network depicted in Figure 7. This acts as a connector between the interface nodes o_1, o_2 and the internal router which it accesses via the nodes n_1, n_2; here $P_m \Leftarrow c?(x)\,.c!\langle x\rangle\,.P_m + c?(x)\,.P_m$.

The network $\mathcal{C} = \Gamma_C \triangleright C$, on the other hand, is defined parametrically. It can be any network that satisfies the following requirements:

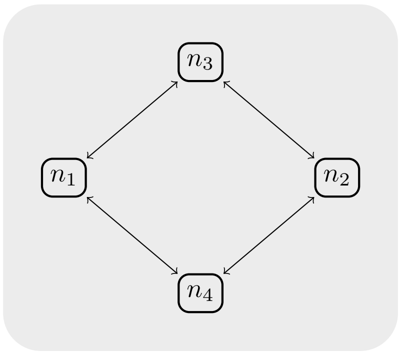
1. $n_1, n_2 \in \text{nodes}(\Gamma_C \triangleright C)$. For the sake of simplicity, we also assume that $\text{nodes}(\Gamma_C \triangleright C) = (\Gamma_C)_V = \{n_1, \cdots, n_k\}$ for some $k > 2$.
2. The connectivity graph Γ_C contains a single connected component.

3. Every node n_i, $i = 1, \cdots, k$ is associated with a channel c_i and a probability distribution $\Lambda_i : \{1, \cdots, k\} \to [0, 1]$. The latter are defined so that $\lceil \Lambda_i \rceil = \{j \mid \Gamma_C \vdash n_i \leftrightarrow n_j\}$, for any $i = 1, \cdots, k$. That is, if node n_i is connected to node n_j, then $\Lambda_i(j) > 0$.
4. $C = \prod_{i \in I} n_i[\![P_i]\!]$, where

$$\begin{aligned} P_i &= c?(x)\,.\left[\bigoplus_{j=1}^{k} \Lambda_i(j) \cdot c_j!\langle x\rangle\,.P_i\right] + c?(x)\,.P_i + \\ &\quad + c_i?(x)\,.\left[\bigoplus_{j=1}^{k} \Lambda_i(j) \cdot c_j!\langle x\rangle\,.P_i\right] + c_i?(x)\,.c!\langle x\rangle\,.P_i, \quad i = 1, 2 \\ P_i &= c_i?(x)\,.(\bigoplus_{j=1}^{k} \Lambda_i(j) \cdot c_j!\langle x\rangle\,.P_i), \qquad i > 2 \end{aligned}$$

Here the derived construct $\bigoplus_{i \in I} p_i \cdot P_i$ is interpreted in the obvious manner as a probability distribution.

Let us explain, informally, one of the possible behaviours of a typical network $\mathcal{N}$.



The connectivity of one possible $\mathcal{C}$, with only two extra nodes n_3, n_4, is given on the left. Upon receiving a message along channel c from the external node o_1, node m_1 will forward it to the node n_1; here note that the external node o_1 also detects the broadcast. Node n_1 forwards the message again to one of its neighbours n_j with a strictly positive probability. Here n_j is selected as the next hop in the routing path by forwarding the message along channel c_j. In fact, node c_j is the only one which can detect messages broadcast along such a channel.

This procedure is iterated until the message v is forwarded to node n_2, at least with some probability, which in turn will forward it to node m_2; a final broadcast from the latter node will cause the message to be detected by the external node o_2. Since the connectivity graph of $\mathcal{C}$ has a single connected component, it is possible to show that, upon being received by node n_1, a message v eventually is delivered to node n_2 almost surely, i.e. with probability 1.

This informal line of reasoning can be used to provide a simulation between the networks $\mathcal{M}$ and $\mathcal{N}$; we can construct a simulation in the extensional $\mathsf{pLTS}_{\mathsf{Nets}}$ containing the pair $(\mathcal{M}, \overline{\mathcal{N}})$, for any $\mathcal{N}$ whose internal component $\mathcal{C}$ satisfies the four constraints given above; thus for any such $\mathcal{N}_c$ we have $\mathcal{M} \sqsubseteq_{may} \mathcal{N}$. The details may be found in [1]

We finish with three remarks about this example. First note that it is necessary to employ our non-standard definition of weak output actions, for a broadcast of network $\mathcal{M}$ to the nodes o_1, o_2 can only be simulated by $\mathcal{N}$ via a sequence of two broadcasts. The first, fired by node m_1, can be detected only by o_1; the second one, fired by node m_2, can be detected only by node o_2, and this only in a probabilistic limit, for which we require the infinitary version of probabilistic weak actions. Secondly note that the network $\mathcal{N}$ implements sequential routing because the nodes m_1, m_2 in $\mathcal{N}$ can non-deterministically decide to ignore a broadcast performed by o_1, o_2 respectively. Finally note that this example emphasises the fact that our formalism can in principle be employed to examine practical routing algorithms. Routing protocols in which the next-hop in a routing path is determined via a probability distribution, as in our example, are of practical significance; see for example, [2].

7 Conclusions

To the best of our knowledge, we believe that our work is the first to apply testing theories to wireless systems, and in particular probabilistic wireless systems. One major strand of research into calculi based on broadcast communications starts with CBS from [15]; here behaviour is defined in terms of various forms of bisimulations. Later developments include local broadcast communication in the Extended CBS of [13] and the use of connectivity graphs in the CBS# of [14].

In [11] a different attempt to formalize wireless networks is made. The authors develop a calculus CWS, where the concepts of node names and location are differentiated; thus, a process is associated both with a node name and a location. Also, every process has a positive real value associated to it, denoting the radius of transmission. The network topology is determined by a metric on locations and a transmision radius. It is worth mentioning that in this calculus the communication between nodes consists of two phases, one to start it and one to end it. The authors also model the possibility of a message whose transmission has started to be corrupted by another transmission, thus modeling collisions.

In [10], a descrete timed calculus for wireless systems (TCWS) is presented; in this case, the authors address the problem of representing collisions in wireless networks, suggesting that formal tools for dealing with interferences in wireless networks can aid in the development of MAC level protocols. The topology of the wireless networks here is described by associating every node a semantic tag representing its set of neighbours. The authors propose a compositional theory for wireless networks based on the notion of reduction barbed congruence; further, they develop a sound proof methodology based on bisimulations over an extensional lts. It is of considerable interest that, despite their targeting at low-level collision prone behaviour, the set of extensional actions they propose (and the activities that can be detected by the external environment) is very similar to those we have suggested.

References

1. Cerone, A., Hennessy, M.: A simple probabilistic broadcast language. Technical Report, Trinity College Dublin, CS-TR-2012-02 (2012), http://www.scss.tcd.ie/~ceronea/works/simpleProbabilisticNetworks.pdf
2. Curran, E., Dowling, J.: Sample: Statistical network link modelling in an on-demand probabilistic routing protocol for ad hoc networks. In: WONS, pp. 200–205. IEEE Computer Society (2005)
3. Deng, Y., 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), http://www.scss.tcd.ie/Matthew.Hennessy/onlinepubs.html
4. Ene, C., Muntean, T.: Testing theories for broadcasting processes. Sci. Ann. Cuza Univ. 11, 214–230 (2002)
5. Ghassemi, F., Fokkink, W., Movaghar, A.: Verification of mobile ad hoc networks: An algebraic approach. Theor. Comput. Sci. 412(28), 3262–3282 (2011)
6. Godskesen, J.C.: Observables for Mobile and Wireless Broadcasting Systems. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 1–15. Springer, Heidelberg (2010)
7. Jurdak, R., Lopes, C.V., Baldi, P.: A survey, classification and comparative analysis of medium access control protocols for ad hoc networks. IEEE Communications Surveys and Tutorials 6(1-4), 2–16 (2004)
8. Lanese, I., Sangiorgi, D.: An operational semantics for a calculus for wireless systems. Theor. Comput. Sci. 411(19), 1928–1948 (2010)
9. Lanotte, R., Merro, M.: Semantic Analysis of Gossip Protocols for Wireless Sensor Networks. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011 – Concurrency Theory. LNCS, vol. 6901, pp. 156–170. Springer, Heidelberg (2011)
10. 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)
11. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. Electr. Notes Theor. Comput. Sci. 158, 331–353 (2006)
12. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
13. Nanz, S., Hankin, C.: Static analysis of routing protocols for ad-hoc networks, March 25 (2004)
14. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. TCS: Theoretical Computer Science 367 (2006)
15. Prasad, K.V.S.: A calculus of broadcasting systems. Science of Computer Programming 25(2-3), 285–327 (1995)
16. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. Nordic J. of Computing 2, 250–273 (1995)
17. Tanenbaum, A.S.: Computer networks. PTR Prentice-Hall

Appendix: Some Definitions

Definition 9 (Hyper-Derivations). *In a pLTS a hyper-derivation consists of a collection of sub-distributions $\varDelta, \varDelta_k^{\rightarrow}, \varDelta_k^{\times}$, for $k \geq 0$, with the following properties:*

$$\begin{array}{lcll}
\varDelta & = & \varDelta_0^{\rightarrow} & + \varDelta_0^{\times} \\
\varDelta_0^{\rightarrow} & \xrightarrow{\tau} & \varDelta_1^{\rightarrow} & + \varDelta_1^{\times} \\
 & \vdots & & \\
\varDelta_k^{\rightarrow} & \xrightarrow{\tau} & \varDelta_{k+1}^{\rightarrow} & + \varDelta_{k+1}^{\times} \\
 & \vdots & &
\end{array}$$

If $\omega(s) =$ false for each $s \in \lceil \varDelta_k^{\rightarrow} \rceil$ and $k \geq 0$ we call $\varDelta' = \sum_{k=0}^{\infty} \varDelta_k^{\times}$ a hyper-derivative *of $\varDelta$, and write $\varDelta \Longrightarrow \varDelta'$.* □

In maximal computations, we require the computation to proceed as long as some internal activity can be performed. To this end, we say that $\varDelta \Longrightarrow\!\!\succ \varDelta'$ if

- $\varDelta \Longrightarrow \varDelta'$,
- for every $s \in \lceil \varDelta_k^{\times} \rceil$, $s \xrightarrow{\tau}$ implies $\omega(s) =$ true. □

This is a mild generalisation of the notion of *extreme derivative* from [3]. Note that the last constraint models exactly the requirement of performing some internal activity whenever it is possible; In other words extreme derivatives correspond to a probabilistic version of maximal computations.

Noninterference via Symbolic Execution[⋆]

Dimiter Milushev, Wim Beck, and Dave Clarke

IBBT-DistriNet, KU Leuven, Heverlee, Belgium

Abstract. Noninterference is a high-level security property that guarantees the absence of illicit information flow at runtime. Noninterference can be enforced statically using information flow type systems; however, these are criticized for being overly conservative and rejecting secure programs. More precision can be achieved by using program logics, but such an approach lacks its own verification tools. In this work we propose a novel, alternative approach: utilizing symbolic execution in combination with ideas from program logics in an attempt to increase the precision of analyses and automate noninterference testing. Dealing with policies incorporating declassification is also explored. The feasibility of the proposal is illustrated using a prototype tool based on the KLEE symbolic execution engine.

Keywords: Noninterference, declassification, symbolic execution, testing.

1 Introduction

Noninterference is a high-level security property, prohibiting information leaks through the executions of a program. The typical program model for expressing noninterference assumes the following: public and secret inputs are given to a program; public and secret outputs are observable as a result of the program runs. In this context, noninterference is a policy stipulating that public outputs of a program should be functionally dependent on public inputs only, and not on secret inputs. The policy has been substantially studied in the language-based security community [16] and typically relies on information flow type systems [15,21,22]; however, these are criticized for being conservative and rejecting many secure programs.

An alternative approach proposes the use of program logics for expressing noninterference. Such an approach was introduced by Darvas, Hähnle and Sands [8], who used dynamic logic to verify noninterference for sequential Java programs. One key observation they made is that noninterference (which is not a property and hence not directly expressible in program logics) on some program P is reducible to a property on the sequential composition $P; P'$ of the program with itself. More precisely, noninterference can be characterized as the following

⋆ This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`).

quadruple: $\{\bar{l} = \bar{l'}\}P; P'\{\bar{l} = \bar{l'}\}$. Here, P' is the same program as P with all variables renamed, $\bar{l}$ are the low variables of P and $\bar{l'}$ are the low variables of P'. Barthe, Argenio and Rezk [4] based their characterization of noninterference in Hoare and temporal logics on similar ideas; they also coined the term *self-composition* for the construct $P; P'$. The program logics approaches provide more precision in specifications, but do not have their own verification tools; in addition, it is not clear how to reuse existing tools and techniques.

Terauchi and Aiken [20] note that self-composition is impractical. They point out that for the purpose of verification of noninterference, some nontrivial, partial-correctness condition that holds between P and P' has to be found; and finding it is impractical. They also argue that in order to be useful for practical verification, self-composition needs to take into account the structure of a self-composed program and the resulting symmetry and redundancy. They propose a type-directed transformation for a simple imperative language to deal with the problems they identify.

Symbolic execution has already been used for verification of secure information flow [8]. The approach offers high precision but unfortunately requires considerable user interaction and verification expertise, needed for adding loop invariants, establishing induction hypotheses, instantiating and unwinding loops etc. Due to this fact, similar approaches are often criticized as being of limited practical significance for developers. In this work we attempt to remedy such limitations and propose the use of symbolic execution as a basis for a noninterference testing tool. This is advantageous because it is automatic and gives developers an efficient, practical way of testing for noninterference bugs. The proposed approach is essentially an under-approximation of the problem and this has two important consequences: on the positive side, it is automatic and precise; on the other hand, it usually cannot discover all bugs. Nevertheless, combined with the observation that typically most bugs are shallow, the approach gives developers a powerful tool, without requiring them to understand and write complex specifications.

More concretely, the advocated approach is based on utilizing symbolic execution in combination with a form of self-composition in an attempt to automate noninterference testing. We start off with Terauchi and Aiken's transformation and accommodate additional language features, such as dealing with procedures and dynamically allocated data structures. The approach essentially interleaves two copies of a program and then uses dynamic symbolic execution to try to extract all possible paths in the program. Conditions on two disjoint program stores are generated in order to express the desired security policy via assert statements. The resultant program is analyzed by the symbolic execution engine: if a bug is found, the program is not secure and the developer is informed about it; the proposed approach typically cannot guarantee that a program is secure. On secure programs, a tool based on this approach will often run indefinitely without producing any error message. On insecure programs it will typically run indefinitely but still discover security bugs and thus would indeed be useful. On concrete programs, our prototype tool has been able to discover instances of all the known patterns of insecurity we have found in the literature.

The contributions of the work are: first, a proposal to use symbolic execution in order to automatically specify and check a notion of plain noninterference and one incorporating declassification; second, an illustration of what is needed to transfer the ideas to a programming language having procedures and dynamic memory allocation (heap); finally, a prototype tool based on the KLEE symbolic execution tool [6] illustrating the feasibility of the approach. The rest of the work is structured as follows. Section 2 provides some background. Sections 3 presents the proposed approach. Section 4 presents our prototypical tool and experimental results. Finally, Sections 5 and 6 are left for the related work and conclusion.

2 Background

2.1 Noninterference

Intuitively, noninterference stipulates that public outputs of a program should be functionally dependent on public inputs only, and not on secret inputs. Define a store $\mathtt{m}$ to be a mapping from program variables from some set Var to values from set $\mathcal{V}$. The notation $\mathtt{m}|_X$ is a restriction of the store to variables from domain X; $(\mathtt{m}, P)$ denotes the final store after execution of program P with initial store $\mathtt{m}$ and $(\mathtt{m}, P) = \perp$ signifies that the program diverges (does not terminate or terminates in an (unobservable) exceptional state). Finally, $\approx_p$ signifies the pointwise extension of equality to stores. The definition of termination insensitive information flow can be formulated as follows:

Definition 1. *(Secure information flow [20]) A program P with high security variables $H = \{h_1, \ldots, h_i\}$ and low security variables $L = \{l_1, \ldots, l_j\}$ is* secure *iff for all possible stores m_1 and m_2 such that $m_1|_L \approx_p m_2|_L$, we have that*

$$((m_1, P) \neq \perp \wedge (m_2, P) \neq \perp) \implies (m_1, P)|_L \approx_p (m_2, P)|_L.$$

There is an obvious way to show that a program P is not secure by Definition 1, namely by finding two stores $\mathtt{m}_i$ and $\mathtt{m}_j$ such that $\mathtt{m}_i|_L \approx_p \mathtt{m}_j|_L$, $(\mathtt{m}_i, P) \neq \perp \wedge (\mathtt{m}_j, P) \neq \perp$, and $(\mathtt{m}_i, P)|_L \not\approx_p (\mathtt{m}_j, P)|_L$.

Program 1.1, also referred to as P_1, illustrates implicit information flow. Let $H = \{i, j\}, L = \{l\}$. Observing the value of variable l discloses whether the average of the two secret values is greater than 1000.

```
int average(int h1, int h2) {
    return (h1+h2)/2; }
int main() {
    int l, i, j;
    if (average(i, j) > 1000) l = 1; else l = 0; }
```

Program 1.1. Implicit information flow

The implicit flow can be detected using Definition 1. Let $\mathtt{m}_1$ and $\mathtt{m}_2$ be such that $\mathtt{m}_1(i) = 1000, \mathtt{m}_1(j) = 900, \mathtt{m}_1(l) = 0, \mathtt{m}_2(i) = 800\ \mathtt{m}_2(j) = 1400$ and $\mathtt{m}_2(l) = 0$. We have that $\mathtt{m}_1|_L \approx_p \mathtt{m}_2|_L$, $(\mathtt{m}_1, P_1) \neq \perp \wedge (\mathtt{m}_2, P_1) \neq \perp$, but at the end of execution $(\mathtt{m}_1, P_1)(l) = 0$ and $(\mathtt{m}_2, P_2)(l) = 1$; thus $(\mathtt{m}_1, P_1)|_L \not\approx_p (\mathtt{m}_2, P_1)|_L$ implies that P_1 is insecure.

2.2 Declassification

Most useful computing systems have to release sensitive information as a part of their functionality (e.g. password checking, shopping for digital content, online games). Thus noninterference is often too strict for realistic systems; the usual solution is weakening the policy with *declassification*, a mechanism for releasing sensitive information. An important problem of declassification is to guarantee precisely *what* is being leaked and to ensure that the mechanism cannot be abused into leaking more [18].

More formally, consider program P on stores $\mathtt{m}_1$ and $\mathtt{m}_2$. Recall that $\approx_p$ signifies the pointwise extension of equality to stores and $\mathtt{m}|_L$ is a restriction of the store to variables from domain L. Let ψ be the predicate defined as $\mathtt{m}_1|_L \approx_p \mathtt{m}_2|_L$. Noninterference can be given as the following quadruple $\{\psi\}(\mathtt{m}_1, P); (\mathtt{m}_2, P)\{\psi\}$. If ψ_{decl} is a predicate on high variables, specifying what is to be declassified by P, then *noninterference with declassification* can be expressed as follows: $\{\psi \wedge \psi_{decl}\}(\mathtt{m}_1, P); (\mathtt{m}_2, P)\{\psi\}$ [3].

For instance, in Program 1.1 the policy might be that it is admissible to reveal some fact about the average (i.e. whether $(average(i, j) > 1000)$ holds) but not more than that. Then, in addition to the usual noninterference condition, ψ_{decl} will be instantiated with $((average(i_1, j_1) > 1000) \Leftrightarrow (average(i_2, j_2) > 1000))$ (note that i_k is shortcut for $\mathtt{m}_k(i)$ for $k \in \{1, 2\}$).

Other dimensions of declassification are about *who* controls information release, *where* in the system does declassification occur and finally *when* can information be declassified [18]. In this work, we focus on the *what* dimension.

2.3 Symbolic Execution

Symbolic execution [13] is a program analysis technique used to investigate the possible execution traces of a program. The idea is to replace program inputs with input symbols and thus instead of executing the program with concrete values, to execute it with symbolic expressions over the input symbols. When the program encounters a conditional branch statement, execution is forked because there are no concrete values to evaluate the condition: whether any or both of these branches are reachable is checked by a constraint solver. Loops can be seen as conditional statements encountered multiple times and they are lazily unrolled, possibly an infinite number of times. The conjunction of all conditions encountered on the branches of a single path is called a *path condition*.

Symbolic execution is a general approach that can be used to check or prove a range of properties of programs. Properties can be expressed using assert statements.

Dynamic symbolic execution, also called concolic execution [19] or DART [11], is a variant of the technique interleaving concrete and symbolic execution. The idea is simple: first, gather the constraints for some path by monitoring program execution with some arbitrary, concrete inputs; then, systematically explore new execution paths by negating parts of the initial path condition. In this work we are going to use KLEE [6], an automatic symbolic execution tool built on top of the LLVM compiler infrastructure; the tool is used for both illustrations of the purposed approach and as a basis of our prototype tool.

3 Approach

3.1 Overview

The approach proposed in this paper starts with partitioning the program input variables into public and secret, and respectively annotating them. This is the only obligation on behalf of the developer, the rest is automatic. Then the variables are made symbolic and a type-directed transformation adopted from the work of Terauchi and Aiken [20] is applied to the program. The transformation is a variant of the self-compositional approach. It (the transformation) provides the method used for interleaving the candidate program with itself, which is needed to express noninterference as a property. We develop certain extensions of the transformation in order to deal with aspects of procedures and dynamically allocated data structures; the latter require reasoning about the heap and a modified definition of noninterference. After the transformation is complete assertions specifying the noninterference policy are placed. Then the symbolic execution tool is used as a program analysis tool for noninterference. If it is able to fully analyze all possible paths in the transformed program (paths have to be finite), a tool based on the approach can decide whether the program is secure or not. Otherwise the tool may either eventually return an error(s) or simply keep running without returning any error. In the latter cases the proposed approach is useful even if it cannot cover the whole state space, as it will still discover bugs in the covered part; moreover, the approach offers precision, i.e. lack of false-positives. Because of the nature of typical bugs (being shallow), this strategy often turns out to be very helpful.

3.2 Transformation of a Basic Language

We start off by illustrating how to transform a program for a minimal language including variable declarations and assignments, while loops and if statements. To illustrate we work with a basic subset of C and also use KLEE notation. Nevertheless, it should be noted that the proposed approach is generic and can be applied to many other language and symbolic execution tool combinations. Some annotations necessary to direct the transformation are identified using special comments "//#"and given next:

high. The subsequent line has one or many secret variables.
assume. The possible variables' values are limited by adding invariants.

The first step is to partition the variables and make them symbolic. Only high variables are denoted. The step is illustrated in Program 1.2.

```
int l;
klee_make_symbolic(&l, sizeof(int), "int l");
//# high
int h;
klee_make_symbolic(&h, sizeof(int), "int h");
l = h + 5;
```

Program 1.2. Trivial noninterference example - labeled

The second step is to determine security types of expressions statically in the usual way: in essence, if an expression depends directly or indirectly on a high variable, it must be high.

The third step is to perform the necessary program transformations given in Figure 1. The rules used here are essentially Terauchi and Aiken's transformation [20] ported to a basic subset of C. The transformation is needed and used in the process of interleaving two copies of the candidate program. Note that the concrete rule applied for an *if* or *while* statement depends on whether the guard has *high* or *low* security type. If the guard is *high* the whole statements are composed sequentially, otherwise the bodies of the statements are interleaved.

$$\frac{c\ \mathit{atomic}}{c \to c; c'} \qquad \frac{c_1 \to c_1^\dagger \qquad c_2 \to c_2^\dagger}{c_1; c_2 \to c_1^\dagger; c_2^\dagger} \qquad \frac{b\ \mathit{has\ low\ security\ type} \qquad c_1 \to c_1^\dagger \qquad c_2 \to c_2^\dagger}{\mathit{if}\ b\ \mathit{then}\ c_1\ \mathit{else}\ c_2 \to \mathit{if}\ b\ \mathit{then}\ c_1^\dagger\ \mathit{else}\ c_2^\dagger}$$

$$\frac{b\ \mathit{has\ low\ security\ type} \qquad c \to c^\dagger}{\mathit{while}\ b\ \mathit{do}\ c \to \mathit{while}\ b\ \mathit{do}\ c^\dagger} \qquad \frac{b\ \mathit{has\ high\ security\ type}}{\mathit{while}\ b\ \mathit{do}\ c \to \mathit{while}\ b\ \mathit{do}\ c; \mathit{while}\ b'\ \mathit{do}\ c'}$$

$$\frac{b\ \mathit{has\ high\ security\ type}}{\mathit{if}\ b\ \mathit{then}\ c_1\ \mathit{else}\ c_2 \to \mathit{if}\ b\ \mathit{then}\ c_1\ \mathit{else}\ c_2; \mathit{if}\ b'\ \mathit{then}\ c_1'\ \mathit{else}\ c_2'}$$

Fig. 1. Type-directed transformation [20]

The fourth and final stage of the transformation is to specify noninterference conditions. These are pre and post conditions derived from the program logic approach and guaranteeing that the program is secure. They assume that the low variables of the two copies are the same (and possibly some extra declassification conditions) and have to assert that the same holds at the end of the run. To illustrate the transformation approach, consider the annotated Program 1.3:

```
        int k; int l;
        //# high
        int h;
        while (k < l) {l = k; k = k+1;}
        if (l > h) l = 1; else l = 0;
```

Program 1.3. Annotated program illustration

It is transformed into Program 1.4.

```
        int k0; int k1;
        klee_make_symbolic(&k0, sizeof(int), "int k0");
        klee_make_symbolic(&k1, sizeof(int), "int k1");
        int l0; int l1;
        klee_make_symbolic(&l0, sizeof(int), "int l0");
        klee_make_symbolic(&l1, sizeof(int), "int l1");
        klee_assume(k0 == k1); klee_assume(l0 == l1);
        int h0; int h1;
        klee_make_symbolic(&h0, sizeof(int), "int h0");
        klee_make_symbolic(&h1, sizeof(int), "int h1");
        while (k0 < l0) {l0 = k0; l1 = k1; k0 = k0+1; k1 = k1+1;}
        if (l0 > h0) l0 = 1; else l0 = 0;
        if (l1 > h1) l1 = 1; else l1 = 0;
        klee_assert(k0 == k1); klee_assert(l0 == l1);
```

Program 1.4. Transformed program illustration

3.3 Procedures

Whereas Terauchi and Aiken develop their transformation for a very basic language, we need to deal with extra language features. One of these features is procedures: the rationale for transforming them is the same as for simple imperative programs. The transformation results in a new procedure with two copies of the parameters; if the original procedure has a *non-void* return type then two potentially different results of the same type are returned and thus have to be placed in a fresh struct.

In different cases variant(s) of the *if* and *while* rules from Fig. 1 have to be used. This depends on whether the procedure is called with arguments having high or low security types (or both) and is based on a respective data flow analysis. Consider Program 1.5 as an instance of a procedure to be transformed:

```
int checkPass(int input, int secret){
    int access;
    if (input == secret){access = 1; return access;}
    else {access = 0; return access;} }
```

Program 1.5. Procedure with non-void return type

The transformed procedure should return a struct of two integers, but that means the original *return* statements have to be replaced with appropriate *goto* statements; these are used to make a transition to the second "copy" and ensure that a properly populated data structure is returned. The resulting transformation, assuming *secret* is passed a high value (thus second version of *if* rule used), is:

```
struct intRet* checkPass2(int input0, int secret0, int input1, int secret1){
    int access0; int access1;
    struct intRet* intR = malloc(sizeof(struct intRet));
    if (input0 == secret0) {access0 = 1; goto second;}
    else {access0 = 0; goto second;}
    second: if (input1 == secret1) {access1 = 1; goto done;}
    else { access1 = 0; goto done;}
    done: intR->ret0 = access0; intR->ret1 = access1;
    return intR; }
```

Program 1.6. Transformed procedure with non-void return type

Next, we illustrate how to transform the *int result* = *checkPass*(*guess*, *pass*) procedure call (assuming the program consists of the call and variable assignments):

```
int result0; int result1; klee_assume(result0 == result1);
int pass0; int pass1;
int guess0; int guess1; klee_assume(guess0 == guess1);
struct intRet* r = checkPass2(guess0, pass0, guess1, pass1);
result0 = r->ret0; result1 = r->ret1;
klee_assert(result0 == result1);
klee_assert(guess0 == guess1);
```

Note that r is a "return" struct with two integer fields and variables have to be symbolic.

3.4 Dynamically Allocated Data Structures and Noninterference

It has already been suggested that different parts of a struct can be high or low. In order to model this and use symbolic execution to check programs allocating memory on the heap, we need to model the heap and change the noninterference definition respectively.

Let F be a set of fields, $\mathcal{L}$ a set of locations and $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{null\}$ be a set of values. A heap $\mathtt{h}$ will be modeled, following prior work [5], as a partial function $\mathtt{h} : \mathcal{L} \rightharpoonup S$; here $S = F \rightharpoonup \mathcal{V}$ is another partial function that models structs; the set of all heaps is *Heap*. Now $(\mathtt{m}, \mathtt{h}, P)$ denotes the final state after executing program P with store $\mathtt{m}$ and heap $\mathtt{h}$. We write $(\mathtt{m}, \mathtt{h}, P) = (\mathtt{m}_f, \mathtt{h}_f)$ to mean that the state evaluates to store $\mathtt{m}_f$ and heap $\mathtt{h}_f$. Let β be a partial bijection on memory locations, used to model the low observer's uncertainty [2]. Let $v, v' \in \mathcal{V}$, L and H be the low and high elements respectively of a typical security lattice. *Value indistinguishability* [5] can be defined as follows:

$$v \sim_{\beta,H} v' \qquad null \sim_{\beta,L} null \qquad \frac{v \in \mathbb{Z}}{v \sim_{\beta,L} v} \qquad \frac{l, l' \in \mathcal{L} \quad \beta(l) = l'}{l \sim_{\beta,L} l'}.$$

Intuitively, two heaps $\mathtt{h}_1$ and $\mathtt{h}_2$ are indistinguishable if there is a bijection that relates each struct s_1 in heap $\mathtt{h}_1$ to its counterpart s_2 in heap $\mathtt{h}_2$; the structs have the same fields (because they are of the same type) and moreover the values of corresponding fields are indistinguishable. This is formally defined as follows: two heaps $\mathtt{h}_1$, $\mathtt{h}_2$ are indistinguishable w.r.t. bijection β denoted $\mathtt{h}_1 \sim_\beta \mathtt{h}_2$ whenever: (1) $dom(\beta) \subseteq dom(\mathtt{h}_1)$ and $rng(\beta) \subseteq dom(\mathtt{h}_2)$; (2) for all $s \in dom(\beta)$ we have that $dom(\mathtt{h}_1(s)) = dom(\mathtt{h}_2(\beta(s)))$ (for every struct in $\mathtt{h}_1$ its corresponding by β struct in $\mathtt{h}_2$ has the same fields) and (3) for all fields $f \in dom(\mathtt{h}_1(s))$ with security level L we have that $\mathtt{h}_1(s)(f) \sim_{\beta,L} \mathtt{h}_2(\beta(s))(f)$, i.e. all field values of β-corresponding structs are *L-indistinguishable*. Similarly all fields with security level H in corresponding structs have field values that are *H-indistinguishable*.

Definition 2. *(Secure information flow [5]) A program P is* secure *iff for all possible stores $\mathtt{m}, \mathtt{m}' \in Var \to \mathcal{V}$ and heaps $\mathtt{h}, \mathtt{h}', \mathtt{h}_f, \mathtt{h}'_f \in Heap$, and partial bijection β such that $(\mathtt{m}, \mathtt{h}, P) \neq \bot$ and $(\mathtt{m}', \mathtt{h}', P) \neq \bot$, and $(\mathtt{m}, \mathtt{h}, P) = (\mathtt{m}_f, \mathtt{h}_f)$ and $(\mathtt{m}', \mathtt{h}', P) = (\mathtt{m}'_f, \mathtt{h}'_f)$, and $\mathtt{m} \sim_\beta \mathtt{m}'$ and $\mathtt{h} \sim_\beta \mathtt{h}'$ imply $\mathtt{m}_f \sim_{\beta'} \mathtt{m}'_f$ and $\mathtt{h}_f \sim_{\beta',L} \mathtt{h}'_f$ for some partial bijection $\beta' \supseteq \beta$.*

The condition $\beta' \supseteq \beta$ actually models the fact that new data structures may be dynamically created at runtime and thus the bijection may become larger. An illustration of the use of the new definition follows. The main function of a simplistic e-banking program is given in Program 1.7.

```
1  int main() {
2      struct bank* bank = createBank(); struct account* account = createAccount(bank);
3      //# high
4      int amount = 100;
5      addToBalance(account, amount); }
```

Program 1.7. Banking program - main

Each procedure call on line 2 declares and creates a struct. Each transformed procedure creates a pair of structs "packed" in another struct (see Section 3.3). The public fields of the struct are assumed equal. The result of the transformation is Program 1.8. The whole program, including procedure transformations, is available as Program 1.14 in Appendix A.

```
int main() {
    struct bankRet* bankr = createBank2();
    klee_assume(bankr->bank0->count == bankr->bank1->count);

    struct accountRet* accr = createAccount2(bankr->bank0, bankr->bank1);
    klee_assume(accr->account0->wealthy == accr->account1->wealthy);
    klee_assume(accr->account0->id == accr->account1->id);

    int amount0; int amount1;
    klee_make_symbolic(&amount0, sizeof(int), "int amount0");
    klee_make_symbolic(&amount1, sizeof(int), "int amount1");

    addToBalance2(accr->account0, amount0, accr->account1, amount1);

    klee_assert(bankr->bank0->count == bankr->bank1->count);
    klee_assert(accr->account0->wealthy == accr->account1->wealthy);
    klee_assert(accr->account0->id == accr->account1->id); }
```

Program 1.8. Banking program - main transformation

In summary, whenever a new struct is allocated on the heap, the respective transformation allocates two structs and makes the appropriate assumptions about the low fields of the struct. At the end of execution, the respective assertions about the low structs have to hold. It should be noted that at this stage and particularly in the implementation of our tool, the bijection compares only the scalar values of the heap structure. The more general case, allowing cycles in the heap, is left for future work.

4 Tool and Experimental Results

This section presents a prototypical tool and some experimental results, demonstrating the potential of the proposed approach.

4.1 Tool Introduction

The proof-of-concept tool that we have built to validate our ideas is written in Perl and is based on KLEE: an automatic symbolic execution tool for high-coverage test generation built on top of the LLVM compiler infrastructure [6]. The tool works with a subset of C, including control flow statements and assignments to scalar variables, procedures and structs, where the fields have to be accessed in the standard way and no pointer arithmetic is allowed. Integer variables with addition and comparison operators are allowed as expressions.

Our tool takes as input a C program with specially annotated high variables, performs some program transformations, adds assertions as necessary and passes the resulting program to KLEE. Based on KLEE's output, the tool can possibly decide whether the tested program is secure or not and inform the developer or keep running indefinitely. In the latter case it cannot cover all paths, nevertheless it might still find a counterexample in the covered part and that would mean that the program is not secure. An optional parameter specifies when to time-out and stop searching. Whenever an error (assertion failure implying interference) is found, the *ktest-tool* tool (a part of the KLEE suite of tools) can be used to inspect and analyze the state that caused it. Additional data on the broken assertions is easily obtainable.

The tool can handle all the sample patterns of explicit and implicit information flow we could find in the literature. Furthermore it can handle patterns of information release. Because the transformations are based on semantic methods, the approach is more precise than information flow type systems resulting in the lack of false positives. On the other hand, the approach suffers from traditional weaknesses of symbolic execution, such as problems with scalability for large numbers of paths, dependence on the power of the constraint solver and difficult interaction with the environment. Moreover, the approach will benefit from further development of test input generation methods for programs with pointers. Despite the mentioned weaknesses, the tool is a very useful noninterference bug finder, as demonstrated in the rest of this section.

4.2 Implicit Flow, Explicit Flow or No Flow

Consider Program 1.9: it would be rejected as insecure by a typical information flow type system.

```
int l;
//# high
int h, j;
if ( (j + h) > 999 ) {l = −1;}
l = h;
l = l − h;
```

Program 1.9. Secure program rejected by flow-sensitive type systems

If we were to consider the program until and including the *if* statement, there would be an implicit flow; the program until and including the following statement ($l = h$) would have both implicit and explicit flows. But Program 1.9 is secure: closer inspection shows that the leaks "neutralize" each other. The results are confirmed by our tool:

```
Program impexno.c secure.
```

4.3 While-Loop Insecure Program [8]

Consider Program 1.10, taken from prior work.

```
int l;
//# high
int h, j;
while (h>0) {h--;l = h;}
```

Program 1.10. While-loop insecure program

In order to prove the insecurity of this program, the theorem proving approach using the tool KeY takes 164 steps [8]; user interaction is needed for a number of steps, such as: establishing the induction hypothesis, instantiation and unwinding of the loop etc. Our tool detects the problem as expected and within $0.2s$ (see Table 1 for detailed statistics):

```
Program while.c insecure. Flow in low variable l detected.
```

The tool is configured to stop after finding an error, but this is optional. More importantly, the developer has to only mark the high variable, which shields away the typical complexity imposed by alternative approaches (e.g. verification ones). The developer does not need to be a verification expert or to think about loop-invariants, unwinding, assertions, etc. but nevertheless has a powerful testing tool at her disposal.

4.4 e-Banking Example

The e-banking program of Section 3.4 is presented next. The interesting, security-related code is in procedure *addToBalance2*:

```
if (amount >= 10000) account->wealthy = true; else account->wealthy = false;
```

Program 1.11. Security-related part of e-Banking example

Whenever the balance gets higher than 10000, flag *wealthy* is set. The field *wealthy* of the struct *account* is public and leaks information about the balance. The latter is confirmed by our tool, producing the following output:

```
Program ebank.c insecure. Flow in low field account->wealthy.
```

4.5 Average Example

Recall that Program 1.1 computes the average of two high variables. Preconditions on the values of variables, such as *//#assume (i* > 0 *&* $j > 0$*)*, can be specified. As already discussed, the program is not secure and the tool terminates with the appropriate error:

```
Program avg.c insecure. Flow in low variable l.
```

It should be noted that our tool is precise in the sense that the errors are reproducible. The values that broke the assertions can be inspected using KLEE's *ktest-tool* in order to analyze the problem. The values generated by the *ktest-tool* are: $l0 = 0$, $l1 = 0$, $i0 = 506$, $i1 = 1609415267$, $j0 = 507$, $j1 = 485081005$.

4.6 Password Examples

Next consider the simple password check in Program 1.12.

```
1     int access, input;
2     //# high
3     int pass;
4     //# declassify (input == pass)
5     if (input == pass) access = 1; else access = 0;
```

Program 1.12. Password check

Password checking programs leak information as a part of their functionality. This can be seen if we consider the program without line 4; even when a given guess is wrong, guessing reveals that the password is or is not equal to the guess. This trivial leak is detected as expected:

```
Program passw.c insecure. Flow in low variable access.
```

As a result of the declassify statement though, the extra condition $((input0 == pass0) == (input1 == pass1))$ is added to the the assumptions; this is of course transparent to developers. In this case, our tool verifies that Program 4.6 is secure:

```
Program passwDecl.c secure.
```

Finally, we consider the following program, illustrating the use of procedures:

```
int checkPass(int input, int secret){
    int access;
    if (input == secret){ access = 1; return access;}
    else { access = 0; return access;} }
int main(){
    int pass; int guess; int result;
    //# declassify (guess == pass)
    result = checkPass (guess, pass);}
```

Program 1.13. Password example with procedures

The program is secure, as expected.

4.7 Statistics

Statistics of the discussed examples are presented in Table 1. Each of the considered programs is characterized by the respective number of instructions, explored paths and generated test cases (given by KLEE). Finally the output of the *time* program is given; *real time* is the elapsed time between start and finish, whereas *user* gives the CPU time spent in user mode and *sys* gives the CPU time in kernel mode. Typically user and sys time tell us how much CPU time the process used. The highest time by this criterion is below $0.25s$. It should be noted that the presented examples include ones whose verification would require a considerable effort if a special purpose type system would have to be developed (for each slightly different definition of security) or a theorem prover would have to be used. Because instruction cycles are cheap nowadays, a tool based on the proposed approach has a high potential of being very useful in everyday development work.

The considered examples are not particularly large, but we have tried embedding them deeper in realistic programs and the results appear to be promising. It should be noted that the definitions considered here (and in the larger part of the security literature) are of termination insensitive notions of security.

Table 1. Statistics for presented programs

Program	Instructions	Paths	Generated Tests	Time		
				real	user	sys
1.9	118	4	4	0.199s	0.062s	0.023s
1.10	57	5	5	0.134s	0.050s	0.024s
1.7	430	4	3	0.095s	0.046s	0.024s
1.1	156	4	3	0.268s	0.212s	0.030s
1.12 (insecure)	76	4	3	0.099s	0.053s	0.024s
1.12 (secure)	76	2	2	0.089s	0.051s	0.021s
1.13	153	2	2	0.108s	0.055s	0.025s

5 Related Work

To the best of our knowledge, we are the first to propose the use of symbolic execution for testing noninterference, boasting the advantages of precision and full automation not available in typical verification approaches. Nevertheless, many of the ideas presented here appear in the substantial literature on verification of noninterference. Proposed solutions traditionally rely on information flow type systems [16], a syntactic approach which offers an overapproximation and thus tends to be too conservative in practice; moreover, a new type system has to be developed every time a slight modification of the needed notion of security is needed (e.g. to allow a specific notion of declassification). On the other hand, many attempts to address noninterference have a semantic flavor [9,12]; such approaches are attractive because they suggest methods to transform the problem so as to benefit from state-of-the-art verification techniques and tools. These approaches gave rise to further work on program-logics based characterizations of noninterference [8,4]: both these rely on the idea of reducing noninterference of a program to a property of the sequential composition of the program with itself. Reasoning about such constructs is facilitated by Terauchi and Aiken's type-directed transformation [20], which takes advantage of the structure of a self-composed program and the resulting symmetry and redundancy.

In the most relevant related work Backes et al. [1] use techniques similar to ours to compute all information leaks in a program and to quantify the leaks using information-theoretic means. Information leaks in their work are characterized by an equivalence relation on secrets and can be expressed as a logical assertion on program variables; this is similar to our approach. They start with a relation expressing noninterference and gradually refine it, when counterexamples are found. Their quantitative analysis is based on computing the number and sizes of equivalence classes. The proposed approach computes an overapproximation of the information leaked by a program (and then checks if such a relation is reachable from the start state) unlike our approach, based on a finer relation (we rely on an underapproximation, only reachable states are considered). Another related and important difference is that our approach does not require a set of experiments to start with, due to the nature of symbolic execution. This is an advantage because it makes the approach more automatic. There are also differences between the approaches on other levels. First, we address a slightly different problem: testing a program for conformance with a base-line information flow policy, possibly augmented with a notion of information release, giving direct feedback to developers and being fully automatic; second, we use symbolic execution, whereas their approach uses off-the-shelf model checkers; finally, we explore qualitative policies only.

Declassification is also a well-studied topic (see [18] for an overview). The idea to use equivalence relations to characterize partial information flow was originally proposed by Cohen [7] and further developed in the literature [23,10]. A number of related articles explore the use of equivalence relations to characterize information release using flow-sensitive type systems [14,17]. Declassification is similarly handled in work using the KeY tool [8], but again the difference is that symbolic execution there is used for verification.

6 Conclusion

We have presented a novel, automatic approach to testing noninterference: the only responsibility of the developer is to identify the secrets in a candidate program and appropriately annotate them. The program is then automatically transformed and assertions are added as needed. Next, dynamic symbolic execution is used to try to break the assertions. Because typically bugs are shallow, the approach has a high potential to be very useful for testing; a major advantage of the approach is precision: any assert violation indicates a concrete, reproducible security bug. Moreover, there is no need to write complex invariants or be an expert in verification to assist the tool in any way and this can be seen as a major advantage for developers.

To illustrate the usefulness of the proposed approach, we have built a prototypical tool based on the symbolic execution tool KLEE. Our tool takes as input an annotated program from a "well-behaved" subset of C, performs the necessary program transformations and passes the resulting program to KLEE. Then, based on KLEE's output the tool informs whether a bug could be found. We have successfully tested the tool on a number of programs known from the literature and exhibiting patterns of insecurity.

Acknowledgements. We would like to thank Dries Vanoverberghe for very insightful and valuable comments on a late draft of the paper and Tatyana Doktorova for many helpful suggestions on the presentation. We also thank the reviewers for their constructive feedback.

References

1. Backes, M., Kopf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, pp. 141–153. IEEE Computer Society, Washington, DC (2009)
2. Banerjee, A., Naumann, D.A.: Stack-based access control and secure information flow. Journal of Functional Programming 15, 131–177 (2005)
3. Banerjee, A., Naumann, D.A., Rosenberg, S.: Towards a logical account of declassification. In: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, pp. 61–66. ACM, New York (2007)
4. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proceedings of the 17th IEEE workshop on Computer Security Foundations, pp. 100–114. IEEE Computer Society, Washington, DC (2004)
5. Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: TLDI 2005, pp. 103–112. ACM, New York (2005)
6. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008, pp. 209–224. USENIX Association, Berkeley, CA (2008)
7. Cohen, E.S.: Information transmission in sequential programs. In: DeMillo, R.A., Dobkin, D.P., Jones, A.K., Lipton, R.J. (eds.) Foundations of Secure Computation, pp. 297–335. Academic Press (1978)
8. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. Technical Report S-412 96, Chalmers University of Technology and Göteborg University (2004)
9. Focardi, R., Gorrieri, R.: A taxonomy of security properties for process algebras. Journal of Computer Security 3(1), 5–34 (1995)
10. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. In: Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004), pp. 186–197. ACM Press, NY (2004)
11. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 213–223. ACM, New York (2005)
12. Joshi, R., Leino, K.R.M.: A semantic approach to secure information flow. Science of Computer Programming 37, 113–138 (2000)

13. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19, 385–394 (1976)
14. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing robust declassification and qualified robustness. Journal of Computer Security 14(2), 157–196 (2006)
15. Pottier, F., Simonet, V.: Information flow inference for ML. SIGPLAN Not. 37, 319–330 (2002)
16. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
17. Sabelfeld, A., Myers, A.C.: A Model for Delimited Information Release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004)
18. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: Proceedings of the 18th IEEE Workshop on Computer Security Foundations, pp. 255–269. IEEE Computer Society, Washington, DC (2005)
19. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. SIGSOFT Software Engineering Notes 30, 263–272 (2005)
20. Terauchi, T., Aiken, A.: Secure Information Flow as a Safety Problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
21. Volpano, D.M., Smith, G.: A Type-Based Approach to Program Security. In: Bidoit, M., Dauchet, M. (eds.) CAAP/FASE/TAPSOFT 1997. LNCS, vol. 1214, pp. 607–621. Springer, Heidelberg (1997)
22. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proceedings 16th IEEE Computer Security Foundations Workshop, pp. 29–43 (July 2003)
23. Zdancewic, S., Myers, A.C.: Robust declassification. In: Proceedings of the 14th IEEE Workshop on Computer Security Foundations, CSFW 2001, pp. 15–23. IEEE Computer Society, Washington, DC (2001)

A Sample Transformation

```
struct account {
    struct account* next; int balance;
    int wealthy; int id;};

struct bank {
    struct account* head;
    int count;};

struct bankRet {
    struct bank* bank0;
    struct bank* bank1;};

struct bankRet* createBank2(){
    struct bank* bank0 = malloc(sizeof(struct bank));
    struct bank* bank1 = malloc(sizeof(struct bank));
    bank0->head = 0; bank1->head = 0;
    bank0->count = 0; bank1->count = 0;
    struct bankRet* bankr = malloc(sizeof(struct bankRet));
    bankr->bank0 = bank0; bankr->bank1 = bank1;
    return bankr; };

struct accountRet {
    struct account* account0; struct account* account1; };

struct accountRet* createAccount2(struct bank* bank0,struct bank* bank1) {
    bank0->count++; bank1->count++;
    struct account* account0 = malloc(sizeof(struct account));
    struct account* account1 = malloc(sizeof(struct account));
    account0->next = bank0->head; account1->next = bank1->head;
    account0->id = bank0->count; account1->id = bank1->count;
    account0->balance = 0; account1->balance = 0;
    account0->wealthy = false; account1->wealthy = false;
    bank0->head = account0; bank1->head = account1;
    struct accountRet* accr = malloc(sizeof(struct accountRet));
    accr->account0 = account0; accr->account1 = account1;
    return accr; }

int getBal(struct account* acct)
{ return acct->balance;}

struct intRet{
    int r1; int r2; };

struct intRet* getBal2(struct account* acct1,struct account* acct2)
{
    struct intRet* intR = malloc(sizeof(struct intRet));
    intR->r1=acct1->balance; intR->r2=acct2->balance;
    return intR;}

void addToBalance2 (struct account* account0, int amount0,
                    struct account* account1, int amount1) {
    if (amount0 >= 10000) account0->wealthy = true;
    else account0->wealthy = false;
    if (amount1 >= 10000) account1->wealthy = true;
    else
    account1->wealthy = false;
    account0->balance += amount0;
    account1->balance += amount1;}
int main() {
//Body of Program 1.8 goes here
}
```

Program 1.14. Transformation and noninterference specification of Program 1.7

Defining Distances for All Process Semantics*

David Romero Hernández and David de Frutos Escrig

Dpto. Sistemas Informáticos y Computación
Facultad CC. Matemáticas, Universidad Complutense de Madrid, Spain
dromeroh@pdi.ucm.es, defrutos@sip.ucm.es

Abstract. Recently several authors have proposed some notions of distance between processes that try to quantify "how far away" is a process to be related with some other with respect to a certain semantics. These proposals are usually based on the simulation game, and therefore are mainly defined for simulation semantics or other semantics more or less close to these. These distances have a local character since only one of the successors of each state is taken into account in their computation. Here, we present an alternative proposal exploiting the fact that processes are trees. We define the distance between two of them as the cost of the transformations that we need to apply to get two processes related by the corresponding semantics. Our new distances can be uniformly defined for all the semantics in the ltbt-spectrum.

1 Introduction and Motivation

We are thirsty, but we hate those boring machines that only offer a few products. But we are very happy with the machine at our institution that offers a wide variety of beverages. So, each day we can go to the machine with our selected chosen item and get our bottle. But if some day the machine is out of that, then we have to choose another drink, and that day we are not so happy... Certainly, if it is only a single kind of drink that is missing we will probably stay very happy, but if something happens and the machine today offers only a single beverage, then we will be probably not so happy...

We have a collection of items in some numbered "collector desk". We look for a product by reminding its assigned number. But today, for some reason, somebody has interchanged two items and then if we look for one of them we will find the other, and we will have to make our job using it, obviously not so well as if we had found the desired item. But if one day the desk collapses and somebody has to put the items in the places without knowing their places, and he is wrong in all the cases, then for sure we will fail when looking for any of the items.

There is a lottery in the club and everybody expects that all the balls corresponding to the sold tickets will be in the bag. But for some reason the set of balls does not exactly corresponds with that of sold tickets. Certainly, the raffle

* Partially supported by the Spanish projects TESIS (TIN2009-14312-C02-01), DESAFIOS10 (TIN2009-14599-C03-01) and PROMETIDOS S2009 / TIC-1465.

is not fair, but how much unfair? An obvious reply will take into account the number of tickets that were not presented in the bag.

All these are simple "real life" situations, that we can easily model by means of a process with some kind of choice (either internal or external), where the number of choices in the initial model is large. This corresponds to the ideal situation, but if something is wrong, the choice is not the same and this would produce a process that does not fully satisfy our expectations. Then, we want to measure how far away we are from the desired behavior.

At the technical level, we want to define adequate distances between processes which measure, in a reasonable way, the gap between any behavior and the corresponding "expected" one. Of course, if we are talking about behaviors, then the first thing to fix is the reference semantics. There are plenty of proposals for process semantics, which have been presented in several versions of the linear-time branching-time (ltbt) spectrum [14, 5].

In the last few years we can find in the literature several proposals for distances between processes associated to a certain range of process semantics, but in all the cases far from being applicable to the whole spectrum [1]. Most of them, if not all, base their definitions on the (bi)simulation game that characterizes (bi)simulations between processes [11, 3, 2]. Although these are branching semantics, their co-inductive characterizations provide a (partially) local way to compare processes by considering, one by one, all the possible transitions from the compared states. The rules of these games state that any a-transition should be replicable by another a-transition of the other process; otherwise, we would have found a proof of non-bisimilarity (or that of non existence of a simulation) of the two compared processes.

Starting from them, the modified distance games allow the defender to reply an a-move by means of another b-move, where we could have $a \neq b$. Then he should pay to the attacker as the provided distance between these two actions, $d(b, a)$, states. Obviously, the attacker tries to maximize his profit by making his appropriate moves, while the defender tries to minimize them with his moves. Finally, the value of this game provides the (bi)simulation distance between the two compared processes w.r.t. the provided distance between actions, d.

Certainly, we could agree about the naturalness of these approaches, which in fact are proved to be correct, in the sense that the distance between two processes is 0, if and only if, they are (bi)similar. But, if we apply these distances to the formalizations of our three examples above, considering the discrete distance between processes (given by $d(a, a) = 0$ and $d(a, b) = 1$ if $a \neq b$) and taking p_n as the corresponding "ideal" behavior, where n is the desired number of choices, p_{n-1} the slightly "incorrect" approximation, and p_1 the poor approximation with a single choice, we obtain $d(p_n, p_{n-1}) = 1$, probably as expected, but a bit surprisingly, we also have $d(p_n, p_1) = 1$. In our opinion, it would be much more informative to get instead $d(p_n, p_1) = n - 1$, in such a way that if we consider the general approximation p_k of the ideal process p_n, which offers exactly k of the actions, then we have $d(p_n, p_k) = n - k$, and also $d(p_k, p_1) = k - 1$.

Why these known distances between processes fail to notice the quantity of choices that are lost? This is simple: just because the "local" character of the distance game. It certainly observes any of the lost actions, but this only happens at different plays of the game, each of them producing a profit $d(a_i, a_1) = 1$ to the attacker, so that the "final" profit (the value of the game, that generates $d(p_n, p_k)$) is always 1, when $k < n$, whatever the number of lost choices, $n - k$ was.

Even if we definitely advocate for a distance which will get $d(p_n, p_k) = n - k$, and in fact we will provide such a distance, we could still look for "justifications" of the distance produced by the game approaches: if we only study the computations of the processes "one by one" (certainly step by step, in order to get the characterizations of the branched semantics, instead of just the trace semantics) then we will never realize that several choices were lost at the same time (we only notice that "each one of them" was lost, but this is not enough).

What is the problem? (and then, how can we solve it?). Simulations define branched behaviors that are roughly trees which consider all computations of each process together [15]. These trees can be seen as "global" values (or full behaviors) of the process. Equality (resp. containment) of trees is defined (in a coalgebraic way) by bisimulation (resp. simulation), and then (in a partially local way) by the bisimulation (resp. simulation) game. We could say that this is the "magic" of (bi)simulation, but when we introduce distances between actions and we try to lift them up to the branched behaviors by means of the distance game, then we find that the obtained values are not able to capture the branched structure, because the value of the game is obtained by the minimax algorithm, which chooses the critical path generated by the application of the optimal strategies of both players, but is not able to "add" the differences observed at different branches. Indeed, we are using *max* instead of *add* when computing the value of the distance games, and then we cannot capture the "global" distances as required by the situations in our introductory examples.

As a matter of fact, the reason why the plain (bi)simulation game is able to capture a branched semantics is because we are interested in checking equality. This can be done by a boolean function which only considers boolean values, e.g. 0 for equal and 1 for unequal. Then, any move that the defender cannot match produces some 1, so the application of max would produce the value 1. But in this discrete domain, max can also be used to compute addition, which in fact coincides with disjunction. Instead, as soon as we have a more informative domain for the values of distances, then max and add become two different operations. It is clear that the first is only able to transmit a partial information about the branched behaviors, while addition collects all the "local" differences to compute a much more reasonable concept of global distance.

Once we have our mechanism to compute our global bisimulation distance, we will see that a quite simple customization, gives us a nice notion of distance for each of the semantics in the ltbt-spectrum. Roughly we just need to combine the preorder defining each of the other semantics in the ltbt-spectrum—see Fig. 1—(or equivalently, the inequalities that are included in their axiomatizations), with the rules which produce the values of our bisimulation distance, .

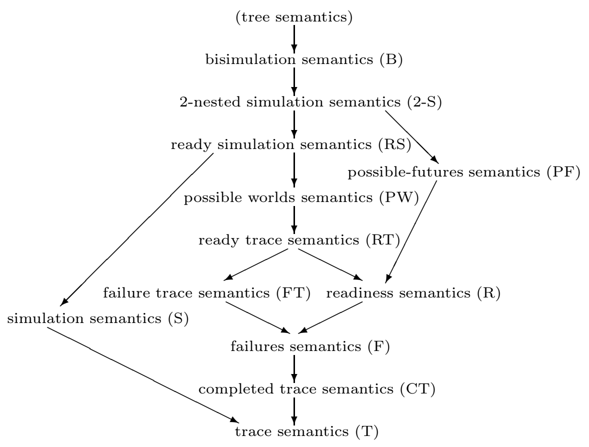

(tree semantics)
bisimulation semantics (B)
2-nested simulation semantics (2-S)
ready simulation semantics (RS)
possible-futures semantics (PF)
possible worlds semantics (PW)
ready trace semantics (RT)
failure trace semantics (FT)
readiness semantics (R)
simulation semantics (S)
failures semantics (F)
completed trace semantics (CT)
trace semantics (T)

Fig. 1. The ltbt-spectrum

2 Preliminaries

All the semantics from the ltbt-spectrum [14, 5, 6] that we consider can be defined over arbitrary (possibly infinite) processes whose operational semantics is defined by means of a labelled transition system (lts) $\mathcal{P} = (Proc, Act, \rightarrow)$. We will use the classical notation $p \xrightarrow{a} p'$ to represent the transitions of processes. Moreover, it is also useful to have a syntactic notation for representing finite processes. We will use BCCSP [14, 5].

Definition 1. *Given a set of actions Act, the set BCCSP(Act) of processes is that defined by the BNF-grammar: $p ::= \mathbf{0} \mid ap \mid p + q$. The very well known operational semantics of BCCSP [14, 5] is defined by:*

$$(1)\ \frac{}{ap \xrightarrow{a} p} \qquad (2)\ \frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \qquad (3)\ \frac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'}$$

In order to simplify the presentation, we start by considering a classic (symmetric) distance between actions $\overline{d} : Act \times Act \rightarrow \mathbb{N}$ with $\overline{d}(a,b) = \overline{d}(b,a)\ \forall a,b \in Act$. Let us recall that any distance has besides to satisfy the following two properties: $\overline{d}(a,b) = 0 \Leftrightarrow a = b$, $\overline{d}(a,c) + \overline{d}(c,b) \leq \overline{d}(a,b)\ \forall a,b,c \in Act$. Later, in Sect. 6, we will discuss when an asymmetric quasi-distance could be used instead, and which is the intuitive meaning of the distances between processes that can be obtained using them.

We can represent any process as a tree (finite or infinite). Then a first approach to the definition of a distance measuring how far away is a process p of being equivalent to some other q, would study the differences between the trees which represent both processes, seeing what we have to change in order to turn them into two equivalent processes. Let us start by considering ordered trees, where we have a set of ordered sons for each node of the tree. We can present these trees as terms $\sum_{i=1}^{n} a_i p_i$, where $n = 0$ produces the empty tree $\mathbf{0}$.

Definition 2. *We say that an ordered tree p is at most at distance d from another tree q, w.r.t. the symmetric distance between actions $\overline{d}$, and then we write $d_{\overline{d}}(p,q) \leq d$, if and only if:*

- $d \geq 0$ *and* $p = q = \boldsymbol{0}$*, or*
- $p = \sum_{i=1}^{n} a_i p_i$*,* $q = \sum_{i=1}^{n} b_i q_i$*, and* $d = \sum_{i=1}^{n} d_i + \sum_{i=1}^{n} \overline{d}(a_i, b_i)$ *with* $d_{\overline{d}}(p_i, q_i) \leq d_i \ \forall i = 1 \dots n$*.*

It is clear that this definition only produces (finite) distances between trees which have exactly the same structure. For instance, for the processes $p = a + b$ and $q = c + d$ we obtain $d_{\overline{d}}(p,q) \leq \overline{d}(a,c) + \overline{d}(b,d)$. However, if we want to compare $r = a$ and $s = b + c$, we will get no finite value d for which $d_{\overline{d}}(r,s) \leq d$, and then we could say that $d_{\overline{d}}(r,s) = \infty$.

Moreover, when comparing two infinite trees we will only obtain a finite distance if the number of disagreements between them is finite. Certainly, this will be the expected result if we simply add the cost of all these mismatches. But it is important to notice that the simple approach here proposed will never been able to compute distances between infinite trees with infinitely many mismatches. Therefore, in the following we will restrict ourselves to the case of finite processes, leaving the case of infinite processes for our conclusions.

It is also true that in this simple scenario when we compare two trees with the same structure, we could directly obtain the distance between them. But we preferred to introduce this indirect presentation using bounds, because this will be later needed when considering more complicated scenarios. Certainly, the order between the summands is important in ordered trees. As a consequence, if we consider $p' = b + a$ and $\overline{d}(a,b) = 1$, we obtain $d_{\overline{d}}(p,p') \leq 2$, and definitely not $d_{\overline{d}}(p,p') \leq 0$.

But trees representing processes are unordered: each node has attached a set of subtrees, and this even implies that no identical sons are allowed. In fact, this corresponds to considering processes "up-to" bisimulation. Then, in order to define a reasonable and well behaved notion of (bound of the) distance between processes, we apply a push-out of the definition above and that of bisimulation. So we get a rewriting procedure where we try to change any of the two compared processes into the other: Either changing one of the actions in a tree by other, but then we need to pay for it, as stated by the function $\overline{d}$; or we simply apply for free to any subtree of them any of the bisimulation axioms:

$$(B1)\ x + y \simeq y + x \qquad\qquad (B2)\ x + x \simeq x$$

$$(B3)\ (x + y) + z \simeq x + (y + z) \qquad\qquad (B4)\ z + \mathbf{0} \simeq z$$

Obviously, this procedure is non-deterministic and different possible applications lead us to several (different) "distances", and this is why we need to talk about "bounds" of the distance between p and q.

Definition 3. *We say that an unordered tree p is at most at distance d from another tree q, w.r.t. the symmetric distance between actions $\overline{d}$, and then we write $d_{\overline{d}}(p,q) \leq d$, if and only if:*

- *(C1) $p = ap'$, $q = bp'$, and $d \geq \overline{d}(a,b)$, or*
- *(C2) $p = p' + r$, $q = q' + r$, and $d \geq d_{\overline{d}}(p',q')$, or*
- *(C3) $p = ap'$, $q = aq'$, and $d \geq d_{\overline{d}}(p',q')$, or*
- *(C4) $d \geq 0$ and q can be obtained from p by application of (B1)-(B4), or*
- *(C5) There exist r, d' and d'' s.t. $d' \geq d_{\overline{d}}(p,r)$, $d'' \geq d_{\overline{d}}(r,q)$ and $d \geq d' + d''$.*

$(C1)$ corresponds to a single application of Def.1 producing a single change at the root of p. $(C2)$ and $(C3)$ allow the contextual application of $(C1)$ at any place, thus generating the possibility to change any action a in p by any other action b, paying $\overline{d}(a,b)$ for it. $(C4)$ introduces the possibility of transforming any process p into another bisimilar q, for free. Finally, $(C5)$ tells us that by adding the costs of the steps of any transformation that produces q from p, we obtain an upper bound of the distance between p and q.

We could obtain "the" distance between two trees by considering the minimal value d for which we have $d_{\overline{d}}(p,q) \leq d$. But unfortunately this corresponds to a global study of the set of derivations that produces the bounds. We prefer to avoid the explicit consideration of those "exact" distances, since it seems not possible to introduce the computation of these minimal values in our approach in a manageable way.

Moreover, it is easy to see that these distances would correspond to the shortest path in the graph whose nodes are processes, and the valued arcs correspond to the cost of the basic allowed transformations between them induced by rules $(C1) - (C4)$; $(C5)$ states somehow the Bellman's optimality principle. As a consequence we do not need "all the strength" of rule $(C5)$ which allows us to compose a path by concatenating two arbitrary paths, but it certainly includes the (needed) case in which the first path is a single step. However, by including this general rule we obtain a more symmetric definition, where those single steps do not need any separate treatment.

Now, by applying $(B1)$ we obtain $d_{\overline{d}}(a+b, b+a) \leq 0$. Moreover, we can compare trees that have not the same structure. For instance, we can transform for free $r = a$ into $r' = a + a$, and then we obtain $d_{\overline{d}}(r', s) \leq \overline{d}(a,b) + \overline{d}(a,c)$, from where we conclude $d_{\overline{d}}(r,s) \leq \overline{d}(a,b) + \overline{d}(a,c)$. Although it could be the case that we could obtain other "lower bounds" of this distance, as we will discuss later in Sect.3 (page 179).

Next, we present another equivalent definition of our bisimulation distance between processes. We consider processes up-to bisimulation, and following the coinductive approach, we will consider a collection of "distance relations" $\{G_m \mid m \in \mathbb{N}\}$, that are those generated by the SOS-rules below:

$$(1)\ \frac{}{p\ G_n\ p} \quad (2)\ \frac{p\ G_n q}{ap\ \ G_{n+\overline{d}(b,a)}\ \ bq} \quad (3)\ \frac{p\ G_n\ p'}{p+q\ \ G_n\ \ p'+q} \quad (4)\ \frac{p\ G_n\ q \quad q\ G_{n'} r}{p\ \ G_{n+n'}\ \ r}$$

Proposition 1. *For all $n \in \mathbb{N}$, we have $p\, G_n\, q$ if and only if $d_{\overline{d}}(p,q) \leq n$.*

Proof. It is clear the correspondence between the rules defining both collections of relations. We will only remark that $(C4)$ corresponds to working up-to bisimilarity, while rule (2) covers both $(C1)$ and $(C3)$ at the same time. □

Remark 1. It would be possible to mix these rules in several ways, even reducing its total number. But we prefer this presentation, where basic transformations are shown in isolation. This definitely simplifies the rule-induction proofs in the following.

3 Simulation Distance

Starting from the bisimulation distance presented above, next we introduce the simulation distance. We start by recalling the definition of simulation.

Definition 4. *A simulation is a relation S between processes such that whenever we have pSq, for every $a \in Act$, if $p \xrightarrow{a} p'$ then, there exists some q', such that $q \xrightarrow{a} q'$ and $p'Sq'$. We say that process p is simulated by process q, or that q simulates p, written $p \sqsubseteq_S q$, if there exists a simulation S such that pSq.*

We want to define by means of rules the relations that indicate how far away is a process p of being simulated by another q. Of course, when q simulates p, the simulation distance between them (in this direction) will be 0. When this is not the case, we will need to change the tree that represents q, to get a process that simulates p, paying for each modification.

Definition 5. *Given two processes p and q, we say that the simulation distance from q to p is at most $m \in \mathbb{N}$, w.r.t. the symmetric distance between actions $\overline{d}$, and then we write $d^S_{\overline{d}}(p,q) \leq m$, if we can derive $p\, G^S_m\, q$ applying the following rules:*

$$(1)\,\frac{p \sqsubseteq_S q}{p\; G^S_n\; q} \quad (2)\,\frac{p\, G^S_n\, q}{ap\; G^S_{n+\overline{d}(b,a)}\; bq} \quad (3)\,\frac{p\, G^S_n\, p'}{p+q\; G^S_n\; p'+q} \quad (4)\,\frac{p\, G^S_n\, q \quad q\, G^S_{n'}\, r}{p\; G^S_{n+n'}\; r}$$

In other words, we can say that the simulation distance is obtained by computing the bisimulation distance up to the similarity relation. This can also be expressed in a transformational way: we look for the "minimal changes" that we need to make in q to get a process q' which simulates p.

Remark 2. Note that in this case we do not need to explicitly say that we work up-to bisimilarity, since when $q \sim q'$, we also have $q' \sqsubseteq_S q$, and then by applying (1) we can transmute q into q' for free, whenever this is needed.

Next we present a very simple example to illustrate how our definition works.

Example 1. We consider the lexicographic distance between actions induced by the lexicographic order, so we have $\overline{d}(a,b) = 1$, $\overline{d}(a,c) = 2$, and so on. Let us consider the processes $p = a(b+c)$ and $q = ab + ad$. Then, it is easy to see that $p \not\sqsubseteq_S q$ and $q \not\sqsubseteq_S p$. Let us start seeing how far away we are of having $q \sqsubseteq_S p$. It is clear that $q \sqsubseteq_S p'$, where p' is obtained from p by turning c into d, so that we define $p' = a(b+d)$. Therefore, we have $d^S_{\overline{d}}(p,q) \leq \overline{d}(c,d) = 1$. Next we see in detail how we can derive $q\, G^S_1\, p$ applying the rules in Def. 5:

$$\dfrac{\dfrac{q \ \sqsubseteq_S \ p'}{q \ G^S_0 \ p'}(1) \qquad \dfrac{\dfrac{d \ G^S_{\overline{d}(c,d)} \ c}{b+d \ G^S_1 \ b+c}(3)}{p' \ G^S_{1+\overline{d}(a,a)} \ p}(2)}{q \ G^S_1 \ p}(4)$$

If we consider the opposite distance, which measures at which extent we have (not) $p \sqsubseteq_S q$, the shortest way to obtain some q' with $p \sqsubseteq_S q'$ is to duplicate (for free) the subtree below a, and then we change one of the b actions into c, paying for it $\overline{d}(b,c)$. So we obtain $q' = a(b+c)+ad$, which produces $d_{\overline{d}}(q,p) \leq \overline{d}(b,c) = 1$. This can be inferred applying our rules as follows:

$$\dfrac{\dfrac{p \ \sqsubseteq_S \ q'}{p \ G^S_0 \ q'}(1) \qquad \dfrac{\dfrac{\dfrac{\dfrac{c \ G^S_1 \ b}{b+c \ G^S_1 \ b+b}(3)}{b+c \ G^S_1 \ b}(B2)}{a(b+c) \ G^S_{1+\overline{d}(a,a)} \ ab}(2)}{q' \ G^S_{1+\overline{d}(a,a)} \ q}(3)}{p \ G^S_1 \ q}(4)$$

Next we compare the definitions of simulation distance based on the simulation game with ours.

Definition 6. *(Simulation game) Given two LTSs, L_1 and L_2, we call configurations the pairs (p,q), with $p \in L_1$ and $q \in L_2$. The simulation game is played by two players: the attacker $\mathbb{A}$ and the defender $\mathbb{D}$. The initial configuration of the game deciding if $p_0 \sqsubseteq_s q_0$, is just the pair (p_0, q_0). A round of the game, when the current configuration is (p,q), proceeds as follows:*

1. *$\mathbb{A}$ chooses a transition in L_1: $p \xrightarrow{a} p'$.*
2. *$\mathbb{D}$ must execute the same action at the other side of the board (L_2): $q \xrightarrow{a} q'$.*
3. *The game proceeds in the same way from the new configuration (p', q').*

The winner of the game is defined by the following rules: (1) Any infinite game is a win for $\mathbb{D}$. (2) $\mathbb{D}$ also wins if $\mathbb{A}$ cannot make any new move. (3) $\mathbb{A}$ wins when he makes a move, that $\mathbb{D}$ cannot reply with a transition from L_2.

Theorem 1. *$p \sqsubseteq_S q$ (resp. $p \not\sqsubseteq_S q$) if and only if $\mathbb{D}$ (resp. $\mathbb{A}$) has a winning strategy for the simulation game starting at (p,q).*

The simulation game can be turned into a (classical) simulation distance game by allowing to reply any a-move by some b-move with $b \neq a$, but then the defender should pay $\overline{d}(b, a)$ to the attacker for the mismatch. The value of the game provides the "classical" simulation distance between p and q [1]. We can obtain a coinductive characterization, which also provides a more general definition covering also infinite processes, as follows:

Definition 7. *A family of relations between processes $(S_n)_{n\in\mathbb{N}}$ is a classical simulation distance family (csdf), w.r.t. the symmetric distance between actions $\overline{d}$, when for each $(p, q) \in S_n$ we have the diagram:*

$$\begin{tikzcd} p \arrow[d, "\forall\, a"'] & S_n & q \arrow[d, "\exists b"] \\ p' & S_{n-\overline{d}(b,a)} & q' \end{tikzcd} \qquad \Longrightarrow$$

We say that p and q are at most at classical simulation distance n, and then we write $d^S_{\overline{d}}(p, q) \leq n$, iff there is some csdf *$(S_n)_{n\in\mathbb{N}}$ such that pS_nq.*

Example 2. Using the distance relation $\overline{d}$ at Example 1, if we apply our Def. 5, we get $d^S_{\overline{d}}(a + d, b + e) \leq 2$, but we cannot obtain $d^S_{\overline{d}}(a + d, b + e) \leq 1$. Instead, we can get a *csdf* taking $S_1 = \{(a + d, b + e)\}$ and $S_0 = \{(\mathbf{0}, \mathbf{0})\}$, because $a + d \xrightarrow{a} \mathbf{0}$ can be replied by $b + e \xrightarrow{b} \mathbf{0}$ with cost 1. If we consider the discrete distance $\overline{d}$ defined by $\overline{d}(a, b) = 1 \Leftrightarrow a \neq b$, then we obtain $d_{\overline{d}}(\sum_{i=1}^n a_i, a_0) \leq n$, but $d_{\overline{d}}(\sum_{i=1}^n a_i, a_0) \not\leq n - 1$, while using the classical simulation game approach we can take $S_1 = \{(\sum_{i=1}^n a_i, a_0) \mid n \in \mathbb{N}\}$ and $S_0 = \{\mathbf{0}, \mathbf{0}\}$, because any move $\sum a_i \xrightarrow{a_i} \mathbf{0}$ can be replied by $a_0 \xrightarrow{a_0} \mathbf{0}$ with cost 1.

Even if we consider that our "global simulation distance", defined at Def. 5, is the most adequate way to turn the simulation relation into a quantitative distance between processes, next we will show the flexibility of our approach showing that a simple variation of the system of rules defining it produces a characterization of the "classical" operational simulation distance, defined at Def. 7. We only need to change rule (3), taking instead the new rule (3′), thus obtaining the revised system:

$$(1)\,\frac{p \sqsubseteq_S q}{p\ H^S_n\ q} \quad (2)\,\frac{p\,H^S_n q}{ap\ H^S_{n+\overline{d}(b,a)}\ bq} \quad (3')\,\frac{p\,H^S_n\,p' \;\; q\,H^S_{n'}\,q'}{p+q\ H^S_{max\{n,n'\}}\ p'+q'} \quad (4)\,\frac{p\,H^S_n\,q \;\; q\,H^S_{n'}r}{p\ H^S_{n+n'}\ r}$$

We will see that the use of max in this rule produces that only the cost of the simulation of the computation that is "harder to simulate" is taken into account when generating the relations H^S_n. As a consequence, the family $(H^S_n)_{n\in\mathbb{N}}$ is a *csdf* that accurately generates the classical simulation distance:

Theorem 2. *1. $(H^S_n)_{n\in\mathbb{N}}$ is a csdf.*
2. If $(S_n)_{n\in\mathbb{N}}$ is a csdf then $S_n \subseteq H^S_n$.

Proof. • $\underline{1|}$ We prove that $(H^S_n)_{n\in\mathbb{N}}$ satisfies the definition of *csdf*, by rule induction on the definition of H^S_n:

$$\begin{tikzcd}
(1): & p \arrow[d, "\forall a"'] & H_n^S & q \arrow[d, "\exists\ b=a"] & (\overset{df}{\Leftarrow} p \sqsubseteq_S q) \\
& p' & H_n^S & q' & (\overset{(1)}{\Leftarrow} p' \sqsubseteq_S q')
\end{tikzcd}$$

$$\begin{tikzcd}
(2): & ap \arrow[d, "a"'] & H^S_{n+\overline{d}(b,a)} & bq \arrow[d, "b"] \\
& p & H^S_{n+\overline{d}(b,a)-\overline{d}(b,a)} & q \\
& & \Downarrow & \\
& p & H_n^S & q
\end{tikzcd}$$

$$\begin{tikzcd}
(3'): & p+q \arrow[d, "a"'] & H_n^S & p'+q' \arrow[d, "b"] \\
& p'' & H^S_{n-\overline{d}(b,a)} & p''' \\
& & & \Uparrow pH_n^S p' \wedge qH^S_{n'} q' \ with\ n \geq n' \\
& p \arrow[d, "a"'] & H_n^S & p' \arrow[d, "b"] & \textit{(by i.h.)} \\
& p'' & H^S_{n-\overline{d}(b,a)} & p'''
\end{tikzcd}$$

$$\begin{tikzcd}
(4): & p \arrow[d, "a"'] & H^S_{n+n'} & r \arrow[d, "c"] & (\Leftarrow^{i.h.(4)}) & p \arrow[d, "a"'] & H_n^S & q \arrow[d, "b"] & q \arrow[d, "b"'] & H^S_{n'} & r \arrow[d, "c"] \\
& p' & H^S_{n+n'-(\overline{d}(c,b)+\overline{d}(b,a))} & r' & & p' & H^S_{n-\overline{d}(b,a)} & q' & q' & H^S_{n'-\overline{d}(c,b)} & r' \\
& & \Downarrow \overline{d}(c,a) \leq \overline{d}(c,b)+\overline{d}(b,c) & & & & & & & & \\
& p' & H^S_{n+n'-\overline{d}(c,a)} & r'
\end{tikzcd}$$

- $\underline{2|}$ We use complete induction on the depth of p:

$$0\ S_n\ q \ \Rightarrow\ 0\ \sqsubseteq_s\ q\ \Rightarrow\ 0\ H_n^S\ q$$

Let $p = ap'_a + r$ and $q = bq'_b + q''$ such that

$$\begin{tikzcd}
p = ap'_a + r \arrow[d, "\forall\ a"'] & S_n & q = bq'_b + q'' \arrow[d, "\exists\ b"] \\
p_a & S_{n-d(b,a)} & q'_b
\end{tikzcd} \quad \Longrightarrow$$

Then we have:

$$p'_a\ S_{n-d(b,a)}\ q'_b\ \Rightarrow\ p'_a\ H^S_{n-d(b,a)}\ q'_b\ \Rightarrow\ ap'_a\ H_n^S\ bq'_b\,.$$

This happens for all the summands of p, which means that up-to idempotence of $+$, we can assume that $p = \sum a_i p'_{a_i}$ and $q = \sum b_i q'_{b_i} + r$, where for all $i \in I$ we have $a_i p'_{a_i}\ H_n^S\ b_i q'_{b_i}$; and finally we conclude $p\ H_n^S\ q$, by applying repeatedly the rule (3), and (1) to get $\mathbf{0}\, H_n^S\, r$. □

It is interesting to note that we have not used the transitivity rule (4) at all in the previous proof, which means that we can obtain the following corollary:

Corollary 1. *If we define $H_n^{S'}$ as H_n^S, but removing the transitivity rule (4), we have that $H_n^{S'}$ is equivalent to H_n^S.*

Proof. From the fact that H_n^S is a *csdf* we immediately obtain that $H_n^{S'}$ is too. But since in the proof of Th.2 we do not use the transitivity rule (4), we have also proved there that for any *csdf* $(S_n)_{n\in\mathbb{N}}$ we have $S_n \subseteq H_n^{S'}$. Then we have $H_n^S \subseteq H_n^{S'}$ and from their definitions we immediately obtain $H_n^{S'} \subseteq H_n^S$, from where we can conclude that H_n^S is equivalent to $H_n^{S'}$. □

Note however, that when we consider the sum between branches in rule (3) instead of the maximum, as done in Def.5, we need indeed the transitivity rule, because in this case it cannot be "derived" from the rest of the rules. The following example shows the necessity of this rule.

Example 3. Consider the processes $p = a$ and $q = b + c$, if we want to simulate q by p, we need to change action a into both b and c. However, it is possible that it would be better to transform first a into some a', and then this a' into b and c. Without the transitivity rule we cannot generate this elaborated transformation, and then we would not get the "desired" global simulation distance. Instead, when we consider the classical simulation distance, by the triangular inequality, it is not useful to transform first a into some a' and then a' into b, because that will be always worse than transforming directly a into b.

This example also illustrates the possible interest of such an elaborated procedure in order to efficiently simulate several branches of the simulated process by a common branch of the simulating one. The cost of the transformation of a into a' is shared by the two branches, and then we only pay once for it. Note that the use (for free) of idempotence allows this double use of a common branch.

4 Bisimulation Distance

Using the bisimulation game, we can define a "classical" bisimulation distance as done in [7]. It measures how far away are two processes of being bisimilar.

Theorem 3 ([10, 12]). *$p \sim q$ (resp. $p \not\sim q$) if and only if $\mathbb{D}$ (resp. $\mathbb{A}$) has a winning strategy for the bisimulation game starting at (p, q).*

Definition 8. *A family $(R_n)_{n\in\mathbb{N}}$ is a classical bisimulation distance family (cbdf), w.r.t. the symmetric distance relation between actions $\overline{d}$, when it satisfies*

$$\begin{tikzcd}
p \arrow[dd, "\forall\, a"'] & R_n & q \arrow[dd, "\exists b"] & & p \arrow[dd, "\exists a"'] & R_n & q \arrow[dd, "\forall\, b"] \\
& \Longrightarrow & & \wedge & & \Longleftarrow & \\
p' & R_{n-\overline{d}(b,a)} & q' & & p' & R_{n-\overline{d}(a,b)} & q'
\end{tikzcd}$$

We say that p and q are at most at classical bisimulation distance n, and then we write $d_{\overline{d}}^B(p,q) \leq n$, iff there is some cbdf *$(R_n)_{n\in\mathbb{N}}$ such that pR_nq.*

From the symmetric definition of bisimulation we immediately obtain that our classical bisimulation distance is also symmetric.

Proposition 2. *For any two processes p, q and any $n \in \mathbb{N}$, we have $d^B_{\overline{d}}(p,q) \leq n$ if and only if $d^B_{\overline{d}}(q,p) \leq n$.*

Following the same ideas that we used in Sect.3, we can obtain a rule system that produces the biggest relations H^B_n that state that the related processes are at most at distance n to be bisimilar.

Definition 9. *We consider the family of relations $(H^B_n)_{n\in\mathbb{N}}$ which are generated by applying the following rules, modulo bisimulation:*

$$(1)\,\frac{}{p\;H^B_n\;p} \quad (2)\,\frac{p\,H^B_n\,q}{ap\;H^B_{n+\overline{d}(b,a)}\;bq} \quad (3')\,\frac{p\,H^B_n\,p' \quad q\,H^B_{n'}\,q'}{p+q\,H^B_{max\{n,n'\}}\,p'+q'} \quad (4)\,\frac{p\,H^B_n\,q \quad q\,H^B_{n'}\,r}{p\;H^B_{n+n'}\;r}$$

It is nice to observe the close similarity between the rules defining this classical bisimulation distance and our previous bisimulation distance in Sect.2: in fact, if we change the max operator in $(3')$ by addition, then it is easy to check that the obtained definition is equivalent to our original one.

Remark 3. It is clear that we can remove the "up-to" bisimulation at the definition above if we explicitly introduce the bisimilarity relation in the definition, by replacing rule (1) by the following rule:

$$(1')\,\frac{p \sim q}{p\;\;H^B_n\;\;q}$$

However, we prefer our first presentation in order to stress the fact that the system of rules that defines the classical simulation distance is obtained from the one above simply adding the similarity relation to produce pairs that are "0-far" away.

We can prove the relationship between the family H^B_n defined above and the "classical" bisimulation distance relations defined at Def.8, exactly as we made for the simulation case.

Theorem 4. *1. $(H^B_n)_{n\in\mathbb{N}}$ is a cbdf.*
2. If $(R_n)_{n\in\mathbb{N}}$ is a cbdf then $R_n \subseteq H^B_n$.

Once again, we do not use rule (4) at the proof above, which allows to derive the following corollary, that is analogous to Cor.1 in Sect.3.

Corollary 2. *If we define $H^{B'}_n$ as H^B_n in Def.9, but removing the transitivity rule* (4)*, then we obtain the same family of relations, that is $H^{B'}_n = H^B_n$, $\forall n \in \mathbb{N}$.*

5 Distances for All the Semantics in the ltbt-Spectrum

Inspired by the connection between the bisimulation and the simulation distances, next we define a general notion of distance between processes. It can be instantiated by any of the different semantics in the ltbt-spectrum. These distances will measure how far away is any process q of being greater than p

with respect to each of the semantic preorders defining the semantics in Fig. 1. Roughly speaking, to obtain these distances, we compute the cost of changing some actions in both p and q in order to obtain two new processes p' and q' which are related under the considered semantics.

We could try to base our general definitions on the "classical" simulation distance. It is defined in a similar way as the "classical" bisimulation distance. The only difference between those two definitions was the use of $\sqsubseteq_S$ at rule (1). This immediately suggests us to define the semantic distances, corresponding to any semantics defined by an order $\sqsubseteq_{\mathcal{L}}$, by means of the following system of rules:

$$(1)\ \frac{p \sqsubseteq_{\mathcal{L}} q}{p\ H^{\mathcal{L}}_n\ q}\quad (2)\ \frac{p\ H^{\mathcal{L}}_n q}{ap\ H^{\mathcal{L}}_{n+\overline{d}(b,a)}\ bq}\quad (3)\ \frac{p\ H^{\mathcal{L}}_n\ p' \quad q\ H^{\mathcal{L}}_{n'}\ q'}{p+q\ H^{\mathcal{L}}_{max\{n,n'\}}\ p'+q'}\quad (4)\ \frac{p\ H^{\mathcal{L}}_n\ q \quad q\ H^{\mathcal{L}}_{n'}\ r}{p\ H^{\mathcal{L}}_{n+n'}\ r}$$

However, when checking some simple examples we see that this "local" approach (based on max) does not produce a "reasonable" distance for some of the most popular semantics in the ltbt-spectrum. Next, we consider the case of ready simulation (RS).

Example 4. Let us consider the processes $p = b + c$ and $q = d + f$. As distance relation $\overline{d}$ between actions, we consider again the lexicographic distance. We can check that the definition above produces

$$\frac{\dfrac{b\ H^{RS}_2\ d \quad c\ H^{RS}_3\ f}{b+c\ H^{RS}_{max\{2,3\}}\ d+f}(3)}{p\ \ H^{RS}_3\ \ q}(df)$$

We infer $p\ H^{RS}_3\ q$, that is the result of the necessary change in the branch which needs the most expensive change. However, this is, by no means, consistent with the definition of ready simulation: In order to have $p \sqsubseteq_{RS} q$, we need that the two processes have the same initial offer. Therefore, we would need to transform the offer $\{d, f\}$ into $\{b, c\}$. We would need changes whose aggregated cost would be (at least) 4—see Example 5—, and not just 3.

Note that this problem does not appear in the simulation case, because the definition of simulation does not contain any "global" factor. But, most of the rest of the semantics, take somehow into account some "global" information that could only be obtained by combining the information taken from several separated computations. This is the case of ready sets at readiness semantics, or even the case of failures defining the failure semantics.

Certainly, we also had $p\ H^B_3\ q$ for the (classical) bisimulation distance, and then we should also expect $p\ H^{\mathcal{L}}_3\ q$ for any semantics coarser than bisimulation. But as we discussed at the end of our introduction, plain bisimilarity is able to check the equality of the offers of two processes even if working in a local way. However, once we need to compare two unequal offers, this local procedure proves to be quite limited. Therefore, we need to recover our first proposal at Sect. 2 that measures the distance between processes by adding the cost of all the changes that we have to do at all the branches of the tree that represents a process. We already saw that it provides two reasonable "global" notions of simulation and bisimulation distances. Based on it, we obtain our general definition of "global" semantic distance between processes:

Definition 10. *Given a semantics $\mathcal{L}$, defined by a preorder $\sqsubseteq_{\mathcal{L}}$, we say that a process q is at global distance at most $m \in \mathbb{N}$ of being better than some other p, w.r.t. the semantics $\mathcal{L}$ and the distance between actions $\overline{d}$, and then we write $gd_{\overline{d}}^{\mathcal{L}}(p,q) \leq n$, if we can infer $p\, G_n^{\mathcal{L}}\, q$, by applying the following rules:*

$$(1)\ \frac{p \sqsubseteq_{\mathcal{L}} q}{p\ G_n^{\mathcal{L}}\ q} \quad (2)\ \frac{p\, G_n^{\mathcal{L}}\, q}{ap\ G_{n+\overline{d}(b,a)}^{\mathcal{L}}\ bq} \quad (3)\ \frac{p\, G_n^{\mathcal{L}}\, p'}{p+q\ G_n^{\mathcal{L}}\ p'+q} \quad (4)\ \frac{p\, G_n^{\mathcal{L}}\, q \quad q\, G_{n'}^{\mathcal{L}}\, r}{p\ G_{n+n'}^{\mathcal{L}}\ r}$$

Example 5. It is easy to check that for the processes in Example 4 and the ready simulation semantics RS, we obtain now the desired distance $gd_{\overline{d}}^{RS}(p,q) \leq 4$, since we can infer applying the rules for $\mathcal{L} = RS$ that:

$$\dfrac{\dfrac{\dfrac{b\,G_1^{RS}\,c}{b+c\,G_1^{RS}\,c+c}(3) \quad \dfrac{\dfrac{c+c \sqsubseteq_{RS} c}{c+c\,G_0^{RS}\,c}(1) \quad \dfrac{c\,G_1^{RS}\,d \quad \dfrac{d \sqsubseteq_{RS} d+d}{d\,G_0^{RS}\,d+d}(1)}{c\,G_{1+0}^{RS}\,d+d}(4)}{c+c\,G_{0+1}^{RS}\,d+d}(4)}{b+c\,G_{1+1}^{RS}\,d+d}(4) \quad \dfrac{d\,G_2^{RS}\,f}{d+d\,G_2^{RS}\,d+f}(3)}{b+c\,G_{2+2}^{RS}\,d+f}(4)}{p\ \ G_4^{RS}\ \ q}(df)$$

Remark 4. As a matter of fact, we have only used rule (1) in the partial case of "idempotence". This means that the computed (bound of the) distance will also be valid for the bisimulation semantics and in fact for any other semantics in the spectrum. Of course, if we consider a coarser semantics, it could be the case that we could obtain a smaller distance by applying (1) in some other way. For instance, for the simulation semantics (S) we will easily obtain $gd_{\overline{d}(p,q)}^{S} \leq 2$.

Generally, we immediately obtain the following result that asserts that our family of distances reflects exactly the hierarchy in the ltbt-spectrum.

Proposition 3. *Whenever we have two semantics $\mathcal{L}_1$ and $\mathcal{L}_2$ and the first is finer than the latter ($\sqsubseteq_{\mathcal{L}_1} \subseteq \sqsubseteq_{\mathcal{L}_2}$), then we have $gd_{\overline{d}}^{\mathcal{L}_1}(p,q) \leq n \Rightarrow gd_{\overline{d}}^{\mathcal{L}_2}(p,q) \leq n$, for all processes p, q and any value $n \in \mathbb{N}$.*

6 Generalizations, Applications and Some Conclusions

In the developments above we have preferred to consider symmetric distances between actions because in particular we wanted to apply all the notions and technical definitions to the case of bisimulation, that is an equivalence relation and therefore symmetric. However, the rest of the semantics are typically defined by means of a preorder, instead of by an equivalence relation. This is why the consideration of asymmetric quasi-distances opens a new and quite interesting space for developments and applications of our theory.

Let us consider the case of the simulation semantics: when we have $p \sqsubseteq_S q$, this reflects that q has all the capabilities of p and possibly some others. The simulation distances presented above reflect how many changes we need to make in q in order to get a process that really simulates p. But it could be the case that q instead of directly offering the same actions offered by p, offers some others that we consider that “do perfectly the work”. This situation is formally covered simply by replacing the symmetric distance between actions by an asymmetric quasi-distance, defined as follows:

Definition 11. *An asymmetric quasi-distance in a set of actions Act is a function* $d : Act \times Act \to \mathbb{N}$ *which satisfies* $d(a, a) = 0\ \forall a \in Act$*, and the triangular inequality* $d(a, b) + d(b, c) \geq d(a, c)\ \forall a, b, c \in Act$*. We will say that* $d(a, b)$ *expresses “how far away” is action* a *of covering the expectation to have a* b*.*

Remark 5. Now we can have $d(b, a) = 0$ even if $b \neq a$, and this would reflect the fact that b totally “simulates” a. Then we could replace without “cost” any occurrence of an action a in the simulated process p using the action b. Of course, now we can have $d(a, b) \neq d(b, a)$, because the cost of replacing a by b could be very different from that of replacing b by a. Finally, any asymmetric quasi-distance induces a symmetric quasi-distance, simply taking $\overline{d}(a, b) = max\{d(a, b), d(b, a)\}$. This becomes a distance if we impose that $a \neq b \Rightarrow \overline{d}(a, b) \neq 0$.

Example 6. If we consider a simple vending machine that returns no change, and a product costs 1€, then from the machine point of view a payment of 2€ for it, could be perfectly assumed. Instead, if the situation is the other way around and we pay 1€ for a product whose cost is 2€, then the company loses 1€. This would be reflected by the asymmetric quasi-distance defined by $d(1€,2€) = 0$ and $d(2€,1€) = 1$. Using it we obtain that the process where we pay 2€ instead of 1€ is at distance 0 of simulating the specification, while when we pay 1€ when a 2€ cost is specified, we would be at distance 1 of satisfying the specification.

Using the fact that all the semantics in the (extended) ltbt-spectrum are connected to some constrained simulation, we could justify the consideration of the corresponding “biased” distances. Instead, it seems not possible to define a reasonable bisimulation distance really based on an asymmetric quasi-distance. Of course, we could always do the task using the induced distance $\overline{d}$, but in this way we are “loosing” the asymmetric information in the original distance d.

We have defined our distances with natural values just to simplify the presentation, but there is no problem at all on using any other totally ordered set, such as $\mathbb{R}^+$. Moreover, if we use fixed values for the weight of any discordance along a computation (or at any place of the trees when considering “global” distances) then the distance between two (infinite) processes would become infinite as soon as the number of discordances between them is also infinite. This would be certainly a problem, for instance, when comparing cyclic programs where any discordance will appear again at any iteration of the compared processes. Of course, the solution to this problem would consist (as proposed, e.g. in [4, 13]) on defining weighted distances. For them the weight of any disagreement at the n-th step of a computation (or at the n-th level of the unfolded processes) will decrease fast enough (for instance, the classical weights used at the literature are those defined by the exponential sequence $\frac{1}{2^n}$).

It is true, however, that we have not discussed how to obtain in a precise way the (bounds for the) distances between two infinite processes, when they "disagree" at infinitely many places. This could be done by using either finite approximations or recursion-induction rules, for the case of finite state processes. But certainly the details need a careful work.

A simpler extension solves the problem of unexpected termination. If we consider for instance our Def.3, we could extend it by adding a fixed payment f, for unexpected termination, taking $d(p, \mathbf{0}) \leq f$ and $d(\mathbf{0}, p) \leq f$, $\forall p \neq \mathbf{0}$. Instead, we could pay for each of the lost actions a quantity q_a, taking $\overline{d}(a\mathbf{0}, \mathbf{0}) \leq q_a$ and $\overline{d}(\mathbf{0}, a\mathbf{0}) \leq q_a$ $\forall a \in Act$. Of course, this second possibility would produce infinite distances if the terminated process was infinite, but weights can be also introduced here if we want to follow this approach.

We consider that starting from the basic (but quite flexible) definitions introduced in this paper we are plenty of more elaborated possibilities, which could be developed by adapting the ideas in our general theory to them. Next, we give a list of interesting directions that we expect to explore in the near future. First, we are working in a definition of *approximated testing*, where we indicate "at which extent" a process passes a test. Using this notion we can quantify the testing procedure by formalizing the quite frequent situation in practice where the specification states the *ideal* behavior of the desired implementations, but some small disagreements are tolerated by the *quality* standards. A dual application of our distances would also provide for free a nice quantification of the notion of *robustness*: given some specification p we would say that a given implementation q is n-robust w.r.t. some semantics $\mathcal{L}$ when any "n-wrong" behavior of q, that is, any q' such that $d_x^{\mathcal{L}}(q', q) \leq n$, satisfies $d_x^{\mathcal{L}}(p, q') = 0$. We can combine our approximated correctness and the quantified robustness proposed above, to define a notion of approximated robustness, where we also allow some small disagreement between p and the n-wrong behaviors of q.

Another generalization would use "contextually defined" distances between actions, that take into account the fact that several occurrences of the same action in a specification could play totally different roles. In such a case, we could specify at each state of the specification which is the distance between actions that we should use locally at each place. The distances between *pure trees*, where the application of the idempotence law is not allowed, will also capture *redundancy*, and then when investigating fault tolerance the previously discussed ideas on approximated robustness could be used to define *approximated fault tolerance.*

Finally, we could also allow negative values at the distances between actions, that would state that whenever we have $d(a, b) = -n$ then using b to simulate a we would be "improving" the quality of the system. This could amortize some other steps where we have the opposite situation. A typical application would appear when comparing two transmission protocols, and is clearly related with the previous work by Vogler and Lüttgen in [9], where "faster than" preorders where studied, and those by Kiehn and Arun-Kumar [8] on *amortized bisimulation.*

References

[1] Černý, P., Henzinger, T.A., Radhakrishna, A.: Quantitative Simulation Games. In: Manna, Z., Peled, D. (eds.) Time for Verification. LNCS, vol. 6200, pp. 42–60. Springer, Heidelberg (2010)

[2] Černý, P., Henzinger, T.A., Radhakrishna, A.: Simulation Distances. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 253–268. Springer, Heidelberg (2010)

[3] Chen, X., Deng, Y.: Game Characterizations of Process Equivalences. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 107–121. Springer, Heidelberg (2008)

[4] de Alfaro, L., Faella, M., Stoelinga, M.: Linear and branching system metrics. IEEE Trans. Software Eng. 35(2), 258–273 (2009)

[5] de Frutos-Escrig, D., Gregorio-Rodríguez, C., Palomino, M.: On the unification of process semantics: equational semantics. ENTCS 249, 243–267 (2009)

[6] de Frutos Escrig, D., Gregorio Rodríguez, C., Palomino, M.: On the Unification of Process Semantics: Observational Semantics. In: Nielsen, M., Kučera, A., Miltersen, P.B., Palamidessi, C., Tůma, P., Valencia, F. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 279–290. Springer, Heidelberg (2009)

[7] Fahrenberg, U., Legay, A., Thrane, C.R.: The quantitative linear-time–branching-time spectrum. In: Chakraborty, S., Kumar, A. (eds.) FSTTCS. LIPIcs, vol. 13, pp. 103–114. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)

[8] Kiehn, A., Arun-Kumar, S.: Amortised Bisimulations. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 320–334. Springer, Heidelberg (2005)

[9] Lüttgen, G., Vogler, W.: Safe Reasoning with Logic LTS. In: Nielsen, M., Kučera, A., Miltersen, P.B., Palamidessi, C., Tůma, P., Valencia, F. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 376–387. Springer, Heidelberg (2009)

[10] Nielsen, M., Clausen, C.: Bisimulation, Games, and Logic. In: Karhumäki, J., Rozenberg, G., Maurer, H.A. (eds.) Results and Trends in Theoretical Computer Science. LNCS, vol. 812, pp. 289–306. Springer, Heidelberg (1994)

[11] Stirling, C.: Modal and Temporal Logics for Processes. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 149–237. Springer, Heidelberg (1996)

[12] Stirling, C.: Bisimulation, modal logic and model checking games. Logic Journal of the IGPL 7(1), 103–124 (1999)

[13] Thrane, C.R., Fahrenberg, U., Larsen, K.G.: Quantitative analysis of weighted transition systems. J. Log. Algebr. Program. 79(7), 689–703 (2010)

[14] van Glabbeek, R.: 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, ch. 1, pp. 3–99. Elsevier (2001)

[15] Winskel, G.: Synchronisation Trees. In: Díaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 695–711. Springer, Heidelberg (1983)

Secure Multi-Execution through Static Program Transformation

Gilles Barthe[1], Juan Manuel Crespo[1], Dominique Devriese[2], Frank Piessens[2], and Exequiel Rivas[1]

[1] IMDEA Software Institute, Madrid, Spain
[2] IBBT-DistriNet Research Group, KU Leuven, Belgium

Abstract. Secure multi-execution (SME) is a dynamic technique to ensure secure information flow. In a nutshell, SME enforces security by running one execution of the program per security level, and by reinterpreting input/output operations w.r.t. their associated security level. SME is sound, in the sense that the execution of a program under SME is non-interfering, and precise, in the sense that for programs that are non-interfering in the usual sense, the semantics of a program under SME coincides with its standard semantics. A further virtue of SME is that its core idea is language-independent; it can be applied to a broad range of languages. A downside of SME is the fact that existing implementation techniques require modifications to the runtime environment, e.g. the browser for Web applications. In this article, we develop an alternative approach where the effect of SME is achieved through program transformation, without modifications to the runtime, thus supporting server-side deployment on the web. We show on an exemplary language with input/output and dynamic code evaluation (modeled after JavaScript's eval) that our transformation is sound and precise. The crux of the proof is a simulation between the execution of the transformed program and the SME execution of the original program. This proof has been machine-checked using the Agda proof assistant. We also report on prototype implementations for a small fragment of Python and a substantial subset of JavaScript.

1 Introduction

Information flow policies are confidentiality and integrity policies that constrain the propagation of data in programs. For instance, such policies can limit how public outputs can depend on confidential inputs, or how high integrity outputs can be influenced by low integrity inputs. A baseline confidentiality policy for information flow security is *non-interference*: given a labeling of input and output channels as either confidential (high, or H) or public (low, or L), a (deterministic) program is non-interferent if there are no two executions with the same public inputs (but possibly different confidential inputs) that lead to different public outputs. This definition of non-interference generalizes from two security levels H and L to an arbitrary partially ordered set of security levels.

Enforcing non-interference and other information flow policies is a challenging problem. Ideally, enforcement mechanisms should achieve potentially conflicting goals, including: i. *soundness:* no illicit flows should arise during execution; ii. *precision:* the execution of secure programs should not be prevented or altered; iii. *practicality:* the cost of the mechanism should be acceptable. Costs can be incurred at development time (for instance additional code annotations), at deployment time (for instance modifications to standard runtime environments) or at run time (for instance performance cost). Despite substantial attention from the research community for several decades, enforcement mechanisms achieving these goals simultaneously have remained elusive.

There are two main classes of enforcement mechanisms for information flow policies. *Static* mechanisms include security type systems [31,17,24], and verification-based approaches [5]. These techniques are sound, and do not incur run time or deployment time costs. However, type-based approaches are not precise, and reject many secure programs. In contrast, verification-based approaches may offer perfect precision (modulo completeness of the underlying program logic). However, both type-based and verification-based approaches have a substantial development time cost as they require annotations in the code. Moreover some language idioms, such as dynamic code evaluation, are not readily amenable to static information flow analysis.

Dynamic techniques, which have received renewed interest in recent years, include run-time monitors [16,29,3,10], and more recently *secure multi-execution (SME)* [14,8]. The cited techniques are sound, and can be more precise than some static techniques. For instance, run-time monitors reject fewer programs than type-based methods[29]; they also require less annotation effort. However, run-time monitors still may reject or alter the behavior of some secure programs. In contrast, SME offers perfect precision (at the cost of potentially modifying the behaviour of insecure programs); it is also practical for developers, since there is no need for security annotations of the code. However, SME is not easy to deploy, as all existing implementations of SME require modifications to the underlying computing infrastructure (OS [8], browser [6,2], virtual machine [14], trusted libraries [18]). Specifically, it is hard to deploy SME for distributed and heterogeneous infrastructures, such as the web.

The key contribution of this paper is a new implementation technique for SME based on static program transformation that eliminates the need to modify the computing infrastructure, while retaining its appealing theoretical properties.

A Motivating Example: JavaScript Advertising

JavaScript code is used in web applications to perform client-side computations. In many scenarios, the fact that scripts run with the same privileges as the website loading the script leads to security problems. One important example are advertisements; these are commonly implemented as scripts and in the absence of security countermeasures, such scripts can leak any information present in the web page that they are part of.

JavaScript advertisements are a challenging application area for information flow security, as they may need some access to the surrounding web page (to be able to provide context-sensitive advertising), and as they can use all of JavaScript's features, including dynamic code evaluation, e.g. in the form of JavaScript's `eval` function, which Richards *et al.*[25] have shown to be widely used on the web. The following code snippet shows a very simple context-sensitive advertisement in JavaScript:

```
1 var keywords = document.getElementById("keywords").textContent;
2 var img = document.getElementById("adimage");
3 img.src = 'http://ads.com/SelectAd.php?keywords='+keywords
```

Line 1 looks up some keywords in the surrounding web page; these keywords will be used by the ad provider to provide a personalized, context-sensitive advertisement. Line 2 locates the element in the document in which the advertisement should be loaded, and finally line 3 generates a request to the advertisement provider site to generate an advertisement (in the form of an image) related to the keywords sent in the request.

Obviously, a malicious advertisement can easily leak any information in the surrounding page to the ad provider or to any third party. Here is a simple malicious ad that leaks the contents of a password field to the ad provider:

```
1 // Malicious: steal a password instead of keywords
2 var password = document.getElementById("password").textContent;
3 var img = document.getElementById("adimage");
4 img.src = 'http://ads.com/SelectAd.php?keywords='+password
```

Information flow security enforcement can mitigate this threat: if one labels the keywords as public information and the password as confidential information, then (treating the network as a public output) enforcing non-interference will permit the non-malicious ad, but block the malicious one.

The example ad script above loads an image from a third-party server. Instead of loading an image, it could also load a script from the server that can then render the ad and further interact with the user (e.g. make the advertisement react to mouse events). In the example below, we illustrate the essence of this technique using the `XMLHttpRequest` API and JavaScript `eval`.

```
1 var keywords = document.getElementById("keywords").textContent;
2 var xmlhttp = new XMLHttpRequest();
3 xmlhttp.open('GET', 'http://ads.com/getAd.php?keywords='+keywords, false);
4 xmlhttp.send(null);
5 eval(xmlhttp.responseText)
```

Lines 2-4 send the keywords to the ad provider, and expect a (personalized) script in response. Line 5 then evaluates the script that was received – and this script could of course be malicious too and try to leak information. Dealing with dynamic generation or loading of new code and its on the fly evaluation further complicates the enforcement of information flow security policies. In particular, since the code to be executed is not available offline, static techniques do not apply.

The enforcement mechanism we develop in this paper will provide effective protection against these security problems of malicious scripts. We propose a program transformation that transforms any script into a script that (1) is guaranteed to be non-interferent, and (2) behaves identically to the original script if that script was non-interferent to begin with.

Summary of Contributions

In summary, the main contributions of this paper are:

- We show that standard SME [14] is sound and precise for a language including dynamic code evaluation.
- We propose a program transformation for sequential programs that simulates the effect of SME, and provide a machine-checked correctness proof.
- We report on two prototype implementations of this program transformation.
- We define a variant of the transformation that targets a concurrent programming language, and prove it correct.

The paper is organized as follows: Section 2 introduces our programming language and defines non-interference. Section 3, 4, 5 and 6 each cover one of the contributions above. Related work is discussed in Section 7.

2 Setting

Syntax. Following [14], a program P is simply a command to be executed by the system. The syntax of commands is defined as follows:

$$\begin{aligned} c ::= {} & x := e \mid \mathbf{input}\ x\ \mathbf{from}\ ic \mid \mathbf{output}\ e\ \mathbf{to}\ oc \mid c; c \\ \mid {} & \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid \mathbf{skip} \mid \mathbf{eval}(e) \end{aligned}$$

Most commands are standard, with the exception of $\mathbf{input}\ x\ \mathbf{from}\ ic$, that assigns the next input from the input channel ic to x, and $\mathbf{output}\ e\ \mathbf{to}\ oc$, that outputs the value of the expression e to the output channel oc—we assume that input and output channels are disjoint. The main extension w.r.t. [14] is the instruction $\mathbf{eval}(e)$, which takes an integer encoding e of a program (in a real language this would be the usual string encoding), decodes it and evaluates it.

Example 1. The command below models a program that exhibits both of the attacks presented in the introduction: the script sends private information (the password) across a public channel (the network) to the ad provider and then receives a (possibly malicious) script which is executed with the same privilege.

$$\begin{array}{l} \mathbf{input}\ keys\ \mathbf{from}\ LKeys; \\ \mathbf{input}\ pass\ \mathbf{from}\ HPass; \\ \mathbf{output}\ keys + pass\ \mathbf{to}\ LReq; \\ \mathbf{input}\ res\ \mathbf{from}\ LReq'; \\ \mathbf{eval}(res) \end{array}$$

Semantics. For simplicity, we assume that expressions are side-effect free, and that they are used with their correct types—e.g. guards of branching statements and loops are boolean expressions. The semantics of expressions is defined as a mapping from memories to values or bottom, where a memory is a (well-typed) mapping from variables to values. Formally, we let $[\![e]\!]\ m$ be the evaluation of e in memory m.

The operational behavior of programs is modelled as a transition relation $\rightsquigarrow$ between configurations. Formally, a configuration is a 5-tuple $\langle c, m, p, I, O\rangle$, where c is a command, m is a memory, I and O are program inputs and outputs, i.e. mappings from input and output channels respectively to lists of values, and p is an input pointer, i.e. a mapping from input channels to natural numbers, that points to the next input to be consumed. A configuration is initial if it is of the form $\langle c, m_0, p_0, I, O_0\rangle$, where m_0 maps every variable to a default value, e.g. 0 for integer variables, p_0 maps every input channel to 0, and O_0 maps every output channel to the empty list.

Fig. 1 provides an excerpt of the transition rules that define the operational semantics.The rules make use of an operation **decode** that turns an integer into a command, and of primitive operations for reading and writing from a channel (we use the notation $l_1 +\!\!+ l_2$ for appending two lists):

$$\mathbf{read}(I, ic, p) = I(ic)(p(ic)) \qquad\qquad \mathbf{write}(O, oc, v) = O[oc \mapsto O(oc) +\!\!+ [v]]$$

We say that an execution of the program P with input I terminates with input pointer p and program output O, and write $\langle P, I\rangle \rightsquigarrow^* \langle p, O\rangle$, iff $\langle P, m_0, p_0, I, O_0\rangle \rightsquigarrow^* \langle \mathbf{skip}, m, p, I, O\rangle$ for some memory m.

$$\frac{}{\langle \mathbf{input}\ x\ \mathbf{from}\ ic, m, p, I, O\rangle \rightsquigarrow \langle \mathbf{skip}, m[x \mapsto \mathbf{read}(I, ic, p)], p[ic \mapsto p(ic) + 1], I, O\rangle}$$

$$\frac{}{\langle \mathbf{output}\ e\ \mathbf{to}\ oc, m, p, I, O\rangle \rightsquigarrow \langle \mathbf{skip}, m, p, I, \mathbf{write}(O, oc, [\![e]\!]m)\rangle}$$

$$\frac{\langle c_1, m, p, I, O\rangle \rightsquigarrow \langle c_1', m', p', I, O'\rangle}{\langle c_1; c_2, m, p, I, O\rangle \rightsquigarrow \langle c_1'; c_2, m', p', I, O'\rangle}$$

$$\frac{}{\langle \mathbf{skip}; c_2, m, p, I, O\rangle \rightsquigarrow \langle c_2, m, p, I, O\rangle}$$

$$\frac{}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, m, p, I, O\rangle \rightsquigarrow \langle c; \mathbf{while}\ b\ \mathbf{do}\ c, m, p, I, O\rangle}\ [\![b]\!]m$$

$$\frac{}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, m, p, I, O\rangle \rightsquigarrow \langle \mathbf{skip}, m, p, I, O\rangle}\ \neg[\![b]\!]m$$

$$\frac{}{\langle \mathbf{eval}(e), m, p, I, O\rangle \rightsquigarrow \langle \mathbf{decode}([\![e]\!]m), m, p, I, O\rangle}$$

Fig. 1. Operational semantics (excerpt)

Security. The notion of program security is defined relative to a partially ordered set $(\mathcal{L}, \leq)$ of security levels, and mappings σ_{in} and σ_{out} from input and output channels to security levels. The mappings induce equivalence relations on inputs, outputs, and input pointers; informally, two inputs, outputs, and input pointers are equal w.r.t. a security level l if they cannot be distinguished by an adversary that has access to channels of level l and lower. Formally, two program inputs I and I' are equal up to l (written $I =_l I'$) iff $I(i) = I'(i)$ for all input channels i such that $\sigma_{in}(i) \leq l$. Likewise, two program outputs O and O' are equal up to l (written $O =_l O'$) iff $O(o) = O'(o)$ for all output channels o such that $\sigma_{out}(o) \leq l$. Finally, two input pointers p and p' are equal up to l (written $p =_l p'$) iff $p(i) = p'(i)$ for all input channels i such that $\sigma_{in}(i) \leq l$.

Definition 1 (Non-interference). *A program P is non-interferent with respect to an execution relation $\Rightarrow^*$ (mapping programs and inputs to input pointers and outputs) if for all security levels $l \in \mathcal{L}$, for all l-equal inputs I and I', i.e. $I =_l I'$, we have that $(P, I) \Rightarrow^* (p_f, O_f)$ if and only if $(P, I') \Rightarrow^* (p'_f, O'_f)$ and $p_f =_l p'_f$ and $O_f =_l O'_f$.*

Note that this definition is *termination-sensitive*: it does not allow termination to depend on information at non-minimal levels. The definition of non-interferent program is obtained by instantiating $\Rightarrow^*$ to $\rightsquigarrow^*$. Example 1 is clearly *not* non-interferent.

3 Secure Multi-Execution: The Operational Approach

We extend the theoretical results of [14] and show that SME remains sound and precise in the presence of dynamic code evaluation.

SME by Example. The central insight of SME is that non-interference can be enforced by executing programs once per security level. In order to guarantee non-interference, the execution at security level l only performs inputs and outputs to channels at level l; moreover, inputs from channels with security levels l' such that $l' \not\leq l$ are replaced by default values and inputs from channels of security levels l' such that $l' < l$ are delayed until the execution corresponding to security level l' reads from them—the result is then available to be reused at security level l.

The precision of SME intuitively follows from the fact that for non-interferent programs, the behavior of the program visible at a level l is by definition not influenced by changes to information at levels not lower than l. Therefore, the execution at any level l will still produce the same behavior at level l as the standard execution of the program, since it receives the same input on all levels lower than l.

Figure 2 illustrates the effect of SME on the malicious script from Section 1 and the two-points lattice of security levels $\{L, H\}$, with $L \leq H$. We treat reading the content of the `password` textbox as input at security level H and setting the URL of the image as output at level L. Hence, the SME execution of the program at level L will receive a default value rather than the real content of the `password` textbox. Subsequently, the execution at level L will compute as URL of the image a value that does not contain any information about the real user password. On the contrary, the execution of the script at security level H does receive the real input, and further computations at level H will be performed based on the password; however, the execution does not output to low channels.

Operational Semantics of SME. Secure multi-execution is described formally through an operational semantics, and is parametrized by a lattice of security levels $\mathcal{L}$, and mappings σ_{in} and σ_{out} associating input and output channels to security levels respectively.

The operational semantics combines a local semantics, and a global semantics. The initial configuration includes a local configuration per security level; each local configuration runs independently of the other, except for input/output operations, where

Execution at L security level.

```
// Malicious: steal a password instead of keywords
var password = ~~document.getElementById("password").textContent~~ undefined;
var img = document.getElementById("adimage");
img.src = 'http://ads.com/SelectAd.php?keywords='+password
```

Execution at H security level.

```
// Malicious: steal a password instead of keywords
var password = document.getElementById("password").textContent
var img = document.getElementById("adimage");
~~img.src = 'http://ads.com/SelectAd.php?keywords='+password~~
```

Fig. 2. Secure Multi-Execution of malicious JavaScript program from Section 1

synchronization is needed. The global semantics capture the synchronization enforced by SME, and are defined relative to a scheduler **select** that, given a set of local configurations, picks the next one to execute. In their work, Devriese and Piessens [14] focus on a scheduler $\textbf{select}_{\text{lowprio}}$ which picks the local configuration corresponding to the lowest security level; other schedulers are considered in [19].

The local semantics are defined as a relation between pairs of local configurations and global states. Local configurations are of the form $\langle c, m, p\rangle_l$, where c is a command, m is a memory and p is a local input pointer and l is the security level associated to the local configuration. Global states consist of a global input pointer r, a program input I and a program output O. The global input pointer r tracks actual input consumption. I.e. for an input channel ic, $r(ic)$ equals $p(ic)$ where p is the local input pointer of the execution at level $\sigma_{in}(ic)$. For details about the semantics, we refer the reader to the original SME paper [14]. The only novelty is the rule in the local semantics for eval:

$$\frac{[\![e]\!]m = v \qquad \mathbf{decode}(v) = c}{\langle eval(e), m, p\rangle_l, r, I, O \Rightarrow\!\!\rhd \langle c, m, p\rangle_l, r, I, O}$$

The global semantics are defined as a relation between configurations. The latter are of the form $\langle L, wq, r, I, O\rangle$, where r, I, O form the global state, L is a set of local configurations, and wq is a queue that maps input channels and message numbers to local configurations waiting for that message to be input. Again, the details are in [14].

We say that a set of local configurations C with input I terminates with final input pointer r_f and program output O_f, and write $\langle C, I\rangle \Rightarrow\!\!\rhd^* \langle r_f, O_f\rangle$, if

$$\langle C, wq_0, r_0, I, O_0\rangle \Rightarrow\!\!\rhd^* \langle [], wq_f, r_f, I, O_f\rangle$$

for some final waiting queue wq_f and where r_0 is the global input pointer mapping all input channels to position 0.

The secure multi-execution of a program P is defined using the global semantics; specifically, we introduce for every program P and security level l the local configuration $P_l = \langle P, m_0, p_0 \rangle_l$, where m_0 is the default memory—as defined in Section 2—and p_0 maps all input channels to 0. Then, we introduce the set of local configurations $P_{lcinit} = [P_{l_1}, \ldots, P_{l_k}]$ where $l_1 \ldots l_k$ is an enumeration of the security levels. Then, we say that the secure multi-execution of the program P with input I terminates with final input pointer r_f and final program output O_f, and write $\langle P, I \rangle \Rightarrow\!\!\rhd^* \langle r_f, O_f \rangle$ iff $\langle P_{lcinit}, I \rangle \Rightarrow\!\!\rhd^* \langle r_f, O_f \rangle$.

Soundness and Precision. SME provides strong security and operational guarantees.

Theorem 1 (Soundness of SME). *For a totally ordered $\mathcal{L}$, any program P is non-interferent under SME, using the* $\mathbf{select}_{lowprio}$ *scheduler.*

The $\mathbf{select}_{\text{lowprio}}$ scheduler requires a total ordering on security levels. If $\mathcal{L}$ is not totally ordered, then it can be extended to a total order in order to apply SME with the $\mathbf{select}_{\text{lowprio}}$ scheduler. In that case execution of P under SME is termination-*insensitively* non-interferent, but termination information may leak between non-comparable levels of $\mathcal{L}$ [19].

Theorem 2 (Precision of SME). *Let P be a non-interferent program. Then, for all program input I, input pointer p_f and program output, O_f, $\langle P, I \rangle \rightsquigarrow^* \langle p_f, O_f \rangle$ implies $\langle P, I \rangle \Rightarrow\!\!\rhd^* \langle p_f, O_f \rangle$.*

The proofs follow along the lines of [14]; additional cases for $\mathbf{eval}$ follow by a direct argument.

4 Secure Multi-Execution by Program Transformation

The instrumented semantics of Section 3 provides a direct, operational interpretation of the effect of secure multi-execution on programs. In this section, we explore an alternative approach in which a program P of the source language is transformed into a program P' whose behavior matches the behavior of P under SME execution. Our results show that one can achieve soundness and precision without modifying the runtime environment.

Informally, one defines for each program P and security level l a transformed program $\mathbf{Tr}(P, l)$ and defines $\mathbf{Tr}(P)$ as the sequential composition of the commands $\mathbf{Tr}(P, l)$, where l ranges over security levels from low to high. This mimicks execution under the SME semantics with the $\mathbf{select}_{\text{lowprio}}$ scheduler. We assume that this sequential composition is done in the same order as the order in which the $\mathbf{select}_{\text{lowprio}}$ scheduler selects executions. For a totally ordered $\mathcal{L}$, this order is fixed, but non-comparable levels can be scheduled in different ways.

SME requires the buffering of inputs so that these inputs can be reused by executions running at higher security levels. We implement these buffers as global lists ($list_{ic}$) and the global input pointer as well as local input pointers are represented as global integer variables ($count_{ic}$ and $count_{ic,l}$ respectively).

For commands that do not perform input/output operations, the command $\mathbf{Tr}(P, l)$ executes P "locally". Specifically, for each variable x of the source program, we introduce variables x_l, where l ranges over security levels; informally, x_l is the local copy of x for the execution corresponding to security level l. Then, we ensure that $\mathbf{Tr}(P, l)$ reads and writes only from/to variables indexed by l. For instance, the transformation of an assignment is defined by the clause:

$$\mathbf{Tr}(x := e, l) = x_l := [e]_l$$

where $[e]_l$ is obtained by replacing occurrences of each variable (say x) by its l-indexed variant (say x_l). The definition of the transformation extends recursively to sequences, branching statements, and loops. In the case of dynamic code evaluation, $\mathbf{Tr}(\mathbf{eval}(e), l)$ should informally compute the value of e locally at level l, decode the resulting value into a command c, compute $c' = \mathbf{Tr}(c, l)$, encode c' into an integer n', and return $\mathbf{eval}(n')$. Hence, $\mathbf{Tr}(\mathbf{eval}(e), l)$ should intuitively be of the form:

$$n := [e]_l; c := \mathbf{decode}(n); c' := \mathbf{Tr}(c, l); n' := \mathbf{encode}(c'); \mathbf{eval}(n')$$

The code snippet is ill-typed and ill-defined in our exemplary language. In a full-fledged language such as JavaScript, one can make the above snippet meaningful, by implementing encoding and decoding functions from strings and abstract syntax trees, and the transformation given by the rules of Fig. 3. For the purpose of this section, we gloss over the details of such implementations and assume the existence for each security level l of a unary operator $\mathbf{trans}_l$ from integers to integers, and define

$$\mathbf{Tr}(\mathbf{eval}(e), l) = \mathbf{eval}(\mathbf{trans}_l([e]_l))$$

Moreover, we assume that $\mathbf{trans}_l$ is correct, i.e. for every integer value k,

$$\mathbf{decode}(\mathbf{trans}_l(k)) = \mathbf{Tr}(\mathbf{decode}(k), l)$$

The most interesting cases of the transformation are for input and output commands. For the latter, $\mathbf{Tr}(P, l)$ is defined by case analysis on the security level of the output channel: a command $\mathbf{output}\ e\ \mathbf{to}\ oc$ is transformed into $\mathbf{output}\ [e]_l\ \mathbf{to}\ oc$ if oc has security level l, and into a $\mathbf{skip}$ statement otherwise. Similarly, for input statements, we define the transformation by case analysis on the security level l' of the input channel—as in the definition of SME. If $l' \not\leq l$, then the input statement is transformed into an assignment of a default value. If $l = l'$, then the transformed command performs the input statement and updates the list of available inputs and the counter representing the number of messages already read from this channel. Finally, if $l' < l$, the transformed command reuses a buffered input value and updates the corresponding counter. Executing the programs $\mathbf{Tr}(P, l)$ sequentially in the order from low to high in an initial memory in which every *count* variable has value 0 and every *list* variable is associated with the empty list, will simulate SME execution under the $\mathbf{select}_{\text{lowprio}}$ scheduler.

$$\mathbf{Tr}(P) = \,;\{\mathbf{Tr}(P, l) \mid l \in \mathcal{L}\}$$

Example 2. We apply the transformation to Example 1. The sequential program obtained is shown in Fig. 4.

Formally, we can prove the following theorems.

Theorem 3. *For every program P and program input I:*

1. *if* $\langle \mathbf{Tr}(P), I\rangle \rightsquigarrow^* \langle p, O\rangle$ *then* $\langle P, I\rangle {\Rightarrow\!\!\triangleright}^* \langle p, O\rangle$*;*
2. *if P is non-interferent and* $\langle P, I\rangle {\Rightarrow\!\!\triangleright}^* \langle p, O\rangle$ *then* $\langle \mathbf{Tr}(P)\rangle \rightsquigarrow^* \langle p, O\rangle$.

We have developed a mechanized proof using Agda, a proof assistant based on the Curry-Howard isomorphism. We refer the reader to the extended version of this paper [4].

The following is an easy corollary of Theorem 3.

Corollary 1. *Statically enforced sequential SME is sound and precise.*

Proof. Soundness follows from Theorem 3, first part and Theorem 1. Precision follows from Theorem 3, second part and Theorem 2. □

5 Implementation

In order to validate our approach, we have developed two prototype implementations. Our first implementation considers a restricted fragment of Python; the fragment essentially corresponds to our exemplary language, with I/O functions `input` and `print` added as built-in functions. It does not support any of Python's more advanced features, but was useful to provide a baseline implementation.

Our second implementation supports a fragment of JavaScript including `eval()`. Both implementations were tested for security and for precision by means of small test scenarios.

We briefly comment on some aspects of the implementations.

Aliasing. The soundness of our transformation relies on applying specific rules for I/O operations. In richer languages such as Python or JavaScript, aliasing becomes a major problem as one cannot statically determine where such operations will be called. To avoid this issue, and to be able to identify I/O operations, we proceed in two steps: first, we wrap primitive I/O functions upfront, i.e. the wrapped function will behave according to the security level associated to the context in which is called. Second, programs are only given access to these wrapped functions. This is achieved using Google Caja [23], which guarantees that the translated program only gets access to properly wrapped APIs. Google Caja will rewrite ("cajole") a program in such a way that it can be guaranteed capability secure, i.e. the modified program will only be able to call API functions which it is passed a reference to and otherwise be isolated from other code.

$$
\begin{array}{lcl}
\mathbf{Tr}(x := e, l) & = & x_l := [e]_l \\
\mathbf{Tr}(\mathbf{output}\ e\ \mathbf{to}\ oc, l) & = & \begin{cases} \mathbf{output}\ [e]_l\ \mathbf{to}\ oc & \text{if } \sigma_{out}(oc) = l \\ \mathbf{skip} & \text{otherwise} \end{cases} \\
\mathbf{Tr}(\mathbf{input}\ x\ \mathbf{from}\ ic, l) & = & \begin{cases} x_l := dv & \text{if } \sigma_{in}(ic) \not\leq l \\ \mathbf{input}\ x\ \mathbf{from}\ ic; \\ list_{ic} := list_{ic} +\!\!+ [x_l]; & \text{if } \sigma_{in}(ic) = l \\ count_{ic} := count_{ic} + 1 \\ x_l := list_{ic}[count_{ic,l}]; \\ count_{ic,l} := count_{ic,l} + 1\} & \text{if } \sigma_{in}(ic) < l \end{cases} \\
\mathbf{Tr}(c_1; c_2, l) & = & \mathbf{Tr}(c_1, l); \mathbf{Tr}(c_2, l) \\
\mathbf{Tr}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, l) & = & \mathbf{if}\ [b]_l\ \mathbf{then}\ \mathbf{Tr}(c_1, l)\ \mathbf{else}\ \mathbf{Tr}(c_2, l) \\
\mathbf{Tr}(\mathbf{while}\ b\ \mathbf{do}\ c, l) & = & \mathbf{while}\ [b]_l\ \mathbf{do}\ \mathbf{Tr}(c, l) \\
\mathbf{Tr}(\mathbf{skip}, l) & = & \mathbf{skip} \\
\mathbf{Tr}(\mathbf{eval}(e), l) & = & \mathbf{eval}(\mathbf{trans}_l([e]_l))
\end{array}
$$

Fig. 3. Syntactic program transformation

```
$\mathbf{input}\ keys_L\ \mathbf{from}\ LKeys;$
$list_{LK} := list_{LK} +\!\!+ [keys_L];$
$count_{LK} := count_{LK} + 1;$
$pass_L := dv;$
$\mathbf{output}\ keys_L + pass_L\ \mathbf{to}\ LReq;$
$\mathbf{input}\ res_L\ \mathbf{from}\ LReq';$
$list_{LR'} := list_{LR'} +\!\!+ [res_L];$
$count_{LR'} := count_{LR'} + 1;$
$\mathbf{eval}(\mathbf{trans}_L(res_L));$
$keys_H := list_{LK}[count_{LK,H}];$
$count_{LK,H} := count_{LK,H} + 1;$
$\mathbf{input}\ pass_H\ \mathbf{from}\ HPass;$
$list_{HP} := list_{HP} +\!\!+ [pass_H];$
$count_{HP} := count_{HP} + 1;$
$res_H := list_{LR'}[count_{LR',H}];$
$count_{LR',H} := count_{LR',H} + 1;$
$\mathbf{eval}(\mathbf{trans}_H(res_H))$
```

Fig. 4. Static transformation applied to malicious ad.

Dynamic Code Evaluation. Our prototype supports an `eval` function (JavaScript's well-known dynamic code evaluation primitive). Since Google Caja does not support dynamic code evaluation, we have developed our own *ad hoc* solution. Our `eval` takes as input a string of code, and sends it to a remote Caja cajoling service; the transformed code is then executed with the same wrapped APIs as the calling code. This proof-of-concept implementation is admittedly inefficient but arguably secure (assuming the calls to Google's cajoling service are reliable) and supports the entire subset of JavaScript that Google Caja supports.

Document Object Model (DOM) The Document Object Model (DOM) APIs that a browser exposes to scripts is structured as a tree corresponding to the HTML structure of the document. The DOM tree can be inspected and modified from within JavaScript. Our prototype supports a limited, read-only, version of the DOM. In particular, it allows the hosting page to assign security levels to parts of the document. The scripts can access the hosting document according to this policy and perform synchronous XMLHttpRequests. Our coverage of the DOM is sufficient for our examples.

Many DOM APIs allow web applications to register callback functions, which will be executed when certain (network, user or other) events occur; Bielova *et al.* [6] discuss how events and callbacks can be supported under secure multi-execution. Extending our transformation to address events and callbacks, and provide support for the full DOM is a significant engineering challenge, which we regard as future work.

6 Transformation to a Concurrent Language

The transformation defined earlier simulates SME with the $\mathbf{select}_{\text{lowprio}}$ scheduler. Kashyap et al. [19] have shown that other scheduling strategies can be useful too. In this section, we present a variant of our transformation towards a language that supports concurrency in order to enable the use of more scheduling strategies.

This revised transformation still takes programs in the sequential subset of the language as input. The concurrency features are only used in the output of the transformation.

Target language. We extend our command language with the following syntax:

$$\begin{aligned} c &::= \ldots \,|\, \mathbf{await}\ b\ \mathbf{then}\ c \\ P &::= \,\|\, (id, c)^* \end{aligned}$$

Intuitively, the command $\mathbf{await}\ b\ \mathbf{then}\ c$ executes c atomically, provided b holds, and blocks otherwise. Then, a program is simply a set of threads; for convenience, we assume that each thread is tagged with a unique identifier. In what follows, we write $\mathbf{atomic}\ c$ as a shorthand for $\mathbf{await}\ \mathbf{true}\ \mathbf{then}\ c$.

The operational behavior of programs is modelled as a transition between configurations. A configuration is a 5-tuple consisting of a program P, a waiting queue wq mapping guards to commands, an input pointer p, a program input I and a program output O. Figure 5 presents the semantics of the language. The thread-local semantics is similar to our sequential language; note however that we introduce another rule for sequence in order to propagate the emission of signals induced by $\mathbf{await}$ commands. The rules for the latter are standard; if the guard holds, then the body of the command is executed atomically. Otherwise, the command blocks and emits a signal, namely the guard in which its blocked. Upon the emission of a signal, the global semantics then inserts the blocked thread associated with the guard into the waiting queue. Further changes in global state trigger the re-evaluation of guards, and threads associated with guards that become true are moved back to the ready list.

We say that an execution of the program P with input I terminates with input pointer p and program output O, and write $\langle P, I\rangle \rightsquigarrow^* \langle p, O\rangle$, if there exists some memory m such that

$$\langle P, wq_0, m_0, p_0, I, O_0\rangle \rightsquigarrow^* \langle [], wq_0, m, p, I, O\rangle$$

$$\frac{\langle c_1, m, p, I, O\rangle \overset{b}{\rightsquigarrow} \langle c_1', m, p, I, O\rangle}{\langle c_1; c_2, m, p, I, O\rangle \overset{b}{\rightsquigarrow} \langle c_1'; c_2, m, p, I, O\rangle}$$

$$\frac{\langle c, m, p, I, O\rangle \rightsquigarrow^* \langle \mathbf{skip}, m', p', I, O'\rangle}{\langle \mathbf{await}\ b\ \mathbf{then}\ c, m, p, I, O\rangle \rightsquigarrow \langle \mathbf{skip}, m', p', I, O'\rangle}\ [\![b]\!]m$$

$$\frac{}{\langle \mathbf{await}\ b\ \mathbf{then}\ c, m, p, I, O\rangle \overset{b}{\rightsquigarrow} \langle \mathbf{await}\ b\ \mathbf{then}\ c, m, p, I, O\rangle}\ \neg[\![b]\!]m$$

(a) Thread-local semantics (excerpts)

$$\frac{\mathbf{select}(P) = (id, \mathbf{skip})}{\langle P, wq, m, p, I, O\rangle \rightsquigarrow \langle P\backslash\{(id, \mathbf{skip})\}, wq, m, p, I, O\rangle}$$

$$\frac{\mathbf{select}(P) = (id, c) \qquad \langle c, m, p, I, O\rangle \overset{b}{\rightsquigarrow} \langle c, m, p, I, O\rangle}{\langle P, wq, m, p, I, O\rangle \rightsquigarrow \langle P\backslash\{(id, c)\}, wq \cup \{(b, (id, c))\}, m, p, I, O\rangle}$$

$$\frac{\begin{array}{c}\mathbf{select}(P) = (id, c) \qquad \langle c, m, p, I, O\rangle \rightsquigarrow \langle c', m', p', I, O'\rangle \\ P' = P\backslash\{(id, c)\} \cup \{(id, c')\} \cup \{(id^*, c^*) | (b, (id^*, c^*)) \in wq \wedge [\![b]\!]m'\} \\ wq' = \{(b, (id^*, c^*)) | (b, (id^*, c^*)) \in wq \wedge \neg[\![b]\!]m'\}\end{array}}{\langle P, wq, m, p, I, O\rangle \rightsquigarrow \langle P', wq', m', p', I, O'\rangle}$$

(b) Global semantics

Fig. 5. Extended semantics

The Transformation. Adapting our transformation to target the concurrent case requires only two changes. First, input will now perform synchronization:

$$\mathbf{Tr^{con}}(\mathbf{input}\ x\ \mathbf{from}\ ic, l) = \begin{cases} x_l := dv & \text{if } \sigma_{in}(ic) \not\leq l \\ \mathbf{atomic}\ (\mathbf{input}\ x_l\ \mathbf{from}\ ic; & \\ list_{ic} := list_{ic} +\!\!+ [x_l]; count_{ic} := count_{ic} + 1) & \text{if } \sigma_{in}(ic) = l \\ \mathbf{await}\ count_{ic,l} < count_{ic}\ \mathbf{then} & \\ (x_l := list_{ic}[count_{ic,l}]; count_{ic,l} := count_{ic,l} + 1) & \text{if } \sigma_{in}(ic) < l \end{cases}$$

Second, instead of defining the overall transformation as a sequential composition, we define it as a parallel one, i.e. $\mathbf{Tr^{con}}(P) = \| \{(l, \mathbf{Tr^{con}}(P, l)) \mid l \in \mathcal{L}\}$.

Example 3. Consider our running example, the malicious ad. Applying the transformations to the example w.r.t. security levels L and H yields the two programs shown in Fig. 6a and Fig. 6b respectively.

The revised transformation again yields executions equivalent to secure multi-execution, now for any scheduling strategy. The proof relies on a simulation result and hinges on the assumption that (informally) schedulers pick the same threads to execute.

```
atomic {
    input $keys_L$ from $LKeys$;
    $list_{LK} := list_{LK}$ ++ $[keys_L]$;
    $count_{LK} := count_{LK} + 1$};
$pass_L := dv$;
output $keys_L + pass_L$ to $LReq$;
atomic {
    input $res_L$ from $LReq'$;
    $list_{LR'} := list_{LR'}$ ++ $[res_L]$;
    $count_{LR'} := count_{LR'} + 1$};
eval(trans$_L(res_L)$);
```

(a) Security level L.

```
await $count_{LK,H} < count_{LK}$ then {
    $keys_H := list_{LK}[count_{LK,H}]$;
    $count_{LK,H} := count_{LK,H} + 1$};
atomic {
    input $pass_H$ from $HPass$;
    $list_{HP} := list_{HP}$ ++ $[pass_H]$;
    $count_{HP} := count_{HP} + 1$};
await $count_{LR',H} < count_{LR'}$ then {
    $res_H := list_{LR'}[count_{LR',H}]$;
    $count_{LR',H} := count_{LR',H} + 1$};
eval(trans$_H(res_H)$);
```

(b) Security level H.

Fig. 6. Static transformation applied to malicious ad.

Theorem 4. *For every program P, and program input I:*

1. *if* $\langle \mathbf{Tr^{con}}(P), I\rangle \rightsquigarrow^* \langle p, O\rangle$ *then* $\langle P, I\rangle \Rightarrow\!\!\triangleright^* \langle p, O\rangle$;
2. *if* $\langle P, I\rangle \Rightarrow\!\!\triangleright^* \langle p, O\rangle$ *then* $\langle \mathbf{Tr^{con}}(P), I\rangle \rightsquigarrow^* \langle p, O\rangle$.

For the proof, we refer to the extended version of this paper [4].

7 Related Work

The work reported on in this paper is related to information flow security, a research area that has received significant attention for many decades. We point the reader to two broad surveys, and then zoom in to recent research that is closely related to our work. Sabelfeld and Myers [28] give an excellent survey on static techniques for information flow enforcement. Le Guernic's PhD thesis [16] surveys dynamic techniques.

Dynamic Techniques for Information Flow Security. Several recent works propose run time monitors for information flow security, often with a particular focus on JavaScript, or on the Web context. These include monitoring algorithms that can handle DOM-like structures [27], dynamic code evaluation [1] and timeouts [26]. Austin and Flanagan [3] develop alternative, more permissive techniques. These run time monitoring based techniques are likely more efficient than the technique proposed in this paper, but they lack the precision of secure multi-execution: such monitors will block the execution of some non-interferent programs.

The idea underlying secure multi-execution was developed independently by several researchers. Capizzi *et al.* [8] propose *shadow executions*: they propose to run two executions of processes for the H (secret) and L (public) security level to provide strong confidentiality guarantees. Cristiá and Mata [12] independently formalize and prototype a similar system for secure multi-execution at operating system level. Devriese and Piessens [14] were the first to prove the strong soundness and precision guarantees that SME offers. They also report on a JavaScript implementation that requires a modified virtual machine. In a somewhat related line of work, Cavadini[9] proposes a technique based on program slicing to obtain secure fragments of insecure programs.

Several authors have improved on these initial results. Kashyap *et al.* [19], generalize the technique of secure multi-execution to a family of techniques that they call *the scheduling approach to non-interference*, and they analyze how the scheduling strategy can impact the security properties offered. Jaskelioff and Russo [18] propose a monadic library to realize secure multi-execution in Haskell. Bielova *et al.* [6] propose a variant of secure multi-execution suitable for reactive systems such as browsers. Finally, Austin and Flanagan [2] develop a more efficient implementation technique.

Finally, some other authors have considered program transformations for information flow security. Chudnov and Naumann [10] propose an inlined information flow monitor, and Birgisson *et al.* [7] propose a transformation towards a capability secure target language. Both approaches share the advantage of not requiring modifications to the operating system or virtual machine, but as with other classical run time monitors, they lack the precision of SME based approaches. In a sense, the approach proposed in this paper combines the advantages of these existing program-transformation based approaches with the advantages of SME (at the same performance cost as SME).

Other Security Techniques for JavaScript. A motivating example for the technique proposed in this paper is providing security for JavaScript script inclusion. Many authors have proposed alternative security mechanisms. Chugh *et al.*[11] develop a novel multi-stage static technique for enforcing information flow security in JavaScript.

Most authors focus on *isolation* or *sandboxing* rather than information flow security: how can scripts be included in web pages without giving them full access to the surrounding page and the browser APIs. Several practical systems have been proposed, including ADSafe [13], Caja [23] and Facebook JavaScript [15]. Maffeis *et al.* [21] formalize the key mechanisms underlying these systems and prove they can be used to create secure sandboxes. They also discuss several other existing proposals; we point the reader to their paper for a more extensive discussion of work in this area.

The capability security approach is of particular relevance to this paper, as we build on the isolation provided by a capability secure language to develop our prototype implementation for JavaScript. Maffeis *et al.* [22] formalize capability safety, and prove a Caja-like subset of JavaScript capability safe. Taly *et al.* [30] propose an approach to verify if APIs offered to sandboxed code are secure.

Ter Louw *et al.* propose AdJail [20], targeted at sandboxing advertisements by isolating them in a separate iframe, and by providing a stub in the original web page that communicates in a controlled way with the sandboxed advertisement.

8 Conclusion

Secure multi-execution is an appealing approach to enforce information flow policies: it is sound and precise, and can be applied to a variety of programming languages. In this paper, we have shown that the effect of SME can be achieved through program transformation, and without the need to modify the underlying computing infrastructure.

Acknowledgments. This research is partially funded by the Research Fund K.U.Leuven, by the IWT-SBO project SPION, and by the EU-funded FP7-projects HATS and WebSand. Dominique Devriese holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO).

The authors are grateful to the anonymous reviewers for their useful and detailed comments on the paper.

References

1. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: CSF, pp. 43–59 (2009)
2. Austin, T., Flanagan, C.: Multiple facets for dynamic information flow. In: POPL (2012)
3. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: PLAS (2010)
4. Barthe, G., Crespo, J.M., Devriese, D., Piessens, F., Rivas, E.: Secure multi-execution through static program transformation: extended version. Technical Report CW620, Department of Computer Science, Katholieke Universiteit Leuven (2012)
5. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSFW, pp. 100–114 (2004)
6. Bielova, N., Devriese, D., Massacci, F., Piessens, F.: Reactive non-interference for a browser model. In: NSS (2011)
7. Birgisson, A., Russo, A., Sabelfeld, A.: Capabilities for information flow. In: PLAS (2011)
8. Capizzi, R., Longo, A., Venkatakrishnan, V.N., Prasad Sistla, A.: Preventing information leaks through shadow executions. In: ACSAC (2008)
9. Cavadini, S.: Secure slices of insecure programs. In: ASIACCS, pp. 112–122 (2008)
10. Chudnov, A., Naumann, D.A.: Information flow monitor inlining. In: CSF, pp. 200–214 (2010)
11. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for Javascript. In: PLDI (2009)
12. Cristiá, M., Mata, P.: Runtime enforcement of noninterference by duplicating processes and their memories. In: WSEGI 2009 (2009)
13. Crockford, D.: Adsafe (December 2009), http://www.adsafe.org/
14. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: IEEE Symposium on Security and Privacy, pp. 109–124 (2010)
15. Facebook. Fbjs (2011), http://developers.facebook.com/docs/fbjs/
16. Le Guernic, G.: Confidentiality Enforcement Using Dynamic Information Flow Analyses. PhD thesis, Kansas State University (2007)
17. Heintze, N., Riecke, J.G.: The SLam calculus: programming with secrecy and integrity. In: Proc. ACM Symp. on Principles of Programming Languages, pp. 365–377 (January 1998)
18. Jaskelioff, M., Russo, A.: Secure multi-execution in haskell. In: PSI (2011)
19. Kashyap, V., Wiedermann, B., Hardekopf, B.: Timing- and termination-sensitive secure information flow: Exploring a new approach. In: Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP 2011, pp. 413–428. IEEE Computer Society, Washington, DC (2011)
20. Louw, M.T., Ganesh, K.T., Venkatakrishnan, V.N.: Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In: USENIX Security Symposium, pp. 371–388 (2010)
21. Maffeis, S., Mitchell, J.C., Taly, A.: Isolating JavaScript with Filters, Rewriting, and Wrappers. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 505–522. Springer, Heidelberg (2009)

22. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted web applications. In: IEEE Symposium on Security and Privacy, pp. 125–140 (2010)
23. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized javascript (January 2008), http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf
24. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Proc. ACM Symp. on Principles of Programming Languages, pp. 228–241 (January 1999)
25. Richards, G., Hammer, C., Burg, B., Vitek, J.: The Eval That Men Do. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 52–78. Springer, Heidelberg (2011)
26. Russo, A., Sabelfeld, A.: Securing timeout instructions in web applications. In: CSF, pp. 92–106 (2009)
27. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking Information Flow in Dynamic Tree Structures. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 86–103. Springer, Heidelberg (2009)
28. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. JSAC 21, 5–19 (2003)
29. Sabelfeld, A., Russo, A.: From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 352–365. Springer, Heidelberg (2010)
30. Taly, A., Erlingsson, U., Miller, M.S., Mitchell, J.C., Nagra, J.: Automated analysis of security-critical javascript apis. In: IEEE Symposium on Security and Privacy (2011)
31. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security 4(2/3), 167–188 (1996)

Synchronous Interface Theories and Time Triggered Scheduling

Benoît Delahaye[1,⋆], Uli Fahrenberg[2], Thomas A. Henzinger[3], Axel Legay[2,1], and Dejan Ničković[4,⋆⋆]

[1] Aalborg University, Denmark
[2] Irisa/INRIA Rennes, France
[3] IST Austria, Klosterneuburg, Austria
[4] Austrian Institute of Technology, Vienna, Austria

Abstract. We propose synchronous interfaces, a new interface theory for discrete-time systems. We use an application to time-triggered scheduling to drive the design choices for our formalism; in particular, additionally to deriving useful mathematical properties, we focus on providing a syntax which is adapted to natural high-level system modeling. As a result, we develop an interface model that relies on a guarded-command based language and is equipped with shared variables and explicit discrete-time clocks. We define all standard interface operations: compatibility checking, composition, refinement, and shared refinement. Apart from the synchronous interface model, the contribution of this paper is the establishment of a formal relation between interface theories and real-time scheduling, where we demonstrate a fully automatic framework for the incremental computation of time-triggered schedules.

1 Introduction

Interface models and theories were developed with the aim to provide a theoretical foundation for compositional design. Interface models describe both input assumptions on a component and its output guarantees and they support *incremental design* and *independent implementability*, two central concepts in component-based design. Interface theories [1,7,13,16,18,21,23,30] and related approaches [9,29] have been subject of active research in the past years, and today provide a strong and stable foundation for component-based design. However, although the theoretical foundations of interface theories can be now considered to be quite solid, the practical applicability of the framework has remained rather limited. One of the reasons is the fact that little attention has been given

⋆ Research partially supported by the Danish-Chinese Center for Cyber Physical Systems (Grant No.61061130541) and VKR Center of Excellence MT-LAB.

⋆⋆ The research leading to these results has received funding from ARTEMIS Joint Undertaking under grant agreement number 269335 and from national fundings (Federal Ministry for Transport, Innovation and Technology and Austrian Research Promotion Agency).

to adapt the modeling languages to the actual engineering needs in the target application domains.

In this paper, we propose *synchronous interfaces* (SI), a new interface theory motivated by an application to time-triggered scheduling and thus providing features that make our model closer to real-life needs of the engineers. In time-triggered communication scheduling, one allocates message transmissions to shared communication channels in a way that respects application-imposed and real-time constraints. The time-triggered scheduling problem can be naturally specified within an interface theory framework, by modeling scheduling constraints as interface guarantees, and considering the environment to be the scheduler. Even though our model has been developed with an eye to time-triggered scheduling, the application domain of the theory is much broader.

Incorrect scheduling of communication messages leads to violation of real-time and contention-freedom constraints, thus resulting in *timing incompatibilities.* This is in contrast to the standard interface theories that are untimed and are focused on reasoning about *value incompatibilities.* Continuous-time extensions of interface theories [15,20] were developed to tackle this problem. While those are of clear interest and can solve interesting problems [14], they suffer from the complexity of handling continuous time in an explicit manner that is often unnecessary in practical application areas. We believe that discrete time provides the right level of abstraction for many application areas, and demonstrate it with the time-triggered scheduling application.

We base the syntax of SI on the model of *reactive modules*, a high-level and general-purpose modeling language that provides a syntax close to procedural guarded-command languages. In addition, we extend our model with *shared variables* that allow simple specification of contention freedom constraints, and explicit discrete-time *clocks* that facilitate modeling timing constraints.

Semantically, a SI is a set of concurrent processes whose behavior is evolving in discrete time. We equip our theory with operations that support incremental design and independent implementability: (1) *well-formedness check* that computes the set of environment choices for which the interface meets its guarantees; (2) *composition* that allows to combine two interfaces and compute the assumptions under which they interact in a compatible way and (3) *refinement* and *shared refinement* that are used to compare behaviors of different interfaces.

The second contribution of this paper is the *incremental* computation of time-triggered schedules, that is resolved as an incremental design problem with SI. We model scheduling constraints as SI guarantees and consider the environment to be the (unknown) scheduler. We apply well-formedness checking to restrict the environment to those schedules that satisfy the scheduling constraints. The composition operator allows to solve scheduling problems incrementally, by decomposing them into subproblems whose restricted environments are combined into a full schedule (using the well-formedness check again).

Related Work. Compositional scheduling for hierarchical real-time systems has been extensively studied in [22,32] and other papers by the same authors, but in a setting which in a sense is complementary to ours. The focus of that work is on computing bounds on resource use under some (simple) schedulers, and on inferring resource bounds for complex systems in a compositional manner, whereas we focus on schedulability, i.e., computation of schedules under given task dependencies and resource bounds. Incremental time-triggered scheduling was also studied in [33], using an approach that computes schedules with an SMT solver, but may miss a feasible schedule.

Another area of related work, similar in spirit but different in methods, is the recent application of timed-automata based formalisms to schedulability problems. In [2], simple job-shop scheduling problems are solved using timed automata, and in [11,31], priced timed automata and games are used for schedulability under resource constraints. Another work in this area is [24], which is using timed automata extended with tasks for solving scheduling problems under uncertainty. Other approaches to solve worst-case scheduling problems are reported in [12,28,34]. A synchronous relational interface theory was proposed in [35], but without the notion of shared variables. Interface theories with shared variables were also proposed in [13] and [16,17]. However, unlike in SI, the information about ownership of the shared variables by individual components within a composed system is not preserved, thus not making them suitable to express time-triggered scheduling problems.

Other examples of component-design based methodologies include the BIP toolset [5,6] and its timed extension [3]. However, while BIP proposes features that are definitively beyond the scope of our work (generation of code, compilers, invariant-based verification), the approach does not permit to reason easily on shared variables, and does not provide (shared) refinement or pruning operators. Observe that several BIP-based approaches [8,25] capable of restraining the behaviors of a distributed system by avoiding deadlocks have the potential to solve scheduling problems. However, a detailed study of (incremental) scheduling problems has not been considered in the mentioned papers, hence it is not clear whether TTEthernet scheduling would easily translate into the BIP framework.

2 Synchronous Interfaces

A synchronous interface comes equipped with a finite set X of typed *variables* which is partitioned into sets $X = \mathit{extX} \cup \mathit{ctrX} \cup \mathit{sharedX}$ of *external*, *controlled*, and *shared* variables. External variables, also called *input* variables in interface theories [19], are controlled by the environment. At each round, the environment sets the values of external variables; the interface can read, but not modify them. Controlled, or *output* variables, are controlled by the interface: in each round, the interface assigns new values to all controlled variables. We further partition $\mathit{ctrX} = \mathit{intfX} \cup \mathit{privX}$ into *interface* and *private* variables. Interface variables can be seen by other SI, while private variables are local; hence private variables do not influence the communication behavior of a SI, and we can safely ignore them. We let $\mathit{obsX} = X \setminus \mathit{privX}$ denote the set of *observable* variables. We use *unprimed* symbols, such as x, to denote a *latched* value, and *primed* symbols, such as x', to denote an *updated* value of the variable x. We naturally extend this notation to sets of variables. The function $\texttt{type}(x)$ returns the type of variable x. In particular, clock variables have the type $\mathbb{C}$.

We follow the approach in [13] and introduce shared variables in the model, to facilitate communication using shared resources. We let the environment ensure the mutual exclusion property. Contrary to [13], we keep additional information on which individual component in the system owns the shared variable at each step of computation. In every computation step, the environment gives write access to a shared variable to at most one interface active in the system. We will define interface semantics following a game-oriented approach, hence this assumption is not a restriction.

Definition 1. *A* guarded command γ *from variables X to Y consists of a* guard p_γ *and an* action Act_γ*. The guard p_γ is a predicate over X, and Act_γ is either a* discrete action*: an expression α_γ from X to Y, or a* wait action*, using the keyword* **wait***.*

We use $\gamma[p_\gamma \setminus p'_\gamma]$ for the operation that consists in replacing the predicate p_γ by the predicate p'_γ. Controlled and shared variables are collected into *atoms*, which additionally contain guarded commands which specify rules for initializing and updating variables.

In interface theories, non-determinism reflects the fact that, given all the available information at a given step of the execution of an interface, several behaviors are possible for its next step. We let the environment resolve non-deterministic choices; this is implemented by assuming that for each $x \in \mathit{sharedX}$, there exists a non-empty set $\mathit{isCtr}_x = \{\mathit{isCtr}_x^A\}$ of external variables, one for each atom A potentially controlling x. A variable isCtr_x^A indicates whether atom A can safely write x at a given step of computation.

Definition 2. *An X-*atom *A consists of a* declaration *and a* body *of guarded commands. We distinguish between atoms defined on controlled variables $\mathit{ctr}(A)$ and those defined on shared variables $\mathit{shared}(A)$.*

- *The atom declaration for $\mathit{ctr}(A)$ consists of sets $\mathit{ctrX}_A \subseteq \mathit{ctrX}$, $\mathit{readX}_A \subseteq X$, and $\mathit{waitX}_A \subseteq X \setminus \mathit{ctrX}_A$ of* controlled*,* read*, and* awaited *variables. The atom body for $\mathit{ctr}(A)$ consists of a set $\mathit{Init}(A)$ of* initial *discrete guarded commands from waitX'_A to ctrX'_A and a set $\mathit{Update}(A)$ of* update *guarded commands from $\mathit{readX}_A \cup \mathit{waitX}'_A$ to ctrX'_A.*
- *The atom declaration for $\mathit{shared}(A)$ consists of sets $\mathit{sharedX}_A \subseteq \mathit{sharedX}$, $\mathit{readX}_A \subseteq X$, and $\mathit{waitX}_A \subseteq X \setminus \mathit{sharedX}_A$ of* shared*,* read*, and* awaited *variables, with $\mathit{isCtr}_x^A \in \mathit{waitX}_A$ for all $x \in \mathit{sharedX}_A$. The atom body for $\mathit{shared}(A)$ consists of a set $\mathit{Init}(A)$ of* initial *discrete guarded commands from waitX'_A to $\mathit{sharedX}'_A$ and a set $\mathit{Update}(A)$ of* update *guarded commands from $\mathit{readX}_A \cup \mathit{waitX}'_A$ to $\mathit{sharedX}'_A$.*

We denote by $P_{\mathit{Init}(A)} = \{p_\gamma \mid \gamma \in \mathit{Init}(A)\}$ and $P_{\mathit{Update}(A)} = \{p_\gamma \mid \gamma \in \mathit{Update}(A)\}$ the sets of predicates declared in initial and in update guarded commands of A. We say that a variable y awaits x, denoted $y \succ_A x$, if $y \in \mathit{ctrX}_A \cup \mathit{sharedX}_A$ and $x \in \mathit{waitX}_A$.

```
module $\mathrm{M_{ex}}$
external $r : \mathbb{B}, b : \mathbb{N}, c : \mathbb{N}$
            $isCtr_x^b : \mathbb{B}, isCtr_x^c : \mathbb{B}$;
shared $x : \mathbb{N}$;

atom $b$ reads $b, x$ awaits $r, isCtr_x^b$
  init
    [] $\neg isCtr_x^{b\prime} \rightarrow$;
  update
    [] $isCtr_x^{b\prime} \wedge \neg r' \rightarrow x' := x + b$;
    [] $isCtr_x^{b\prime} \wedge r' \rightarrow x' := 0$;
    [] $\neg isCtr_x^{b\prime} \rightarrow$;

atom $c$ reads $c, x$ awaits $r, isCtr_x^c$
  init
    [] $\neg isCtr_x^{c\prime} \rightarrow$;
  update
    [] $isCtr_x^{c\prime} \wedge \neg r' \rightarrow x' := x + c$;
    [] $isCtr_x^{c\prime} \wedge r' \rightarrow x' := 0$;
    [] $\neg isCtr_x^{c\prime} \rightarrow$;
```

Fig. 1. An example of a SI

Definition 3. *A* synchronous interface *(SI) M consists of a* declaration X_M *and a* body $\mathcal{A}_M$*, where X_M is a finite set of variables, and $\mathcal{A}_M = ctr(\mathcal{A}_M) \cup shared(\mathcal{A}_M)$ is a finite set of X_M-atoms for which $\bigcup_{A \in ctr(\mathcal{A}_M)} ctrX_A = ctrX_M$ and $\bigcup_{A \in shared(\mathcal{A}_M)} sharedX_A = sharedX_M$, $ctrX_{A_1} \cap ctrX_{A_2} = \emptyset$ for all atoms $A_1, A_2 \in ctr(\mathcal{A}_M)$ with $A_1 \neq A_2$, and such that the transitive closure $\succ_M = (\bigcup_{A \in \mathcal{A}_M} \succ_A)^+$ is asymmetric.*

These conditions ensure that the atoms in M control exactly the variables in $ctrX_M \cup sharedX_M$, that each variable in $ctrX_M$ is controlled by exactly one atom in $\mathcal{A}_M$, and that the await dependencies between variables in $\mathcal{A}_M$ are acyclic. A linear order $A_1, \ldots, A_n$ of the atoms in $\mathcal{A}_M$ is *consistent* if for all $1 \leq i < j \leq n$, the awaited variables in A_i are disjoint from the control variables in A_j. The asymmetry of $\succ_M$ guarantees the existence of a consistent order of atoms in $\mathcal{A}_M$. We denote by $P_M = \bigcup_{A \in \mathcal{A}_M} P_{Init(A)} \cup \bigcup_{A \in \mathcal{A}_M} P_{Update(A)}$ the set of all predicates declared in the guarded commands of M. Remark that in our examples, we name atoms by the set of variables they control. This is only possible when all atoms have disjoint sets of controlled variables.

Example 1. Consider the SI $\mathrm{M_{ex}}$ given in Figure 1. $\mathrm{M_{ex}}$ consists of two external integer variables b, c, three external Boolean variables $r, isCtr_x^b, isCtr_x^c$, and a shared integer variable x. Intuitively, $\mathrm{M_{ex}}$ models a simple additive controller that works as follows. The shared variable x is either incremented or reset at each time step in which the module controls x. $\mathrm{M_{ex}}$ controls x whenever $(isCtr_x^b \vee isCtr_x^c = \mathtt{t})$. In this case, if $r = \mathtt{t}$, then x is reset. Else, if atom b (resp. c) controls x $(isCtr_x^b = \mathtt{t})$, then x is incremented by the value of b (resp. c). If none of these atoms assign a value to the variable, then the environment will do.

3 Semantics

The intuition about the semantics of a SI is as follows: in each round, the environment assigns arbitrary values of correct type to external variables. Then, the atoms are executed in a (abitrary) static consistent order. As we do not assume that modules are input-enabled, there may be valuations of the external variables for which one or several atoms cannot be executed. Such configurations result in *deadlock states*. A given valuation of the variables is *reachable* if there exists a succession of rounds of the atom ending in this valuation.

Formally, the semantics of a SI is given by a *labeled transition system* (LTS). Given a SI M with set of variables X_M, we denote by $V[X_M]$ the set of valuations on variables in X_M. A *state* s of an interface M is a valuation in $V[X_M]$. We write $\Sigma_M = \Sigma_{X_M}$ for the set of states of M. Given a state $s \in \Sigma_M$ and $Y \subseteq X_M$, we denote by $s[Y]$ the projection of the state s to the valuations of variables in Y. Note that we will define the semantics in a way which keeps enough information about the syntax to be able to go back from semantics to syntax; this is important for several of the operations which we define in the next section, as these are defined only at the semantics level.

Given a state s of an interface M, we denote by $\mathtt{safe}^{sv}_M(s)$ the predicate that indicates whether the state is safe with respect to shared variables in M; formally, $\mathtt{safe}^{sv}_M(s) = \mathtt{t}$ iff $\forall x \in sharedX_M, \bigwedge_{A,A' \in M, A \neq A'} s[isCtr^A_x] \wedge s[isCtr^{A'}_x] = \mathtt{f}$. Intuitively, a state s of M is safe if and only if, for all shared variables x, there is at most one atom that controls x.

Definition 4. *Let X, Y, and $Z \subseteq Y$ be sets of variables and γ a guarded command from X to Y. We define the semantics $[\![\gamma]\!] \subseteq \Sigma_X \times \Sigma_Y$ of γ as follows:*

- *If γ is of the form $p_\gamma \rightarrow \alpha_\gamma$, where $\alpha_\gamma : V[X] \rightarrow V[Z]$, then $(s,t) \in [\![\gamma]\!]$ iff (1) $s \models p_\gamma$; (2) $\forall z \in Z, t[z] = \alpha_\gamma(s)[z]$; (3) $\forall y \in Y \backslash Z$ such that $\mathtt{type}(y) \neq \mathbb{C}$, $t[y] = s[y]$ and (4) $\forall y \in Y \backslash Z$ such that $\mathtt{type}(y) = \mathbb{C}$, $t[y]. = s[y] + 1$*
- *If γ is of the form $p_\gamma \rightarrow$ **wait**, then $(s,t) \in [\![\gamma]\!]$ iff (1) $s \models p_\gamma$, (2) $\forall y \in Y$ such that $\mathtt{type}(y) = \mathbb{C}$, $t[y] = s[y] + 1$, (3) $\forall y \in Y$ such that $\mathtt{type}(y) \neq \mathbb{C}$, $t[y] = s[y]$, and (4) $t \models p_\gamma$.*

Let A be an atom from X to Y and let Γ_A be either $Init(A)$ or $Update(A)$, i.e., a finite set of guarded commands. Then, Γ_A defines a relation $[\![\Gamma_A]\!] \subseteq \Sigma_X \times \Sigma_Y$ such that $(s,t) \in [\![\Gamma_A]\!]$ iff $(s,t) \in [\![\gamma]\!]$ for some $\gamma \in \Gamma_A$.

The semantics of SI is a LTS whose states represent valuations of variables, and whose transitions correspond to complete rounds of updates for all atoms A_i in a static consistent order $A_1, \dots A_n$.

Definition 5. *The semantics of a SI M is the LTS $[\![M]\!] = (S_M, S^0_M, \rightarrow_M, L_M)$ with $S_M = V[X_M] \cup \{s_{init}\}$, $S^0_M = \{s_{init}\}$, $L_M \subseteq P_M^{\mathcal{A}_M}$ the set of all functions $l : \mathcal{A}_M \rightarrow P_M$ for which $l(A) \in P_A$ for all $A \in \mathcal{A}_M$, and $\rightarrow_M$ defined as follows:*

- *$(s_{init}, l, t) \in \rightarrow_M$ iff $\mathtt{safe}^{sv}_M(t)$ and there exist $\gamma_1, \dots, \gamma_n$ such that for all $1 \leq i \leq n$, $\gamma_i \in Init(A_i), l(A_i) = p_{\gamma_i}$ and there exists $s^0 \in V[X_M]$ such that $t = [\![Init(A_n)]\!] \circ \cdots \circ [\![Init(A_1)]\!](s^0)$.*
- *$(s, l, t) \in \rightarrow_M$ iff $\mathtt{safe}^{sv}_M(s)$, $\mathtt{safe}^{sv}_M(t)$, and there exist $\gamma_1, \dots, \gamma_n$ such that for all $1 \leq i \leq n$, $\gamma_i \in Update(A_i), l(A_i) = p_{\gamma_i}$ and $t = [\![Update(A_n)]\!] \circ \cdots \circ [\![Update(A_1)]\!](s)$.*

Note that we label each transition by the predicates of the guarded commands that are effectively executed during the round, hence we preserve full syntactic information about the interface in its semantics. In the following, we may omit this labelling in our notations when we do not need the information.

```
module M                                      | module GS(M)
external a : ℕ;                               | external a : ℕ;
interface b, c : ℕ;                           | interface b, c : ℕ;
                                              |
atom b awaits a        atom c awaits b        | atom b awaits a        atom c awaits b
  initupdate             initupdate           |   initupdate             initupdate
    [] a′ ≤ 5 → b′ := 1;   [] b′ ≤ 1 → c′ := 1; |     [] a′ ≤ 5 → b′ := 1;   [] b′ ≤ 1 → c′ := 1;
    [] a′ ≥ 2 → b′ := 2;                      |     [] false → b′ := 2;
```

Fig. 2. An example of guard strengthening. Left: M, right: $\mathsf{GS}(M)$

A *trajectory* of a SI M is a *finite* sequence of states $s_0, s_1, \ldots, s_n$ in $[\![M]\!]$ such that: (1) $s_0 = s_{init}$; (2) $(s_i, s_{i+1}) \in \rightarrow_M$ for all $0 \le i < n$; and (3) no deadlock states are reachable from s_n. The sequence $s_1[obsX_M], \ldots, s_n[obsX_M]$ of observable valuations is called a *trace* of M; the *trace language* $L(M)$ of M is the set of traces of M. In our optimistic approach, computing environments that cannot result in deadlock states amounts to projecting $L(M)$ onto the external variables. Note that we can compute these environments if the interface has a finite representation of its trace language, i.e. if $[\![M]\!]$ has a finite state space. We say that M is *well-formed* if $L(M) \neq \emptyset$.

4 Operations

We now describe some operations on SI which will allow us to use SI as an interface theory. Note that we will also use some of these operations for incremental scheduling in Section 5; but notably shared refinement is not used in incremental scheduling, yet a necessary ingredient in any interface theory.

Guard Strengthening. Given a well-formed interface M, we are interested in computing an equivalent module (in terms of infinite executions) in which no deadlock states are reachable. This *guard strengthening* $\mathsf{GS}(M)$ is computed by strengthening the guards of M in such a way that deadlocks are forbidden. An illustration of guard strengthening is given in Figure 2.

Semantically, the construction relies on a notion of recursively *pruning* deadlock states together with states which inevitably lead to them: Let M be a SI and let $[\![M]\!] = (S_M, S_M^0, \rightarrow_M, L_M)$ be its associated LTS. Define the function $\mathrm{Succ}_X : S_M \times V[X] \rightarrow 2^{S_M}$ by

$$\mathrm{Succ}_X(q, v) = \{q' \mid (q, q') \in \rightarrow_M \text{ and } q'[X] = v\}\,.$$

This function gives all successors of q in $[\![M]\!]$ which for variables in X match the valuation v. Next we define a mapping Pred which outputs the controllable predecessors of a subset $B \subseteq S_M$:

$$\mathrm{Pred}(B) = \{s \mid \forall v \in V[extX_M] : \\ \mathrm{Succ}_{extX_M}(s, v) \neq \emptyset \Rightarrow \mathrm{Succ}_{extX_M}(s, v) \cap B \neq \emptyset\}.$$

Denote by Pred* the closure of Pred, let $B = \{s \in S_M \mid \forall t \in S_M : (s,t) \notin \rightarrow_M\}$ be the deadlock states and $B^* = \text{Pred}^*(B)$. Intuitively, these are states from which the environment cannot prevent M from reaching a deadlock.

The *pruning* of $[\![M]\!]$ is then given by the LTS $\rho([\![M]\!]) = (S_M \backslash B^*, S_M^0 \backslash B^*, \rightarrow'_M, L_M)$, where $\rightarrow'_M = \{(s,l,s') \in \rightarrow_M \mid s' \notin B^*\}$. Intuitively, pruning $[\![M]\!]$ removes all bad states and transitions leading to them, which reduces the state-space of M without affecting its language. Note that if M is not well-formed, then $\rho([\![M]\!])$ has no initial states; we then say that the pruning of M is empty.

For guard strengthening at the *syntactic* level, matching the pruning of the semantics, we proceed as follows: for each initial set of guarded commands $\mathit{Init}(A)$ of an atom A in $\mathcal{A}_M$ and $\gamma \in \mathit{Init}(A)$, the predicate $p_\gamma \in \gamma$ is replaced by

$$\tilde{p}_\gamma = \bigvee\nolimits_{(s_{init}, t \in S_M \backslash B^*, (s,l,t) \in \rightarrow_M, l(A) = p_\gamma)} \bigwedge\nolimits_{(x \in X_M[waitX_A])} x' = t[x].$$

Similarly, for each update set of guarded commands $\mathit{Update}(A)$ and $\gamma \in \mathit{Update}(A)$, the predicate $p_\gamma \in \gamma$ is replaced by

$$\tilde{p}_\gamma = \bigvee\nolimits_{(s, t \in S_M \backslash B^*, (s,l,t) \in \rightarrow_M, l(A) = p_\gamma)} \bigwedge\nolimits_{(x \in X_M[waitX_A])} x' = t[x] \wedge \bigwedge\nolimits_{(x \in X_M[readX_A])} x = s[x].$$

Intuitively, this method amounts to an enumeration, for every guarded command, of possible valuations of read and awaited variables that cannot reach a bad state. Replacing original predicates with associated enumerations prevents exactly bad behaviors. It follows, by construction, that $[\![\mathsf{GS}(M)]\!] \equiv \rho([\![M]\!])$, hence also that M is well-formed if and only if $[\![\mathsf{GS}(M)]\!]$ is not empty. As the trace language $L(M)$ by definition only includes traces which cannot be extended to a deadlock state, we also have $L(M) = L(\mathsf{GS}(M))$.

Parallel Composition. We introduce a synchronous parallel composition operation which combines two compatible SI. We say that SI M and N are *composable* if (1) the interface variables of M and N are disjoint; and (2) the await dependencies between the observable variables of M and N are acyclic, that is the transitive closure $(\succ_M \cup \succ_N)^+$ is asymmetric. Observe that the sets of shared variables may overlap, and that private variables are not taken into consideration here: these are not visible from the outside, hence in case of private variables with the same name, we consider that they are different and belong to different name spaces.

We say that two composable SI are *compatible* if there exists an environment in which they can be composed without reaching deadlock states. Informally, the synchronous composition P of M and N consists in the union of their atoms, where some controlled variables in M can constrain external variables in N, and vice-versa. An execution of P thus consists in an update of all the remaining external variables, followed by an update of the controlled and shared variables of M and N.

Definition 6. *Let M and N be two composable SI. Define an intermediate module P by $privX_P = privX_M \cup privX_N$, $intfX_P = intfX_M \cup intfX_N$, $extX_P =$*

$extX_M \cup extX_N \setminus intfX_P$, $sharedX_P = sharedX_M \cup sharedX_N$, and finally, $\mathcal{A}_P = \mathcal{A}_M \cup \mathcal{A}_N$. M and N are compatible *if P is well-formed, in which case we define $M \parallel N = \mathsf{GS}(P)$.*

The below theorem shows that parallel composition is *associative*, hence allowing incremental composition. The theorem follows directly from the fact that pruning does not affect the trace language.

Theorem 1. *For composable SI M_1, M_2, and M_3, $L(M_1 \parallel (M_2 \parallel M_3)) = L((M_1 \parallel M_2) \parallel M_3)$.*

Refinement. Refinement of SI allows comparing interfaces. Informally, if N refines M, then N works in at least all the environments where M works, and all the behaviors of N defined in these environments are also behaviors of M. Hence refinement for SI is similar to alternating simulation for I/O automata [4]. For valuations $v \in V[extX_M]$, we define the set $ctr_M(v)$ of shared variables that are controlled by M according to v by $ctr_M(v) = \{x \in sharedX_M \mid (\bigvee_{A \in M} v[isCtr_x^A]) = \mathtt{t}\}$, and we let $noctr_M(v) = extX_M \cup (sharedX_M \setminus ctr_M(v))$.

Definition 7. *Let M and N be SI and $[\![M]\!] = (S_M, S_M^0, \rightarrow_M, L_M)$, $[\![N]\!] = (S_N, S_N^0, \rightarrow_N, L_N)$. We say that N* refines *M, written as $N \leq M$, if $extX_M \subseteq extX_N$, $intfX_N \subseteq obsX_M$, $sharedX_N = sharedX_M$, and there exists a relation $R \subseteq S_N \times S_M$ such that $(s_{init}, t_{init}) \in R$ and for all $(s, t) \in R$, we have*

- *$(s,t) \neq (s_{init}, t_{init})$ implies that $s[extX_M \cup intfX_N \cup sharedX_M] = t[extX_M \cup intfX_N \cup sharedX_M]$*
- *for all $v \in V[extX_M]$ and $v' \in V[sharedX_M \setminus ctr_M(v)]$ it holds that if $\mathrm{Succ}_{noctr(v)}(t, v \cup v') \neq \emptyset$, then also $\mathrm{Succ}_{noctr(v)}(s, v \cup v') \neq \emptyset$, and then for all $s' \in \mathrm{Succ}_{noctr(v)}(s, v \cup v')$, there exists $t' \in \mathrm{Succ}_{noctr(v)}(t, v \cup v')$ such that $s'Rt'$.*

The relation between refinement and trace languages is as follows: for a SI M, let $\mathrm{adm}(M) = \{w \in extX_M^* \mid \exists w' \in L(M). w'{\downarrow_{extX_M}} = w\}$ be the set of all *admissible external valuations*; here $w'{\downarrow_{extX_M}}$ denotes the projection of w' to external variables, hence we are collecting all traces of valuations of external variables which *do not block* execution of M. Then:

Theorem 2. *For SI N, M with $N \leq M$ we have $\{w \in L(N) \mid w{\downarrow_{extX_M}} \in \mathrm{adm}(M)\} \subseteq L(M)$.*

The next theorem shows that SI theory supports *independent implementability*: Refinement is compatible with parallel composition in the sense that components may be refined individually.

Theorem 3. *Given SI M_1, M_1', M_2, M_2' with M_1 and M_2 compatible, if $M_1' \leq M_1$ and $M_2' \leq M_2$, then M_1' is compatible with M_2' and $M_1' \parallel M_2' \leq M_1 \parallel M_2$.*

Shared Refinement. We finish this section by mentioning that there is also a notion of *shared refinement* for SI which supports component reuse in different parts of a design. The shared refinement of two SI M_1 and M_2 is the SI $M = M_1 \wedge M_2$ which is the product of the state spaces in LTSs of M_1 and M_2, with appropriate transitions ensuring that M_1 and M_2 evolve synchronously along the same transitions. Hence M accepts inputs that satisfy any of the assumptions from M_1 and M_2, and it provides outputs that satisfy both guarantees of M_1 and M_2. In particular, M can be used to implement two different aspects of a single component. Moreover, M is the *smallest* such SI in the sense of the theorem below.

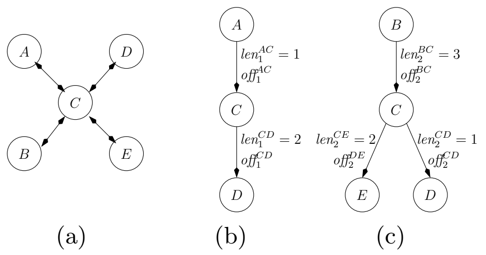
Theorem 4. *Given two SI M_1 and M_2, we have that: (1) $M_1 \wedge M_2 \leq M_1$; (2) $M_1 \wedge M_2 \leq M_2$; and (3) for all SI M' such that $M' \leq M_1$ and $M' \leq M_2$, also $M' \leq M_1 \wedge M_2$.*

5 Incremental TTEthernet Scheduling with SI

In this final section we present a methodology for solving scheduling problems using the synchronous interface theory developed in this paper. We concentrate on the particular application of *TTEthernet* scheduling [26], but our framework is sufficiently general that it also allows application to other scheduling and job-shop problems.

A specification of a TTEthernet network consists of a *physical topology*, a set of *frames* and a set of time-triggered scheduling *constraints*. The physical topology is an undirected graph consisting of a set of vertices, corresponding to communicating *devices* (end-systems or switches), and edges, representing bi-directional communication links, called *data-flow links*, between devices. A frame specifies a message that is sent over the network, and is represented by a tree that defines the route for the message delivery from a sender device to a set of receiver devices. Every edge in the tree represents the frame on a particular data-flow link, and is characterized by its *period* (relative deadline for the frame arrival from the sender to its receiver), *length* (value denoting frame delivery duration on the data-flow link) and *offset* (actual time slot at which the frame is sent from a sender to a receiver device). Like in [33], we assume, without loss of generality, that the frame period *Period* is the same for all frames on all data-flow links in the specification. Finally, time-triggered scheduling constraints are defined over the offset values of frames on data-flow links. To simplify presentation, we consider only two most common types of TT scheduling constraints: (1) *contention free* (CF) constraints that impose to any reasonable schedule to forbid simultaneous presence of two frames on the same data-flow link; and (2) *path-dependent* (PD) constraints that impose correct flow of a frame through data-flow links, ensuring that a device cannot send a frame before receiving it.

In a TTEthernet network specification, the only non-fixed values are the offset parameters of frames in data-flow links. A schedule that satisfies



CF:	$\mathit{off}_1^{CD} \geq \mathit{off}_2^{CD} + len_2^{CD} \vee$ $\mathit{off}_2^{CD} \geq \mathit{off}_1^{CD} + len_1^{CD}$
PD:	$\mathit{off}_1^{CD} \geq \mathit{off}_1^{AC} + len_1^{AC}$ $\mathit{off}_2^{CE} \geq \mathit{off}_2^{BC} + len_2^{BC}$ $\mathit{off}_2^{CD} \geq \mathit{off}_2^{BC} + len_2^{BC}$

(d)

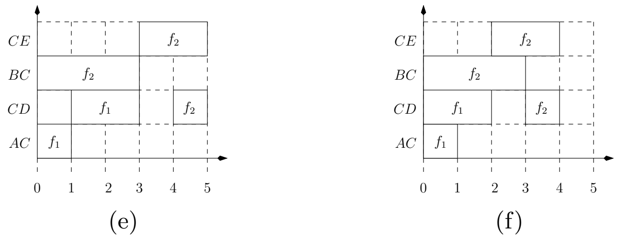


Fig. 3. Specification of a TTEthernet network: (a) network topology N; (b), (c) specification of frame routes f_1 and f_2; (d) constraints on offset values; (e) feasible schedule; (f) infeasible schedule

the specification corresponds to an assignment of concrete values to offset parameters which satisfies all the constraints. The TT scheduling problem consists in computing such a schedule from a specification.

We introduce our methodology by way of an example below, but in essence, it proceeds as follows:

1. Introduce a SI *Clock* which keeps track of time within a period.
2. Model each frame as a SI, including transmission length, path dependency, and shared resources.
3. Use parallel composition and well-formedness check to incrementally reject all non-feasible offset values.
4. If any feasible offset values remain after the preceding step, then any of these constitutes a feasible schedule. Otherwise the problem is unschedulable.

Figure 3 depicts an example of a time-triggered scheduling problem for a particular TTEthernet network specification. The input to the scheduling problem consists of the network topology N (Figure 3(a)) and two frames f_1, f_2 (Figures 3(b) and (c)). The contention-freedom and path-dependency constraints induced by the frames are depicted in Figure 3(d). Solving the scheduling problem specified in Figure 3 consists in computing the feasible schedules that satisfy all the requirements of the specification. Figures 3(e) and (f) depict two schedules, one that satisfies and another that violates the specification.

```
module Clock
interface $clk_P : \mathbb{C}$;
atom $clk_P$ reads $clk_P$
  init
    [] t → $clk_P{}' := 0$;
  update
    [] $(clk_P \geq P-1) \to clk_P{}' := 0$;
    [] $clk_P < P-1 \to$ wait;
```

Fig. 4. SI *Clock*

To solve the example scheduling problem, we first introduce a SI *Clock* as depicted in Figure 4 which measures the relative time within every period using an explicit clock variable clk_P. This clock is visible to all other interfaces in the system. Then we model the two frames f_1 and f_2 as two independent interfaces M_1 and M_2; this will allow to solve the problem incrementally. The application of the well-formedness check operator on the composition $Clock \| M_i$ computes the set of all feasible partial schedules that are consistent with the scheduling (path-dependency) constraints of the frame f_i. We then use the parallel composition of *Clock* with M_1 and M_2 to combine compatible partial schedules for f_1 and f_2, effectively removing all schedules that violate the contention-freedom constraint.

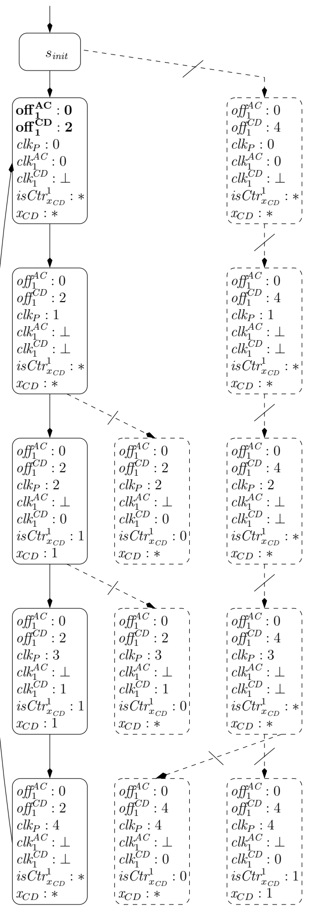
```
module $M_1$
external $soff_1^{AC} : [0,P)$, $soff_1^{CD} : [0,P)$, $clk_P : \mathbb{C}$;
         $isCtr^1_{x_{CD}} : \mathbb{B}$;
interface $off_1^{AC} : [0,P)$, $off_1^{CD} : [0,P)$
interface $clk_1^{AC} : \mathbb{C}$; $clk_1^{CD} : \mathbb{C}$;
shared $x_{CD} : \mathbb{B}$;

atom $off_1^{AC}, off_1^{CD}$ awaits $soff_1^{AC}, soff_1^{CD}$
  init
    [] $soff_1^{CD\prime} \geq soff_1^{AC\prime} + len_1^{AC} \to$
       $off_1^{AC\prime} := soff_{1AC}{}', off_1^{CD\prime} := soff_1^{CD\prime}$;
  update
    [] t →;

atom $clk_1^{AC}$ awaits $off_1^{AC}, clk_P$
  init
    [] $off_1^{AC\prime} = 0 \to clk_1^{AC\prime} := 0$;
    [] $off_1^{AC\prime} \neq 0 \to clk_1^{AC\prime} := \perp$;
  update
    [] $clk_1^{AC\prime} < len_1^{AC} \wedge clk_P{}' < P-1 \to$ wait;
    [] $clk_1^{AC\prime} = len_1^{AC} \wedge clk_P{}' < P \to clk_1^{AC\prime} := \perp$;

atom $clk_1^{CD}$ awaits $off_1^{CD}, clk_P$
  init
    [] $off_1^{CD\prime} \neq 0 \to clk_1^{CD\prime} := \perp$;
  update
    [] $clk_1^{CD\prime} < len_1^{CD} \wedge clk_P{}' < P-1 \to$ wait;
    [] $clk_1^{CD\prime} = len_1^{CD} \wedge clk_P{}' < P \to clk_1^{CD} := \perp$;

atom $x_{CD}$ awaits $clk_1^{CD}, isCtr^1_{x_{CD}}$
  initupdate
    [] $isCtr^1_{x_{CD}} \wedge clk_1^{CD\prime} \in [0, len_1^{CD}) \to x_{CD}{}' :=$ t;
    [] $\neg isCtr^1_{x_{CD}} \wedge clk_1^{CD\prime} \notin [0, len_1^{CD}) \to$
```



s_{init}
$\mathbf{off_1^{AC}} : \mathbf{0}$, $\mathbf{off_1^{CD}} : \mathbf{2}$, $clk_P : 0$, $clk_1^{AC} : 0$, $clk_1^{CD} : \perp$, $isCtr^1_{x_{CD}} : *$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 4$, $clk_P : 0$, $clk_1^{AC} : 0$, $clk_1^{CD} : \perp$, $isCtr^1_{x_{CD}} : *$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 2$, $clk_P : 1$, $clk_1^{AC} : \perp$, $clk_1^{CD} : \perp$, $isCtr^1_{x_{CD}} : *$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 4$, $clk_P : 1$, $clk_1^{AC} : \perp$, $clk_1^{CD} : \perp$, $isCtr^1_{x_{CD}} : *$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 2$, $clk_P : 2$, $clk_1^{AC} : \perp$, $clk_1^{CD} : 0$, $isCtr^1_{x_{CD}} : 1$, $x_{CD} : 1$
$off_1^{AC} : 0$, $off_1^{CD} : 2$, $clk_P : 2$, $clk_1^{AC} : \perp$, $clk_1^{CD} : 0$, $isCtr^1_{x_{CD}} : 0$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 4$, $clk_P : 2$, $clk_1^{AC} : \perp$, $clk_1^{CD} : \perp$, $isCtr^1_{x_{CD}} : *$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 2$, $clk_P : 3$, $clk_1^{AC} : \perp$, $clk_1^{CD} : 1$, $isCtr^1_{x_{CD}} : 1$, $x_{CD} : 1$
$off_1^{AC} : 0$, $off_1^{CD} : 2$, $clk_P : 3$, $clk_1^{AC} : \perp$, $clk_1^{CD} : 1$, $isCtr^1_{x_{CD}} : 0$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 4$, $clk_P : 3$, $clk_1^{AC} : \perp$, $clk_1^{CD} : \perp$, $isCtr^1_{x_{CD}} : *$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 2$, $clk_P : 4$, $clk_1^{AC} : \perp$, $clk_1^{CD} : \perp$, $isCtr^1_{x_{CD}} : *$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 4$, $clk_P : 4$, $clk_1^{AC} : \perp$, $clk_1^{CD} : 0$, $isCtr^1_{x_{CD}} : 0$, $x_{CD} : *$
$off_1^{AC} : 0$, $off_1^{CD} : 4$, $clk_P : 4$, $clk_1^{AC} : \perp$, $clk_1^{CD} : 0$, $isCtr^1_{x_{CD}} : 1$, $x_{CD} : 1$

Fig. 5. Synchronous interface M_1: (a) syntax; (b) part of its pruned semantics

We now encode the frame f_1 as a SI M_1, shown in Figure 5. The environment (scheduler) owns the variables soff_1^{AC} and soff_1^{CD} (line 2), that are used to propose in the initial state the offset values for the message of frame f_1 on the data flow AC and CD, respectively. The interface M_1 checks in line 8 whether the proposed values satisfy the path-dependency constraint, and accordingly rejects the offsets, or accepts them and copies them into the controlled variables off_1^{AC} and off_1^{CD}. The atom depicted in lines $6-10$ controls a local clock clk_1^{AC}, that measures the time length of the message transmitted on the data flow link AC by the frame f_1. The clock clk_1^{AC} is reset when the corresponding offset value is reached, and the atom ensures that the transmission of the message is finished before the end of the period P. The atom that controls the local clock clk_1^{CD}, depicted in lines $18-23$, does the same monitoring of the message transmitted by f_1 on the data flow link CD. Finally, the last atom controls the shared variable x_{CD}, that models the shared resource (data flow link) CD. It ensures that when the frame f_1 is given access to the data flow link CD (via the external variable $\mathit{isCtr}^1_{x_{CD}}$), it is not preempted before the message transmission is done.

In order to compute the partial feasible schedules for f_1, one needs to apply the well-formedness check on $\mathit{Clock}\|M_1$, which amounts to generating the pruned semantics graph of this composition (also shown (in parts) in Figure 5).

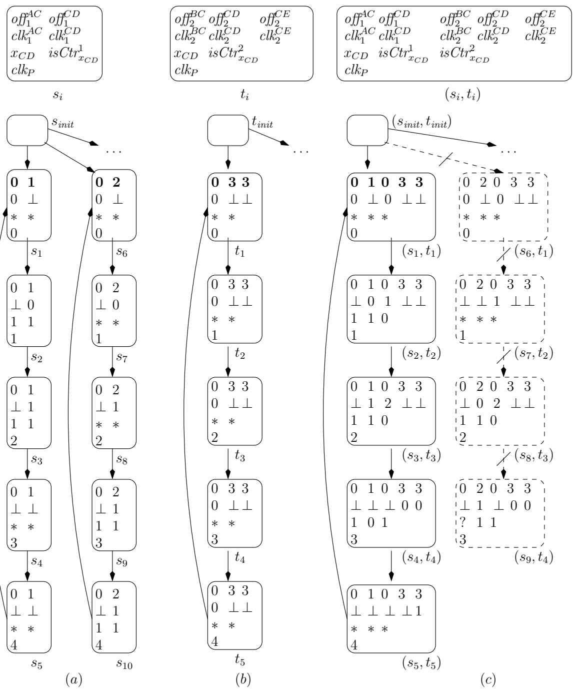


Fig. 6. Parallel composition of M_1 and M_2: fragments of (a) $\rho(\llbracket \mathit{Clock}\|M_1\rrbracket)$; (b) $\rho(\llbracket \mathit{Clock}\|M_2\rrbracket)$ and (c) $\rho(\rho(\llbracket \mathit{Clock}\|M_1\rrbracket)\|\rho(\llbracket \mathit{Clock}\|M_2\rrbracket))$

The well-formedness checking results in pruning all states that lead to a deadlock, i.e., it removes all states where the offsets that are proposed by the environment result in a violation of a scheduling constraint. The partial feasible schedules are encoded as the valuations of off_1^{AC} and off_1^{CD} in the remaining initial states.

The encoding of the scheduling problem for f_2 into a synchronous interface M_2, and the corresponding computation of the partial feasible schedules for f_2 is done in a similar way. Given the pruned transition systems $\rho([\![\mathit{Clock}\|M_1]\!])$ and $\rho([\![\mathit{Clock}\|M_2]\!])$, the parallel composition combines the two systems and removes the joint behaviors that are not compatible. In our example, it amounts to remove all the behaviors in which the mutual exclusion property on the access to the shared variable x_{CD} is violated, thus falsifying the contention-freedom scheduling constraint. The pruned transition system of the composition encodes exactly all feasible schedules of the original problem. Figure 6 depicts two fragments of the transition systems for $\mathit{Clock}\|M_1$ and $\mathit{Clock}\|M_2$ and of the pruned semantics of their composition.

6 Conclusion and Further Work

We present in this paper a simple yet powerful model for *synchronous interfaces*. Contrary to most other interface models one finds in the literature, the modeling language we use is inspired by a specific application domain, resulting in a model that resembles a high-level programming language. At the same time, we allow explicit use of time and of shared variables that are treated in a flexible way, resulting in a rich model which satisfies most common engineering needs. We develop our model into an *interface theory*, allowing for high-level reasoning and component-based design using (shared) refinement, composition and pruning. We propose to use our interface theory as an elegant solution for incremental computation of time-triggered schedules.

In the future, we plan to implement the SI framework and apply it to different scheduling problems. We also believe that the state-based type of analysis on SI makes our approach a good candidate for development of efficient and flexible heuristics, by assigning value functions to states and restricting the search space to the assigned values. Finally, we plan to extend our approach in order to incorporate deeper information about the platform on which the system is running, like in the spirit of recent works [10,27] done in the context of the untimed BIP and UPPAAL frameworks.

References

1. Aarts, F., Vaandrager, F.: Learning I/O Automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 71–85. Springer, Heidelberg (2010)
2. Abdeddaïm, Y., Asarin, E., Maler, O.: Scheduling with timed automata. TCS 354(2), 272–300 (2006)
3. Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: EMSOFT, pp. 229–238. ACM (2010)

4. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating Refinement Relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)
5. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
6. Basu, A., Mounier, L., Poulhiès, M., Pulou, J., Sifakis, J.: Using BIP for modeling and verification of networked systems – a case study on TinyOS-based networks. In: NCA, pp. 257–260. IEEE Computer Society (2007)
7. Bauer, S.S., Mayer, P., Schroeder, A., Hennicker, R.: On Weak Modal Compatibility, Refinement, and the MIO Workbench. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 175–189. Springer, Heidelberg (2010)
8. 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)
9. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple Viewpoint Contract-Based Specification and Design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 200–225. Springer, Heidelberg (2008)
10. Bourgos, P., Basu, A., Bozga, M., Bensalem, S., Sifakis, J., Huang, K.: Rigorous system level modeling and analysis of mixed HW/SW systems. In: MEMOCODE, pp. 11–20. IEEE (2011)
11. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite Runs in Weighted Timed Automata with Energy Constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
12. Burns, A.: Preemptive priority based scheduling: An appropriate engineering approach. In: PRTS, pp. 225–248 (1994)
13. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Mang, F.Y.C.: Synchronous and Bidirectional Component Interfaces. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 414–427. Springer, Heidelberg (2002)
14. David, A., Larsen, K.G., Legay, A., Nyman, U., Wąsowski, A.: ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 365–370. Springer, Heidelberg (2010)
15. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: HSCC, pp. 91–100. ACM (2010)
16. de Alfaro, L., da Silva, L.D., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable Interfaces. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 81–105. Springer, Heidelberg (2005)
17. de Alfaro, L., Faella, M.: An Accelerated Algorithm for 3-Color Parity Games with an Application to Timed Games. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 108–120. Springer, Heidelberg (2007)
18. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC / SIGSOFT FSE, pp. 109–120 (2001)
19. de Alfaro, L., Henzinger, T.A.: Interface Theories for Component-Based Design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
20. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed Interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)

21. Doyen, L., Henzinger, T.A., Jobstmann, B., Petrov, T.: Interface theories with component reuse. In: EMSOFT, pp. 79–88. ACM (2008)
22. Easwaran, A., Shin, I., Sokolsky, O., Lee, I.: Incremental schedulability analysis of hierarchical real-time components. In: EMSOFT, pp. 272–281. ACM (2006)
23. Emmi, M., Giannakopoulou, D., Păsăreanu, C.S.: Assume-Guarantee Verification for Interface Automata. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 116–131. Springer, Heidelberg (2008)
24. Fersman, E., Krčál, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. I&C 205(8), 1149–1172 (2007)
25. Graf, S., Peled, D., Quinton, S.: Monitoring Distributed Systems Using Knowledge. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 183–197. Springer, Heidelberg (2011)
26. Kopetz, H., Ademaj, A., Grillinger, P., Steinhammer, K.: The time-triggered ethernet (TTE) design. In: ISORC, pp. 22–33. IEEE Computer Society (2005)
27. Mikučionis, M., Larsen, K.G., Rasmussen, J.I., Nielsen, B., Skou, A., Palm, S.U., Pedersen, J.S., Hougaard, P.: Schedulability Analysis Using Uppaal: Herschel-Planck Case Study. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 175–190. Springer, Heidelberg (2010)
28. Palm, S.: Herschel-Planck ACC ASW: sizing, timing and schedulability analysis. Tech. rep., Terma A/S (2006)
29. Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: SEFM, pp. 377–381. IEEE Computer Society (2008)
30. Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: Modal interfaces: unifying interface automata and modal specifications. In: EMSOFT, pp. 87–96. ACM (2009)
31. Rasmussen, J.I., Larsen, K.G., Subramani, K.: On using priced timed automata to achieve optimal scheduling. FMSD 29(1), 97–114 (2006)
32. Shin, I., Lee, I.: Compositional real-time scheduling framework. In: RTSS, pp. 57–67. IEEE Computer Society (2004)
33. Steiner, W.: An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks. In: RTSS, pp. 375–384 (2010)
34. Terma A/S. Software timing and sizing budgets. Tech. rep., Terma A/S, Issue 9
35. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM Trans. Program. Lang. Syst. 33(4), 14 (2011)

TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs

Samira Tasharofi[1], Rajesh K. Karmani[1], Steven Lauterburg[2], Axel Legay[3], Darko Marinov[1], and Gul Agha[1]

[1] Department of Computer Science, University of Illinois, Urbana, IL 61801, USA
{tasharo1,rkumar8,marinov,agha}@illinois.edu
[2] Salisbury University, Salisbury, MD 21801, USA
stlauterburg@salisbury.edu
[3] INRIA, Campus de Beaulieu, France
alegay@irisa.fr

Abstract. To detect hard-to-find concurrency bugs, testing tools try to systematically explore all possible interleavings of the transitions in a concurrent program. Unfortunately, because of the nondeterminism in concurrent programs, exhaustively exploring all interleavings is time-consuming and often computationally intractable. Speeding up such tools requires pruning the state space explored. Partial-order reduction (POR) techniques can substantially prune the number of explored interleavings. These techniques require defining a *dependency relation* on transitions in the program, and exploit independency among certain transitions to prune the state space.

We observe that actor systems, a prevalent class of programs where computation entities communicate by exchanging messages, exhibit a dependency relation among co-enabled transitions with an interesting property: *transitivity*. This paper introduces a novel dynamic POR technique, TransDPOR, that exploits the transitivity of the dependency relation in actor systems. Empirical results show that leveraging transitivity speeds up exploration by up to two orders of magnitude compared to existing POR techniques.

1 Introduction

Concurrent programs are becoming increasingly important as multicore and networked computing systems become the norm. A model of concurrent programming that has been gaining popularity is the *actor model* [1]. The actor model is used in many systems such as ActorFoundry, Asynchronous Agents, Charm++, E, Erlang, and Scala.[1] Actor programs consist of computing entities called actors (each with its own local state and thread of control) that communicate by exchanging messages asynchronously. An actor *configuration* consists of the local state of the actors and a set of *pending messages*. In response to receiving

[1] For a more extensive list of actor systems, refer to [14].

a message, an actor can update its local state, send messages, or create new actors. At each step in the computation of an actor system [2], an actor from the system is scheduled to process one of its pending messages. Assuming that this processing terminates, the actor system transitions to a new configuration.

Concurrent systems, such as actor systems, present a significant challenge for the testing and verification community. Such systems can exhibit exponentially many different interleavings of concurrent transitions. In the case of actors, the execution of an actor program can have different results from an exponentially large number of potential interleavings of messages. The nondeterminism in actor systems stems from the fact that multiple messages sent to the same actor may be processed in different orders, thus resulting in different configurations, and only some specific interleavings/configurations may reveal bugs.

A naïve exploration that would explore all the interleavings to reach all possible system configurations does not scale. Partial-order reduction (POR) techniques [5,6,8,9,11,20,22,24–27,29] can be applied to help mitigate the resulting state-space explosion by exploring a representative subset of all possible interleavings. POR techniques have been widely used for testing and verification of concurrent protocols and software, including in tools such as SPIN [13], VeriSoft [8], and Java PathFinder [28].

To prune state-space exploration, POR techniques explore a *subset* of the set of enabled transitions in each configuration. This subset should be selected such that by exploring only the transitions in the subset, all the properties of interest are guaranteed to be preserved. For example, in one of the popular POR techniques, this subset is defined as a *persistent set* [9]. POR techniques require the definition of a *dependency relation* between transitions in the system and then exploit the independency between certain transitions to compute this subset. A valid dependency relation is a reflexive and symmetric (but *not necessarily transitive*) binary relation on the transitions.

Traditionally, dependencies among transitions, such as in *persistent sets* [9] proposed by Godefroid, were computed via static analysis. More recently, Flanagan and Godefroid introduced a POR algorithm, called dynamic POR (DPOR), that relies on dynamic analysis for computing dependencies [6]. More precisely, this algorithm maintains for each configuration a *backtrack* set and updates the backtrack sets during the execution of a test program. Flanagan and Godefroid proved that the computed backtrack sets are persistent sets [6]. They show that DPOR can significantly improve on POR techniques based on static analysis by computing smaller persistent sets. Note that DPOR is *stateless*, i.e., it does not store states/configurations across different executions.

In this paper, we leverage the fact that actors do not share their states, and we define a dependency relation between the transitions that is *transitive* on the transitions enabled in the same configuration (called co-enabled transitions). We present a new stateless dynamic POR algorithm, called *TransDPOR* which extends DPOR to take advantage of the transitive dependency relations in actor systems. We show that TransDPOR in some cases explores fewer configurations/transitions than DPOR, but it never explores more. TransDPOR is complete like DPOR, i.e., when the state space is acyclic, TransDPOR can reach every deadlock or local safety violation in the system (space limitations do not allow us to provide a proof of these properties in this version of the paper).

We implemented TransDPOR in Basset [17], a tool for the systematic testing of actor programs written in the Scala programming language [12] or the ActorFoundry library for Java [21]. TransDPOR code is publicly available with Basset at `http://mir.cs.illinois.edu/basset`. We compare TransDPOR and DPOR (we previously adapted the original DPOR algorithm to work for actor systems [18]) on eight programs without bugs and three programs with bugs. The experimental results show that TransDPOR reduces the number of transitions executed during state space exploration by $2.39x$ on *average* and *up to* $163.80x$ over DPOR. When we combine TransDPOR and DPOR with sleep sets (a traditional POR technique) [7], we find that TransDPOR can find bugs *up to* $2.56x$ faster than DPOR.

2 Illustrative Example

To illustrate how TransDPOR works, we use the simple actor program shown in Figure 1. It has four actors: one *master* (which is the entry point of the program), one *registry server*, and two *workers*. The registry keeps track of the actors registered in the system. The master first registers itself by sending its ID to the registry. It then creates two workers and sends them each a message with the registry's ID. After receiving the message, each worker sends its ID to the registry. In the comments for *send* statements, we labeled each of the five messages: $worker1$ and $worker2$ receive messages w_1 and w_2 respectively, and the registry eventually receives three messages—r_0, r_1, and r_2. In this example, the nondeterminism is the order in which the registry receives these three messages and thus assigns the values for its three local variables. For example, the program could have a bug if it assumes that r_0 is received before r_1 and r_2.

We observe that without any assumption one would have to explore up to 5! permutations of the messages exchanged between actors. We will see that this number reduces considerably by using DPOR algorithms that consider a basic property in the actor model. In the actor model, actors have no shared states but only communicate by exchanging messages. Since processing a message in one actor cannot change the states of other actors, only the transitions that process the messages sent to the same actor are *dependent*. Hence, when exploring actor systems, to reach all local safety violations and deadlocks, it suffices to explore different interleavings of processing messages in each actor, i.e., it is not necessary to explore interleavings of processing messages across different actors. For example, if m_a and m_b respectively stand for processing message m_a in actor a and message m_b in actor b, it suffices to explore only one of interleavings $m_a.m_b$ or $m_b.m_a$.

Figure 1 shows the state spaces that DPOR and TransDPOR explore for our example program. Each node represents a configuration, and each edge shows a transition labeled with the message being processed.

```
master:
1:registry:=create(Registry);
2:send(registry,id);/*r0*/
3:worker1:=create(Worker);
4:worker2:=create(Worker);
5:send(worker1,registry);/*w1*/
6:send(worker2,registry);/*w2*/

registry:
1:id0:=receive();
2:id1:=receive();
3:id2:=receive();

worker1:
1:r:=receive();
2:send(r,id);/*r1*/

worker2:
1:r:=receive();
2:send(r,id);/*r2*/
```

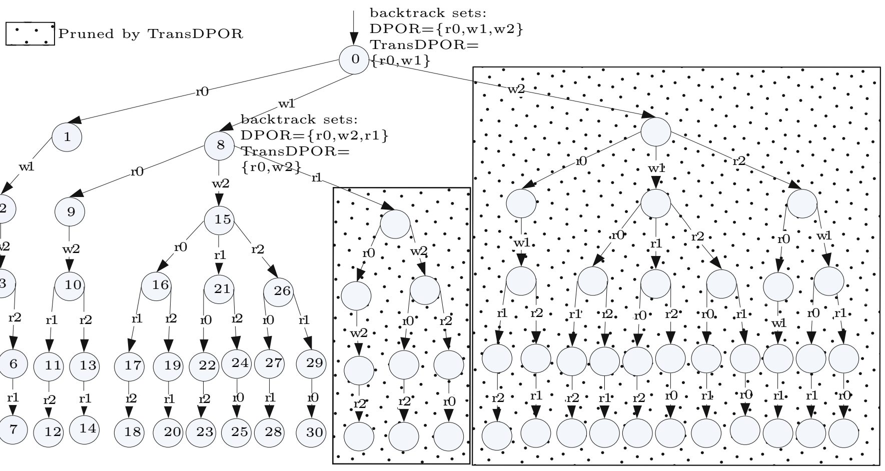


Fig. 1. The registry example and the state space explored by DPOR and TransDPOR

Let us first focus on DPOR. Specifically, this algorithm first executes an actor program to obtain an execution path and for each configuration keeps a *backtrack* set of all messages to be explored from that configuration. These sets start empty but grow as DPOR discovers dependencies among transitions. In our example, DPOR first executes the path $r_0.w_1.w_2.r_1.r_2$. Then, the algorithm observes that r_1 and r_2 are sent to the same *registry* actor, which makes them dependent. DPOR thus adds r_2 to the backtrack set of configuration 3. Moreover, because r_1 and r_2 are dependent with r_0 but not enabled in the initial configuration, DPOR adds the messages that can produce r_1 and r_2—namely, w_1 and w_2—to the backtrack set of the configuration 0 so that different interleavings of those messages with r_0 can be explored. After the first path, DPOR backtracks in a depth-first manner to configuration 3 and executes $r_0.w_1.w_2.r_2.r_1$. It then backtracks to configuration 0 and explores w_1 and w_2 from that configuration. Observe that some redundant paths such as $r_0.w_2.w_1.r_2.r_1$ have been removed from the exploration. In the end, DPOR explores 24 paths. Recall that DPOR is stateless, i.e., it does not store the history of previous explored paths and every time it backtracks to a configuration it chooses a message from backtrack set and runs the program nondeterministically.

While the above pruning is already an improvement over full exploration, it does not fully exploit the semantics of the actor model. More precisely, one can observe that adding w_1 to the backtrack set of the initial configuration would be enough to explore all possible permutations of the messages processed in each actor, i.e., all paths of the subtree that starts with w_2 are redundant. Intuitively this is because only *registry* can receive more than one message and different permutations of the three messages sent to the registry have been explored in the previous paths. The same holds for the backtrack set of configuration 8 where the subtree that starts with r_1 is redundant.

Our new POR algorithm TransDPOR detects these redundant paths and as a result explores only 10 paths in this example (those *not* included in dotted boxes in Figure 1). The main idea in TransDPOR is to add (at most) one new message to the backtrack set for a configuration. After the newly added message is explored, *only if it is necessary* TransDPOR adds more messages to the backtrack set. TransDPOR implements this idea by attaching a boolean flag, *freeze* flag, to each configuration. It only adds a message to the backtrack set of a particular configuration if the *freeze* flag of that configuration is not set. While initially this flag is not set in any configuration, TransDPOR sets this flag when it adds a message to the backtrack set of a configuration. It resets the flag when it backtracks to a configuration and explores a new message from that configuration.

In the example, when TransDPOR adds w_1 to the backtrack set of configuration 0, it sets the *freeze* flag and that prevents the addition of w_2 to the backtrack set of configuration 0. The same situation happens for the backtrack set of configuration 8. That leads to smaller backtrack sets than DPOR for the two configurations 0 and 8. This reduction is allowed due to the *transitivity* of the dependency relation between the transitions that may be co-enabled in a configuration, and we show that this reduction does not miss any bug that DPOR can find. An example of adding all messages can be seen in configuration 15 (all three messages r_0, r_1, and r_2 are added to the backtrack set of the configuration). In this configuration, after exploring r_0, both r_1 and r_2 are dependent with r_0, but TransDPOR only adds r_1 to the backtrack set of configuration 15. The *freeze* flag prevents the addition of r_2 to the backtrack set at the same time. Due to the transitivity of the dependency relation, r_1 and r_2 are also dependent. Thus, after exploring r_1 from configuration 15, the *freeze* flag is reset and r_2 is added to the backtrack set of configuration 15. The algorithm will end up adding all three messages r_0, r_1, and r_2 to the backtrack set of configuration 15 and will not miss any permutation of these three messages.

3 Actor Semantics

While the above example relied on an intuitive understanding of actors, we now define the semantics of actor programs precisely. Formally, we view actor programs as state-transition systems. A (global) *state* of an actor program, termed a *configuration*, in notation $\kappa = \langle \alpha, \mu \rangle$, consists of a map $\alpha : \mathbb{A} \to \mathbb{L}$, where $\mathbb{A}$ are actor identifiers and $\mathbb{L}$ are possible local states, and a set of *pending* messages $\mu \subseteq \mathbb{M}$, where $\mathbb{M}$ is the set of all possible messages in the system. We use $\mathbb{K}$ to denote the set of all configurations in a system and *pending*(κ) to denote the set of pending messages for $\kappa \in \mathbb{K}$. Each message is a tuple of *receiver actor*, *content*, and *unique message identifier*. Conceptually, the messages in μ can be partitioned according to their receiver actor, i.e., μ is a union of disjoint message sets, one for every actor in the system.

At each step in execution, an actor processes a message from its message set: the actor removes the message from its set and potentially updates its local state, sends messages to other actors or itself, and creates new actors. The processing can be viewed as a single, atomic macro-step [2] because actors do not share state [14]. The actor model allows *constraints* that enable or disable processing of some message by an actor depending on its local state. Formally, for an actor a, its constraint $c_a \subseteq \mathbb{L} \times \mathbb{M}$ is a predicate on the local state of the actor and the set of messages.

Definition 1. *The* transition t_m *for a message* m *is a partial function* $t_m : \mathbb{K} \rightharpoonup \mathbb{K}$*. For a given* $\langle \alpha, \mu \rangle \in \mathbb{K}$*, let the receiver of* m *be actor* a *with the local state* s *and constraint* c_a*; the transition* t_m *is enabled if* $t_m(\langle \alpha, \mu \rangle)$ *is defined (i.e.,* $\alpha(a) = s$ *and* $m \in \mu$*) and* $\langle s, m \rangle \in c_a$*. If* t_m *is enabled, it can be executed and produces a new configuration, updating the local state of the actor from* s *to* s'*, sending messages* $out_s(t_m)$*, and creating new actors with their initial local state* $new_s(t_m)$*:*

$$\langle \alpha, \mu \rangle \xrightarrow{t_m} \langle \alpha[a \mapsto s'] \cup new_s(t_m), \mu \backslash \{m\} \cup out_s(t_m) \rangle$$

We denote $msg(t_m) = m$ and $actor(t_m) = a$. We denote with $out(t_m)$ and $new(t_m)$ the sets of all new messages and actors, respectively, that the transition t_m can create for *any* local state s. Observe that as is usual in actor semantics, we assume that the behavior of an actor in response to a message is *deterministic.* Moreover, we assume that all transitions terminate–this is a standard assumption in testing programs. Thus, the execution of a transition t in a configuration κ leads to a unique successor κ' (up to the choice of fresh identifiers for new actors and messages).

4 Definitions for Partial-Order Reduction

We introduce several terms and definitions required for presenting our TransDPOR algorithm, following the DPOR presentation style of Flanagan and Godefroid [6]. Then we present an important property of the actor model that will be used to improve DPOR. Let τ be the set of all transitions in the system and τ^* be the set of all transition sequences. We write $\kappa \overset{\omega}{\Rightarrow} \kappa'$ to denote that the execution of finite sequence $\omega \in \tau^*$ leads from κ to κ'. A configuration in which no transition is enabled is called a *deadlock* or *terminating* configuration. A *transition sequence* S of an actor system is a (finite) sequence of transitions $t_1.t_2 \ldots t_n$ where there exist configurations $\kappa_0, \ldots, \kappa_n$ such that κ_0 is the initial configuration and $\kappa_0 \xrightarrow{t_1} \kappa_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} \kappa_n$. A transition sequence that ends in a deadlock or terminating configuration is called an *execution path* of the system.

We define an actor system as a transition system $A_G = \langle \mathbb{K}, \Delta, \kappa_0 \rangle$, where $\Delta = \{\langle \kappa, \kappa' \rangle \mid \exists t \in \tau : \kappa \xrightarrow{t} \kappa'\}$ and κ_0 is the initial configuration. We first recap the general definition for a valid dependency relation between transitions [6], then we show how to adapt it to the actor model.

Definition 2. *Let t_1 and t_2 be two transitions of an actor system. We say that t_1 and t_2 are* independent *if for all configurations κ in the state space A_G of the system:*

- *if t_1 is enabled in κ and $\kappa \xrightarrow{t_1} \kappa'$, then t_2 is enabled in κ iff t_2 is enabled in κ' (i.e., independent transitions cannot disable or enable each other); and*
- *if t_1 and t_2 are enabled in κ, there is a unique configuration κ' such that $\kappa \overset{t_1,t_2}{\Rightarrow} \kappa'$ and $\kappa \overset{t_2,t_1}{\Rightarrow} \kappa'$ (i.e., enabled independent transitions must commute).*

The reflexive and symmetric binary relation D is a valid dependency relation on τ iff $D = \{(t_1, t_2) | t_1, t_2 \text{are not independent transitions}\}$. The pair of transitions (t_1, t_2) are said to be dependent iff they belong to a valid dependency relation.

We observe that for actor programs, a transition t_m cannot be enabled until the receiver actor for m is created and the message m is sent (m becomes pending). Second, once a message m is sent to an actor a, only transitions of the actor a can enable or disable the transition t_m that processes the message m. In other words, the constraint c_a does not depend on the global state but only on the local state of a and the message m. Therefore, we can easily show that two transitions t_1 and t_2 are independent if $actor(t_1) \neq actor(t_2)$, $msg(t_1) \notin out(t_2)$, and $actor(t_1) \notin new(t_2)$. Based on these observations, we can cast Definition 2 in the actor programs setting to obtain the following proposition.

Proposition 1. *Two transitions $t_1, t_2 \in \tau$ are* dependent *iff one of the following conditions holds:*

- *$actor(t_1) = actor(t_2)$; or*
- *$msg(t_1) \in out(t_2)$ or $msg(t_2) \in out(t_1)$; or*
- *$actor(t_1) \in new(t_2)$ or $actor(t_2) \in new(t_1)$.*

Based on Proposition 1, one can extract an important property of our model, which will be used to improve over DPOR. We say that two transitions t_1 and t_2 *may be co-enabled* if there may exist some configuration in which both t_1 and t_2 are enabled. For a shorthand, we introduce a binary relation on transitions called *race relation*; we say that a pair of transitions $\langle t_1, t_2 \rangle$ are *in race* if they are dependent and may be co-enabled. A key observation is that if $\langle t_1, t_2 \rangle$ are in race, then $actor(t_1) = actor(t_2)$. Indeed, while our definition of dependency allows two other cases ($msg(t_1) \in out(t_2)$ or $actor(t_1) \in new(t_2)$), the transitions that satisfy those two other cases can never be co-enabled (because those cases require that the message or actor for t_1 be created after the execution of t_2). As a result, the following proposition holds.

Proposition 2. *The race relation is reflexive, symmetric, and transitive.*

Given a transition sequence, two adjacent transitions that are independent can be permuted without changing the behavior of the transition sequence. To formalize the set of equivalent transition sequences, we recap the happens-before relation presented in [6].

Definition 3. *The* happens-before *relation $\rightarrow_S$ for a transition sequence $S = t_1 \ldots t_n$ is the smallest relation on $\{1, \ldots, n\}$ such that (1) if $i \leq j$ and t_i is dependent with t_j, then $i \rightarrow_S j$; and (2) $\rightarrow_S$ is transitively closed.*

Since happens-before relation is a partial order [6], we introduce the following equivalence relation:

Definition 4. *Two transition sequences S_1 and S_2 are equivalent iff they have the same set of transitions, and they are linearizations of the same happens-before relation.*

We use $[S]$ to denote the set of transition sequences that are equivalent to S.

5 TransDPOR: A New DPOR Algorithm

Figure 2 presents our TransDPOR algorithm, which explores the state space of an actor system dynamically in a depth-first manner. The underlined parts are the differences between TransDPOR and the original DPOR [6] adapted for actors. The input to the algorithm is a transition sequence S (Line 1). Notation-wise, for a sequence $S = t_1 \ldots t_n$: $dom(S)$ is the set $\{1, \ldots, n\}$; S_i for $i \in dom(S)$ is transition t_i; $pre(S, i)$ for $i \in dom(S)$ is the configuration in which t_i is executed; and $last(S)$ is the configuration reached after executing S. We denote with $next(\kappa, m)$ the transition that processes message m in the configuration κ. Following [6], we also use a variant of the happens-before relation to determine if some messages are sent as the result of executing other transitions:

Definition 5. *In a transition sequence S, the relation $i \rightarrow_S m$ holds for $i \in dom(S)$ and message m iff either (1) $m \in out(S_i)$ or (2) $\exists j \in dom(S)$ such that $i \rightarrow_S j$ and $m \in out(S_j)$.*

Like DPOR, TransDPOR maintains a backtrack set $backtrack(\kappa)$, which keeps the messages to be explored from each configuration κ in the input sequence S. The main difference is that, in addition, TransDPOR also uses a boolean flag $freeze(\kappa)$. As explained in Section 2, this flag can prevent adding some messages to $backtrack(\kappa)$, and hence it reduces the size of $backtrack(\kappa)$. As we shall see, because of the transitivity of the race relation, TransDPOR can use this flag to improve over DPOR.

TransDPOR starts by finding the current configuration κ for the input sequence S (Line 2). For every message m in $pending(\kappa)$ (Line 3), it considers the transition $next(\kappa, m)$ for processing that message. It finds the *last* transition i in the sequence S which is in the race with $next(\kappa, m)$, i.e., $actor(S_i) = actor(next(\kappa, m))$ and $i \not\rightarrow_S m$. If the *freeze* flag is set in $pre(S, i)$, the algorithm does not update the backtrack set for $pre(S, i)$ (Line 4′); this line does not exist in DPOR and is a major difference between TransDPOR and DPOR. Effectively, this step prevents additional messages in $backtrack(pre(S, i))$ until the previously added message is explored by the algorithm. Due to the transitivity of our race relation, we prove that the messages not added right away are added later if

```
0 : Initially: $Explore(\emptyset)$;

1 : $Explore(S)$ {
2 :     let $\kappa = last(S)$;
3 :     for all messages $m \in pending(\kappa)$ {
4 :         if $\exists i = max(\{i \in dom(S) \mid S_i$ is dependent and
                                   may be co-enabled with $next(\kappa, m)$ and $i \not\to_S m\})$ {
$\underline{4'}$ :             $\underline{\text{if } (\neg freeze(pre(S,i)))}$ {
5 :                 let $E = \{m' \in enabled(pre(S,i)) \mid m' = m$ or $\exists j \in dom(S) \mid j > i$ and
                                         $m' = msg(S_j)$ and $j \to_S m$ $\underline{\text{and}}$
                                         $\underline{j = min(\{j \in dom(S) \mid j > i \text{ and } j \to_S m\})}\}$;
6 :                 if $(E \setminus \underline{backtrack(pre(S,i))} \neq \phi)$ {
                      add any $m' \in E$ to $backtrack(pre(S,i))$;
                      $\underline{freeze(pre(S,i)) := true}$;
                    }
7 :                 $\underline{/*}$ else add all $m$ in $enabled(pre(S,i))$ to $backtrack(pre(S,i))$ $\underline{*/}$;
$\underline{7'}$ :             $\underline{\}}$
8 :         }
9 :     }
10 :    if $(\exists m \in enabled(\kappa))$ {
11 :        $backtrack(\kappa) := \{m\}$;
12 :        let $done = \emptyset$;
13 :        while $(\exists m \in (backtrack(\kappa) \setminus done))$ {
14 :            add $m$ to $done$;
$\underline{14'}$ :            $\underline{freeze(\kappa) := false}$;
15 :            $Explore(S.next(\kappa, m))$;
16 :        }
17 :    }
18 : }
```

Fig. 2. The TransDPOR algorithm (The differences with DPOR are underlined)

necessary to explore them from κ. However, as TransDPOR does not add them right away, it may terminate faster than DPOR. If *freeze*$(pre(S,i))$ is not set, the algorithm next finds the message that should be added to *backtrack*$(pre(S,i))$ by computing the set E (Line 5) from the messages whose transitions are enabled in $pre(S,i)$. If m is enabled in $pre(S,i)$ it is added to E $(m' = m)$; otherwise a message m' is added to E if its transition is the *first* transition after S_i that happens before m (in this case, m is produced as a result of executing other transitions after S_i). Note that our approach for computing E differs from DPOR in that it finds the *minimum* index $j > i$ such that j happens before m, while DPOR finds *all* $j > i$ such that j happens before m. As a result of this change, E in our case has at most one element. After computing E, if it contains a message that is not already in *backtrack*$(pre(S,i))$, the message is added to *backtrack*$(pre(S,i))$, and the *freeze* flag is set (Line 6). In DPOR, if E is not empty, it can have more than one message, and the algorithm nondeterministically chooses one message to add to *backtrack*$(pre(S,i))$ (hence "add any").

If E is empty, then m is in *pending*$(pre(S,i))$ but t_m is not enabled in $pre(S,i)$. Intuitively, because of the transitivity of race relation, every enabled message in $pre(S,i)$ that can enable t_m would be in race with S_i (all of them belong to the same actor) and would be added to *backtrack*$(pre(S,i))$ either in the next iteration of the for loop or in the recursive calls to $Explore$. Therefore, TransDPOR does not add anything to *backtrack*$(pre(S,i))$ at this point (Line 7 is effectively deleted). In contrast, in DPOR, if E is empty, the algorithm adds *all* messages from E to $backtrack(pre(S, i))$.

After Lines 3-9 update the backtrack set of configurations seen previously in the sequence S, Lines 10-17 process the messages from the current configuration κ. The algorithm nondeterministically chooses an enabled message from κ (Line 10) to initialize $backtrack(\kappa)$ (Line 11). It then processes all messages from the backtrack set that have not been explored before (Line 13). Every time the algorithm backtracks to κ and explores a new message, it adds that message to the *done* set and resets the *freeze* flag (Line 14′). The algorithm finally recursively calls itself with the transition sequence extended with the $next(\kappa, m)$.

In our example in Section 2, once TransDPOR adds w_1 to $backtrack(\kappa_0)$, it sets $freeze(\kappa_0)$, which prevents from adding w_2 to $backtrack(\kappa_0)$. After the algorithm explores w_1 from κ_0, it resets $freeze(\kappa_0)$, but because none of the messages in paths that start with w_1 from κ_0 is in the race with w_1, no message is added to the $backtrack(\kappa_0)$ (w_2 is not added). Similarly, in κ_8, adding w_2 to $backtrack(\kappa_8)$ prevents from adding r_1. After exploring w_2 from κ_8, because none of the messages r_0, r_1, and r_2 are in the race with w_2, no message is added to $backtrack(\kappa_8)$ (r_1 is not added). On the other hand, consider κ_{15}. After exploring r_0, both r_1 and r_2 are in the race with r_0. Because of the *freeze* flag, the algorithm only adds r_1 to $backtrack(\kappa_{15})$. After exploring r_1 from κ_{15}, because of *transitivity* of race relation, r_2 is also in the race with r_1 and it is eventually added to $backtrack(\kappa_{15})$.

When the loop terminates, the exploration from κ is finished, and the algorithm backtracks to the previous configuration. Note that the algorithm is stateless, i.e., it does not store states/configurations across different execution paths. However, it may store configurations for the current path on a stack, depending on the implementation strategy.

It is trivial to show that TransDPOR never explores more execution paths than DPOR. As a result of the changes that we have made in Lines 4′ and 7 of the algorithm, in each call to $Explore(S)$, we add either fewer or the same number of messages to the backtrack set of each configuration κ.

Theorem 1 states that by starting from a fix initial configuration if A_G is acyclic, TransDPOR will explore at least one execution path from each set of equivalent execution paths in A_G, i.e., it can detect any *deadlock* and *local safety violation* in the program [8].

Theorem 1. *In a program $\mathcal{P}$, by starting from an initial configuration, let A_G be the* acyclic *state-space graph and A_R be the reduced state space explored by TransDPOR. If Ω_G and Ω_R denote the set of execution paths of $\mathcal{P}$ in A_G and A_R respectively, then $\forall \omega \in \Omega_G, \exists \omega' \in \Omega_R$ such that $\omega' \in [\omega]$.*

6 Implementation and Evaluation

To compare TransDPOR and DPOR, we implemented TransDPOR in the Basset tool [17]. Basset provides an extensible environment for testing Java-based actor programs written in the Scala Actors library [12] or ActorFoundry [21]. We use vector clocks [16] to track the happens-before relation at runtime as shown in [24].

We use eight different subject programs in our evaluation. Each actor program was either originally implemented in ActorFoundry or ported to it for this evaluation. `fibonacci` computes the n^{th} element in the Fibonacci sequence. In this case, we show the result for $n = 5$. `quicksort` is a distributed sorting implementation using a standard divide-and-conquer strategy to carry out the computation. `pi` is a porting of a publicly available [23] MPI example, which computes an approximation of π by distributing the task among a set of worker actors. The results shown here are for a configuration with five worker actors. `pipesort` is a modified version of the sorting algorithm used in the dCUTE study [24]. `chameneos` is an implementation of the `chameneos-redux` benchmark from the Great Language Shootout (`http://shootout.alioth.debian.org`). `leader` is an implementation of a leader election algorithm previously used in the dCUTE study [24]. `shortpath` is an implementation of the Chandy-Misra shortest path algorithm [4]. This subject appears twice in the results: once for a graph with 4 nodes (`shortpath4`) and once for a graph with 5 nodes (`shortpath5`), where the two graphs are dissimilar. `regsim` is a server registration simulation. The numbers with the name of subjects, if available, represent the values of program parameters. We performed all experiments using Sun's JVM 1.6.0_20-b02 on a 2.93GHz Intel Core(TM)i7 running Ubuntu release 10.04.

Our recent work [18] shows that the effectiveness of DPOR techniques is highly sensitive to the *order* in which messages are explored. However, one cannot easily determine before the exploration which order will work the best. For that reason, we present results for three ordering heuristics, *ECA*, *LCA*, and *FIFO*. FIFO sorts the messages based on the time they are sent in the ascending order. ECA

Table 1. Comparison of TransDPOR and DPOR

		DPOR				TransDPOR				Speedup			
Heur.	Subject	# of Paths	# of Trans	time [sec]	mem [MB]	# of Paths	# of Trans	time [sec]	mem [MB]	# of Paths	# of Trans	time [sec]	mem [MB]
FIFO	fib5	40	203	5	176	40	203	4	176	1.00x	1.00x	1.25x	1.00x
ECA		327	1650	28	455	203	1051	18	377	1.61x	1.57x	1.56x	1.21x
LCA		16	91	3	173	16	91	3	159	1.00x	1.00x	1.00x	1.09x
FIFO	quicksort6	368	1586	26	343	368	1586	26	463	1.00x	1.00x	1.00x	0.74x
ECA		3822	16766	264	381	1519	6992	115	751	2.52x	2.40x	2.30x	0.51x
LCA		32	156	4	197	32	156	4	179	1.00x	1.00x	1.00x	1.10x
FIFO	pi5	120	931	16	265	120	931	16	374	1.00x	1.00x	1.00x	0.71x
ECA		120	931	16	263	120	931	16	374	1.00x	1.00x	1.00x	0.70x
LCA		19845	156070	2509	451	312	2452	40	376	63.61x	63.65x	62.73x	1.20x
FIFO	pipesort4	1791	8562	101	375	755	3541	45	375	2.37x	2.42x	2.24x	1.00x
ECA		288	1293	17	374	288	1293	18	450	1.00x	1.00x	0.94x	0.83x
LCA		5970	32385	361	375	2221	11999	136	451	2.69x	2.70x	2.65x	0.83x
FIFO	chameneos2	3240	19459	233	376	600	3673	44	376	5.40x	5.30x	5.30x	1.00x
ECA		19683	118197	1360	550	1728	10554	123	374	11.39x	11.20x	11.06x	1.47x
LCA		216	1231	16	375	216	1231	16	453	1.00x	1.00x	1.00x	0.83x
FIFO	leader4	18098	107780	26872	336	14984	86889	37516	341	1.21x	1.24x	0.72x	0.99x
ECA		11957	68373	1207	240	11909	68125	1312	266	1.00x	1.00x	0.92x	0.90x
LCA		39238	236330	4301	634	27287	163030	3120	525	1.44x	1.45x	1.38x	1.21x
FIFO	shortpath4	238	910	12	261	238	910	12	261	1.00x	1.00x	1.00x	1.00x
ECA		392	1464	20	262	392	1464	19	260x	1.00x	1.00x	1.05x	1.01x
LCA		640	2158	27	260	370	1337	17	264	1.73x	1.61x	1.59x	0.98x
FIFO	shortpath5	528	2443	32	454	528	2443	33	372	1.00x	1.00x	0.97x	1.22x
ECA		2658	8476	104	368	1170	3737	49	261	2.27x	2.27x	2.12x	1.41x
LCA		1865	7076	93	375	1272	4704	61	375	1.47x	1.50x	1.52x	1.00x
FIFO	regsim	211750	590835	14440	989	1320	3607	64	76	160.42x	163.80x	225.63x	13.01x
ECA		208034	591454	14782	989	1950	5434	93	79	106.68x	108.84x	158.95x	12.52x
LCA		720	1962	34	64	720	1962	36	66	1.00x	1.00x	0.94x	0.97x
									Max	160.42x	**163.80x**	225.63x	13.01x
									Average	2.39x	**2.39x**	2.38x	1.18x

sorts messages according to the creation time of the receiving actor in ascending order; messages for the *earliest created actor* are considered first. LCA is similar to ECA but sorts the actors in descending order of their creation time.

To illustrate the speedup that can be achieved using TransDPOR, we performed a set of nine experiments which compare explorations performed using DPOR with explorations performed using TransDPOR. Table 1 shows the results for these experiments. For each subject and DPOR technique, we show the number of paths executed in their entirety while exploring the specified subjects, the total number of transitions executed (across all execution paths), the total exploration time in seconds, and memory usage in MB. Since the length of paths might be different in a program, and the time is dependent on the platform and noise in the system, we focus on the number of explored transitions as the primary metric for comparison.

The experiments suggest that TransDPOR can explore up to *over two orders of magnitude* fewer transitions than DPOR. In all the experiments TransDPOR has a speedup for at least one heuristic, and it is *never* the case that the use of TransDPOR results in more executed transitions than DPOR. Although the speedup in transitions executed can at times be small (e.g., only $1.24x$ or less for `leader`), it can be also quite significant. For `chameneos`, the speedup is over $11x$, and for `regsim`, it is over $163x$. The `regsim` experiment using DPOR did not complete in 4 hours for either FIFO or ECA.

***Combining with Sleep Sets*:** Sleep sets is a POR technique based on the history of exploration [7]. Specifically, sleep sets record the transitions that have already been explored from a particular configuration, and avoid exploring them in successor configurations until some condition is met. Sleep sets can further prune the number of transitions and configurations that are explored [8]. In the case where the state space is acyclic (which is the assumption in this paper), sleep sets can be combined with dynamic POR in exactly the same way as static POR [6]. We implemented a variant of TransDPOR that is combined with sleep sets and compared it with the combination of DPOR with sleep sets.

In addition to the eight programs used in our initial experiment, we added three more programs. These programs have such a large state space that the exploration times out without sleep sets. `diningphil` is an implementation of the dining philosopher protocol in ActorFoundry. `minesweeper` is a simulation of the minesweeper game written using the Scala Actors library. `le-erlang` is an implementation of a fault-tolerant leader election algorithm for Erlang that had been running on Ericsson switches. Some bugs were found in the program by Arts et al. [3] in in the presence of node failures. We re-implemented the buggy program in ActorFoundry in order to test it using our tool.

The results are presented in Table 2. For `le-erlang`, our tool was able to find all the previously known bugs in the algorithm (in the presence of node failures). We also tested the algorithm for four processes and *without a failure-recovery scenario*. To our surprise, our tool detected a *new bug*, which allows the program to reach a state in which no leader is elected. We contacted the developers and they confirmed the new bug.

Table 2. Comparison of TransDPOR+Sleep sets and DPOR+Sleep sets

		DPOR+ S				TransDPOR+ S				Speedup			
Heur.	Subject	# of Paths	# of Trans	time [sec]	mem [MB]	# of Paths	# of Trans	time [sec]	mem [MB]	# of Paths	# of Trans	time [sec]	mem [MB]
FIFO	fib5	16	101	3	173	16	101	3	173	1.00x	1.00x	1.00x	1.00x
ECA		16	139	4	159	16	139	4	173	1.00x	1.00x	1.00x	0.92x
LCA		16	91	3	174	16	91	3	174	1.00x	1.00x	1.00x	1.00x
FIFO	quicksort6	32	179	5	181	32	179	5	181	1.00x	1.00x	1.00x	1.00x
ECA		32	272	7	269	32	272	7	270.00x	1.00x	1.00x	1.00x	1.00x
LCA		32	156	5	194	32	156	5	193	1.00x	1.00x	1.00x	1.01x
FIFO	pi5	120	931	17	264	120	931	17	376	1.00x	1.00x	1.00x	0.70x
ECA		120	931	17	266	120	931	17.00	263.00	1.00x	1.00x	1.00x	1.01x
LCA		120	1236	22	377	120	990	18	456	1.00x	1.25x	1.22x	0.83x
FIFO	pipesort4	288	1448	20	378	288	1422	20	377	1.00x	1.02x	1.00x	1.00x
ECA		288	1293	19	376	288	1293	18	376	1.00x	1.00x	1.06x	1.00x
LCA		288	1944	27	376	288	1935	27	376	1.00x	1.00x	1.00x	1.00x
FIFO	chameneos2	216	1681	23	378	216	1453	20	377	1.00x	1.16x	1.15x	1.00x
ECA		216	1826	24	374	216	1530	21	376	1.00x	1.19x	1.14x	0.99x
LCA		216	1231	17	376	216	1231	17	375	1.00x	1.00x	1.00x	1.00x
FIFO	leader4	492	3125	43	454	492	3097	43	372	1.00x	1.01x	1.00x	1.22x
ECA		492	3267	45	376	492	3218	42	377	1.00x	1.02x	1.07x	1.00x
LCA		492	3311	46	680	492	3311	46	377	1.00x	1.00x	1.00x	1.80x
FIFO	shortpath4	126	473	8	261	126	473	8	262	1.00x	1.00x	1.00x	1.00x
ECA		126	489	8	260	126	489	8	260	1.00x	1.00x	1.00x	1.00x
LCA		126	522	9	262	126	502	8	262	1.00x	1.04x	1.13x	1.00x
FIFO	shortpath5	296	1408	22	375	296	1408	22	375	1.00x	1.00x	1.00x	1.00x
ECA		296	1031	16	264	296	997	17	265	1.00x	1.03x	0.94x	1.00x
LCA		296	1228	20	376	296	1218	19	451	1.00x	1.01x	1.05x	0.83x
FIFO	regsim	720	3453	37	376	720	2019	22	377	1.00x	1.71x	1.68x	1.00x
ECA		720	4054	47	375	720	2152	26	452	1.00x	1.88x	1.81x	0.83x
LCA		720	1962	22	264	720	1962	22	375	1.00x	1.00x	1.00x	0.70x
FIFO	regsim-2-level	1296	8636	141	427	1296	5537	92	417	1.00x	1.56x	1.53x	1.02x
ECA		1296	14990	267	558	1296	7486	129	560	1.00x	2.00x	2.07x	1.00x
LCA		1296	6481	115	381	1296	6295	111	381	1.00x	1.03x	1.04x	1.00x
FIFO	diningphil	31	1375	38	524	31	1375	37	524	1.00x	1.00x	1.03x	1.00x
ECA		31	2082	55	348	31	1662	44	366	1.00x	1.25x	1.25x	0.95x
LCA		31	1333	38	374	29	1147	33	407	1.07x	1.16x	1.15x	0.92x
									Max	1.07x	**2.00x**	2.07x	1.80x
									Average	1.00x	**1.13x**	1.13x	0.98x
Buggy programs (Exploration stops at first bug instance.)													
FIFO	diningphil (deadlock)	1	15	2	112	1	15	2	112	1.00x	1.00x	1.00x	1.00x
ECA		16	915	29	340	16	767	22	342	1.00x	1.19x	1.32x	0.99x
LCA		1	15	2	112	1	15	2	112	1.00x	1.00x	1.00x	1.00x
FIFO	minesweeper (deadlock)	1	29	3	163	1	29	2	163	1.00x	1.00x	1.50x	1.00x
ECA		2710	15577	484	717	2710	15381	499	744	1.00x	1.01x	0.97x	0.96x
LCA		6993	69350	1950	1041	6993	55430	1651	893	1.00x	1.25x	1.18x	1.17x
FIFO	le-erlang3-failure (safety)	457	1976	41	462	452	1944	27	427	1.01x	1.02x	1.52x	1.08x
ECA		30	174	4	241	30	174	4	236	1.00x	1.00x	1.00x	1.02x
LCA		233738	759109	14401	2020	93640	296229	3557	1557	2.50x	2.56x	4.05x	1.30x
FIFO	le-erlang4 (no leader, new)	2209	11006	169	605	2191	10146	155	586	1.01x	1.08x	1.09x	1.03x
ECA		1	27	2	112	1	27	2	112	1.00x	1.00x	1.00x	1.00x
LCA		198713	698759	14405	2011	88505	277440	3924	1383	2.25x	2.52x	3.67x	1.45x
									Max	2.50x	**2.56x**	4.05x	1.45x
									Average	1.16x	**1.22x**	1.40x	1.08x

The combination with sleep sets reduces the improvement as sleep sets already prune many redundant transitions; however TransDPOR is always equal to or better than DPOR in terms of paths and transitions. For seven experiments, TransDPOR provides the speedup of over $1.20x$ for at least one heuristic. Note that it is not obvious from the program what may be a good heuristic, and the results table suggests the same. For example, ECA is the best heuristic for `le-erlang`, and FIFO is the best for `minesweeper`. Moreover, different components of a single application, such as the `regsim-2-level`, may have different good heuristics for exploration.

Overall, the results suggest that our algorithm combined with sleep sets outperforms the combination of DPOR and sleep sets. We achieved speedup as high as $2x$ for the `regsim` benchmark as shown in Table 2. TransDPOR is also very efficient when exploring programs with bugs. We consistently find the bug faster than DPOR. Even in the presence of sleep sets, we were able to find the bugs up to $2.56x$ faster than DPOR.

7 Related Work

One of the earliest POR approaches is based on computing persistent (or stubborn) sets [8,9,27] and the related technique of ample sets [22]. Persistent sets can be computed statically or dynamically. Using static analysis for computing persistent sets [8] suffers from conservative approximation, resulting in coarse persistent sets. Therefore, *dynamic POR* (DPOR) techniques [6], which compute the persistent sets on the fly, have been proposed to yield more accurate persistent sets. Another variation of persistent (or stubborn) is *weak persistent sets* [9,27], which can generate smaller sets and lead to better reduction. This reduction needs additional knowledge about the transitions that enable and disable each other, which may not be easy to compute during the exploration.

Sen and Agha proposed a DPOR technique for testing multi-threaded programs [25], as well as for testing distributed message-passing programs [24]. Both papers present an operational definition of the set of transitions to be explored from a state, and the presented algorithms are conceptually similar to that in [6]. Kastenberg and Rensink proposed a new DPOR which is based on *probe sets* for handling dynamic creation and destroying of processes and objects [15]. Probe sets relies on abstract enabling and disabling relations among actions, rather than associated sets of concurrent processes. The authors show that their technique leads to a better reduction in comparison to persistent sets.

Recently a new partial-order reduction technique called cartesian POR was proposed, which is based on *cartesian semantics* [11] and stateful exploration. The authors provide an operational definition, and present a dynamic algorithm that overcomes the acyclic state space restriction in stateless approaches. The technique is shown to improve over optimal persistent sets for some examples. The cartesian approach trades space for time since the approach requires storing program states precisely.

Lei and Carver [19] propose a technique that explores only one interleaving from each partial order. However their technique assumes FIFO channels and requires a non-trivial amount of memory for storing interleavings that are yet to be explored. Message Passing Interface (MPI) [10] is a popular environment for writing message-passing programs. MPI programs are more constrained than actor programs. Specifically, MPI processes assume FIFO channels and usually have matched sends and receives. Vakkalanka et al. [26] proposed a stateless DPOR technique for MPI programs, called POE, which exploits these constraints. POE can produce only one interleaving for large MPI programs that do not have an MPI wild-card receive.

8 Conclusions and Future Work

We have proposed a new stateless DPOR technique called TransDPOR for message-passing (actor) systems. We exploit the transitivity of a dependency relation between co-enabled transitions in actor systems to achieve faster exploration than the existing DPOR based on persistent sets. Experimental results suggest that TransDPOR can substantially reduce the number of transitions executed during state space exploration by up to 163.80x compared to DPOR, and it can detect bugs up to 2.56x faster than DPOR. TransDPOR code is available with Basset at `http://mir.cs.illinois.edu/basset`.

Although we applied our algorithm for message-passing systems, we believe that the technique discussed in this paper may be applicable to shared-memory multi-threaded programs if a dependency relation between the co-enabled transition is defined so that it is transitive. However, the detailed discussion in this regard is outside the scope of this paper. We also plan to explore the possibility of combining our algorithm with stateful exploration.

Acknowledgments. We would like to thank Yaniv Eytani, Bobak Hadidi, Vilas Jagannath, Timo Latvala, and P. Madhusudan for discussions and other assistance during the course of this project, and Nicholas Chen, Stas Negara, Milos Gligoric, Mohsen Vakilian, and the participants of the software engineering seminar at UIUC for comments on a draft of this paper. This material is based upon work partially supported by US Department of Energy under Grant No. DOE DE-FG02-06ER25752 and the National Science Foundation under Grant Nos. CCF-0916893, CNS-0851957, CCF-0746856, and CNS-0509321, and by the AFRL and the AFOSR under agreement number FA8750-11-2-0084.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Agha, G., Mason, I.A., Smith, S., Talcott, C.: A foundation for actor computation. Journal of Functional Programming 7(01), 1–72 (1997)
3. Arts, T., Claessen, K., Svensson, H.E.: Semi-formal Development of a Fault-Tolerant Leader Election Protocol in Erlang. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 140–154. Springer, Heidelberg (2005)
4. Chandy, K.M., Misra, J.: Distributed computation on graphs: Shortest path algorithms. ACM (1982)
5. Esparza, J.: Model checking using net unfoldings. Science of Computer Programming 23(2-3), 151–195 (1994)
6. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121 (2005)
7. Godefroid, P.: Using Partial Orders to Improve Automatic Verification Methods. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 176–185. Springer, Heidelberg (1992)
8. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, p. 142. Springer, Heidelberg (1996)
9. Godefroid, P., Pirottin, D.: Refining Dependencies Improves Partial-Order Verification Methods. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 438–449. Springer, Heidelberg (1993)
10. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI: The Complete Reference. The MPI-2 Extensions, vol. 2 (1998)

11. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian Partial-Order Reduction. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007)
12. Haller, P., Odersky, M.: Actors That Unify Threads and Events. In: Murphy, A.L., Ryan, M. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 171–190. Springer, Heidelberg (2007)
13. Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
14. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: A comparative analysis. In: PPPJava, pp. 11–20 (2009)
15. Kastenberg, H., Rensink, A.: Dynamic Partial Order Reduction Using Probe Sets. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 233–247. Springer, Heidelberg (2008)
16. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM (1978)
17. Lauterburg, S., Dotta, M., Marinov, D., Agha, G.: A framework for state-space exploration of Java-based actor programs. In: ASE, pp. 468–479 (2009)
18. Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 308–322. Springer, Heidelberg (2010)
19. Lei, Y., Carver, R.H.: Reachability testing of concurrent programs. IEEE Transactions on Software Engineering 32, 382–403 (2006)
20. McMillan, K.: Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 164–177. Springer, Heidelberg (1993)
21. Open Systems Laboratory, University of Illinois at Urbana-Champaign. The Actor Foundry: A Java-based Actor Programming Environment (September 1998)
22. Peled, D.: All from One, One for All: On Model Checking Using Representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
23. Pi code, http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/simplempi/main.htm
24. Sen, K., Agha, G.: Automated Systematic Testing of Open Distributed Programss. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 339–356. Springer, Heidelberg (2006)
25. Sen, K., Agha, G.: A Race-Detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 166–182. Springer, Heidelberg (2007)
26. Vakkalanka, S.S., Gopalakrishnan, G., Kirby, R.M.: Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 66–79. Springer, Heidelberg (2008)
27. Valmari, A.: Stubborn sets for reduced state space generation. In: ATPN, pp. 491–515 (1991)
28. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. 10(2), 203–232 (2003)
29. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole Partial Order Reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008)

Verification of Ad Hoc Networks with Node and Communication Failures

Giorgio Delzanno[1], Arnaud Sangnier[2], and Gianluigi Zavattaro[3]

[1] University of Genova, Italy
[2] LIAFA, Univ. Paris Diderot, Sorbonne Paris Cité, CNRS, France
[3] University of Bologna, INRIA - FOCUS Research Team, Italy

Abstract. We investigate the impact of node and communication failures on the decidability and complexity of parametric verification of a formal model of ad hoc networks. We start by considering three possible types of node failures: intermittence, restart, and crash. Then we move to three cases of communication failures: nondeterministic message loss, message loss due to conflicting emissions, and detectable conflicts. Interestingly, we prove that the considered decision problem (reachability of a control state) is decidable for node intermittence and message loss (either nondeterministic or due to conflicts) while it turns out to be undecidable for node restart/crash, and conflict detection.

1 Introduction

Broadcast communication is often used in networks in which individual nodes have no precise information about the underlying connection topology (e.g. ad hoc wireless networks). As shown in [13,10,11,16,17,4], this type of communication can naturally be specified in models in which a network configuration is represented as a graph and in which individual nodes run an instance of a given protocol specification. A protocol typically specifies a sequence of control states in which a node can either send a message (emitter role), wait for a message (receiver role), or perform an update of its internal state. Broadcast communication can be represented here as a simultaneous update of the state of the emitter node and of the state of its neighbors. This semantics of broadcast is often termed *selective* in contrast with broadcast messages that simultaneously reach all nodes of a network.

Already at this level of abstraction, verification of ad hoc network protocols turns out to be a very difficult task. A formal account of this problem is given in [3,4], where the *control state reachability problem* is proved to be undecidable for selective broadcast communication. The control state reachability problem consists in verifying the existence of an initial network configuration (with unknown size and topology) that may evolve into a configuration in which at least one node is in a given control state. If such a control state represents a protocol error, then this problem naturally expresses (the complement of) a safety verification task in a setting in which nodes have no information a priori about the size and connection topology of the underlying network. The analysis in [3,4] works under the assumption that the underlying network and communication model are both reliable. This is a quite strong assumption since ad hoc networks have several sources of unreliability: from node failures to conflicts caused by interferences among different transmissions.

In this paper we study the impact of node and communication failures on the control state reachability problem for ad hoc network protocols. We start our analysis by introducing node failures in a model of selective broadcast. For this purpose, we consider an intermittent semantics in which a node can be (de)activated at any time. As a first result, we show that control state reachability becomes decidable under the intermittent semantics. Decidability seems strictly related to the assumption that nodes cannot directly take decisions that depend on the current activation state (e.g. change state when the node is turned on). We then consider two restricted types of node failure, i.e., node crash (a node can only be deactivated) and node restart (when it is activated, it restarts in a special restart state). We show that for these two semantics, the verification task becomes undecidable.

We consider then different types of communication failures. We first consider a semantics in which a broadcast is not guaranteed to reach all neighbors of the emitter nodes (message loss). Control state reachability is again decidable in this case. We then introduce a semantics for selective broadcast specifically designed to capture possible conflicts during a transmission. Basically, a transmission of a broadcast message is split into two different phases: a starting and an ending phase. During the starting phase, receivers connected to the emitter move to a transient state. While being in the transient state, a reception from another node generates a conflict. In the ending phase an emitter always moves to the next state whereas connected receivers move to their next state only when no conflicts have been detected. Time-out can be modeled here by allowing receivers to abandon a transmission at any time. In our model we also allow several emitters to simultaneously start a transmission. Decidability holds only when receivers ignore corrupted messages by remaining in their original state. Moreover, for the verification task in the decidable variants we show that it is possible to resort to the polynomial time reachability algorithm for a model of ad hoc networks with nondeterministic mobility presented in [2].

Related Work. Formal models of broadcast communication have been considered in several works in the literature such as [14,16,17,6,5,8,10,11,12]. Perfect synchronous semantics for broadcast communication in mobile and ad hoc networks have been proposed in [14,16,17,5]. Verification problems for broadcast protocols have been studied in the different context of hardware protocols [6]. In all the above mentioned works a transmission is modelled as an atomic step in which the emitter node and the connected receiver nodes simultaneously update their current state. Decidability of reachability problems like those we consider here (coverability) is considered only in the case of synchronous broadcast for fully connected networks [6].

Delays in between the instant in which the emitter starts a transmission and the instant in which the transmission ends have been considered in a timed semantics [10,11] in which every message has an associated non-zero transmission time, or in form of non-atomic transitions (start and end phase are kept distinct) as in [12]. In all these approaches a broadcast communication is split into several phases to model scenarios in which different transmission periods of different emitters overlap. Following [12] in the present paper we consider an untimed semantics for explicitly representing conflicts. Differently from other models, our semantics allows multiple nodes to start a communication in the same instant, a model that seems closer to real scenarios.

In [3,4] we have studied decision problems for verification of models of ad hoc networks with seective broadcast communication with perfect semantics and no conflicts. In this paper we lift our studies to unreliable networks and consider semantics for broadcast communication with conflicts. Communication failures (e.g. message loss and insertion) are commonly considered when facing verification problems for communication protocols as in the case of unreliable FIFO channels [1]. Differently from works like [1], we evaluate here the impact of communication failures in a communication model with broadcast communication restricted to neighbour nodes and in which reachability is formulated for an initial configuration with arbitrary size and topology.

2 Ad Hoc Networks

Definition 1. *A Q-graph is a labeled undirected graph $\gamma = \langle V, E, L \rangle$, where V is a finite set of* nodes, *$E \subseteq V \times V$ is a symmetric relation representing a finite set of* edges*, and L is a labeling function from V to a set of labels Q (in our setting they represent control states).*

We use $L(\gamma)$ to represent all the labels present in γ (i.e. the image of the function L). The nodes belonging to an edge are called the *endpoints* of the edge. For an edge $\langle u, v \rangle$ in E, we use the notation $u \sim_\gamma v$ and say that the vertices u and v are adjacent to each other in the graph γ. We omit γ, and simply write $u \sim v$, when it is made clear by the context.

A configuration is a Q-graph and we assume that each node of the graph is a process that runs a common predefined protocol defined by a communicating automaton with a finite set Q of control states. Communication is achieved via selective broadcast: the effect of a broadcast is local to the vicinity of the sender. The initial configuration is any graph in which all the nodes are labeled by an initial control state. Note that even if Q is finite, there are infinitely many possible configurations (the number of Q-graphs). We next formalize the above intuition.

Definition 2. *A process is a tuple $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$, where Q is a finite set of control states, Σ is a finite alphabet, $R \subseteq Q \times (\{\tau\} \cup \{!!a, ??a \mid a \in \Sigma\}) \times Q$ is the transition relation, and $Q_0 \subseteq Q$ is a set of initial control states.*

The label τ represents the capability of performing an internal action, and the label $!!a$ ($??a$) represents the capability of broadcasting (receiving) a message

$a \in \Sigma$. For $q \in Q$ and $a \in \Sigma$, we define the set $R_a(q) = \{q' \in Q \mid \langle q, ??a, q' \rangle \in R\}$ which contains states that can be reached from the state q when receiving the message a.

The network semantics associated to a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$ is given by the transition system $AHN(\mathcal{P}) = \langle \mathcal{C}, \Rightarrow, \mathcal{C}_0 \rangle$, where $\mathcal{C}$ is the set of Q-graphs (network configurations), $\mathcal{C}_0$ is the set of Q_0-graphs (initial configurations), and $\Rightarrow \subseteq \mathcal{C} \times \mathcal{C}$ is the transition relation defined as follows: for $\gamma = \langle V, E, L \rangle$, we have $\gamma \Rightarrow \gamma'$ iff $\gamma' = \langle V, E, L' \rangle$ and one of the following conditions holds:

Local: $\exists v \in V$ s.t. $(L(v), \tau, L'(v)) \in R$, and $L(u) = L'(u)$ for all u in $V \setminus \{v\}$;
Broadcast: $\exists v \in V$ s.t. $(L(v), !!a, L'(v)) \in R$ and for every $u \in V \setminus \{v\}$, we have:
- if $u \sim v$ and $R_a(L(u)) \neq \emptyset$ (u can receive a), then $L'(u) \in R_a(L(u))$,
- $L(u) = L'(u)$, otherwise.

An execution in $AHN(\mathcal{P})$ is a sequence $\gamma_0\gamma_1 \ldots$ such that $\gamma_0 \in \mathcal{C}_0$ and $\gamma_i \Rightarrow \gamma_{i+1}$ for $i \geq 0$. We use $\Rightarrow^*$ to denote the reflexive and transitive closure of $\Rightarrow$.

Observe that a broadcast message a sent by v is delivered only to the subset of neighbors interested in it; such a neighbor u has then to update its state with a new state taken from $R_a(L(u))$. All the other nodes (including neighbors not interested in a) simply ignore the message. Also notice that the topology is static, i.e., the set of nodes and edges remain unchanged during an execution.

As an example of an ad hoc network and of its semantics, consider a process consisting of the following rules: (A, τ, C), $(C, !!m, D)$, $(B, ??m, C)$, and $(A, ??m, C)$. As shown in Figure 1, starting from a configuration with only A and B nodes, an A node first moves to C and then sends m to his/her neighbors. In turn, they forward the message m to their neighbors, and so on.

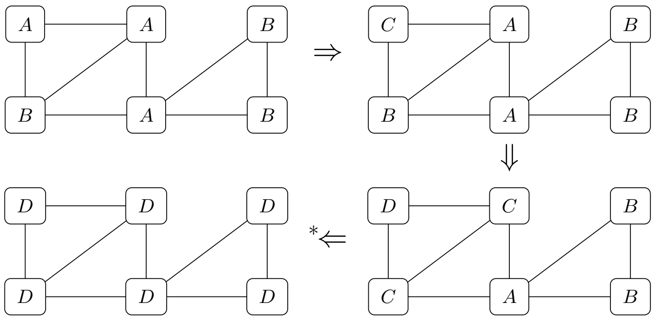


Fig. 1. Example of normal execution

The network semantics formalized by the transition system $\Rightarrow$ assumes fixed topology. Formally, if $\gamma \Rightarrow \gamma'$ then $\gamma = \langle V, E, L \rangle$ and $\gamma' = \langle V, E, L' \rangle$ share the same nodes and edges and can differ only in the labeling function. In [3] we have formalized also nondeterministic mobility as follows. Given a process

$\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$ the mobile network semantics is given by the transition system $MAHN(\mathcal{P}) = \langle \mathcal{C}, \rightsquigarrow, \mathcal{C}_0 \rangle$, where $\mathcal{C}$ and $\mathcal{C}_0$ are as in the definition of $AHN(\mathcal{P})$ and $\rightsquigarrow \subseteq \mathcal{C} \times \mathcal{C}$ is the transition relation defined as follows: for $\gamma = \langle V, E, L \rangle$, we have $\gamma \rightsquigarrow \gamma'$ iff $\gamma' = \langle V, E', L' \rangle$ and one of the following conditions holds:

State Transition: $\gamma \Rightarrow \gamma'$;
Mobility: $E' \subseteq V \times V$ and $L' = L$.

Observe that all the transitions of the original $AHN(\mathcal{P})$ transition system are included by the state transition rule, while the mobility rule adds transitions that modify the edges arbitrarily while preserving the labeling function.

2.1 Safety Analysis: The Control State Reachability Problem

Following [3,4] we consider decision problems related to verification of safety properties. We remark that in our formulation the size and topology of the initial configurations is not fixed a priori. The problem that we consider is *control state reachability* (COVER) defined as follows:

Input: A process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$ with $AHN(\mathcal{P}) = \langle \mathcal{C}, \Rightarrow, \mathcal{C}_0 \rangle$ and a control state $q \in Q$.
Output: Yes, if $\exists \gamma \in \mathcal{C}_0$ and $\gamma' \in \mathcal{C}$ s.t. $\gamma \Rightarrow^* \gamma'$ and $q \in L(\gamma')$; no, otherwise.

If q represents an error state, COVER amounts at checking whether there exists an initial configuration (among the infinitely many possible ones) from which a configuration containing a node in the error state is reachable.

In [3], we prove the following result.

Theorem 1. COVER *is undecidable.*

In the following we will also consider COVER for the mobile network semantics: in that case the transitions $\gamma \rightsquigarrow \gamma'$ will be taken into account instead of $\gamma \Rightarrow \gamma'$. In [3] we have proved that COVER turns out to be decidable with spontaneous (i.e. non-deterministic) mobility. Indeed, in this setting the topology of the network cannot be exploited to build structures that could be applied to model an unbounded storage. In a more recent work [2], we have characterized its complexity.

Theorem 2. COVER *for mobile ad hoc networks is* PTIME*-complete.*

We will also study different semantics for ad hoc networks and we will consider COVER for these semantics. However, sometimes the labelled graphs representing the configurations will have more information in their labels than only the control state of the process, for these cases, COVER will correspond to the reachability of a configuration in which there exists a node whose label contains the desired control state.

3 Node Failures

3.1 Intermittent Nodes

We start our analysis from a semantic variant that models intermittent nodes. We modify the network semantics by using a flag, which is set to $\mathtt{A}$ [resp. to $\mathtt{D}$] to denote an active [resp. deactivated] node.

Definition 3. *Given a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$, an i-configuration is a $(Q \times \{\mathtt{A}, \mathtt{D}\})$-graph and an initial i-configuration is a $(Q_0 \times \{\mathtt{A}, \mathtt{D}\})$-graph.*

We use $\mathcal{C}^{int}$ [resp. $\mathcal{C}_0^{int}$] to denote the set of i-configurations [resp. initial i-configurations] associated to a process definition $\mathcal{P}$. Given a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$, the semantics of the corresponding ad hoc network with intermittent nodes is given by the transition system $AHN_i(\mathcal{P}) = \langle \mathcal{C}^{int}, \dashrightarrow, \mathcal{C}_0^{int} \rangle$ where the transition relation $\dashrightarrow \subseteq \mathcal{C}^{int} \times \mathcal{C}^{int}$ is defined as follows: for $\gamma = \langle V, E, L \rangle$, we have $\gamma \dashrightarrow \gamma'$ iff $\gamma' = \langle V, E, L' \rangle$ and one of the following conditions holds:

Local: $\exists v \in V$ s.t. $L(v) = \langle q, \mathtt{A} \rangle$, $L'(v) = \langle q', \mathtt{A} \rangle$, $(q, \tau, q') \in R$, and $L(u) = L'(u)$ for all u in $V \setminus \{v\}$;

Broadcast: $\exists v \in V$ s.t. $L(v) = \langle q, \mathtt{A} \rangle$, $(q, !!a, q') \in R$, $L'(v) = \langle q', \mathtt{A} \rangle$, and for every u in $V \setminus \{v\}$:
- if $u \sim v$ and $L(u) = \langle q'', \mathtt{A} \rangle$ and $R_a(q'') \neq \emptyset$, then $L'(u) = \langle q''', \mathtt{A} \rangle$ with $q''' \in R_a(u)$;
- $L(u) = L'(u)$, otherwise.

Intermittence: $\exists v \in V$ s.t. $L(v) = \langle q, \mathtt{A} \rangle$ [resp. $L(v) = \langle q, \mathtt{D} \rangle$], $L'(v) = \langle q, \mathtt{D} \rangle$ [resp. $L(v) = \langle q, \mathtt{A} \rangle$] , and $L(u) = L'(u)$ for all u in $V \setminus \{v\}$.

Note that the transition relation is defined as in the previous section with only two differences: the transitions already present in the previous definition now apply only to active nodes (i.e. those with the flag $\mathtt{A}$); additional transitions allow one node to move from the active to the passive state, and vice versa. We denote by $\dashrightarrow^*$ the reflexive and transitive closure of $\dashrightarrow$.

An example of ad hoc network protocol and of its semantics under node intermittence, consider the following protocol: $(A, !!m, D)$, $(C, !!m, D)$, $(B, ??m, C)$, and $(A, ??m, C)$. As shown in Figure 2, the top-left node is initially deactivated. It then activates, sends a message, and only active neighbors react, and so on.

We now prove that COVER is PTIME-complete also for ad hoc networks with intermittent nodes. This result follows from a the correspondence between $AHN_i(\mathcal{P})$ and $MAHN(\mathcal{P})$ formalized by the following proposition.

Proposition 1. *Consider a process definition $\mathcal{P}$ and a control state q. A configuration γ s.t. $q \in L(\gamma)$ is reachable from an initial configuration in $AHN_i(\mathcal{P})$* if and only if *a configuration γ' s.t. $q \in L(\gamma')$ is reachable from an initial configuration in $MAHN(\mathcal{P})$.*

Proof. We start from the *only if* part. Consider the initial state $\gamma_0 = \langle V, E, L_0 \rangle$ and the execution $\gamma_0 \dashrightarrow^* \gamma$ in $AHN_i(\mathcal{P})$ with $q \in L(\gamma)$. A similar execution can be reproduced also in $MAHN(\mathcal{P})$. Consider the initial configuration

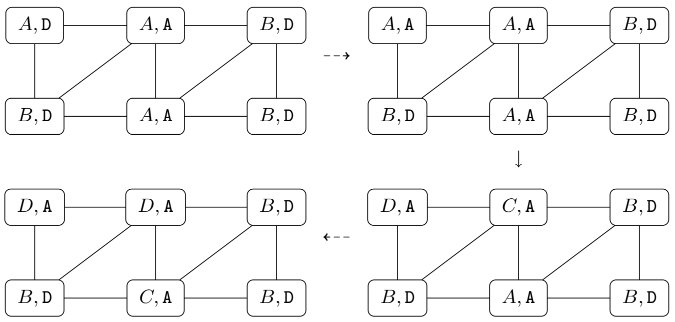


Fig. 2. Example of execution with intermittent nodes

$\gamma_0' = \langle V, E, L_0' \rangle$ with, for every $v \in V$, $L_0'(v) = q_v$ assuming $L_0(v) = \langle q_v, \mathtt{A} \rangle$ or $L_0(v) = \langle q_v, \mathtt{D} \rangle$. Consider now the following execution $\gamma_0' \rightsquigarrow^* \gamma'$ constructed from the above execution $\gamma_0 \dashrightarrow^* \gamma$ as follows. All the **Local** and **Broadcast** transitions are faithfully reproduced, while the **Intermittence** transitions are mimicked by a **Mobility** transition: in case of deactivation of one node the **Mobility** transition disconnects such node from its neighbors, while in case of node activation the **Mobility** transition restores the previously removed edges. It is easy to see that $q \in L(\gamma')$.

We now move to the *if* part. Consider the initial state $\gamma_0' = \langle V', E', L_0' \rangle$ and the execution $\gamma_0' \rightsquigarrow^* \gamma'$ in $MAHN(\mathcal{P})$ with $q \in L(\gamma')$. A similar execution can be reproduced also in $AHN_i(\mathcal{P})$. Consider the initial configuration $\gamma_0 = \langle V', E, L_0 \rangle$ with $E = V' \times V'$ (i.e. γ_0 is a complete graph) and, for every $v \in V'$, $L_0(v) = \langle q_v, \mathtt{A} \rangle$ assuming $L_0'(v) = q_v$. Consider now the following execution $\gamma_0 \dashrightarrow^* \gamma$ constructed from the above execution $\gamma_0' \rightsquigarrow^* \gamma'$ as follows. All the **Local** transitions are faithfully reproduced; the **Broadcast** transitions are reproduced by a protocol that first deactivates the nodes that are not neighbors of the emitter in the corresponding mobile network execution, then the broadcast actions is mimicked, and then the previously deactivated nodes are re-activated; the **Mobility** transitions are not reproduced. It is easy to see that $q \in L(\gamma)$. □

As a simple corollary of the above Proposition and Theorem 2 we obtain the following.

Theorem 3. COVER *for ad hoc networks with intermittent nodes is* PTIME-*complete.*

3.2 Node Crash and Restart

We now consider two variants of the semantics with intermittence. In the first one, modelling node crash, nodes can only be deactivated. In the second one, modelling node restart, nodes can also be reactivated but then they restart from a given special state.

Given process $\mathcal{P}$, its transition system with node crash denoted by $AHN_{cr}(\mathcal{P})$, is defined as the transition system $AHN_i(\mathcal{P})$ where the **Intermittence** transitions are replaced by the following **Crash** transitions:

Crash: $\exists v \in V$ s.t. $L(v) = \langle q, \mathtt{A}\rangle$, $L'(v) = \langle q, \mathtt{D}\rangle$, and $L(u) = L'(u)$ for all u in $V \setminus \{v\}$.

Note that with this semantics, nodes that have been turned off (or deactivated) cannot be activated again.

The variant with restart requires the indication of the restart state in the process. So a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0, q_r\rangle$ now includes a restart state $q_r \in Q$. The transition system $AHN_r(\mathcal{P})$ with node restart for $\mathcal{P}$, is defined as the transition system $AHN_i(\langle Q, \Sigma, R, Q_0\rangle)$ where the **Intermittence** transitions are replaced by the following **Restart** transitions:

Restart: $\exists v \in V$ s.t. $L(v) = \langle q, \mathtt{A}\rangle$ [resp. $L(v) = \langle q, \mathtt{D}\rangle$], $L'(v) = \langle q, \mathtt{D}\rangle$ [resp. $L'(v) = \langle q_r, \mathtt{A}\rangle$] and $L(u) = L'(u)$ for all u in $V \setminus \{v\}$.

In this case, besides the transitions turning off nodes, there are also transitions that turn on one node by changing its internal state to the restart state q_r. The following theorem then holds.

Theorem 4. COVER *with node crash [resp. with node restart] is undecidable.*

Proof. The proof is by reduction from the undecidability of COVER for ad hoc networks (Theorem 1). We first consider the model with node crash. Let $\mathcal{P}$ be a process. It is trivial to see that a computation leading to a configuration that exposes the control state q in $AHN(\mathcal{P})$ has a corresponding computation in $AHN_{cr}(\mathcal{P})$ (in which no **Crash** transition is performed).

Consider now a computation in $AHN_{cr}(\mathcal{P})$ leading to a configuration that exposes the control state q. It is not restrictive to assume that the state q is exposed by a node that did not crash during the computation (we can always consider the last step in q before the node crashes). Consider now a computation in $AHN(\mathcal{P})$ that performs the same **Local** and **Broadcast** transitions (but not the **Crash** transitions). It is easy to see that the nodes that did not crash during the computation in $AHN_{cr}(\mathcal{P})$ are in the same state also in the computation of $AHN(\mathcal{P})$. Hence also the latter computation leads to a configuration exposing the control state q.

The undecidability can be proved as in [3] where we present how to translate a two counter machine (a Turing powerful formalism) into a protocol $\mathcal{P}$ for ad hoc network without failures. Such protocol $\mathcal{P}$ should be slightly modified as follows to work also under intermittence. Let $\mathcal{P} = \langle Q, \Sigma, R, Q_0\rangle$; the modified protocol is defined as $\mathcal{P}' = \langle Q', \Sigma', R', \{q_0\}, q_0\rangle$ where $q_0 \notin Q$ and R' is obtained from R by adding the following rules: $(q_0, !!init, q_0')$ and (q_0', τ, q) for all $q \in Q_0$ and $(q, ??init, q_{err})$ for all $q \in Q$ and this assuming that $q_0', q_{err} \in Q' \setminus Q$. The idea of this encoding is that the unique initial state and the restart state are the same, but when a node comes back to the initial state while simulating the protocol $\mathcal{P}$, if it goes to q_0' it sends all his neighbors (which are in state belonging to Q) into the deadlock state q_{err}. This ensures that if a node is turned off and is reactivated, it cannot play a role in the simulation of the protocol $\mathcal{P}$ by $\mathcal{P}'$. $\square$

4 Communication Failures

4.1 Message Loss

The first type of failures corresponds to nondeterministic message loss: when a message is broadcasted, some of the receivers could not receive it.

A process $\mathcal{P}$ is defined as usual. The corresponding transition system $AHN_l(\mathcal{P})$ is defined as $AHN(\mathcal{P})$ where the **Broadcast** transitions are replaced by the following **Message loss** transitions:

Message Loss: $\exists v \in V$ s.t. $(L(v), !!a, L'(v)) \in R$ and for every $u \in V \setminus \{v\}$
- if $u \sim v$ and $R_a(L(u)) \neq \emptyset$ (reception of a in u is enabled), then $L'(u) \in R_a(L(u))$ or $L'(u) = L(u)$,
- $L(u) = L'(u)$, otherwise.

The main difference with the transition system $AHN(\mathcal{P})$ is that during the performance of a broadcast, some of the potential receivers could remain in their internal state. This is similar to what happens in the model with intermittent nodes when one is deactivated. Starting from this observation it is easy to show that there exists a computation leading to a configuration that exposes the control state q in $AHN_l(\mathcal{P})$ iff there exists a corresponding computation in $AHN_i(\mathcal{P})$. From this consideration, we deduce the following theorem.

Theorem 5. COVER *for ad hoc networks with message loss is* PTIME*-complete.*

Proof. Consider a process definition $\mathcal{P}$. As in Theorem 3 we show that there exists an execution in $AHN_l(\mathcal{P})$ leading to a configuration exposing the control state q *if and only if* there exists an execution in $AHN_i(\mathcal{P})$ leading to a configuration exposing q.

Consider an execution leading to a configuration that exposes the control state q in $AHN_l(\mathcal{P})$. It has the following corresponding execution in $AHN_i(\mathcal{P})$: it is sufficient to mimic **Broadcast** transitions by executing before the broadcast a sequence of **Intermittence** transitions that switch off the nodes that do not receive the message, and by performing after the broadcast the **Intermittence** transitions on the same nodes.

Consider now an execution in $AHN_i(\mathcal{P})$ leading to a configuration that exposes the control state q. This execution can be mimicked in $AHN_l(\mathcal{P})$ simply by assuming that the nodes that are deactivated during a specific phase of the execution in $AHN_i(\mathcal{P})$, lose the messages that are broadcasted in that phase in the corresponding execution in $AHN_l(\mathcal{P})$. $\square$

4.2 Conflict

The second type of failures we consider corresponds to transmission conflicts. Here we consider conflicts due to the contemporaneous emission of messages: if a node has (at least two) neighbors that contemporaneously broadcast a message, then such a node is unable to correctly receive the emitted messages. The modeling of this phenomenon requires a significant modification of the formal semantics. First of all we need to introduce a notion of internal state.

Internal State. The internal state of a node is characterized by the current state according to the process behavior, and by two additional flags indicating whether the node is currently emitting or receiving a message. Formally, given a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$ we define the set of states $\mathcal{S} = \big\{[q, x, y] \mid q \in Q, x \in \{\perp\} \cup \Sigma, y \in \{\perp, \mathtt{rcv}, \mathtt{cnfl}\}\big\}$. The field denoted with x represents whether the node is or is not in a transmission state ($\perp$ means no transmission, while $a \in \Sigma$ denotes transmission of message a). The field y represents whether the node is not receiving ($\perp$) or it is currently receiving correctly a message ($\mathtt{rcv}$) or the reception has been damaged due to a conflict ($\mathtt{cnfl}$). The initial states are defined as follows: $\mathcal{S}_0 = \{[q, \perp, \perp] \mid q \in Q_0\}$. Notice that nodes in their initial state are neither receiving nor emitting.

The use of triples simplifies the definition of the semantics. In the figures we also use a more compact notation without distinction between transmission and reception state, e.g., $[q, \perp, \perp]$ is simplified as q, $[q, a, \perp]$ as $[q, a]$, $[q, \perp, \mathtt{rcv}]$ as $[q, \mathtt{rcv}]$, etc.

Network Semantics. The semantics of a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$ with conflicts is given by the transition system $AHN_{co}(\mathcal{P}) = \langle \mathcal{C}^{co}, \rightrightarrows, \mathcal{C}_0^{co} \rangle$ where $\mathcal{C}^{co}$ is the set of $\mathcal{S}$-graphs and the set of initial configurations $\mathcal{C}_0^{co}$ is the set of $\mathcal{S}_0$-graphs.

Before giving the formal definition of the transition relation $\rightrightarrows \subseteq \mathcal{C}^{co} \times \mathcal{C}^{co}$, we define the function *emitter* which associates to a $\mathcal{S}$-graph $\gamma = \langle V, E, L \rangle$ and to a node $u \in V$, the set $\mathit{emitter}(\gamma, u) = \{v \mid u \sim v \text{ and } L(v) = [q, a, y] \text{ for some } a \in \Sigma \text{ and } y \in \{\perp, \mathtt{rcv}, \mathtt{cnfl}\}\}$ of nodes adjacent to u in γ which are in a transmission state.

Given a configuration $\gamma = \langle V, E, L \rangle$, we have that $\gamma \rightrightarrows \gamma'$ iff $\gamma' = \langle V, E, L' \rangle$ and one of the following conditions holds:

Local/Time-Out: $\exists v \in V$ s.t. $L(v) = [q, \perp, y]$, $y \in \{\perp, \mathtt{cnfl}, \mathtt{rcv}\}$, $(q, \tau, q') \in R$, $L'(v) = [q', \perp, \perp]$, and $L(u) = L'(u)$ for all $u \in V \setminus \{v\}$;

Start Broadcast: $\exists v_1, \ldots, v_l \in V$ s.t. $\cup_{j \in \{1 \ldots l\}} \mathit{emitter}(\gamma, v_j) = \emptyset$, $L(v_i) = [q_i, \perp, \perp]$, $(q_i, !!a_i, q_i') \in R$, $L'(v_i) = [q_i', a_i, \perp]$ $\forall i \in \{1 \ldots l\}$ and the following conditions hold:
- $\forall u \in V \setminus \{v_1, \ldots, v_l\}$ s.t. $u \sim v_i$ for some $i \in \{1 \ldots l\}$ and $L(u) = [r, \perp, y]$ with $y \in \{\mathtt{rcv}, \perp\}$ we have:
  - if $y = \mathtt{rcv}$ then $L'(u) = [r, \perp, \mathtt{cnfl}]$;
  - if $y = \perp$ and $u \not\sim v_j$ $\forall j \in \{1 \ldots l\} \setminus \{i\}$ then $L'(u) = [r, \perp, \mathtt{rcv}]$;
  - if $y = \perp$ and $u \sim v_j$ for some $j \in \{1 \ldots l\} \setminus \{i\}$ then $L'(u) = [r, \perp, \mathtt{cnfl}]$;
- $L(u) = L'(u)$ otherwise;

End Broadcast: $\exists\ v \in V$ s.t. $L(v) = [q, a, \perp]$, $L'(v) = [q, \perp, \perp]$ and we have:
- $\forall u \in V$ s,t. $u \sim v$ and $L(u) = [r, \perp, y]$, with $y \in \{\mathtt{rcv}, \mathtt{cnfl}\}$, and $\mathit{emitter}(\gamma, u) = \{v\}$ we have:
  - if $y = \mathtt{rcv}$ and $\exists\ r'$ s.t. $(r, ??a, r') \in R$ then $L'(u) = [r', \perp, \perp]$;
  - if $y = \mathtt{rcv}$ and $\nexists\ r'$ s.t. $(r, ??a, r') \in R$ or $y = \mathtt{cnfl}$ then $L'(u) = [r, \perp, \perp]$;
- $L(u) = L'(u)$ otherwise.

The local rule models internal and time-out steps (a node non-deterministically decides to abandon a transmission). In the start rule we select a set of node that have the capability of sending a broadcast and check that no other node in their vicinity is currently transmitting. The selected emitters simultaneously start transmitting. Receiving nodes connected to a single emitter move to the `rcv` state, and to the `cnfl` state in case of connection with more than one emitter (e.g. a selected node and an emitter that started transmitting in a previous step). In the ending rule an emitter moves to its next state. A receiver connected to such a node moves to the next state only if it is still in the `rcv` state (no conflicts occurred in between the start and end phases).

As an example of ad hoc networks and of its semantics in the model with conflicts, consider the process $(S, !!m, T), (R, ??m, Q)$, and the execution in Figure 3. In the initial configuration we have three senders in state S $(a, b, c$ from left to right), and three receivers in state R $(d, e, f$ from left to right). Nodes a and b can simultaneously start transmitting m, since no other node is currently transmitting in their vicinity. Node d simultaneously moves to a conflict state (it is connected to both emitters), while node e moves to a reception state. When c starts transmitting m (again there are no other emitters in its vicinity), node e is forced to enter a conflict state, whereas node f goes to a reception state. When a stops transmitting, d goes back to the original state (a conflict occurred). If now c stops transmitting, f receives the message and moves to its next state Q (no conflicts occurred). Finally when b stops transmitting, e goes back to the original state (a conflict occurred). Other possible executions are obtained, e.g., by selecting only one of the nodes a, b for starting a transmission (the other node has to remain silent since it is connected to an active emitter) and by nondeterministically allowing receiver nodes to abandon a transmission.

Theorem 6. COVER *for ad hoc networks with conflicts is* PTIME-*complete.*

Proof. Consider a process $\mathcal{P}$. Following our usual proof technique, we show that there exists an execution in $AHN_{co}(\mathcal{P})$ leading to a configuration exposing the control state q *if and only if* there exists an execution in $AHN_i(\mathcal{P})$ leading to a configuration exposing q.

It is easy to see that a computation leading to a configuration that exposes the control state q in $AHN_{co}(\mathcal{P})$ has a corresponding computation in $AHN_i(\mathcal{P})$: the **Local** transitions are faithfully reproduced, the **Start broadcast** transitions are not mimicked, and the **End broadcast** transitions are simulated via a protocol that first turns off the nodes that do not receive the message or that detect a conflict, then executes the broadcast, and then turns on the samenodes.

It is more complex to show that a computation in $AHN_i(\mathcal{P})$ that leads to a configuration that exposes the control state q can be reproduced in $AHN_{co}(\mathcal{P})$.

We first assume, without loss of generality, that in the process $\mathcal{P}$ there is at least one state with an outgoing broadcast transition which is reachable from an initial state $q_0 \in Q_0$ doing only internal steps. If this is not the case, there is no communication in the system and the analysis of COVER can be trivially done by checking whether the target state q is reachable from an initial state in the automaton defining the process behavior doing only internal steps. Consider

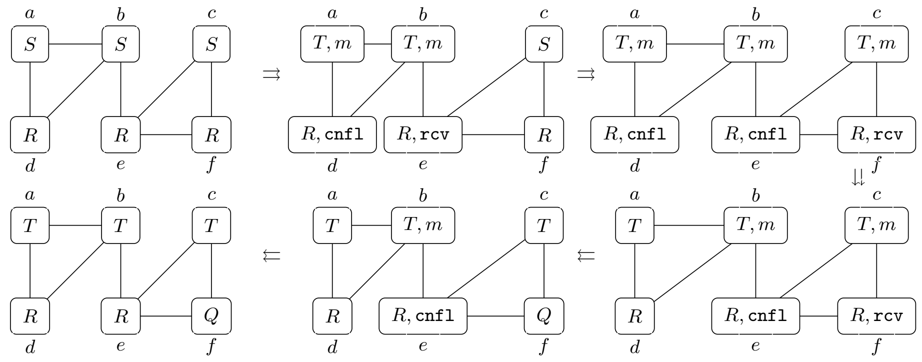


Fig. 3. Example of execution with conflicts

now the computation in $AHN_i(\mathcal{P})$ that leads to a configuration that exposes the control state q. Let γ_0 be the initial configuration in the considered computation, and let $loss(u)$ be the number of messages that the node u loses during the computation when it was turned off.

We now show the existence of an initial configuration in $AHN_{co}(\mathcal{P})$ able to reproduce such computation. This initial configuration contains γ_0 plus a set of additional nodes used to generate conflicts.

Namely, we connect to each node u of the initial configuration $loss(u)$ additional nodes $Noise(u)$: each node in $Noise(u)$ is connected only with its corresponding node u.

Each node u simulates the behavior of the corresponding node in the computation in $AHN_i(\mathcal{P})$. The nodes in $Noise(u)$ are initially in the state q_0. The simulation of the transitions in the computation in $AHN_i(\mathcal{P})$ is as follows. First of all, for every node u we consider local transitions for nodes in $Noise(u)$ in state q_0 leading them to a state ready to perform a broadcast. Then the transitions are simulated as follows.

- **Local** transitions are faithfully reproduced.
- **Intermittence** transitions are not mimicked.
- To simulate **Broadcast** transitions performed by one node, say v, we proceed as follows: we partition the potential receivers in two groups, (i) those that actually receive the message and (ii) those that do not receive it as they are turned off. For each node u in group (ii) we take an attacker node $n \in Noise(u)$ ready to start a transmission and let n perform a **Start broadcast** transition. Simultaneously node u moves to the `rcv`-state. Node v performs then a broadcast (it executes both the **Start** and the **End broadcast** transitions). Since u and v are connected, u detects a conflicting transmission and moves to the `cnfl`-state. Finally, node n ends the transmission.

  Note that the nodes corresponding to (i) receive the broadcast messages, while those corresponding to (ii) do not receive it, due to the conflict generated by the interferring transmissions generated by the attacker node n.

By assumption on the cardinality of $Nodes(u)$, therefore an attack can be executed every time node u is switched off in the computation with intermittent semantics. □

4.3 Conflict Detection

We now define a variant of the semantics in order to capture the notion of conflict detection. In fact, even though a node that receives overlapping signal emissions is unable to reconstruct the emitted messages, it can infer that (at least) two neighbors have contemporaneously emitted their messages. This can be considered in our model of ad hoc networks by adding *conflict detection* transitions to the processes. Such transitions can be executed by nodes at the end of a receive phase during which more than one neighbor has performed a broadcast. Formally, we slightly modify the definition of the *Internal State* and of the *Network Semantics* of the previous section.

Internal State. The new definition of $\mathcal{P}$ is as usual with the unique difference that we can have transitions of the form (q, ρ, q') in R, representing conflict detection (where ρ is a new symbol).

Network Semantics. Given a process $\mathcal{P}$, the transition system $AHN_{cd}(\mathcal{P})$ characterizing the semantics with conflict detection is defined as $AHN_{co}(\mathcal{P})$ except that the **End broadcast** transitions are replaced by the following **End broadcast II** transitions:

End Broadcast II: $\exists v \in V$ s.t. $L(v) = [q, a, \perp]$, $L'(v) = [q, \perp, \perp]$ and we have:
- $\forall u \in V$ s.t. $u \sim v$, $L(u) = [r, \perp, y]$, with $y \in \{\mathtt{rcv}, \mathtt{cnfl}\}$, and *emitter* $(\gamma, u) = \{v\}$:
  - if $y = \mathtt{rcv}$ and $\exists r'$ s.t. $(r, ??a, r') \in R$ then $L'(u) = [r', \perp, \perp]$;
  - if $y = \mathtt{cnfl}$ and $\exists r'$ s.t. $(r, \rho, r') \in R$ then $L'(u) = [r', \perp, \perp]$;
  - if $y = \mathtt{rcv}$ and $\nexists r'$ s.t. $(r, ??a, r') \in R$, or $y = \mathtt{cnfl}$ and $\nexists r'$ s.t. $(r, \rho, r') \in R$, then $L'(u) = [r, \perp, \perp]$;
- $L(u) = L'(u)$ otherwise.

As an example of ad hoc networks and of its semantics with conflict detection, consider the process $(S, !!m, T), (R, ??m, Q), (R, \rho, Er)$, and the execution in Figure 4. It consists of the same steps as those in Figure 3 up to ending phases of broadcast messages. Receiver that detect a conflict move here to the special Er states. Note that in the step from the fourth to the fifth configuration only the node in the leftmost down corner detects a conflict. The other receiver R is connected to two different emitters, so it will apply the detection only in the next step.

Theorem 7. COVER *for ad hoc networks with conflict detection is undecidable.*

Proof. The proof is by reduction from the undecidability of COVER for ad hoc networks with node restart (Theorem 4). Consider a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0, q_r \rangle$

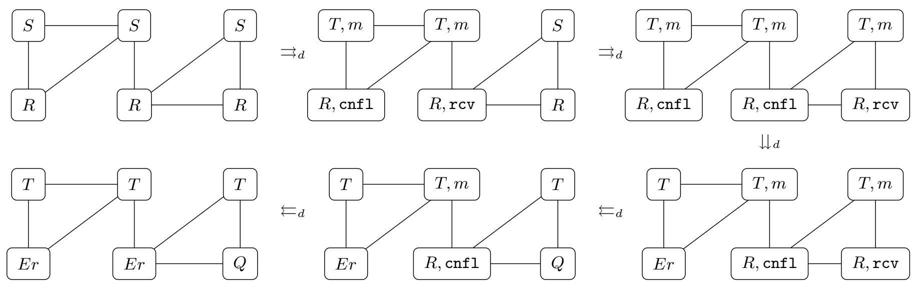


Fig. 4. Example of execution with conflict detections (indicated as $\rightrightarrows_d$)

for ad hoc networks with node restart (q_r being the restart state). Consider now the process $\mathcal{P}' = \langle Q \cup \{q_i\}, \Sigma, R', Q_0 \rangle$, for ad hoc networks with conflict detection, defined as $\mathcal{P}$ with the following additional transitions: for each node $q \in Q$ we have a transition labeled with ρ leading to the additional state q_i, from which there is only one outgoing transition labeled with τ leading to the restart state q_r.

We first show that given a computation in $AHN_r(\mathcal{P})$ leading to a configuration that exposes the control state q, there exists a corresponding computation in $AHN_{cd}(\mathcal{P}')$. As in Theorem 6 we make the nonrestrictive assumption that in the process $\mathcal{P}$ there is at least one state with an outgoing broadcast transition which is reachable from an initial state $q_0 \in Q_0$ doing only internal steps. Let γ be the initial configuration of the considered computation in $AHN_r(\mathcal{P})$. For each node u in γ we denote with $restart(u)$ the number of restarts performed by u during the computation. We now show the existence of an initial configuration γ' of $AHN_{cd}(\mathcal{P}')$ from which the computation is simulated. The configuration γ' is as γ with the difference that each node u has exactly $restart(n) \times 2$ additional neighbors that are used to generate conflicts. These additional nodes are connected only to the corresponding node u. The simulation of the computation proceeds as follows. At the beginning the additional nodes in state q_0 perform the local transitions leading them to a state ready to perform a broadcast. Then the simulation starts.

- **Local** transitions are reproduced faithfully.
- A transition that deactivates the node u is simulated via the following protocol: two of the additional nodes connected to u perform a **Start broadcast** transition and then execute the **End broadcast II**. Due to the emission conflict, the node u moves to the internal state q_i.
- A transition that activates the node u is reproduced by an internal transition from the state q_i of u to the restart state q_r.
- Finally, **Broadcast** transitions are mimicked by performing in sequence a **Start** and an **End broadcast II** transition.

We now show that a computation in $AHN_{cd}(\mathcal{P}')$ leading to a configuration that exposes the control state q has a corresponding computation in $AHN_r(\mathcal{P})$.

In the simulated computation the **Local** transitions are reproduced faithfully, the **Start broadcast** transitions are not mimicked, while **End broadcast II** transitions are simulated by the following protocol.

Assume that the node that completes its signal emission in the **End broadcast II** transition is u, and let a be the emitted message. The neighbors of u able to receive a can be partitioned in three groups:

(i) those that correctly receive message a,
(ii) those that perform a conflict detection transition during the execution of the **End broadcast II** transition,
and (iii) those that do not change their internal state because they are still under the effect of another signal emission.

The simulation of the transition in $AHN_r(\mathcal{P})$ proceeds as follows. The nodes, corresponding to those in (ii) and (iii), that are not currently crashed perform a **Crash** transition, then the **Broadcast** transition is executed. Notice that at the end of this protocol the nodes in (ii) are in the intermediary state q_i in the computation in $AHN_{cd}(\mathcal{P}')$, while they are crashed in the corresponding computation in $AHN_r(\mathcal{P})$. The **Local** transitions that move the nodes form the state q_i to q_r are reproduced in $AHN_r(\mathcal{P})$ by **Restart** transitions. □

5 Conclusion

In this paper we have compared different types of semantics for modelling unreliability in protocols based on broadcast communication. The comparison is based on the study of decidability and undecidability of the coverability problem (reachability of a network with at least a node in an error state for an initial configuration of unknown size and shape). Coverability is commonly used to formulate violations of properties like mutual exclusion (and more in general to locally reason on errors generated by a fixed set of processes independently from the global configuration). Coverability turns out to be undecidable for models in which individual nodes have special transitions to the detect the occurrence of a failure (e.g. crash with restart, conflict detection). Removing this feature from the model completely change the corresponding expressive power, often making coverability decidable. Decidability results are obtained by means of reduction to coverability in a model with spontaneous movement, for which we have given a PTIME algorithm in [2]. Among possible future directions we plan to investigate the impact of node and communication failures in richer models of broadcast communication that could be used to model for instance routing strategy or time division protocols.

References

1. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inf. Comput. 127(2), 91–101 (1996)
2. Delzanno, G., Sangnier, A., Traverso, R., Zavattaro, G.: Reachability Problems in Mobile Ad Hoc Networks. Technical report available on arXiv

3. 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)
4. Delzanno, G., Sangnier, A., Zavattaro, G.: On the Power of Cliques in the Parameterized Verification of Ad Hoc Networks. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 441–455. Springer, Heidelberg (2011)
5. Ene, C., Muntean, T.: A broadcast based calculus for Communicating Systems. In: IPDPS 2001, p. 149 (2001)
6. Esparza, J., Finkel, A., Mayr, R.: On the verification of Broadcast Protocols. In: LICS 1999, pp. 352–359 (1999)
7. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 253–272. Springer, Heidelberg (2007)
8. Godskesen, J.C.: A Calculus for Mobile Ad Hoc Networks. In: Murphy, A.L., Ryan, M. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 132–150. Springer, Heidelberg (2007)
9. Ladner, R.E.: The circuit value problem is logspace complete for P. SIGACT News, 18–20 (1977)
10. Merro, M.: An observational theory for Mobile Ad Hoc Networks. Inf. Comput. 207(2), 194–208 (2009)
11. 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)
12. Lanese, I., Sangiorgi, D.: An operational semantics for a calculus for wireless systems. TCS 411(19), 1928–1948 (2010)
13. Nanz, S., Hankin, C.: A Framework for security analysis of mobile wireless networks. TCS 367(1-2), 203–227 (2006)
14. Prasad, K.V.S.: A Calculus of Broadcasting Systems. SCP 25(2-3), 285–327 (1995)
15. Saksena, M., Wibling, O., Jonsson, B.: Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)
16. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A Process Calculus for Mobile Ad Hoc Networks. In: Wang, A.H., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 296–314. Springer, Heidelberg (2008)
17. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: Query-Based Model Checking of Ad Hoc Network Protocols. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 603–619. Springer, Heidelberg (2009)

Verification of Timed Erlang Programs Using McErlang[⋆]

Clara Benac Earle and Lars-Åke Fredlund

Babel group, DLSIIS, Facultad de Informática, Universidad Politécnica de Madrid
{cbenac,lfredlund}@fi.upm.es

Abstract. There is a large number of works that apply model checking to timed *specifications*, however, there are far fewer attempts at model checking concurrent *programs* for which correct timed behaviour is crucial. In this work we explore the formal verification of timed programs written in the Erlang concurrent programming language, in its full complexity, using the McErlang model checker.

We have extended the McErlang model checker with a timed semantics, similar to the timed semantics Lamport has developed for TLA and TLC, but with a few notable differences. In the paper we present the resulting semantics, its implementation in McErlang, and evaluate it using a number of examples. Among the examples is a process supervision component for controlling the processes in an Erlang application, which provides fault-tolerance.

1 Introduction

Timed semantics for concurrent formalisms is by now a very well-studied field. In the field of real-time semantics a very successful technique for specifying and verifying systems are the timed automata [1]. Similarly there exist numerous discrete-timed specification formalisms, or timed formalisms for which a discrete time domain has been extensively studied, e.g., in process algebra for Timed CSP [2], TCCS [3], TPCCS [4], LOTOS [5], to mention but a few.

Uppaal [6] is the currently most well-known model checker for real-time systems. It provides an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays etc.).

While tools and specification formalisms like Uppaal are quite successful at real-time verification, there is still a need to reason about timed behaviour in other specification and programming languages, using dedicated model checkers. However, many of these model checkers do not implement tailored real-time verification algorithms like Uppaal. Rather, they are used to check timed behaviour by discretizing time, and by using normal model checking algorithms for the LTL

⋆ This work has been partially supported by the following projects: DESAFIOS10 (TIN2009-14599-C03-00), PROMETIDOS (P2009/TIC-1465), pSAFECER (GA 269265) and nSAFECER (GA 295373).

and CTL logics. The SPIN [7] model checker is perhaps the most well-known explicit-state model checker that follows this approach.

In recent years the approach of checking timed behaviour using "untimed" model checkers has received renewed attention, with the publication of Lamport's article [8] on real-time model checking using the TLC model checker. Other recent works along the same lines include [9] and [10].

In this work we address the verification of timed programs written in the Erlang functional concurrent programming language using the McErlang model checker [11]. The approach taken is similar to Lamport's approach to real-time model checking in [8], and also inspired by the timed automata framework, but with a few notable differences. As in Lamport's work, the transitional semantics is defined over a global system state. In contrast to that article there is no explicit clock tick (the minimum observable "time quanta"), rather the tick is derived from the time constraints for the processes in the system, and the granularity of the tick can even vary through the lifetime of the verified system. Moreover, there is no explicit clock process which is responsible for the progress of time. As demonstrated, such processes can be programmed, and if desired, included in a verification.

In related work for Erlang, Guo et.al. [12], verify untimed Erlang programs by translating Erlang into the μCRL process algebra. However, the translation addresses only the high-level concurrency libraries of Erlang. In [13], Guo and Derrick consider the verification of timed Erlang/OTP components by means of translating Erlang into μCRL, however without considering the problem of generating finite models (state graphs).

In Sect. 2 and 3 we provide a brief introduction to the Erlang programming language and the McErlang model checker. Then, in Sect. 4, we provide an intuition for the timed semantics for Erlang, and Sect. 5 defines a high-level formal semantics. In Sect. 6 we evaluate the efficiency of the timed semantics; Sect. 7 summarizes the results.

2 Erlang

Erlang [14,15] is a functional concurrent programming language created by the Ericsson company in the 1980s. Ericsson is still maintaining the main Erlang implementation, but it is available as open source since 1998. The chief strength of the language is that it provides excellent support for concurrency, distribution and fault tolerance on top of a dynamically typed and strictly evaluated functional programming language. Concurrency is achieved by lightweight processes communicating through asynchronous message passing. Although Erlang is not a new language, it has experienced considerable growth in users in recent years. This is due in most part to its focus on message passing instead of variable sharing as the main communication mechanism, which enables programmers to write robust and clean code for modern multiprocessor and distributed systems.

Today Erlang is used by Ericsson and many other companies (T-Mobile (UK), and many smaller start-up companies such as e.g. LambdaStream in Spain and

Klarna in Sweden) to develop industrial applications, often implementing crucial internet server-side applications. Examples include a high-speed ATM switch developed at Ericsson with over a million lines of Erlang code which had to meet very challenging requirements on software reliability and overall system availability [16,17], parts of Facebook chat, Apache CouchDB – a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API, etc.

Handling a large number of processes easily turns into an unmanageable task, and therefore Erlang programmers mostly work with higher-level language components. The OTP component library is the most used, it offers design patterns such as: a generic server component (for client-server communication), a finite state machine component, generic TCP/IP communication, and a supervisor component for easy structuring of fault-tolerant systems.

2.1 Handling Time in Erlang

As Erlang is relatively well known, we will just describe the main language features which concern timing, i.e., the receive statement and timestamps.

The Receive Statement. The basic mechanism for handling time dependent behaviour in Erlang is the timeout clause of a receive statement:

```
receive
  Pat1 when Guard1 -> Expr1;
  ...
  PatN when GuardN -> ExprN
after Deadline -> TimeoutExpr
end
```

The intuitive semantics of the receive statement is as follows. If a message matches a pattern `PatI`, and the guard `GuardI` (which may contain variables bound by the match) evaluates to true (and moreover no earlier pattern `Patj` matches, or the guard `GuardJ` does not evaluate to true), the message is removed from the mailbox and evaluation continues with expression `ExprI` under the matching binding.

Concretely, the oldest message in the process mailbox is first matched against the clauses according to the above procedure. If no pattern and guard match this message, the same sequence of tests continues with the second oldest message, and so on. If no message matches, the process waits for the reception of a matching message for *at least* `Deadline` milliseconds, until it times out and starts executing the expression `TimeoutExpr`.

A zero deadline corresponds to the case when, if no matching message is in the mailbox, the timeout can happen at once. The special atom `infinity` may also be used as a time deadline, signifying waiting forever without timing out.

Timestamps. The API call `now()` returns the time elapsed since 00:00 GMT, January 1, 1970 as a tuple `{MegaSeconds,Seconds,MicroSeconds}`.

3 McErlang

McErlang [11,18] is an explicit-state model checker for programs written in Erlang. Similarly to most explicit state model checkers, McErlang checks concurrent programs against specifications in full linear temporal logic (LTL) using on-the-fly state space exploration algorithms.

The main idea behind the design of McErlang is to re-use as much of the normal Erlang language implementation as possible, but adding a model checking capability. To achieve this, the tool replaces the part of the Erlang runtime system which implements concurrency and message passing, while still using the runtime system for the evaluation of the sequential part of the input programs. McErlang has built-in support for some Erlang OTP component behaviours that are used in almost all serious Erlang programs such as the supervisor component (for implementing fault-tolerant applications) and the generic server component (implementing a client–server component). The presence of such high-level components in the model checker significantly reduces the gap between original program and the verifiable model, compared to other model checkers.

McErlang has been used in several complex case studies: in the model checking of a Video-on-Demand-server [19], in the verification of agent based RoboCup teams [20], and for the verification of an industrial Erlang process supervision component [21]. The timed extension of McErlang described in this paper is available at GitHub [22].

4 A Timed Extension

In the following subsections we first provide an intuition for the untimed Erlang semantics in McErlang, then introduce the timed extension and timestamps.

4.1 An Untimed Semantics

Previously, there was only an untimed semantics implemented in McErlang. In the untimed semantics, if a receive statement cannot be executed because there is no matching message in the process mailbox, the timeout is enabled (unless the deadline is `infinity`). However, another process can send a receivable message to the process and thus disable the timeout. This corresponds to treating timeouts as nondeterministic choices. As an example, consider code below.

```
P1 = spawn(fun () -> receive Msg -> ok
                     after 1000 -> bad
                     end
         end),
spawn(fun () -> P1!hello end).
```

Two processes are spawned, the first (`P1`) waiting to receive a message, and timing out after one second (1000 milliseconds) if no message can be received, and the second process sending the message `hello` to the first.

The state graph of the above program as generated by McErlang is depicted in Fig. 1 below. Note that only side effects are depicted; as the receive statement is not considered a side effect it is not shown. We colour the states where the `hello` message was received light grey, whereas states where a timeout occurred first are grey, and the arrow of the timeout transition is bold. Clearly there is a race condition in the program: if the message `hello` is sent first from `P1` to the second process a timeout never happens. Alternatively the timeout can happen first; in this case the message is sent anyway but is never received.

Non-timed Actions Are Infinitely Fast. A semantics option, implemented in McErlang, provides a semantics where non-timeout actions are infinitely fast compared to timeouts, i.e., always giving precedence to non-timeout actions. In practice this turns out to be a useful abstraction in many scenarios where the actions of the verified program can be safely assumed to be infinitely fast compared to timed (external) actions.

The state graph for the above program, when this option is enabled, is shown in Fig. 2. Since a timeout is infinitely slow compared to other actions, the timeout never happens since it is disabled by the reception of the `hello` message.

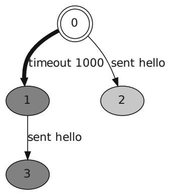


Fig. 1. State graph with no precedence

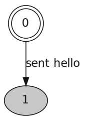


Fig. 2. State graph with precedence

4.2 Adding Explicit Time

The main changes needed in McErlang to implement a timed semantics are to record the current time in the state representation of a running program, and to modify the behaviour of the receive statement in the model checker so that when handling timeouts, the current time is taken into account.

To keep compatibility with normal Erlang code we let the current time be a tuple `{MegaSeconds,Seconds,MicroSeconds}`, and its initial value is `{0,0,0}`.

Clearly the presence of a non-infinity timeout clause in a receive statement specifies a *minimum* waiting period until a timeout happens. In the Erlang documentation there is of course no guarantee for exactly when, after a timer has elapsed, the corresponding timeout happens (as it depends on the operating system, the hardware, etc). However, in this work, as is usual for timed calculi, we also want to be able to specify a *maximum* waiting period until a timeout happens (a notion sometimes called *urgency* in timed calculi).

To specify the urgency of a state the function `mce_erl:urgent(MaximumWait)` is provided. The parameter `MaximumWait` specifies the maximum number of milliseconds the process can remain in the current state, if it has transitions enabled.

Moreover we reinterpret the notion of an infinitely fast computation (i.e., where normal actions are infinitely fast compared to timeouts) as one where every state has an associated implicit call to `mce_erl:urgent(0)` signifying that no time can pass if the state has a transition enabled.

For instance, we can add the line `mce_erl:urgent(1500)` to the running program example (see Fig. 3) to force a timeout to happen before some moment in time. The first process will now wait between 1 and 1.5 seconds for a message to arrive before timing out. Since we have not specified when the second process sends a message both possibilities (timeout or no timeout) remain possible. Its state graph is depicted in Fig. 4; each state is labelled by the current time value in milliseconds.

```
P1 =
  spawn
  (fun () ->
     mce_erl:urgent(1500),
     receive Msg -> ok
     after 1000 -> bad
     end
   end),
spawn(fun () -> P1!hello end).
```

Fig. 3. Program 2

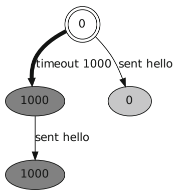


Fig. 4. State graph with urgency and a clock

It may be surprising to see that time does not progress in Fig. 4 after the timeout, and that the timeout occurs exactly at the earliest possible moment. This is because the semantics discretizes the progress of time: time makes a discrete jump between two consecutive time values. So what are the consecutive time values? In normal discrete-time semantics there is often an implicit clock, with a *tick* value which defines the minimum distance between two time values:

$$t_0 = 0 \qquad t_1 = 1 * tick \qquad t_2 = 2 * tick \qquad \cdots$$

where the t_i's are the time values at different states. In our semantics, in contrast, there is no implicit clock process nor a hard-wired tick clock value increment. Rather, each timeout clause in a receive statement represents a clock tick.

However, we can easily implement an *explicit* clock process, which constantly increments the time value with a tick increment as seen in Fig. 5. Note that if we add an unbounded clock process to the program in Fig. 3, then its state graph becomes infinite. To obtain a finite state graph for now (in Sect. 5.1 we overcome this restriction) the bounded clock in Fig. 6 is used instead.

The state graph of program 2, where we add a clock process with a 500ms tick and a duration of 2500ms with the call `spawn(fun () ->clock(500,2500) end)`, is shown in Fig. 7. The timeout transitions in the graph are either timeouts by a process (corresponding to bold arrows as usual) or timeouts by the clock process. In the graph we can see that the timeout behaviour (the bold transition)

```
clock(Tick) ->
  mce_erl:urgent(0),
  receive
  after Tick -> clock(Tick)
  end.
```

Fig. 5. An unbounded clock

```
clock(Tick,0) -> ok;
clock(Tick,N) when N>0 ->
  mce_erl:urgent(0),
  receive
  after Tick ->
    clock(Tick,N-Tick)
  end.
```

Fig. 6. A bounded clock

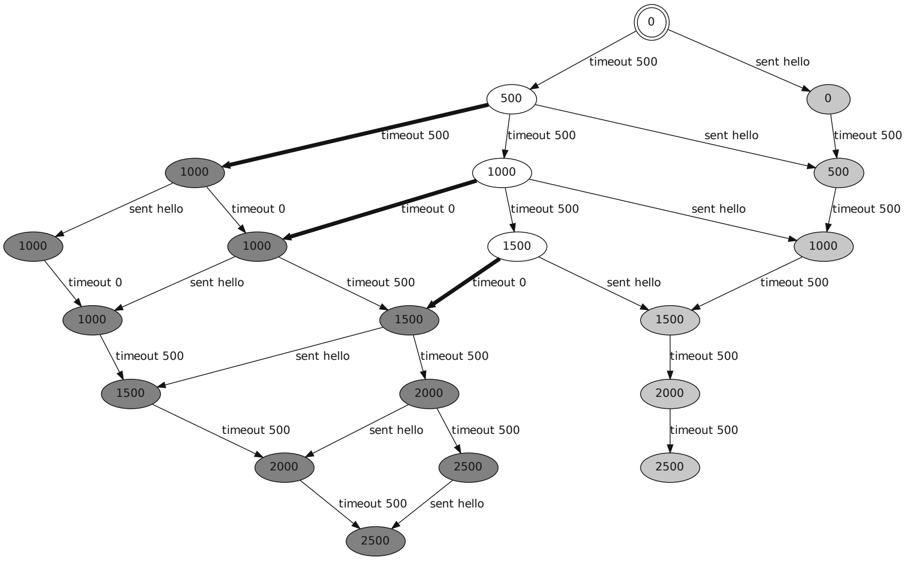


Fig. 7. State graph with an explicit bounded clock process

is enabled after 500ms, and stays enabled until after 1500ms. Note also the white state labelled with 1500 (ms); since the timeout transition with value 0 is urgent, the clock process cannot make a transition which increases time.

4.3 Supporting Timestamps

Clocks have an important role to play in decidable real-time calculi, recording the moment an event takes place, relative to other clocks and the general progress of time. In Erlang the function `now()` returns the time elapsed since 00:00 GMT, January 1, 1970 as a tuple `{MegaSeconds,Seconds,MicroSeconds}`. Most Erlang programs that handle time simply call `now`, store the result somewhere, and later compare the old value with the current time. This programming discipline is reminiscent of the use of clocks in timed automata formalisms.

The discussion on how to obtain finite models with time is deferred to Sect. 5.1; however, a crucial requirement is that only a finite number of values returned from calls to `now()` are "alive" (i.e., accessible from some program variable) in any state. It would be possible to tailor a static analysis to track calls to `now()`, and the flow of the resulting values. Instead we provide the programmer with a new API to obtain time stamps, and to directly manipulate the lifetime of timestamps. Direct calls to `now()` are forbidden. In general adapting a program to use this new API is trivial; an example is provided below, and Sect. 6.2 contains a further discussion.

The new API located in the module `mce_erl_time` has the following functions:

`now()`	– returns the current time
`nowRef()`	– stores the current time in a clock reference
`was(Ref)`	– returns the time stored in a clock reference
`forget(Ref)`	– explicitly destroys an old clock reference

There are a number of restrictions on the use of this API. First, times obtained from calls to `now()` may never be remembered by the program, but only used in comparisons against previously recorded clocks. Moreover, for the soundness of model checking it is forbidden to compare a clock (or the current time) against an absolute time value, only relative comparisons are permitted, e.g., checking how much time has passed since some system event occurred. That is, it is not allowed to check whether `now()` returns some concrete date and time.

To illustrate the clock API we show below the coding of a fragment of the lamp example [23] of real-time calculi. A lamp is initially *off*, and when a button is pressed shines with *low* intensity. If a button is pressed again within 5 milliseconds, the lamp shines with *bright* intensity, otherwise if the new button press arrives later, the lamp is switched *off*. In the code fragment below we show the lamp controller, which receives button presses from a user, and operates the lamp hardware by sending messages to `PhysicalLamp`. The function `compareTimeStamps_ge(T1,T2)` returns true if `T1` is a later or identical timestamp as `T2`; `addTimeStamps(T1,T2)` computes a new time stamp which is the sum of its argument time stamps, and `milliSecondsToTimeStamp(N)` computes the time stamp corresponding to `N` milliseconds.

```
lamp(PhysicalLamp) ->
  receive
    press ->
      PressTime = mce_erl_time:nowRef(),
      PhysicalLamp!low,
      receive
        press ->
      case compareTimeStamps_ge
               (mce_erl_time:now(),
            addTimeStamps(milliSecondsToTimeStamp(5),
                          mce_erl_time:was(PressTime))) of
        true ->
          PhysicalLamp!off,
          mce_erl_time:forget(PressTime), ...;
            false ->
          PhysicalLamp!bright,
          mce_erl_time:forget(PressTime), ...
          end
      end
  end.
```

5 A Semi-formal Timed Semantics

In the following we describe how a timed semantics can be obtained by modifying an untimed semantics. We assume the presence of a non-hierarchical "global" structured operational semantics for Erlang states, which is the basis of the implementation of the McErlang model checker (unpublished work). For a non-global state, hierarchical, semantics of Erlang programs see [24,25].

In the untimed "global" semantics, an Erlang state s is, informally, a tuple $\langle \mathit{Nodes}, \mathit{Ether} \rangle$ consisting of a set of nodes (Nodes) and a datastructure (Ether) storing the messages in transit between nodes. To obtain a timed Erlang state we add the current time $\mathit{Time} \equiv \langle \mathit{MegaSeconds}, \mathit{Seconds}, \mathit{MicroSeconds} \rangle$ and a set of clocks, i.e., tuples $\langle \mathit{ClockId}, \mathit{Time} \rangle$ created by calling `nowRef()`, to the tuple: $\langle \mathit{Nodes}, \mathit{Ether}, \mathit{Time}, \mathit{Clocks} \rangle$. A node $\langle \mathit{Processes}, \mathit{Dictionary}, \mathit{Registry}, \mathit{Links}, \mathit{Monitors} \rangle$ is a collection of processes, a node global variable store, a process registry (for associating symbolic names to processes), and process links and monitors (for handling fault tolerance). Finally a process $\langle \mathit{Pid}, \mathit{Dictionary}, \mathit{Mailbox}, \mathit{Expr} \rangle$ has a mailbox, a process dictionary (an imperative memory), a unique process identifier, and the currently executing expression Expr. The contents of a typical Erlang system state is depicted symbolically in Fig. 8. Solid lines depict inter-node message passing, dashed lines intra-node message passing.

An Erlang untimed action is, informally, a side effect (e.g., a message sent between two processes, registering a symbolic name for a process, etc) or a process internal action. To the untimed actions we add the timed actions corresponding to timeouts, which cause time to progress, and actions corresponding to creating and modifying clocks. We let α range over the actions.

Given that we can compute the untimed transition relation written $s \xrightarrow{\alpha} s'$, which is defined as an structural operational semantics, we obtain the timed transition relation $s \xrightarrow[pre_t]{\alpha} s'$ by copying the transition rules from the untimed semantics *except* the rules for timeouts, and by adding a few transition rules concerning timeouts and clock handling to the timed semantics.

For timeout handling we add two rules:

$$\frac{\begin{array}{c} p \in \mathit{processes}(s) \qquad p.\mathit{expr} \equiv \texttt{receive}\ \mathit{clauses}\ \texttt{after}\ \mathit{deadline}\ \texttt{->}\ e\ \texttt{end} \\ \mathit{deadline} \neq \texttt{infinity} \qquad \mathit{absdeadline} = s.\mathit{time} + \mathit{deadline} \\ p' = p \text{ where } p'.\mathit{expr} = \texttt{absreceive}\ \mathit{clauses}\ \texttt{after}\ \mathit{absdeadline}\ \texttt{->}\ e\ \texttt{end} \end{array}}{s \xrightarrow[pre_t]{} s[p'/p]}$$

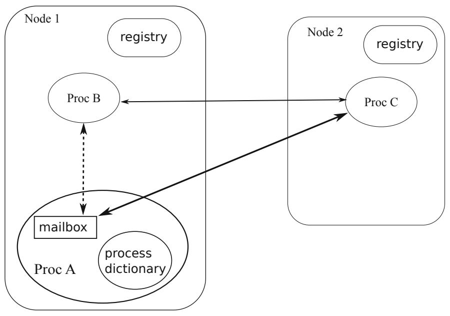


Fig. 8. An Erlang multi-node system

$$\frac{p \in \mathit{processes}(s) \qquad p.\mathit{expr} \equiv \texttt{absreceive}\ \mathit{clauses}\ \texttt{after}\ \mathit{absdeadline}\ \texttt{->}\ e\ \texttt{end} \quad \neg\mathit{receivable}(p) \quad s' = s \text{ where } s'.\mathit{time} = \mathit{absdeadline} \qquad p' = p \text{ where } p'.\mathit{expr} = e}{s \xrightarrow[\mathit{pre}_t]{\mathit{timeout}(\mathit{absdeadline})} s'[p'/p]}$$

The first side effect free rule simply replaces a time relative deadline with an absolute deadline; to clarify the semantics we introduce the new synthetic keyword `absreceive` for such time absolute receive statements.

In the second rule, in a state s, if there is a process p which is executing a receive statement, and which cannot receive a message, then there is a transition labelled by the action *timeout*(*deadline*) to a new state where the current time has increased, and where the currently executing expression of p has been replaced. Similar transition rules are added for handling clocks.

Finally we constrain the resulting transition relation to enforce global urgency constraints on the progress of time, using the following high-level Erlang function which first calculates the transitions for a state using the function `transitions`, and then returns a reduced set of transitions compatible with the notion of urgency introduced in Sect. 4.2.

```
timeRestrict(State) ->
  Now = State#state.time,
  Transitions = transitions(State),
  timeRestrict(Transitions,Now,[],infinity,[]).

timeRestrict([],_,Untimed,_,Timed) -> Untimed++Timed;
timeRestrict(Transitions,Now,Untimed,MostUrgent,Timed) ->
  [Transition|Rest] = Transitions,
  MinWait = calculate_minwait(Transition),
  MaxWait = calculate_maxwait(Transition),
  if
    MinWait==infinity orelse MinWait > MostUrgent ->
      timeRestrict(Rest,Now,Untimed,MostUrgent,Timed);
    MaxWait==infinity andalso MinWait==Now ->
      NewNonTimed = [Transition|NonTimed],
      timeRestrict(Rest,Now,NewNonTimed,MostUrgent,Timed);
    MinWait =< MaxWait < MostUrgent ->
      NewTimed = [Transition|restrict(MaxWait,TimerEntries)],
      timeRestrict(Rest,Now,NonTimed,MaxWait,Newtimed);
    MinWait =< MostUrgent =< MaxWait ->
      NewTimed = [Transition|TimerEntries],
      timeRestrict(Rest,Now,NonTimed,MostUrgent,NewTimed)
  end.

restrict(MaxWait,[]) -> [];
restrict(MaxWait,[Transition|Rest]) ->
  MinWait = calculate_minwait(Transition),
  if
    MinWait =< MaxWait -> [Transition|restrict(MaxWait,Rest)];
    MinWait > MaxWait -> restrict(MaxWait,Rest)
  end.
```

The `timeRestrict` function is called with the following arguments: a list of transitions which is reduced, the current system time (`Now`), a list of untimed transitions (`Untimed`), the time when the most urgent transition seen so far must be taken (`MostUrgent`), and a list of transitions which are enabled to be executed sometime before the `MostUrgent` deadline. The function first classifies a timed transition: informally the minimum waiting period is the timeout value in a receive statement, whereas the maximum waiting period is the urgency (specified using a call to `mce_erl:urgent`). Having `infinity` as the wait limit signifies that the process will wait forever.

5.1 Finite Models

To obtain finite models, i.e., finite state graphs, for timed programs, we note that the actions of a timed program normally depends only on the passage time, not on the absolute value of the time parameter of the system state. Similarly, in the specification logic we make statements only about the relative value of clocks and the system time parameter. Thus, for any given system state there is typically an infinite number of equivalent states, which differ only in that the time system parameter is distinct, but the relative values of clocks and the system time parameter is the same for all these “equivalent” states.

Thus, to obtain finite models, the obvious strategy is to normalize system states, and to generate the state space modulo such normalization. During state space generation, before adding a new node to the state graph, we should check whether its normalization (another state) is already in the graph. If it is, we do not need to consider the new state further. If it is not, we add the normalized state to the state graph, and continue exploring the behaviour of the new state. In [8] this procedure is referred to as model checking under symmetry.

It turns out to be trivial to add such a normalization to McErlang. The untimed McErlang tool already provided an “abstraction” feature, whereby a user-defined state abstraction function can be used to transform a state before storing it in the state table; exactly what is needed to implement time normalization. The normalization procedure modifies a state according to the following:

– The system time parameter is reset to `{0,0,0}`
– The values of clocks (created using `nowRef()`) are decreased by the old time
– Timeouts in (absolute) receive statements are decreased by the old time (but greater or equal to the new time)

As an example, the state graph for the program in Fig. 3, with the unbounded clock in Fig. 5, and using the above normalization, is depicted in Fig. 9. Note that states are no longer labeled by the current time, as the normalization collapses many states with distinct times into a single state.

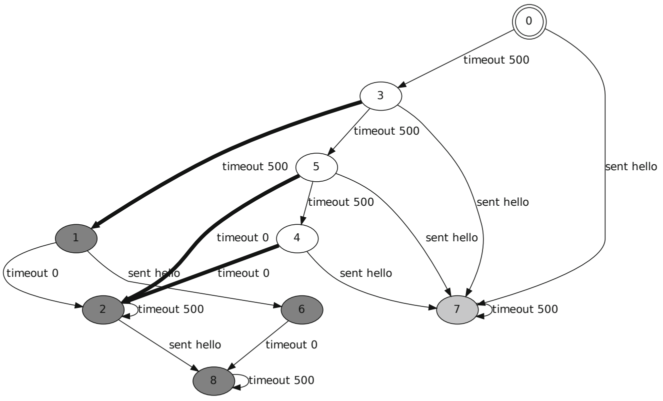


Fig. 9. State graph for the program in Fig. 4 with an explicit unbounded clock process, and finite model abstraction

Note that normalization does not guarantee finite state spaces. A program may for instance create an unbounded number of clock references, leading to an infinite state graph.

6 Experiments

To evaluate the resulting semantics and its implementation we below provide some initial benchmark results, and report on the challenges in obtaining a verifiable timed model from a crucial Erlang software component used in industry.

6.1 Efficiency

We evaluate the efficiency of the resulting implementation by checking Fischer's mutual exclusion algorithm [26]. Fig. 10 contains an implementation of the algorithm in Erlang. The global node dictionary extension of McErlang is used to implement reading and writing to the shared variable; see the functions `read` and `write`. The entering of the critical region of a process `Id` is indicated by a synthetic probe action `mce_erl:probe({enter,Id})`. The `latest(Tick,Wait,F)` function calls the function parameter `F` at most `Wait` milliseconds later; the time interval is partitioned into slices of maximum size `Tick` (the slices could be smaller if other clocks are defined).

We check mutual exclusion in a range of experiments characterized by the parameters of the `start(N,Tick,D,T)` function: `N` is the number of processes, `Tick` is the time tick, `D` is the maximum time to wait until writing to the shared variable, and `T` is the minimum time to wait until reading from the shared variable. Note that in this experiment we do not assume that internal actions are infinitely fast compared to timers.

Table 1. Execution times for Fischer's algorithm

N	Tick	D	T	Time (secs.)	Number of states
4	1	1	2	0.1s	2034
5	1	1	2	0.7s	13738
6	1	1	2	7.6s	89051
7	1	1	2	50.3s	580080
5	1	2	3	4.4s	81452
5	1	3	4	12.7s	268793
5	1	4	5	36.7s	704901
5	1	5	6	73.7s	1522179

As seen in Table 1 the size of the state space is exponential in the number of processes (`N`). In the last four rows the effect of an increase in the values of the timers is indicated. To verify the correctness of the algorithm a simple monitor checks that `enter` and `exit` probe actions strictly alternate. As expected, if writing is slower than reading, e.g., `D` > `T`, the algorithm works correctly and otherwise a counterexample is quickly found. Overall, the size of state spaces and the execution times are reasonable.

6.2 Expressive Power

As a final example we consider the verification of the nos_supervisor library [21]; this is a crucial software component used in several industrial projects at the LambdaStream company [27].

A supervisor is a process in charge of starting, stopping and monitoring a set of children (processes). Basically whenever a child process terminates the supervisor should restart it, i.e., spawn a new process executing the task of the terminated child. A supervisor typically supervises not only process workers, but also other supervisors, defining a hierarchical structure as shown in Fig. 11.

```
start(N,Tick,D,T) ->
  write(0),
  lists:foreach
    (fun (Id) -> spawn(fun () -> idle(Id,Tick,D,T) end) end,
     lists:seq(1,N)).

idle(Id,Tick,D,T) ->
  case read() of
    0 -> set(Id,Tick,D,T);
    _ -> idle(Id,Tick,D,T)
  end.

set(Id,Tick,D,T) ->
  latest(Tick,D,fun () -> setting(Id,Tick,D,T) end).

setting(Id,Tick,D,T) ->
  write(Id),
  sleep(T),
  testing(Id,Tick,D,T).

testing(Id,Tick,D,T) ->
  case read() of
    Id -> mutex(Id,Tick,D,T);
    _ -> idle(Id,Tick,D,T)
  end.

mutex(Id,Tick,D,T) ->
  mce_erl:probe({enter,Id}),
  write(0),
  mce_erl:probe({exit,Id}),
  idle(Id,Tick,D,T).

read() ->
  case mcerlang:nget(id) of
    N when is_integer(N),N>=0 -> N
  end.

write(V) ->
  mcerlang:nput(id,V).

%% Support code

sleep(Milliseconds) ->
  receive after Milliseconds -> ok end.

latest(_Tick,0,F) ->
  mce_erl:urgent(0), F();
latest(Tick,Wait,F) ->
  mce_erl:urgent(0),
  mce_erl:choice
    ([fun () -> mce_erl:urgent(0), F() end,
      fun () ->
        mce_erl:urgent(0),
        receive after Tick -> latest(Tick,Wait-Tick,Fun) end
      end]).
```

Fig. 10. Fischer's mutual exclusion algorithm in Erlang

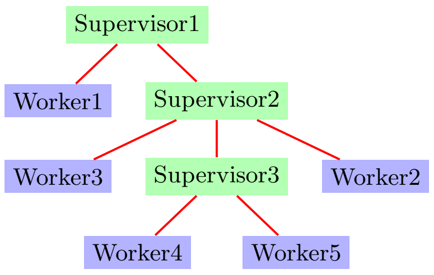


Fig. 11. A supervision tree

The nos_supervisor is implemented in around 760 lines of Erlang code. To enable untimed verification of part of its functionality in [21] we had to modify the functions below:

```
%% Check if restarting a child again is admissible
add_restart(#child_spec{restart_intensity = infinity}) -> [];
add_restart(Spec=#child_spec{state = ChildState}) ->
  {MaxR,MaxT,Final} = Spec#child_spec.restart_intensity,
  Restarts = ChildState#child_state.restarts,
  check_restarts(MaxR, Final,
                 filter_restarts(MaxT, [now() | Restarts])).

%% Remove restarts older than MaxT
filter_restarts(MaxT, [H | Restarts]) ->
  F = fun(Restart) -> difference(Restart, H) < MaxT end,
  [H | lists:takewhile(F, Restarts)].

check_restarts(MaxR, Final, Restarts) ->
  case length(Restarts) > MaxR of
    true -> Final;
    false -> Restarts
  end.
```

These functions define the restarting policies of the supervisor. The function `add_restart` is called when a child process should be restarted due to having terminated abnormally. However, a constraint on restarting is that the child process may not have been restarted more than `MaxR` times within `MaxT` seconds. If this constraint is violated, the supervisor terminates all child processes and then itself.

Correctness properties are specified as safety monitors that inspect the actions of a the supervisor and processes interacting with the supervisor; see [21] for details. Using untimed McErlang we were not able to verify time dependent properties for the supervisor component, but had to resort to expressing the timing checks as a nondeterministic choice.

Using timed McErlang there is no need to change any of the 760 lines of code, although a few lines of code had to be added to delete the time clocks created using `nowRef()`. Of course we still need to create verification scenarios that explore the behaviour of the supervisor in detail, i.e., programming child process terminating abnormally and being restarted in narrow time intervals.

7 Conclusions

We have implemented a timed semantics for the Erlang programming language in the McErlang model checker, and have demonstrated that the resulting tool is capable of verifying timed systems. Compared to other similar semantics our semantics has a few interesting characteristics such as e.g. the absence of a dedicated time tick.

Currently the timed implementation is undergoing a study as to its suitability as a workbench for analysing Timed Rebeca programs. Earlier work [28] has implemented a translation from Timed Rebeca to Erlang, and has used untimed McErlang to simulate and test the resulting programs against correctness properties specified as safety monitors. In recent work, the new timed McErlang model checker, and the language extensions to specify urgency and clocks, are being used to verify Timed Rebeca programs.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. TCS 126, 183–235 (1994)
2. Ouaknine, J.: Discrete analysis of continuous behaviour in real-time concurrent systems. PhD thesis, Oxford University (2001)
3. Moller, F., Tofts, C.M.N.: Behavioural Abstraction in TCCS. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 559–570. Springer, Heidelberg (1992)
4. Hansson, H., Jonsson, B.: A calculus for communicating systems with time and probabitilies. In: IEEE Real-Time Systems Symposium, pp. 278–287 (1990)
5. Léonard, L., Leduc, G.: A formal definition of time in LOTOS. Formal Asp. Comput. 10(3), 248–266 (1998)
6. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT 1(1-2) (1997)
7. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23, 279–295 (1997)
8. Lamport, L.: Real-Time Model Checking Is Really Simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
9. Wang, H., MacCaull, W.: Verifying real-time systems using explicit-time description methods. In: Andova, S., McIver, A., D'Argenio, P.R., Cuijpers, P.J.L., Markovski, J., Morgan, C., Núñez, M. (eds.) QFM. EPTCS, vol. 13 (2009)
10. van den Berg, L., Strooper, P.A., Winter, K.: Introducing Time in an Industrial Application of Model-Checking. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 56–67. Springer, Heidelberg (2008)
11. Fredlund, L.Å., Svensson, H.: McErlang: a model checker for a distributed functional programming language. In: Proceeding of the 12th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP). ACM, Freiburg (2007)
12. Guo, Q., Derrick, J., Hoch, C.: Verifying Erlang Telecommunication Systems with the Process Algebra μCRL. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 201–217. Springer, Heidelberg (2008)

13. Guo, Q., Derrick, J.: Verification of timed Erlang/OTP components using the process algebra mucrl. In: Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, pp. 55–64 (2007)
14. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice-Hall (1996)
15. Cesarini, F., Thompson, S.: Erlang Programming – A Concurrent Approach to Software Development. O'Reilly Media (2009)
16. Blau, S., Rooth, J., Axell, J., Hellstrand, F., Buhrgard, M., Westin, T., Wicklund, G.: AXD 301: A new generation ATM switching system. Computer Networks 31(6), 559–582 (1999)
17. Wiger, U., Ask, G., Boortz, K.: World-class product certification using Erlang. SIGPLAN Not. 37, 25–34 (2002)
18. McErlang: web page (April 2012), https://babel.ls.fi.upm.es/trac/McErlang/
19. Fredlund, L.-å., Sánchez Penas, J.J.: Model Checking a Video–on–Demand Server Using McErlang. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 539–546. Springer, Heidelberg (2007)
20. Benac Earle, C., Fredlund, L.-Å., Iglesias, J.A., Ledezma, A.: Verifying Robocup Teams. In: Peled, D.A., Wooldridge, M.J. (eds.) MoChArt 2008. LNCS, vol. 5348, pp. 34–48. Springer, Heidelberg (2009)
21. Castro, D., Gulías, V.M., Benac Earle, C., Fredlund, L.Å., Rivas, S.: A case study on verifying a supervisor component using McErlang. ENTCS 271, 23–40 (2011)
22. (April 2012), https://github.com/fredlund/McErlang-DTime
23. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
24. Fredlund, L.Å.: A Framework for Reasoning about Erlang Code. PhD thesis, Royal Institute of Technology, Stockholm, Sweden (2001)
25. Svensson, H., Fredlund, L.Å.: A more accurate semantics for distributed Erlang. In: Proc. of the SIGPLAN Workshop on Erlang. ACM, New York (2007)
26. Gafni, E., Mitzenmacher, M.: Analysis of timing-based mutual exclusion with random times. In: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 13–21. ACM Press (1999)
27. LambdaStream, S.L.: web page (April 2012), http://www.lambdastream.com/
28. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. In: FOCLASA. EPTCS, vol. 58, pp. 1–19 (2011)

Author Index

Agha, Gul 219
Albert, Elvira 35
Aman, Bogdan 52

Barthe, Gilles 186
Beck, Wim 152
Benac Earle, Clara 251
Bensalem, Saddek 118
Bozga, Marius 118

Cerone, Andrea 135
Ciobanu, Gabriel 52
Clarke, Dave 152
Crespo, Juan Manuel 186

de Frutos Escrig, David 169
Delahaye, Benoît 203
Delzanno, Giorgio 235
Desel, Jörg 67
Devriese, Dominique 186

Fahrenberg, Uli 203
Flores-Montoya, Antonio E. 35
Fredlund, Lars-Åke 251

Genaim, Samir 35

Hennessy, Matthew 135
Henzinger, Thomas A. 203

Johnson, Taylor T. 18
Junttila, Tommi 84

Karmani, Rajesh K. 219
Kindermann, Roland 84
Klai, Kais 67
Koutny, Maciej 52

Lanese, Ivan 1
Lauterburg, Steven 219
Legay, Axel 203, 219
Lienhardt, Michael 1

Marinov, Darko 219
Mezzina, Claudio Antares 1
Milushev, Dimiter 152
Mitra, Sayan 18

Ničković, Dejan 203
Niemelä, Ilkka 84

Piessens, Frank 186
Prabhakar, Pavithra 101

Quilbeuf, Jean 118

Rivas, Exequiel 186
Romero Hernández, David 169

Sangnier, Arnaud 235
Sifakis, Joseph 118
Stefani, Jean-Bernard 1

Tasharofi, Samira 219

Viswanathan, Mahesh 101

Zavattaro, Gianluigi 235