Mario Bravetti
Gianluigi Zavattaro (Eds.)

LNCS 5710

CONCUR 2009 – Concurrency Theory

20th International Conference, CONCUR 2009
Bologna, Italy, September 2009
Proceedings

Springer

Mario Bravetti   Gianluigi Zavattaro (Eds.)

CONCUR 2009 - Concurrency Theory

20th International Conference, CONCUR 2009
Bologna, Italy, September 1-4, 2009
Proceedings

Springer

Volume Editors

Mario Bravetti
University of Bologna, Department of Computer Science
Mura A. Zamboni 7, 40127 Bologna, Italy
E-mail: bravetti@cs.unibo.it

Gianluigi Zavattaro
University of Bologna, Department of Computer Science
Mura A. Zamboni 7, 40127 Bologna, Italy
E-mail: zavattar@cs.unibo.it

Library of Congress Control Number: 2009932911

CR Subject Classification (1998): D.1.3, D.4.1, F.1.2, H.2.4, D.3, C.2.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN	0302-9743
ISBN-10	3-642-04080-2 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-04080-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12734689 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 20th Conference on Concurrency Theory (CONCUR 2009), held in Bologna, September 1–4, 2009. The purpose of the CONCUR conference is to bring together researchers, developers, and students in order to advance the theory of concurrency and promote its applications. This year the CONCUR conference was in its 20th edition, and to celebrate 20 years of CONCUR, the conference program included a special session organized by the IFIP Working Groups 1.8 "Concurrency Theory" and 2.2 "Formal Description of Programming Concepts" as well as an invited lecture given by Robin Milner, one of the fathers of the concurrency theory research area.

This edition of the conference attracted 129 submissions. We wish to thank all their authors for their interest in CONCUR 2009. After careful discussions, the Program Committee selected 37 papers for presentation at the conference. Each of them was accurately refereed by at least three reviewers (four reviewers for papers co-authored by members of the Program Committee), who delivered detailed and insightful comments and suggestions. The conference Chairs warmly thank all the members of the Program Committee and all their sub-referees for the excellent support they gave, as well as for the friendly and constructive discussions. We would also like to thank the authors for having revised their papers to address the comments and suggestions by the referees.

The conference program was enriched by the outstanding invited talks by Martin Abadi, Christel Baier, Corrado Priami and, as mentioned above, Robin Milner.

The conference this year was jointly organized with the 7th International Conference on Computational Methods in Systems Biology (CMSB 2009) and the 6th International Workshop on Web Service and Formal Methods (WS-FM 2009) emphasizing, on the one hand, the close connections and similarities between concurrent, artificial systems, and biological, natural systems and, on the other hand, the current pervasiveness of concurrent, distributed and mobile computing technologies. The invited talk by Corrado Priami, held in conjunction with CMSB, and the one by Robin Milner, held in conjunction with WS-FM, were a witness of the commonalities of interests of these conferences and of the corresponding research communities. As additional co-located events, CONCUR 2009 included the following satellite workshops: 16th International Workshop on Expressiveness in Concurrency (EXPRESS), Second Interaction and Concurrency Experience (ICE), 11th International Workshop on Verification of Infinite-State Systems (INFINITY), Third Workshop on Membrane Computing and Biologically Inspired Process Calculi (MeCBIC), 7th International Workshop on Security Issues in Concurrency (SecCo), 6th Workshop on Structural Operational Semantics (SOS) and Young Researchers Workshop on Concurrency Theory (YR-CONCUR).

We would like to thank all the people who contributed to the organization of CONCUR 2009, and the generous support from the Alma Mater Studiorum – Università degli Studi di Bologna and from Microsoft Research Cambridge. We are also grateful to Andrei Voronkov, who allowed us to use the wonderful free conference software system EasyChair, which we used for the electronic submission of papers, the refereeing process and the Program Committee work.

September 2009

Mario Bravetti
Gianluigi Zavattaro

Organization

Steering Committee

Roberto Amadio	Université Paris Diderot, France
Jos C.M. Baeten	Eindhoven University of Technology, The Netherlands
Eike Best	Carl von Ossietzky University of Oldenburg, Germany
Kim Larsen	Aalborg University, Denmark
Ugo Montanari	University of Pisa, Italy
Scott Smolka	SUNY at Stony Brook, USA

Program Committee

Luca Aceto	Reykjavik University, Iceland
Jos C.M. Baeten	Eindhoven University of Technology, The Netherlands
Michele Boreale	University of Florence, Italy
Mario Bravetti (Co-chair)	University of Bologna, Italy
Luís Caires	Universidade Nova de Lisboa, Portugal
Philippe Darondeau	INRIA Rennes - Bretagne Atlantique, France
Wan Fokkink	Vrije Universiteit Amsterdam, The Netherlands
Cédric Fournet	Microsoft Research, Cambridge, UK
Robert J. van Glabbeek	Kensington Res. Lab., Sydney, Australia
Matthew Hennessy	Trinity College Dublin, Ireland
Holger Hermanns	Saarland University, Germany
Thomas Hildebrandt	IT University of Copenhagen, Denmark
Jane Hillston	University of Edinburgh, UK
Barbara König	University of Duisburg-Essen, Germany
François Laroussinie	Université Paris Diderot-Paris 7, France
Ugo Montanari	University of Pisa, Italy
Uwe Nestmann	Technische Universität Berlin, Germany
Manuel Núñez	Universidad Complutense de Madrid, Spain
Catuscia Palamidessi	Ecole Polytechnique, Palaiseau, France
Joachim Parrow	Uppsala University, Sweden
Shaz Qadeer	Microsoft Research, Redmond, USA
Julian Rathke	University of Southampton, UK
Philippe Schnoebelen	Ecole Normale Superieure Cachan, France
Nobuko Yoshida	Imperial College London, UK
Gianluigi Zavattaro (Co-chair)	University of Bologna, Italy

Organizing Committee

Mario Bravetti (Co-chair)
Cinzia Di Giusto
Ivan Lanese (Workshops Chair)
Fabrizio Montesi
Jorge A. Perez
Sylvain Pradalier
Cristian Versari
Antonio Vitale
Gianluigi Zavattaro (Co-chair)

Additional Reviewers

Lucia Acciai
Luca de Alfaro
Austin Anderson
Suzana Andova
Jesus Aranda
Mohamed Faouzi Atig
Eric Badouel
Christel Baier
Massimo Bartoletti
Nick Benton
Beatrice Berard
Josh Berdine
Jasper Berendsen
Bernard Berthomieu
Nathalie Bertrand
Dietmar Berwanger
Karthikeyan Bhargavan
Christoph Blume
Frank de Boer
Benedikt Bollig
Filippo Bonchi
Johannes Borgstrom
Ahmed Bouajjani
Anne Bouillard
Patricia Bouyer
Laura Bozzelli
Andrea Bracciali
Aaron Bradley
Jeremy Bradley
Thomas Brihaye
Niklas Broberg
H.J. Sander Bruggink
Roberto Bruni
Michele Bugliesi
Mikkel Bundgaard
Marzia Buscemi
Diletta R. Cacciagrano
Silvio Capobianco
Arnaud Carayol
Marco Carbone
Franck Cassez
Diego Cazorla
Krishnendu Chatterjee
Konst. Chatzikokolakis
Han Chen
Taolue Chen
Stefano Chessa
Matteo Cimini
Gabriel Ciobanu
David Clark
Thomas Colcombet
Mario Coppo
Ricardo Corin
Pieter Cuijpers
Pedro D'Argenio
Arnaud Dacosta-Lopes
Mads Dam
Malo Danielou
Alexandre David
Christian Dax
Søren Debois
Giorgio Delzanno
Stéphane Demri
Yuxin Deng
Rocco De Nicola
Liliana D'Errico
Gregorio Diaz
Cinzia Di Giusto
Dino Distefano
Christian Eisentraut
Tayfun Elmas
Zoltan Esik
Azadeh Farzan
Carla Ferreira
Bernd Finkbeiner
Alain Finkel
Dana Fisman
Riccardo Focardi
Adrian Francalanza
Laurent Fribourg
Sibylle Fröschle
David de Frutos-Escrig
Murdoch Gabbay
Maurizio Gabbrielli
Fabio Gadducci
Vashti Galpin
Deepak Garg
Blaise Genest
Sonja Georgievska
Prodromos Gerakios
Nils Gesbert
Fatemeh Ghassemi
Stephen Gilmore

Hugo Gimbert
Stefania Gnesi
Jens Chr. Godskesen
Thomas Goethel
Ganesh Gopalakrishnan
Daniele Gorla
Carlos Gregorio-Rodriguez
Marcus Groesser
Davide Grohmann
Claudio Guidi
Aarti Gupta
Serge Haddad
Ernst Moritz Hahn
Magnús Halldórsson
Tingting Han
Arnd Hartmanns
Tobias Heindel
Loïc Hélouët
Tom Hirschowitz
Florian Horn
Mathieu Hoyrup
Mathias Huelsbusch
Atsushi Igarashi
Radu Iosif
Petr Jancar
Thierry Jeron
Mathias John
Bengt Jonsson
Marcin Jurdzinski
Vineet Kahlon
Joost-Pieter Katoen
Shin-ya Katsumata
Klaus Keimel
Nicholas Kidd
Martin Kot
Vasileios Koutavas
Lars Kristensen
Antonín Kučera
Orna Kupferman
Dietrich Kuske
Jim Laird
Akash Lal
Ivan Lanese
Diego Latella
Axel Legay
Jérôme Leroux
Cedric Lhoussaine
Luis Llana
Markus Lohrey
Natalia López
Michele Loreti
Michael Luttenberger
Bas Luttik
Tiejun Ma
Hermenegilda Macia
Sergio Maffeis
Rudolf Mak
Nicolas Markey
Richard Mayr
Larissa Meinicke
Hernan Melgratti
Massimo Merro
Antoine Meyer
Marino Miculan
Kees Middelburg
Paolo Milazzo
Alice Miller
Dale Miller
Samuel Mimram
Faron Moller
David Monniaux
Alberto Montresor
Arjan Mooij
Carroll Morgan
Remi Morin
Christophe Morvan
Dejan Ničković
Joachim Niehren
Johan Nordlander
Gethin Norman
Carlos Olarte
Luca Padovani
Nikolaos Papaspyrou
David Parker
Dirk Pattinson
Romain Pechoux
Fernando L. Pelayo
Wojciech Penczek
Kirstin Peters
Sophie Pinchinat
Nir Piterman
Nuno Preguiça
Cristian Prisacariu
Riccardo Pucella
Willard Thor Rafnsson
Michel Reniers
Arend Rensink
Bernhard Reus
Pierre-Alain Reynier
Ahmed Rezine
Ismael Rodriguez
Christian Rohner
Cristobal Rojas
Fernando Rosa-Velardo
Jan Rutten
Arnaud Sangnier
Vladimiro Sassone
Jens-Wolfhard Schicke
Ina Schieferdecker
Sven Schneider
Stefan Schwoon
Peter Sewell
Ali Sezgin
Natalia Sidorova
Marjan Sirjani
A. Prasad Sistla
Michael Smith
Pawel Sobocinski
Ana Sokolova
Michael Spear
Jeremy Sproston
Sam Staton
Rob van Stee
Grégoire Sutre
Nikhil Swamy
Grzegorz Szubzda
Tachio Terauchi
Francesco Tiezzi
Tayssir Touili
Nikola Trcka
Richard Trefler
Yih-Kuen Tsay
Emilio Tuosto
Frank Valencia
Miguel Valero

Valentin Valero	Walter Vogler	Ralf Wimmer
Daniele Varacca	Marc Voorhoeve	Dominik Wojtczak
Enrico Vicario	Edsko de Vries	Nicolas Wolovick
Björn Victor	Igor Walukiewicz	James Worthington
Hugo Vieira	Andrzej Wasowski	Hans Zantema
Maria Grazia Vigliotti	Adam Welc	Lijun Zhang
Erik de Vink	Tim Willemse	Roberto Zunino

Table of Contents

Perspectives on Transactional Memory

Martín Abadi[1,2] and Tim Harris[1]

[1] Microsoft Research
[2] University of California, Santa Cruz

Abstract. We examine the role of transactional memory from two perspectives: that of a programming language with atomic actions and that of implementations of the language. We argue that it is difficult to formulate a clean, separate, and generally useful definition of transactional memory. In both programming-language semantics and implementations, the treatment of atomic actions benefits from being combined with that of other language features. In this respect (as in many others), transactional memory is analogous to garbage collection, which is often coupled with other parts of language runtime systems.

1 Introduction

The name "transactional memory" [21] suggests that a transactional memory (TM) is something similar to an ordinary memory, though perhaps with a slightly different interface and different properties. In particular, the interface would include means of initiating and committing transactions, as well as means of performing memory accesses. These memory accesses may be within transactions, and perhaps also outside transactions. The interface may provide other operations for aborting transactions, for delaying their execution, or for nesting them in various ways. As for the properties, we would expect certain guarantees that differentiate TM from ordinary memory. These properties should include, in particular, that all memory accesses within a successful transaction appear atomic.

Some interesting recent research aims to define TM more precisely, along these lines [9, 16–19, 29, 32]. TM may be modeled as a shared object that supports operations such as read, write, begin-transaction, and commit-transaction, with requirements on the behavior of these operations. Some of this research also examines particular implementations, and whether or not they satisfy those requirements.

Another recent line of research studies programming-language constructs that may be built over TM—typically `atomic` blocks [25] or other constructs for atomic actions [3, 6, 22]. A variety of semantics have been provided at different levels of abstraction. Some semantics model atomic actions that execute without the interleaving of operations of other threads. We call these "strong" semantics. Other semantics model low-level details of common implementations, such as conflict-detection mechanisms and roll-backs.

Despite encouraging progress, much work remains. In particular, the research to date does not fully explore the relation between the first style of definition (where TM is a shared object) and the second style of definition (modeling language constructs rather than TM *per se*).

In this paper, we argue that these gaps are not surprising, nor necessarily bad. Indeed, we find limiting the view that a TM is something similar to a memory, with a slightly different interface and properties. Although this view can sometimes be reasonable and useful, in many settings a clear delineation of TM as a separate, memory-like object is neither necessary nor desirable.

We consider TM from two perspectives: that of the programming language with atomic actions and that of the implementation of the atomicity guarantees.

– From the former perspective, we are primarily interested in the possibility of writing correct, efficient programs. The syntax and semantics of these programs may reflect transactional guarantees (for instance, by including `atomic` blocks), but they need not treat TM as a separate object. Indeed, the language may be designed to permit a range of implementations, rather than just those based on TM.
– From the latter perspective, we are interested in developing efficient implementations of programming languages. Implementations may fruitfully combine the TM with other aspects of a runtime system and with static program analysis, thus offering stronger guarantees at a lesser cost.

These two perspectives are closely related, and some of the same arguments appear from both perspectives.

Despite these reservations, we do recognize that, sometimes, a clear delineation of TM is possible and worthwhile. We explore how this approach applies in some simple language semantics, in Section 2. In Section 3, we consider the difficulties of extending this approach, both in the context of more sophisticated semantics and in actual implementations. We argue that it is best, and perhaps inevitable, to integrate the TM into the semantics and the implementations. In Section 4, we examine the question of the definition of TM through the lens of the analogy with garbage collection. We conclude in Section 5.

2 Transactional Memory in Semantics: A Simple Case

In our work, we have defined various semantics with atomicity properties [3, 5, 6]. Some of the semantics aim to capture a programmer's view of a language with high-level atomicity guarantees. Other semantics are low-level models that include aspects of implementations, for instance logs for undoing eager updates made by transactions and for detecting conflicts between concurrent transactions. Moore and Grossman [25] have defined some analogous semantics for different languages. Remarkably, although all these semantics specify the behavior of programs, none of them includes a separate definition of TM. Rather, the TM is closely tied to the rest of the semantics.

$$
\begin{array}{rcl}
b \in \text{BExp} & = & \ldots \\
e \in \text{NExp} & = & \ldots \\
C, D \in \text{Com} & = & \texttt{skip} \\
 & | & x := e \qquad (x \in \text{Vars}) \\
 & | & C; D \\
 & | & \texttt{if } b \texttt{ then } C \texttt{ else } D \\
 & | & \texttt{while } b \texttt{ do } C \\
 & | & \texttt{async } C \\
 & | & \texttt{unprotected } C \\
 & | & \texttt{block}
\end{array}
$$

Fig. 1. Syntax

In this section we illustrate, through a simple example, the style of those semantics. We also consider and discuss a variant in which TM is presented more abstractly and separately.

More specifically, we consider a simple imperative language and an implementation with transaction roll-back. This language omits many language features (such as memory allocation) and implementation techniques (such as concurrent execution of transactions). It is a fragment of the AME calculus [3], and a small extension (with `unprotected` sections) of a language from our previous work [6]. Both the high-level semantics of the language (Section 2.2) and a first version with roll-back (Section 2.3) treat memory as part of the execution state, with no separate definition of what it means to be a correct TM. On the other hand, a reformulation of the version with roll-back (Section 2.4) separates the semantics of language constructs from the specification of TM.

2.1 A Simple Language

The language that we consider is an extension of a basic imperative language, with a finite set of variables Vars, whose values are natural numbers, and with assignments, sequencing, conditionals, and while loops (IMP [36]). Additionally, the language includes constructs for co-operative multi-threading:

– A construct for executing a command in an asynchronous thread. Informally, `async` C forks off the execution of C. This execution is asynchronous, and will not happen if the present thread keeps running without ever yielding control, or if the present thread blocks without first yielding control. The execution of C will be atomic until C yields control, blocks, or terminates.
– A construct for running code while allowing preemption at any point. Informally, `unprotected` C yields control, then executes C without guaranteeing C's atomicity, and finally yields control again.
– A construct for blocking. Informally, `block` halts the execution of the entire program.

We define the syntax of the language in Figure 1. We do not detail the usual constructs on numerical expressions, nor those for boolean conditions.

$$\begin{array}{lll}
\langle \sigma, T, \mathcal{E}[x := e] \rangle & \longrightarrow \langle \sigma[x \mapsto n], T, \mathcal{E}[\mathtt{skip}] \rangle & \text{if } \sigma(e) = n \\
\langle \sigma, T, \mathcal{E}[\mathtt{skip}; C] \rangle & \longrightarrow \langle \sigma, T, \mathcal{E}[C] \rangle & \\
\langle \sigma, T, \mathcal{E}[\mathtt{if}\ b\ \mathtt{then}\ C\ \mathtt{else}\ D] \rangle & \longrightarrow \langle \sigma, T, \mathcal{E}[C] \rangle \quad \text{if } \sigma(b) = \mathtt{true} & \\
\langle \sigma, T, \mathcal{E}[\mathtt{if}\ b\ \mathtt{then}\ C\ \mathtt{else}\ D] \rangle & \longrightarrow \langle \sigma, T, \mathcal{E}[D] \rangle \quad \text{if } \sigma(b) = \mathtt{false} & \\
\langle \sigma, T, \mathcal{E}[\mathtt{while}\ b\ \mathtt{do}\ C] \rangle & \longrightarrow \langle \sigma, T, \mathcal{E}[\mathtt{if}\ b\ \mathtt{then}\ \ldots\ \mathtt{else}\ \ldots] \rangle & \\
\langle \sigma, T, \mathcal{E}[\mathtt{async}\ C] \rangle & \longrightarrow \langle \sigma, T.C, \mathcal{E}[\mathtt{skip}] \rangle & \\
\langle \sigma, T, \mathcal{E}[\mathtt{unprotected}\ C] \rangle & \longrightarrow \langle \sigma, T.\mathcal{E}[\mathtt{unprotected}\ C], \mathtt{skip} \rangle & \\
\langle \sigma, T, \mathcal{E}[\mathtt{unprotected}\ \mathtt{skip}] \rangle & \longrightarrow \langle \sigma, T.\mathcal{E}[\mathtt{skip}], \mathtt{skip} \rangle & \\
\langle \sigma, T.C, \mathtt{skip} \rangle & \longrightarrow \langle \sigma, T, C \rangle &
\end{array}$$

Fig. 2. Transition rules of the abstract machine

2.2 High-Level Strong Semantics

A first semantics for our language is given in terms of small-step transitions between states. A state $\langle \sigma, T, C \rangle$ consists of the following components:

- a store σ, which is a mapping of the finite set Vars of variables to the set of natural numbers;
- a finite multiset of commands T, which we call the thread pool;
- a distinguished active command C.

We write $\sigma[x \mapsto n]$ for the store that agrees with σ except at x, which is mapped to n. We write $\sigma(b)$ for the boolean denoted by b in σ, and $\sigma(e)$ for the natural number denoted by e in σ. We write $T.C$ for the result of adding C to T. As usual, a context is an expression with a hole [ ], and an evaluation context is a context of a particular kind. Given a context $\mathcal{C}$ and a command C, we write $\mathcal{C}[C]$ for the result of placing C in the hole in $\mathcal{C}$. We use the evaluation contexts defined by the grammar:

$$\mathcal{E} = [\ ] \mid \mathcal{E}; C \mid \mathtt{unprotected}\ \mathcal{E}$$

Figure 2 gives rules that specify the transition relation (eliding straightforward details for `while` loops). According to these rules, when the active command is `skip`, a command from the pool becomes the active command. It is then evaluated as such until it produces `skip`, yields, or blocks. No other computation is interleaved with this evaluation. When the active command is not `skip`, each evaluation step produces a new state, determined by decomposing the active command into an evaluation context and a subexpression. Yielding happens when this subexpression is a command of the form `unprotected` C.

This semantics is a strong semantics in the sense that any `unprotected` sections in the thread pool will not run while an active command is running and does not yield.

2.3 A Lower-Level Semantics with Roll-Back

A slightly lower-level semantics allows roll-back at any point in a computation. (Roll-back may make the most sense when the active command is blocked, but

$$
\begin{array}{lll}
\langle S, \sigma, T, \mathcal{E}[x := e]\rangle & \longrightarrow \langle S, \sigma[x \mapsto n], T, \mathcal{E}[\texttt{skip}]\rangle & \text{if } \sigma(e) = n \\
\langle S, \sigma, T, \mathcal{E}[\texttt{skip}; C]\rangle & \longrightarrow \langle S, \sigma, T, \mathcal{E}[C]\rangle & \\
\langle S, \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D]\rangle & \longrightarrow \langle S, \sigma, T, \mathcal{E}[C]\rangle & \text{if } \sigma(b) = \texttt{true} \\
\langle S, \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D]\rangle & \longrightarrow \langle S, \sigma, T, \mathcal{E}[D]\rangle & \text{if } \sigma(b) = \texttt{false} \\
\langle S, \sigma, T, \mathcal{E}[\texttt{while } b \texttt{ do } C]\rangle & \longrightarrow \langle S, \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } \ldots \texttt{ else } \ldots]\rangle & \\
\langle S, \sigma, T, \mathcal{E}[\texttt{async } C]\rangle & \longrightarrow \langle S, \sigma, T.C, \mathcal{E}[\texttt{skip}]\rangle & \\
\langle S, \sigma, T, \mathcal{E}[\texttt{unprotected } C]\rangle & \longrightarrow \langle S, \sigma, T.\mathcal{E}[\texttt{unprotected } C], \texttt{skip}\rangle & \\
\langle S, \sigma, T, \mathcal{E}[\texttt{unprotected skip}]\rangle & \longrightarrow \langle S, \sigma, T.\mathcal{E}[\texttt{skip}], \texttt{skip}\rangle & \\
\langle S, \sigma, T.C, \texttt{skip}\rangle & \longrightarrow \langle\langle \sigma, T.C\rangle, \sigma, T, C\rangle & \\
\langle\langle \sigma_0, T_0\rangle, \sigma, T, C\rangle & \longrightarrow \langle\langle \sigma_0, T_0\rangle, \sigma_0, T_0, \texttt{skip}\rangle &
\end{array}
$$

Fig. 3. Transition rules of the abstract machine, with roll-back

it is convenient to allow roll-back at any point.) For this purpose, the semantics relies on extended states $\langle\langle \sigma_0, T_0\rangle, \sigma, T, C\rangle$ with two additional components: an extra store σ_0 and an extra thread pool T_0. Basically, the current σ and T are saved as σ_0 and T_0 when a transaction starts, and restored upon roll-back. Figure 3 gives the rules of the semantics. Only the last two rules operate on the additional state components.

Our work and that of Moore and Grossman include more elaborate semantics with roll-back [3, 5, 25]. Those semantics model finer-grain logging; in them, roll-back is not a single atomic step. Some of the semantics are weak, in the sense that `unprotected` sections may execute while transactions are in progress, and even during the roll-back of transactions. We return to this complication in Section 3.2, where we also consider concurrency between transactions.

2.4 Separating the Transactional Memory

Figure 4 presents a reformulation of the semantics of Section 2.3. States are simplified so that they consist only of a thread pool and an active command. Memory is treated through labels on the transition relation. These labels indicate any memory operations, and also the start and roll-back of atomic computations. The labels for start and roll-back include a thread pool (which could probably be omitted if thread pools were tracked differently). The commit-point of atomic computations can remain implicit.

A separate definition can dictate which sequences μ of labels are legal in a computation $\langle T, \texttt{skip}\rangle \longrightarrow^*_\mu \langle T', \texttt{skip}\rangle$. This definition may be done axiomatically. One of the axioms may say, for instance, that for each label **back**T in μ there is a corresponding, preceding label **start**T, with the same T, and with no intervening other **start** or **back** label. Another axiom may constrain reads and writes, and imply, for example, that the sequence $[x \mapsto 1][x = 2]$ is not legal. Although such axiomatic definitions can be elegant, they are both subtle and error-prone. Alternatively, the definition may have an operational style. For this purpose we define a transition relation in Figure 5, as a relation on triples of the

$$\begin{array}{lcl}
\langle T, \mathcal{E}[x := e]\rangle & \longrightarrow_{[e=n][x\mapsto n]} & \langle T, \mathcal{E}[\texttt{skip}]\rangle \\
\langle T, \mathcal{E}[\texttt{skip}; C]\rangle & \longrightarrow & \langle T, \mathcal{E}[C]\rangle \\
\langle T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D]\rangle & \longrightarrow_{[b=\texttt{true}]} & \langle T, \mathcal{E}[C]\rangle \\
\langle T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D]\rangle & \longrightarrow_{[b=\texttt{false}]} & \langle T, \mathcal{E}[D]\rangle \\
\langle T, \mathcal{E}[\texttt{while } b \texttt{ do } C]\rangle & \longrightarrow & \langle T, \mathcal{E}[\texttt{if } b \texttt{ then } \ldots \texttt{ else } \ldots]\rangle \\
\langle T, \mathcal{E}[\texttt{async } C]\rangle & \longrightarrow & \langle T.C, \mathcal{E}[\texttt{skip}]\rangle \\
\langle T, \mathcal{E}[\texttt{unprotected } C]\rangle & \longrightarrow & \langle T.\mathcal{E}[\texttt{unprotected } C], \texttt{skip}\rangle \\
\langle T, \mathcal{E}[\texttt{unprotected skip}]\rangle & \longrightarrow & \langle T.\mathcal{E}[\texttt{skip}], \texttt{skip}\rangle \\
\langle T.C, \texttt{skip}\rangle & \longrightarrow_{\mathbf{start}_{T.C}} & \langle T, C\rangle \\
\langle T, C\rangle & \longrightarrow_{\mathbf{back}_{T'}} & \langle T', \texttt{skip}\rangle
\end{array}$$

Fig. 4. Transition rules of the abstract machine, with roll-back, reformulated

$$\begin{array}{lcll}
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{[e=n]} & \langle \sigma_0, T, \sigma\rangle & \text{if } \sigma(e) = n \\
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{[b=v]} & \langle \sigma_0, T, \sigma\rangle & \text{if } \sigma(b) = v \\
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{[x\mapsto n]} & \langle \sigma_0, T, \sigma[x \mapsto n]\rangle & \\
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{\mathbf{start}_{T'}} & \langle \sigma, T', \sigma\rangle & \\
\langle \sigma_0, T, \sigma\rangle & \longrightarrow_{\mathbf{back}_{T}} & \langle \sigma_0, T, \sigma_0\rangle &
\end{array}$$

Fig. 5. Operational definition of legal sequences of memory operations

form $\langle \sigma_0, T, \sigma\rangle$. Given an initial triple S, we say that the sequence of memory operations μ is legal if there is another triple S' such that $S \longrightarrow_{\mu}^{*} S'$.

Figures 4 and 5 amount to a decomposition of Figure 3, separating the definition of TM from the language semantics. Having a clear delineation of TM can be helpful for factoring semantics. Further, one may study how to implement memory systems that satisfy the definition in Figure 5—for instance, with various forms of logging.

3 Difficulties in Separating Transactional Memory

In this section we discuss the difficulties of having a separate TM in the context of more sophisticated semantics and in actual implementations. We have explored several such semantics and implementations, particularly focusing on avoiding conflicts between transactional and non-transactional memory accesses [2–5]. Recent research by others [8, 28] develops implementations with similar goals and themes, though with different techniques. Our observations in this section are drawn primarily from our experience on the implementation of `atomic` blocks and the AME constructs.

In Section 3.1, we consider systems with memory allocation, for which the difficulties appear somewhat interesting but mild. In Section 3.2, we consider concurrency, and the important but delicate distinction between strong semantics (of the kind presented in Section 2) and the property of strong atomicity [7] that may be ensured by a TM. In Section 3.3, we identify areas where aspects of the implementation of atomic actions can either be provided by a TM with strong guarantees or be layered over a TM with weaker guarantees.

We conclude that, in such settings, it is beneficial and perhaps inevitable to integrate TM with other parts of semantics and implementations (for instance, with static analysis, garbage collection, scheduling, and virtual-memory management).

3.1 Memory Allocation

In Section 2, as in some works in the literature, the operations on memory do not include allocation. However, allocation must be taken into account in the context of TM.

In particular, some semantic definitions say that roll-backs do not undo allocations [3, 20, 25]. This choice simplifies some of the theory, and it is also important in practice: it helps ensure that—no matter what else happens—a dangling pointer will not be dereferenced after a roll-back. Thus, this choice represents a sort of defense in depth.

Adding allocation to the TM interface does not seem particularly challenging. However, we may wonder whether allocation is the tip of an iceberg. Class loading, initialization, finalization, exceptions, and perhaps other operations may also have interesting interactions with transactions. A definition of TM that considers them all may well become unwieldy.

Implementations vary a great deal in how allocation is treated inside transactions. Some consider the memory-management work to be part of transactions—for example, the memory manager may be implemented using transactional reads and writes to its free-lists. In other cases, the memory manager is integrated with the transactional machinery—for example, maintaining its own logs of tentative allocations and de-allocations that can be made permanent when a transaction commits and undone when a transaction aborts.

3.2 Concurrency, and Strong Atomicity vs. Strong Semantics

Much of the appeal of TM would not exist if it were not for the possibility that transactions execute in parallel with one another and also possibly with non-transactional code. We can extend the semantics of Section 2 to account for such concurrency. The semantics do get heavier and harder, whether TM is built into the semantics or is treated as a separate module. In the latter case, the operations on transactions may include transaction identifiers, and the reads and writes may be tied to particular transactions via those identifiers.

Such semantics may reflect the choice of particular implementation techniques (for instance, the use of eager updates or lazy updates). Nevertheless, the definitions should guarantee that a class of "correctly synchronized" programs run (or appear to run) with strong semantics. Researchers have explored various definitions of correct synchronization, for instance with static separation between transactional and non-transactional data [3, 20, 25], dynamic separation with

```
Thread 1                Thread 2

atomic {                tmp1 = ready;
  ready = true;         if (tmp1 == true) {
  data = 1;               tmp2 = data;
}                       }
```

Fig. 6. Initially `ready` is `false`. Under strong semantics, if `tmp1` is `true` then `tmp2` must be `1`.

operations for moving data between those two modes [1, 2, 5], and dynamic notions of data races between transactional and non-transactional code [3, 10, 29].

If TM is a separate module, we should also make explicit its own guarantees in the presence of concurrency. With this style of definition, if we wish to obtain strong semantics for all programs in a language, the TM should probably ensure strong atomicity, which basically means that transactions appear atomic not only with respect to one another but also with respect to non-transactional memory accesses. Unfortunately, the exact definition of strong atomicity is open to debate, and the debate might only be settled in the context of a particular language semantics. Stronger notions of strong atomicity might be needed to provide strong semantics when the language implementation is aggressive, and weaker notions of strong atomicity might suffice in other cases. The relation between strong semantics (for a programming language) and strong atomicity (for a TM) is particularly subtle.

Strong Semantics Without Strong Atomicity. Some languages have strong semantics but do not rely upon a TM with strong atomicity. For instance, in STM-Haskell, a type system guarantees that the same locations are not concurrently accessed transactionally and non-transactionally [20]. In other systems, scheduling prevents transactional and non-transactional operations being attempted concurrently [26]. These cases suggest that, at the very least, the TM should be closely tied to program analysis and scheduling.

Strong Atomicity Without Strong Semantics. Strong atomicity does not suffice for strong semantics, in particular because of program transformations. Figure 6 provides an example due to Grossman *et al.* [15]. After this code runs, under strong semantics, if `tmp1` is `true` then `tmp2` must be 1. However, a conventional optimizing compiler might perform Thread 2's read from `data` before the thread's read from `ready`. (For instance, an earlier read from `data` may still be available in a register.)

Arguments in favor of strong atomicity (in particular from a hardware perspective) often seem to overlook such examples. Unless these examples are regarded as racy, program transformations currently in use should be considered incorrect. As these examples illustrate, strong atomicity does not remove the need for a notion of "correct synchronization" in programs. For racy programs that do not satisfy the correctness criterion, very few guarantees can be given in the presence of transformations by optimizing compilers and weak processor memory models.

3.3 Implementation Options at Multiple Layers

In practical implementations of language constructs, we encounter implementation options at multiple layers, and it would seem premature to fix a specific TM interface that would mandate one option or another. We consider several examples.

Implementing Strong Atomicity. In one of our implementations [4], we rely on off-the-shelf memory protection hardware for detecting possible conflicts between transactional accesses and normal (non-transactional) accesses. We organize the virtual address space of a process so that its heap is mapped twice. One mapping is used in transactions, while the other mapping is used in normal execution. This organization lets us selectively prevent normal access to pages while they remain accessible transactionally. We use this mechanism to detect possible conflicts between transactional accesses and normal accesses at the granularity of pages; we use an existing TM to detect conflicts between transactions at the granularity of objects.

This design provides a foundation that is sound but slow on conventional hardware. We introduce a number of optimizations. We allow the language runtime system to operate without triggering access violations. We use static analysis to identify non-transactional operations that are guaranteed not to conflict with transactions (these operations can bypass the page-level checks) and to identify transactions that are guaranteed not to conflict with normal accesses (these transactions need not revoke normal-access permissions on the pages involved).

Both the design and the optimizations raise a number of questions on the notion of TM. If a TM is a separate entity, should it know about virtual addresses, physical addresses, or both? How should it relate to memory protection? How can its guarantees be adjusted in the presence of program analysis?

Tolerating Inconsistent Views of Memory. With some TM implementations, a transaction can continue running as a “zombie” [12] after experiencing a conflict. Zombie transactions can have errant behavior that is not permitted by strong semantics. Consider the example in Figure 7. This program is correctly synchronized under all of the criteria that we have studied, so a correct implementation must not loop endlessly. However, if the TM does not guarantee that Thread 1 will see a consistent view of memory, it is possible for `temp1==0` and `temp2==1`, and consequently for Thread 1 to loop.

This flaw can be corrected by modifying the TM to give stronger guarantees (for instance, opacity [19]). Alternatively, the language runtime system can sandbox the effects of zombie transactions by adding periodic validation work to loops, and containing any side-effects from zombie transactions (for example, not raising a `NullReferenceException` in response to an access violation from a zombie transaction).

```
Thread 1                          Thread 2

atomic {                          atomic {
  temp1 = x1;                       x1 ++;
  temp2 = x2;                       x2 ++;
  if (temp1 != temp2) {           }
    while (1) { }
} }
```

Fig. 7. Initially `x1==x2==0`. Under strong semantics, Thread 1 must not loop.

```
Thread 1                          Thread 2

atomic {                          while (true) {
  x = 100;                          atomic {
  x_initialized = true;               if (x_initialized) break;
}                                 } }
                                  Console.Out.Writeline(x);
```

Fig. 8. A publication idiom. Under strong semantics, if Thread 2 sees `x_initialized` true, then it must print `100`.

Granular Safety. Some TM implementations present data granularity problems. For instance, rolling back a write to a single byte might also roll back the contents of other bytes in the same memory word.

There are several viable techniques for avoiding this problem. First, the TM implementation may be strengthened to maintain precise access granularity (say, at the cost of extra book-keeping by tracking reads and writes at a byte level). Second, if correct synchronization is defined by static separation, then transactional and non-transactional data can be allocated on separate machine words. Third, various dynamic mechanisms can be used for isolating transactional and non-transactional data.

Only the first of these options places granular-safety requirements on the TM implementation. The other two options require that other parts of the language implementation be aware of the particular granularity at which the TM operates.

Ordering Across Threads. In privatization and publication idioms [3, 12, 13, 31, 34], a piece of data is accessed transactionally depending on the value of another piece of data, as for example in Figure 8. These idioms are frequently considered to be correctly synchronized. However, naïve implementations over TM may not execute them correctly. For the example in Figure 8, an implementation that uses lazy updates may allow Thread 2's non-transactional read of `x` to occur before Thread 1 has finished writing back its transactional update.

Such idioms require that if non-transactional code executes after an atomic action, then the non-transactional code must see all the side effects of preceding atomic actions. This guarantee is provided by TMs with strong atomicity.

Alternatively, it can be layered over a weaker TM by adding synchronization barriers when transactions start and commit [24], or it can be provided by page-based separation of transactional and non-transactional data [4]. The performance trade-offs between these approaches are complicated, and there is no clear approach to favor in defining a clean common TM interface.

4 The Garbage-Collection Analogy

Grossman has drawn a compelling analogy between TM and garbage collection, comparing the programming problems that they aim to solve, and the features and limitations of the techniques [14]. In this section we consider what this analogy says about the problem of defining TM.

Garbage collection can be regarded as a tool for implementing "garbage-collected memory". A "garbage-collected memory" is simply an "infinite memory" over which one does not need to worry about freeing space. It is both common and helpful to describe language semantics over such a memory.

On the other hand, there is no canonical definition of garbage collection. Although language implementations may have internal interfaces that are defined with various degrees of precision and generality, there seems to be no separate, clean, portable garbage-collection interface.

Typically, garbage collection—much like TM—is coupled with a compiler and other parts of a language implementation. Some collectors exploit virtual-memory page-protection hardware (e.g., [23, 35]) and static analysis (e.g., [11, 33]). Much like TM, also, garbage collection can interact with program transformations. For instance, the trick of exchanging the contents of two reference-typed fields by using three XOR operations is correct only if the garbage collector will not see the intermediate non-reference values. Furthermore, program transformations can affect when finalizers (which run when an object becomes unreachable) will be eligible to run.

On this basis, garbage-collection machinery and TM machinery are indeed analogous. In the world of TM, however, there is no easy counterpart to garbage-collected memory, that is, to infinite memory. In our opinion the best candidate is not TM, but rather the concept of atomic action. Both infinite memory and atomic actions can be used in specifying language semantics, and both have a wide range of concrete implementations.

5 Conclusion

In this paper we examine transactional memory from the perspectives of language semantics and implementations. We believe that, from both perspectives, it is often impractical to define a separate, clean, portable internal TM interface.

Nevertheless, it may be productive to study the definition of TM interfaces, and to examine whether or not particular TM implementation techniques are compatible with them.

Furthermore, in some cases, programmers may use a TM interface directly, rather than via `atomic` blocks (for instance, for manipulating data structures). However, that TM interface need not coincide with one used internally by an implementation of `atomic` blocks, nor with a hardware TM interface, since hardware might provide lower-level primitives [27, 30]. Finally, a compiler framework may usefully include a common interface for multiple TM implementations—but this interface is likely to be specific to a particular framework, and much broader than that of TM as a shared object.

In summary, the question of what is transactional memory seems to remain open, and may well deserve further investigation. However, in our opinion, it is at least as worthwhile to study languages and implementation techniques based on transactional ideas, even without a separate definition of transactional memory.

Acknowledgements. We are grateful to Dan Grossman, Rachid Guerraoui, Leslie Lamport, and Greg Morrisett for discussions on the subject of this paper.

References

1. Abadi, M., Birrell, A., Harris, T., Hsieh, J., Isard, M.: Dynamic separation for transactional memory. Technical Report MSR-TR-2008-43, Microsoft Research (March 2008)
2. Abadi, M., et al.: Implementation and use of transactional memory with dynamic separation. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 63–77. Springer, Heidelberg (2009)
3. Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: POPL 2008: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2008, pp. 63–74 (2008)
4. Abadi, M., Harris, T., Mehrara, M.: Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In: PPoPP 2009: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009, pp. 185–196 (2009)
5. Abadi, M., Harris, T., Moore, K.F.: A model of dynamic separation for transactional memory. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 6–20. Springer, Heidelberg (2008)
6. Abadi, M., Plotkin, G.D.: A model of cooperative threads. In: POPL 2009: Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2009, pp. 29–40 (2009)
7. Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing transactional semantics: The subtleties of atomicity. In: WDDD 2005: Proc. 4th Workshop on Duplicating, Deconstructing and Debunking, June 2005, pp. 48–55 (2005)
8. Bronson, N.G., Kozyrakis, C., Olukotun, K.: Feedback-directed barrier optimization in a strongly isolated STM. In: POPL 2009: Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2009, pp. 213–225 (2009)
9. Cohen, A., O'Leary, J.W., Pnueli, A., Tuttle, M.R., Zuck, L.D.: Verifying correctness of transactional memories. In: FMCAD 2007: Proc. 7th International Conference on Formal Methods in Computer-Aided Design, November 2007, pp. 37–44 (2007)

10. Dalessandro, L., Scott, M.L.: Strong isolation is a weak idea. In: TRANSACT 2009: 4th ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (February 2009)
11. Detlefs, D., Nandivada, V.K.: Compile-time concurrent marking write barrier removal. Technical Report SMLI-TR-2004-142, Sun Microsystems (December 2004)
12. Dice, D., Shalev, O., Shavit, N.N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
13. Dice, D., Shavit, N.: What really makes transactions faster? In: TRANSACT 2006, 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (June 2006)
14. Grossman, D.: The transactional memory / garbage collection analogy. In: OOPSLA 2007: Proc. 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, October 2007, pp. 695–706 (2007)
15. Grossman, D., Manson, J., Pugh, W.: What do high-level memory models mean for transactions? In: MSPC 2006: Proc. 2006 Workshop on Memory System Performance and Correctness, October 2006, pp. 62–69 (2006)
16. Guerraoui, R., Henzinger, T.A., Singh, V.: Completeness and nondeterminism in model checking transactional memories. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 21–35. Springer, Heidelberg (2008)
17. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 305–319. Springer, Heidelberg (2008)
18. Guerraoui, R., Henzinger, T., Singh, V.: Model checking transactional memories. In: PLDI 2008: Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2008, pp. 372–382 (2008)
19. Guerraoui, R., Kapałka, M.: On the correctness of transactional memory. In: PPoPP 2008: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2008, pp. 175–184 (2008)
20. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP 2005: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, June 2005, pp. 48–60 (2005)
21. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA 1993: Proc. 20th International Symposium on Computer Architecture, May 1993, pp. 289–301 (1993)
22. Jagannathan, S., Vitek, J., Welc, A., Hosking, A.: A transactional object calculus. Sci. Comput. Program. 57(2), 164–186 (2005)
23. Kermany, H., Petrank, E.: The compressor: concurrent, incremental, and parallel compaction. In: PLDI 2006: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2006, pp. 354–363 (2006)
24. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R., Saha, B., Welc, A.: Practical weak-atomicity semantics for Java STM. In: SPAA 2008: Proc. 20th Symposium on Parallelism in Algorithms and Architectures, June 2008, pp. 314–325 (2008)
25. Moore, K.F., Grossman, D.: High-level small-step operational semantics for transactions. In: POPL 2008: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2008, pp. 51–62 (2008)
26. Oancea, C.E., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: SPAA 2009: Proc. 21st ACM Symposium on Parallelism in Algorithms and Architectures (August 2009)

27. Saha, B., Adl-Tabatabai, A.-R., Jacobson, Q.: Architectural support for software transactional memory. In: MICRO 2006: Proc. 39th IEEE/ACM International Symposium on Microarchitecture, June 2006, pp. 185–196 (2006)
28. Schneider, F.T., Menon, V., Shpeisman, T., Adl-Tabatabai, A.-R.: Dynamic optimization for efficient strong atomicity. In: OOPSLA 2008: Proc. 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, October 2008, pp. 181–194 (2008)
29. Scott, M.L.: Sequential specification of transactional memory semantics. In: TRANSACT 2006: 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (June 2006)
30. Shriraman, A., Dwarkadas, S., Scott, M.L.: Flexible decoupled transactional memory support. In: ISCA 2008: Proc. 35th International Symposium on Computer Architecture, June 2008, pp. 139–150 (2008)
31. Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization techniques for software transactional memory. Technical Report 915, CS Dept, U. Rochester (February 2007)
32. Tasiran, S.: A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research (April 2008)
33. Vechev, M.T., Bacon, D.F.: Write barrier elision for concurrent garbage collectors. In: ISMM 2004: Proc. 4th International Symposium on Memory Management, October 2004, pp. 13–24 (2004)
34. Wang, C., Chen, W.-Y., Wu, Y., Saha, B., Adl-Tabatabai, A.-R.: Code generation and optimization for transactional memory constructs in an unmanaged language. In: CGO 2007, International Symposium on Code Generation and Optimization, March 2007, pp. 34–48 (2007)
35. Wegiel, M., Krintz, C.: The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. In: ASPLOS 2008: Proc. 13th International Conference on Architectural Support for Programming Languages and Operating Systems, March 2008, pp. 91–102 (2008)
36. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge (1993)

The Effect of Tossing Coins in Omega-Automata

Christel Baier[1], Nathalie Bertrand[2], and Marcus Größer[1]

[1] Technische Universität Dresden, Faculty of Computer Science, Germany
[2] INRIA Rennes Bretagne Atlantique, France

Abstract. In this paper we provide a summary of the fundamental properties of probabilistic automata over infinite words. Such probabilistic automata are a variant of standard automata with Büchi or other ω-regular acceptance conditions, such as Rabin, Streett, parity or Müller, where the nondeterministic choices are resolved probabilistically. Acceptance of an infinite input word can be defined in different ways: by requiring that (i) almost all runs are accepting, or (ii) the probability for the accepting runs is positive, or (iii) the probability measure of the accepting runs is beyond a certain threshold. Surprisingly, even the qualitative criteria (i) and (ii) yield a different picture concerning expressiveness, efficiency, and decision problems compared to the nondeterministic case.

Introduction

Automata over infinite objects play a central role for verification purposes, reasoning about infinite games and logics that specify nondeterministic behaviors. Many variants of ω-automata have been studied in the literature that can be classified according to their inputs (e.g., words or trees), their acceptance conditions (e.g., Büchi, Rabin, Streett, Muller or parity acceptance) and their branching structure (e.g., deterministic, nondeterministic, or alternating). See, e.g., [9,18] for an overview of automata over infinite objects.

Although probabilistic finite automata (PFA) have attracted many researchers, see e.g. [8,12,13,15], probabilistic language acceptors for infinite words just have recently been studied. The formal definition of probabilistic ω-automata is the same as for nondeterministic ω-automata, the only difference being that all choices are resolved by probabilistic distributions. Acceptance of an infinite word $\sigma = a_1 a_2 a_3 \ldots$ can then be defined by imposing a condition on the probability of the accepting runs for σ. We consider here three types of accepted languages. The *probable semantics* requires positive probability for the accepting runs, the *almost-sure semantics* requires that the accepting runs for σ have probability 1, while the *threshold semantics* relies on some fixed threshold $\lambda \in]0,1[$ and requires that the acceptance probability is greater than λ.

Given the well-known fact that PFA are more expressive that NFA and that many relevant decision problems for PFA are undecidable, it is no surprise that PBA with the threshold semantics are more powerful than NBA and that the emptiness problem and other decision problems are undecidable for them. The definition of the accepted language under the probable semantics via the criterion *"the probability for the accepting runs is* > 0*"* appears to be the natural adaption of the definition of the accepted language of a nondeterministic automaton which relies on the criterion *"there is at least* *one accepting run"*. One therefore might expect that probabilistic and nondeterministic ω-automata are rather close and enjoy similar properties. This, however, is not the case. The class of languages that are accepted by a PBA with the probable semantics strictly subsumes the class of ω-regular languages and it is closed under union, intersection and complementation. Furthermore, there are ω-regular languages that are recognizable by probable PBA of linear size while the sizes of smallest NBA for these languages grow exponentially. The price we have to pay for this extra power of probabilistic automata is that basic problems like checking emptiness, universality or equivalence are undecidable for PBA under the probable semantics.

The almost-sure semantics of PBA is "weaker" in the sense that each almost-sure PBA can be transformed into an equivalent PBA with the probable semantics, but not vice versa. Moreover, the class of languages that are recognizable by PBA with the almost-sure semantics does not cover the full class of ω-regular languages, it is not closed under complementation and contains non-ω-regular languages. On the positive side, the emptiness and universality problem for almost-sure PBA are decidable.

Organization. Section 1 recalls the definition of nondeterministic ω-automata with Büchi, Rabin or Streett acceptance conditions and introduces their probabilistic variants. Results on the expressiveness and efficiency of probabilistic Büchi automata are summarized in Section 2. Composition operators for PBA and probabilistic automata with Rabin and Streett acceptance are considered in Section 3. Decision problems for PBA will be discussed in Section 4. Finally, Section 5 contains some concluding remarks.

The material of this paper is a summary of the results presented in the papers [2,3]. Further details can be found there and in the thesis by Marcus Größer [10].

1 Probabilistic ω-Automata

We assume some familiarity with classical nondeterministic automata over finite or infinite words; see e.g. [9,18]. We just recall some basic concepts of nondeterministic ω-automata, and then present the definition of probabilistic ω-automata.

Definition 1 (Nondeterministic ω-automata). *A nondeterministic ω-automaton is a tuple $\mathcal{N} = (Q, \Sigma, \delta, Q_0, \mathsf{Acc})$, where*

- *Q is a finite nonempty set of states,*
- *Σ is a finite nonempty input alphabet,*
- *$\delta : Q \times \Sigma \to 2^Q$ is a transition function that assigns to each state q and letter $a \in \Sigma$ a (possibly empty) set $\delta(q,a)$ of states,*
- *$Q_0 \subseteq Q$ is the set of* initial states*,*
- Acc *is an* acceptance condition *(which will be explained later).*

$\mathcal{N}$ is called deterministic if $|Q_0| = 1$ and $|\delta(q,a)| = 1$ for all $q \in Q$ and $a \in \Sigma$.

Given an input word $\sigma = a_1 a_2 a_3 \ldots \in \Sigma^\omega$, a run for σ in $\mathcal{N}$ is a maximal state-sequence $\pi = q_0 q_1 q_2 \ldots$ such that $q_0 \in Q_0$, $q_{i+1} \in \delta(q_i, a_{i+1})$ for all $i \geq 0$. Maximality means that either π is infinite or ends in state q_n if $\delta(q_n, a_{n+1}) = \emptyset$. Each finite run $q_0 q_1 \ldots q_n$

is said to be *rejecting*. The acceptance condition Acc imposes a condition on infinite runs and declares which of the infinite runs are *accepting*. We write $\mathsf{inf}(\pi)$ for the set of states $p \in Q$ such that $p = q_i$ for infinitely many indices $i \geq 0$. Several acceptance conditions are known for nondeterministic ω-automata. We will consider three types of acceptance conditions:

Büchi: A Büchi acceptance condition Acc is a subset F of Q. The elements in F are called *final* or *accepting states*. An infinite run $\pi = q_0 q_1 q_2 \ldots$ is called (Büchi) accepting if π visits F infinitely often, i.e., $\mathsf{inf}(\pi) \cap F \neq \emptyset$.

Streett: A Streett acceptance condition Acc is a finite set of pairs (H_l, K_l) consisting of subsets H_l, K_l of Q, i.e., $\mathsf{Acc} = \{(H_1,K_1),\ldots,(H_\ell,K_\ell)\}$. An infinite run $\pi = q_0 q_1 q_2 \ldots$ is called (Streett) accepting if for each $l \in \{1,\ldots,\ell\}$ we have:

$$\mathsf{inf}(\pi) \cap H_l \neq \emptyset \text{ or } \mathsf{inf}(\pi) \cap K_l = \emptyset.$$

Rabin: A Rabin acceptance condition Acc is syntactically the same as a Streett acceptance condition, i.e., a finite set $\mathsf{Acc} = \{(H_1,K_1),\ldots,(H_\ell,K_\ell)\}$ where $H_l, K_l \subseteq Q$ for $1 \leq l \leq \ell$. An infinite run $\pi = q_0 q_1 q_2 \ldots$ is called (Rabin) accepting if there is some $l \in \{1,\ldots,\ell\}$ such that

$$\mathsf{inf}(\pi) \cap H_l = \emptyset \text{ and } \mathsf{inf}(\pi) \cap K_l \neq \emptyset.$$

Note that the semantics of Streett and Rabin acceptance conditions are duals of each other, i.e., for each infinite run π we have: π is accepting according to the Rabin condition Acc iff π is rejecting (i.e., not accepting) according to the Streett condition Acc. Furthermore, a Büchi acceptance condition F can be viewed as a special case of a Streett or Rabin condition with a single acceptance pair, namely $\{(F,Q)\}$ for the Streett condition and $\{(\emptyset,F)\}$ for the Rabin condition. The *accepted language* of a nondeterministic ω-automaton $\mathcal{N}$ with the alphabet Σ, denoted $\mathcal{L}(\mathcal{N})$, is defined as the set of infinite words $\sigma \in \Sigma^\omega$ that have at least one accepting run in $\mathcal{N}$.

$$\mathcal{L}(\mathcal{N}) \; = \; \{\sigma \in \Sigma^\omega : \text{there exists an accepting run for } \sigma \text{ in } \mathcal{N}\}$$

In what follows, we write NBA to denote a nondeterministic Büchi automaton, NRA for nondeterministic Rabin automata and NSA for nondeterministic Streett automata. Similarly, the notations DBA, DRA and DSA are used to denote deterministic ω-automata with a Büchi, Rabin or Streett acceptance condition.

It is well-known that the classes of languages that can be accepted by NBA, DRA, NRA, DSA or NSA are the same. These languages are often called ω-*regular* and represented by ω-regular expressions, i.e., finite sums of expressions of the form $\alpha\beta^\omega$ where α and β are ordinary regular expressions (representing regular languages over finite words) and the language associated with β is nonempty and does not contain the empty word. In the sequel, we will identify ω-regular expressions with the induced ω-regular language. In what follows, we write **NBA** for the class of ω-regular languages. While deterministic ω-automata with Rabin and Streett acceptance (DRA and DSA) cover the full class of ω-regular languages, DBA are less powerful as, e.g., the language $(a+b)^* a^\omega$ cannot be recognized by a DBA. Hence, the class **DBA** of DBA-recognizable languages is a proper subclass of **NBA**.

The syntax of probabilistic ω-automata is the same as in the nondeterministic case, except that the transition function specifies probabilities. That is, for any state p and letter $a \in \Sigma$ either p does not have any a-successor or there is a probability distribution for the a-successors of p.

Definition 2 (Probabilistic ω-automata). *A probabilistic ω-automaton is a tuple $\mathcal{P} = (Q, \Sigma, \delta, \mu_0, \mathsf{Acc})$, where*

- *Q is a finite nonempty set of states,*
- *Σ is a finite nonempty input alphabet,*
- *$\delta : Q \times \Sigma \times Q \to [0,1]$ is a transition probability function such that for all $p \in Q$ and $a \in \Sigma$ either $\sum_{q \in Q} \delta(p,a,q) = 1$ or $\delta(p,a,q) = 0$ for all $q \in Q$,*
- *μ_0 is the* initial distribution*, i.e., a function $\mu_0 : Q \to [0,1]$ such that $\sum_{q \in Q} \mu_0(q) = 1$,*
- *Acc is an* acceptance condition *(as for nondeterministic ω-automata).*

We refer to the states $q_0 \in Q$ where $\mu_0(q_0) > 0$ as initial states. If p is a state such that $\delta(q,a,p) > 0$ then we say that q has an outgoing a-transition to state p.

Acceptance conditions can be defined as in the nondeterministic case. In this paper, we just regard Büchi, Rabin and Streett acceptance and use the abbreviations PBA, PRA and PSA for probabilistic Büchi automata, probabilistic Rabin automata, and probabilistic Streett automata, respectively.

The intuitive operational behavior of a probabilistic ω-automaton $\mathcal{P}$ for a given input word $\sigma = a_1 a_2 \ldots \in \Sigma^\omega$ is as follows. Initially $\mathcal{P}$ chooses at random an initial state p_0 according to the initial distribution μ_0. If $\mathcal{P}$ has read the first i input symbols $a_1, \ldots, a_i$ and the current state is p_i then $\mathcal{P}$ moves with probability $\delta(p_i, a_{i+1}, p)$ to state p and tries to consume the next input symbol a_{i+2} in state $p = p_{i+1}$. If there is no outgoing a_{i+1}-transition from the current state p_i, then $\mathcal{P}$ rejects. As in the nondeterministic case, the resulting state-sequence $\pi = p_0 p_1 p_2 \ldots \in Q^* \cup Q^\omega$ is called a *run* for σ in $\mathcal{P}$. If $\mathcal{P}$ rejects in state p_i, i.e., $\delta(p_i, a_{i+1}, \cdot)$ is the null function, then the obtained run is finite (and ends in state p_i). If the automaton never rejects while reading the letters a_i of the input word $\sigma = a_1 a_2 a_3 \ldots \in \Sigma^\omega$, the generated run is an infinite state-sequence $\pi = p_0 p_1 p_2 \ldots \in Q^\omega$. Acceptance of a run according to a Büchi, Rabin or Streett acceptance condition is defined as in the nondeterministic setting.

Semantics of probabilistic ω-automata. We distinguish three semantics that differ in the requirements on the *acceptance probability*. The formal definition of the acceptance probability relies on the view of an input word $\sigma \in \Sigma^\omega$ as a *scheduler* when $\mathcal{P}$ is treated as a Markov decision process, i.e., an operational model for a probabilistic system where in each state q the letters that can be consumed in q are treated as actions that are enabled in q. Given a word/scheduler $\sigma = a_1 a_2 a_3 \ldots \in \Sigma^\omega$, the behavior of $\mathcal{P}$ under σ is given by a Markov chain $\mathcal{M}_\sigma$ where the states are pairs (q,i) where $q \in Q$ stands for the current state and i is a natural number ≥ 1 that denotes the current word position. Stated differently, state (q,i) in the Markov chain $\mathcal{M}_\sigma$ stands for the configuration that $\mathcal{P}$ might have reached after having consumed the first $i-1$ letters $a_1, \ldots, a_{i-1}$ of the input word σ. Assuming that $\delta(q, a_{i+1}, \cdot)$ is not the null function, the transition probabilities from state (q,i) are given by the distribution $\delta(q, a_{i+1}, \cdot)$, i.e., from state

(q,i) the Markov chain $\mathcal{M}_\sigma$ moves with probability $\delta(q,a_{i+1},p)$ to state $(p,i+1)$. In case that $\delta(q,a_{i+1},\cdot)=0$ then (q,i) is an absorbing state, i.e., a state without any outgoing transition. The runs for σ in $\mathcal{P}$ correspond to the paths in $\mathcal{M}_\sigma$. We can now apply the standard concepts for Markov chains to reason about the probabilities of infinite paths and define the acceptance probability $\mathsf{Pr}^{\mathcal{P}}(\sigma)$ for the infinite word σ in $\mathcal{P}$ as the probability measure of the accepting runs for σ in the Markov chain $\mathcal{M}_\sigma$.

Probable semantics. A probabilistic ω-automaton $\mathcal{P}$ accepts an infinite input word σ if the acceptance probability $\mathsf{Pr}^{\mathcal{P}}(\sigma)$ is positive.
Almost-sure semantics. $\mathcal{P}$ accepts an infinite input word σ if almost all runs for σ are accepting.[1]
Threshold semantics. Given a threshold $\lambda \in]0,1[$, $\mathcal{P}$ accepts an infinite input word σ if the acceptance probability $\mathsf{Pr}^{\mathcal{P}}(\sigma)$ is greater than λ.

Thus, there are three versions of accepted language of a PBA $\mathcal{P}$:

$$\mathcal{L}^{>0}(\mathcal{P}) \overset{\text{def}}{=} \{\sigma \in \Sigma^\omega : \mathsf{Pr}^{\mathcal{P}}(\sigma) > 0\}$$
$$\mathcal{L}^{=1}(\mathcal{P}) \overset{\text{def}}{=} \{\sigma \in \Sigma^\omega : \mathsf{Pr}^{\mathcal{P}}(\sigma) = 1\}$$
$$\mathcal{L}^{>\lambda}(\mathcal{P}) \overset{\text{def}}{=} \{\sigma \in \Sigma^\omega : \mathsf{Pr}^{\mathcal{P}}(\sigma) > \lambda\}$$

Sometimes we will add the subscript "Büchi", "Streett' or "Rabin" to make clear which type of acceptance condition is assumed and write, for instance, $\mathcal{L}^{>0}_{\text{Büchi}}(\mathcal{P})$, $\mathcal{L}^{>0}_{\text{Rabin}}(\mathcal{P})$ or $\mathcal{L}^{>0}_{\text{Streett}}(\mathcal{P})$ to denote the accepted language of a PBA, PRA or PSA, respectively, under the probable semantics.

$$\begin{aligned}
\mathbf{PBA}^{>0} &= \text{class of languages } L \text{ such that } L = \mathcal{L}^{>0}(\mathcal{P}) \text{ for some PBA } \mathcal{P},\\
\mathbf{PBA}^{=1} &= \text{class of languages } L \text{ such that } L = \mathcal{L}^{=1}(\mathcal{P}) \text{ for some PBA } \mathcal{P},\\
\mathbf{PBA}^{>\lambda} &= \text{class of languages } L \text{ such that } L = \mathcal{L}^{>\lambda}(\mathcal{P}) \text{ for some PBA } \mathcal{P}
\end{aligned}$$

Analogous notations will be used for Rabin and Streett acceptance conditions. *Equivalence* of ω-automata means that their accepted languages agree. The *size* of a Büchi automaton is the number of states. The size of a Rabin or Streett automaton denotes the number of states plus the number of acceptance pairs.

Example 1 (Probabilistic Büchi automata). In the pictures for probabilistic ω-automata, the probability $\delta(q,a,p)$ is attached to the a-transition from q to p. If $\delta(q,a,p)=1$ then the edge is simply labeled with a. Similarly, if there is a single initial state q_0 (i.e., $\mu_0(q_0)=1$, while $\mu_0(q)=0$ for all other states q) we simply depict an incoming arrow to q_0. For PBA, we depict the accepting states (i.e., the states $q \in F$) by squares, non-accepting states by circles.

The left part of Figure 1 shows a PBA $\mathcal{P}$ over the alphabet $\Sigma=\{a,b\}$ and, e.g., $\delta(q_0,a,q_0)=\delta(q_0,a,q_1)=\frac{1}{2}$ and $\delta(q_0,b,q_0)=1$. The initial distribution assigns probability 1 to state q_0. The Büchi acceptance condition is given by $F=\{q_1\}$. We have:

[1] The formulation "almost all runs have property X" means that the probability measure of the runs where property X does not hold is 0.

$$\mathcal{L}^{>0}(\mathcal{P}) = (a+b)^* a^\omega, \ \ \mathcal{L}^{=1}(\mathcal{P}) = b^* a^\omega, \ \ \mathcal{L}^{>\frac{1}{5}}(\mathcal{P}) = b^* a b^* a b^* a^\omega$$

We briefly explain why $\mathcal{L}^{>0}(\mathcal{P})$ agrees with the language $(a+b)^* a^\omega$.

"$\supseteq$": The behavior of $\mathcal{P}$ for an an infinite input word $\sigma \in (a+b)^* a^\omega$ is as follows. With positive probability $\mathcal{P}$ stays in the initial state q_0 until the last b in σ has been read. From then on, $\mathcal{P}$ moves almost surely to the accepting state q_1 and stays there forever when reading the infinite suffix a^ω. Thus, $\mathrm{Pr}^{\mathcal{P}}(\sigma) > 0$ for all $\sigma \in (a+b)^* a^\omega$. This yields that $(a+b)^* a^\omega \subseteq \mathcal{L}^{>0}(\mathcal{P})$.

"$\subseteq$": If $\sigma \in \mathcal{L}^{>0}(\mathcal{P})$ then the number of b's that appear in σ is finite because only state q_1 is accepting and only letter a can be read in state q_1. Hence, each accepted word $\sigma \in \mathcal{L}^{>0}(\mathcal{P})$ must have the suffix a^ω.

Given a word $\sigma = c_1 c_2 \ldots c_\ell b a^\omega$ with $c_i \in \{a, b\}$, the precise acceptance probability is $\mathrm{Pr}^{\mathcal{P}}(\sigma) = \left(\frac{1}{2}\right)^k$ where $k = |\{i \in \{1,\ldots,\ell\} : c_i = a\}|$, while $\mathrm{Pr}^{\mathcal{P}}(a^\omega) = 1$. This yields $\mathcal{L}^{=1}(\mathcal{P}) = b^* a^\omega$ and $\mathcal{L}^{>\frac{1}{5}}(\mathcal{P}) = b^* a b^* a b^* a^\omega$.

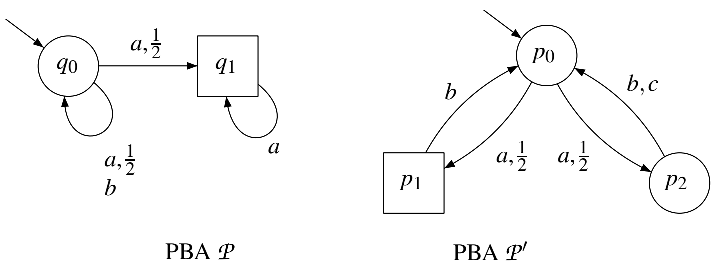


Fig. 1. Examples for PBA $\mathcal{P}$ (left) and $\mathcal{P}'$ (right)

Obviously, for each PBA $\mathcal{P}$ all words in $\mathcal{L}^{>0}(\mathcal{P})$ have at least one accepting run. Thus, $\mathcal{L}^{>0}(\mathcal{P})$ is always contained in the accepted language of the NBA that results by ignoring the transition probabilities. For the PBA $\mathcal{P}$ in Figure 1, the language $\mathcal{L}^{>0}(\mathcal{P})$ even agrees with the language of the underlying NBA. This, however, does not apply to all PBA. For an example, we regard the PBA $\mathcal{P}'$ shown in the right part of Figure 1 which has the alphabet $\Sigma = \{a, b, c\}$. The underlying NBA accepts the language $\big((ac)^* ab\big)^\omega$, while the accepted language of the PBA $\mathcal{P}'$ with the probable semantics is

$$\mathcal{L}^{>0}(\mathcal{P}') \ = \ (ab+ac)^*(ab)^\omega.$$

Given an input word $\sigma \in (ab+ac)^*(ab)^\omega$, say $\sigma = x(ab)^\omega$ where $x \in (ab+ac)^*$, then with positive probability $\mathcal{P}'$ generates the run fragment $p_0 p_2 p_0 p_2 \ldots p_0 p_2 p_0$ when reading x. For the remaining suffix $(ab)^\omega$, $\mathcal{P}'$ can always consume the next letter and almost surely $\mathcal{P}'$ will visit p_1 and p_2 infinitely often. Thus, $\mathrm{Pr}^{\mathcal{P}'}(\sigma) > 0$ and $\sigma \in \mathcal{L}(\mathcal{P}')$. Vice versa, we have to show that $\mathcal{L}^{>0}(\mathcal{P}')$ is a subset of $(ab+ac)^*(ab)^\omega$. It is obvious that all accepted words $\sigma \in \mathcal{L}(\mathcal{P}')$ belong to $((ac)^* ab)^\omega$. Any word σ in $(ab+ac)^\omega$ with infinitely many c's is rejected by $\mathcal{P}'$ as almost all runs for σ are finite and end in state p_1, where the next input symbol is c and cannot be consumed in state p_1. $\square$

2 Expressiveness of Probabilistic Büchi Automata

Each DBA can be viewed as a PBA (we just have to assign probability 1 to all edges in the DBA and deal with the initial distribution that assigns probability 1 to the unique initial state). As shown in Example 1 there is a PBA for the language $(a+b)^*a^\omega$ which cannot be accepted by DBA. As a consequence we obtain that the class of DBA-recognizable languages is a subclass of $\mathbf{PBA}^{=1}$ and a proper subclass of $\mathbf{PBA}^{>0}$ (and any threshold language $\mathbf{PBA}^{>\lambda}$).

In this section, we will study the expressiveness of PBA in more detail. We start with the observation that the three semantics form a hierarchie:

Theorem 1 (Hierarchy of PBA-languages). *For all* $\lambda, \nu \in]0,1[$*, we have:*

$$\boldsymbol{PBA}^{=1} \subset \boldsymbol{PBA}^{>0} \subset \boldsymbol{PBA}^{>\lambda} = \boldsymbol{PBA}^{>\nu}$$

Furthermore, the inclusions $\boldsymbol{PBA}^{=1} \subset \boldsymbol{PBA}^{>0}$ *and* $\boldsymbol{PBA}^{>0} \subset \boldsymbol{PBA}^{>\lambda}$ *are strict.*

In the sequel, we will write **TPBA** for the threshold classes $\mathbf{PBA}^{>\lambda}$.

Proof sketch. Let us briefly sketch the proof ideas for Theorem 1. The inclusion $\mathbf{PBA}^{=1} \subset \mathbf{PBA}^{>0}$ has been established in [2] by providing a transformation that constructs for a given PBA $\mathcal{P}$ with the almost-sure semantics an equivalent PBA $\mathcal{P}'$ with the probable semantics. The rough idea is that $\mathcal{P}'$ guesses probabilistically some word position i and then checks whether, from position i on, $\mathcal{P}$ will never enter an accepting state $p \in F$. Since the definition of $\mathcal{P}'$ relies on a certain powerset construction, this transformation can yield an exponential blow-up.

To prove the inclusion $\mathbf{PBA}^{>0} \subset \mathbf{PBA}^{>\lambda}$ we start with a PBA $\mathcal{P}$ with the probable semantics and turn $\mathcal{P}$ into an equivalent PBA $\mathcal{T}$ with threshold λ. $\mathcal{T}$ arises from $\mathcal{P}$ by adding a fresh trap state p_{acc} (i.e., $\delta(p_{\text{acc}}, a, p_{\text{acc}}) = 1$ for all $a \in \Sigma$) which is declared to be accepting. The initial distribution of $\mathcal{T}$ assigns probability λ to state p_{acc}. With the remaining probability $1-\lambda$, $\mathcal{T}$ simulates $\mathcal{P}$. Then, $\Pr^{\mathcal{T}}(\sigma) = \lambda + (1-\lambda)\Pr^{\mathcal{P}}(\sigma)$ for all words σ. Hence, $\mathcal{L}^{>0}(\mathcal{P}) = \mathcal{L}^{>\lambda}(\mathcal{T})$.

An analogous construction is applicable to prove that $\mathbf{PBA}^{>\lambda} \subseteq \mathbf{PBA}^{>\nu}$ if $\lambda < \nu$. Vice versa, suppose we are given a PBA $\mathcal{T}$ with threshold λ. An equivalent PBA $\mathcal{T}'$ with threshold ν where $\lambda > \nu$ arises from $\mathcal{T}$ by adding a fresh trap state p that is non-accepting. $\mathcal{T}'$ enters p initially with probability $1 - \frac{\nu}{\lambda}$ and mimicks $\mathcal{T}$ with the remaining probability $\frac{\nu}{\lambda}$. Then, $\Pr^{\mathcal{T}'}(\sigma) = \frac{\nu}{\lambda} \cdot \Pr^{\mathcal{T}}(\sigma)$ for all words σ, and therefore $\mathcal{L}^{>\lambda}(\mathcal{T}) = \mathcal{L}^{>\nu}(\mathcal{T}')$.

Using results established by Paz [14] for probabilistic finite automata, one can show that for each irrational threshold λ the language L over the alphabet $\{0,1,c\}$ consisting of all words $b_1 b_2 \ldots b_n c^\omega$ where $\sum_{1 \le i \le n} b_i \cdot 2^{-i}$ is strictly larger than λ can be recognized by a PBA with threshold λ, while there is no PBA for L with the probable semantics. An example for a language that is accepted by a PBA with the probable semantics, but not by an almost-sure PBA will be provided in Theorem 2. □

Probabilistic finite automata (PFA) with the acceptance criterion "the accepting runs have a positive probability measure" can be viewed as nondeterministic finite automata, and hence, they have exactly the power of regular languages. One might expect that an analogous result can be established for PBA with the probable semantics and the class of ω-regular languages. This, however, is not the case. PBA with the probable semantics are more powerful than NBA. In the proof sketch of the following theorem we will provide examples for a non-ω-regular language in $\mathbf{PBA^{>0}}$ that cannot be recognized by an almost-sure PBA.

Theorem 2 (PBA versus ω-regular languages). *NBA is strictly contained in* $\mathbf{PBA^{>0}}$, *while* $\mathbf{NBA} \not\subseteq \mathbf{PBA^{=1}}$ *and* $\mathbf{PBA^{=1}} \not\subseteq \mathbf{NBA}$.

Proof sketch. A transformation from NBA into an equivalent probable PBA is obtained by using NBA that are *deterministic-in-limit*. These are NBA such that for each state p that is reachable from some accepting state $q \in F$ and each letter $a \in \Sigma$ state p has at most one outgoing a-transition. That is, as soon as an accepting state has been reached, the behavior from then on is deterministic. Courcoubetis and Yannakakis [7] presented an algorithm that turns a given NBA $\mathcal{N}$ into an equivalent NBA $\mathcal{N}_{\text{det}}$ that is deterministic-in-limit. A PBA $\mathcal{P}$ where $\mathcal{L}^{>0}(\mathcal{P})$ agrees with the language of $\mathcal{N}_{\text{det}}$ (and $\mathcal{N}$) is obtained by resolving the nondeterministic choices in $\mathcal{N}_{\text{det}}$ via distributions that assign non-zero probabilities to all transitions.

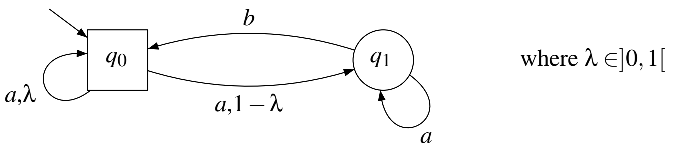


Fig. 2. PBA $\mathcal{P}_\lambda$ that accepts with the probable semantics the non-ω-regular language L_λ

The above shows that the class of ω-regular languages is contained in $\mathbf{PBA^{>0}}$. We now provide an example for a language $L \in \mathbf{PBA^{>0}}$ which is not ω-regular. For L we can take the language

$$L_\lambda \overset{\text{def}}{=} \left\{ a^{k_1} b a^{k_2} b a^{k_3} b \ldots : \prod_{i=1}^{\infty} \left(1 - \lambda^{k_i}\right) > 0 \right\}$$

where λ is an arbitrary real number in the open interval $]0,1[$. The convergence condition which requires the infinite product over the values $1-\lambda^{k_i}$ to be positive can easily be shown to be non-ω-regular. Furthermore, it can be shown that there is no almost-sure PBA that accepts L_λ. However, a PBA $\mathcal{P}_\lambda$ with the probable semantics is shown in Figure 2. To see why $L_\lambda = \mathcal{L}^{>0}(\mathcal{P}_\lambda)$, let us first observe that all words in $\mathcal{L}^{>0}(\mathcal{P}_\lambda)$ must contain infinitely many b's. As $\mathcal{P}_\lambda$ cannot consume two consecutive b's, all words in $\mathcal{L}(\mathcal{P}_\lambda)$ have the form $a^{k_1} b a^{k_2} b a^{k_3} b \ldots$ where $k_1, k_2, \ldots$ is a sequence of positive natural numbers. We now show that

$$\mathsf{Pr}^{\mathcal{P}_\lambda}(a^{k_1} b a^{k_2} b a^{k_3} b \ldots) = \prod_{i=1}^{\infty} \left(1 - \lambda^{k_i}\right)$$

The factors $1-\lambda^{k_i}$ stand for the probability to move from state q_0 to q_1 when reading the subword a^{k_i}. With the remaining probability λ^{k_i}, the automaton $\mathcal{P}_\lambda$ stays in state q_0, but then letter b at position $k_1+\ldots+k_i+i$ of the input word $a^{k_1}ba^{k_2}ba^{k_3}b\ldots$ cannot be consumed and $\mathcal{P}_\lambda$ rejects. Hence, the probability for run fragments of the form $q_0\ldots q_0q_1\ldots q_1\,q_0$ that are generated while reading the subword $a^{k_i}b$ is precisely $1-\lambda^{k_i}$. This yields that the infinite product of these values agrees with the acceptance probability for the input word $a^{k_1}ba^{k_2}ba^{k_3}b\ldots$.

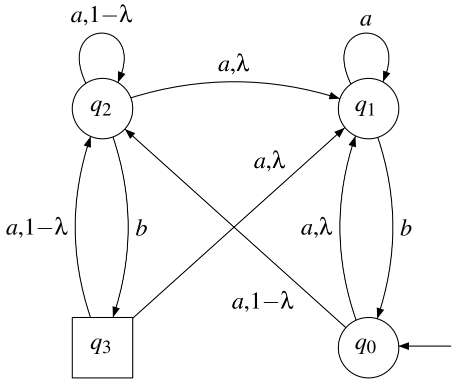


Fig. 3. PBA $\tilde{\mathcal{P}}_\lambda$ that accepts the non-ω-regular language $\tilde{L}_\lambda$

It remains to show that **NBA** $\not\subseteq$ **PBA**$^{=1}$ and **PBA**$^{=1}$ $\not\subseteq$ **NBA**. An example for an ω-regular language that is not recognizable by an almost-sure PBA is $(a+b)^*a^\omega$. Moreover, for the PBA $\tilde{\mathcal{P}}_\lambda$ in Figure 3 the acceptance probability of each word is either 0 or 1 and the accepted language $\mathcal{L}^{>0}(\tilde{\mathcal{P}}_\lambda)=\mathcal{L}^{=1}(\tilde{\mathcal{P}}_\lambda)$ consists of all words

$$a^{k_1}bca^{k_2}bca^{k_3}bc\ldots \text{ where } k_1,k_2,\ldots\geq 1 \text{ and } \prod_{i=1}^{\infty}(1-\lambda^{k_i})=0.$$

As this language is clearly not ω-regular, this shows **PBA**$^{=1}$ $\not\subseteq$ **NBA**. □

Relevance of the concrete transition probabilities. For the threshold semantics it is clear that the precise transition probabilities might be crucial for the accepted language. That is, if we are given a PBA and modify the nonzero transition probabilities then also the accepted threshold language might change, even for fixed threshold. However, one might expect that the precise transition probabilities are irrelevant for the probable and almost-sure semantics as they just rely on a qualitative criterion. Indeed, in the context of finite-state Markov decision processes it is known that whether or not a given linear time property holds with positive probability or almost surely just depends on the underlying graph, but not on the concrete transition probabilities. However, the languages of PBA under the probable and almost-sure semantics are sensitive to the distributions for the successor states. Consider again the PBA $\mathcal{P}_\lambda$ and $\tilde{\mathcal{P}}_\lambda$ of Figures 2 and 3 and two values λ and $\nu\in]0,1[$ with $\lambda<\nu$. For any sequence $(k_i)_{i\geq 1}$ of natural numbers k_i where the infinite product over the values $1-\nu^{k_i}$ converges to some positive value, also the infinite product over the values $1-\lambda^{k_i}$ is positive, as we have $1-\nu^{k_i} < 1-\lambda^{k_i}$. In fact, whenever $\lambda < \nu$ then there are sequences $(k_i)_{i\geq 1}$ such that the product of the values $1-\lambda^{k_i}$ converges to some positive real number, while the product of the values $1-\nu^{k_i}$ has value 0. Thus, $\mathcal{L}^{>0}(\mathcal{P}_\nu)$ is a proper subset of $\mathcal{L}^{>0}(\mathcal{P}_\lambda)$ and $\mathcal{L}^{=1}(\tilde{\mathcal{P}}_\nu)$ a proper superset of $\mathcal{L}^{=1}(\tilde{\mathcal{P}}_\lambda)$.

PBA for ω-regular languages. Using a criterion that has some similarities with the condition for isolated cutpoints for PFA, [3] presents a characterization of a subclass of PBA with the probable semantics that covers exactly the ω-regular languages. An alternative purely syntactic condition has been provided recently by Chadha, Sistla and Viswanathan [5]. Concerning the efficiency of probable PBA as an automata-model for ω-regular languages, it turns out that PBA are not comparable to their nondeterministic counterparts. There are examples for families $(L_n)_{n\geq 1}$ of ω-regular languages that are accepted by NBA of linear size, while each PBA $\mathcal{P}$ with $\mathcal{L}^{>0}(\mathcal{P}) = L_n$ has $\Omega(2^n)$ states. An example for such a family of languages is $\big((a+b)^*a(a+b)^nc\big)^\omega$. Vice versa, the languages $L_n = \big\{xy^\omega : x,y \in \{a,b\}^*, |y| = n\big\}$ can be recognized by probable PBA of size $O(n)$, while any NSA for L_n has $\Omega(2^n/n)$ states. (See [3].)

For languages over finite words, "pumping lemmas" provide a useful tool to prove that certain languages are not recognizable by some type of automata. Indeed also for PBA with the probable semantics one can also establish such a pumping lemma:

Theorem 3 (Pumping lemma for $\mathbf{PBA}^{>0}$). *For each PBA $\mathcal{P}$ and each word $\sigma \in \mathcal{L}^{>0}(\mathcal{P})$ and all $n \in \mathbb{N}$ there exist finite words u,v,w and an infinite word x such that (1) $\sigma = uvwx$, (2) $|u| = n$, $|vw| \leq |\mathcal{P}|$ and $|w| \geq 1$ and (3) $uvw^kx \in \mathcal{L}^{>0}(\mathcal{P})$ for all $k \in \mathbb{N}$.*

Using Theorem 3 one can, e.g., show that the ω-context-free language $\{a^nb^n\sigma : \sigma \in (a+b)^\omega, n \geq 0\}$ is not contained in $\mathbf{PBA}^{>0}$.

3 Composition Operators for PBA

The most important composition operators for any class of languages over infinite words are the standard set operations union, intersection and complementation. We consider here only the probable and almost-sure semantics. To the best of our knowledge, composition operators for PBA with the threshold semantics have not been studied yet.

Theorem 4. ***PBA*$^{>0}$** *is closed under union, intersection and complementation, while* ***PBA*$^{=1}$** *is closed under union and intersection, but not under complementation.*

Union. Given two PBA $\mathcal{P}_1$ and $\mathcal{P}_2$ with the probable semantics and initial distributions μ_1 and μ_2, respectively, we consider the PBA $\mathcal{P}_1 \uplus \mathcal{P}_2$ that arises from the disjoint union of $\mathcal{P}_1$ and $\mathcal{P}_2$ with the initial distribution $\mu(q) = \frac{1}{2}\mu_i(q)$ if q is a state in $\mathcal{P}_i$. If F_1 and F_2 are the sets of accepting states in $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively, then $\mathcal{P}$ requires to visit $F_1 \cup F_2$ infinitely often. We then have $\mathcal{L}^{>0}(\mathcal{P}_1 \uplus \mathcal{P}_2) = \mathcal{L}^{>0}(\mathcal{P}_1) \cup \mathcal{L}^{>0}(\mathcal{P}_2)$. For almost-sure PBA $\mathcal{P}_1$, $\mathcal{P}_2$, we consider the product automaton $\mathcal{P}$ of $\mathcal{P}_1$ and $\mathcal{P}_2$ (which runs $\mathcal{P}_1$ and $\mathcal{P}_2$ in parallel) and declare all states $\langle q_1, q_2\rangle$ with $q_1 \in F_1$ or $q_2 \in F_2$ to be accepting. Then, $\mathcal{L}^{=1}(\mathcal{P}) = \mathcal{L}^{=1}(\mathcal{P}_1) \cup \mathcal{L}^{=1}(\mathcal{P}_2)$.

Intersection. Given two PBA $\mathcal{P}_1$ and $\mathcal{P}_2$ with the probable semantics, we can reuse the ideas of an intersection operator on NBA and use a product construction $\mathcal{P}_1 \times \mathcal{P}_2$ with a generalized Büchi condition requiring that (i) an accepting state of $\mathcal{P}_1$ is visited infinitely often and (ii) an accepting state of $\mathcal{P}_2$ is visited infinitely often. This generalized PBA can then be transformed into an equivalent PBA with the probable semantics as in the nondeterministic case, using two copies of $\mathcal{P}_1 \times \mathcal{P}_2$. For PBA $\mathcal{P}_1$ and $\mathcal{P}_2$ with the almost-sure semantics, intersection can be realized as union for PBA under the probable semantics. That is, $\mathcal{L}^{=1}(\mathcal{P}_1 \uplus \mathcal{P}_2) = \mathcal{L}^{=1}(\mathcal{P}_1) \cap \mathcal{L}^{=1}(\mathcal{P}_2)$.

Complementation. In the proof of Theorem 2 we saw that $(a+b)^*a^\omega \notin \mathbf{PBA}^{=1}$, while its complement $(a^*b)^\omega$ is DBA-recognizable and therefore belongs to $\mathbf{PBA}^{=1}$. Thus, $\mathbf{PBA}^{=1}$ is not closed under complementation.

Let us now address the question how to complement a given PBA $\mathcal{P}$ with the probable semantics. The rough idea is to transform the given PBA $\mathcal{P}$ into an equivalent PRA $\mathcal{P}_R$ (probabilistic automaton with Rabin acceptance). Complementation is then realized by switching from $\mathcal{P}_R$ to a PSA $\mathcal{P}_S$ (probabilistic automaton with Streett acceptance) for the complement language and finally transform $\mathcal{P}_S$ into an equivalent PBA with the probable semantics. For the transformation $\mathcal{P}_R \rightsquigarrow \mathcal{P}_S$ we rely on the duality of Rabin and Streett acceptance. However, complementing the acceptance condition does not complement the accepted language if there are words with acceptance probability strictly between 0 and 1. For this reason, in the first step we transform $\mathcal{P}$ into an equivalent 0/1-PRA $\mathcal{P}_R$ which means a PRA such that $\mathsf{Pr}^{\mathcal{P}_R}(\sigma) \in \{0,1\}$ for all words σ. Hence, $\mathcal{L}^{>0}_{\text{Büchi}}(\mathcal{P}) = \mathcal{L}^{>0}_{\text{Rabin}}(\mathcal{P}_R) = \mathcal{L}^{=1}_{\text{Rabin}}(\mathcal{P}_R)$. We can then deal with $\mathcal{P}_R = \mathcal{P}_S$ to obtain $\mathsf{Pr}^{\mathcal{P}_S}(\sigma) = 1 - \mathsf{Pr}^{\mathcal{P}_R}(\sigma) \in \{0,1\}$ and therefore

$$\mathcal{L}^{>0}_{\text{Streett}}(\mathcal{P}_S) = \Sigma^\omega \setminus \mathcal{L}^{>0}_{\text{Rabin}}(\mathcal{P}_R)$$

The schema for the complementation of probable PBA semantics is sketched in Figure 4. This rough schema has some similarities with the complementation of NBA via Safra's determinisation operator [16]. In this approach, one first switches from a given NBA $\mathcal{N}$ to an equivalent DRA $\mathcal{D}_R$ which yields a DSA $\mathcal{D}_S$ for the complement language. Finally $\mathcal{D}_S$ can be transformed into an equivalent NBA.

The idea for the first step in Figure 4 is to design a 0/1-PRA $\mathcal{P}_R$ that generates up to n sample runs of $\mathcal{P}$ and checks whether at least one of them is accepting, where n is the number of states in $\mathcal{P}$. If so then $\mathcal{P}_R$ accepts, otherwise it rejects. For the details of this construction we refer to [2,10].

$$\begin{tikzcd}
\begin{array}{c}\text{PBA } \mathcal{P} \text{ with}\\ \mathcal{L}^{>0}(\mathcal{P}) = L\end{array} \arrow[r, rightsquigarrow] & \begin{array}{c}\text{0/1-PRA } \mathcal{P}_R\\ \text{for } L\end{array} \arrow[r, rightsquigarrow] & \begin{array}{c}\text{0/1-PSA } \mathcal{P}_S\\ \text{for } \overline{L}\end{array} \arrow[r, rightsquigarrow] & \begin{array}{c}\text{PBA } \overline{\mathcal{P}} \text{ with}\\ \mathcal{L}^{>0}(\overline{\mathcal{P}}) = \overline{L}\end{array}
\end{tikzcd}$$

Fig. 4. Complementation of a PBA under the probable semantics

The last step of Figure 4 relies on the transformation from PSA to PBA. [3] presents an algorithm to transform a given PSA $\mathcal{P}_S$ with ℓ acceptance pairs into an equivalent PBA with the probable semantics of size $O(\ell^2|\mathcal{P}_S|)$. It is worth noting that there are families $(L_n)_{n\geq 0}$ of languages $L_n \subseteq \Sigma^\omega$ that are recognizable by nondeterministic Streett automata of size $O(n)$, while each nondeterministic Büchi automaton for L_n has 2^n or more states [17]. Thus, the polynomial transformation from Streett to Büchi acceptance is specific for the probabilistic case.

In particular, the transformations between PBA, PRA and PSA show that for probabilistic automata with Rabin or Streett acceptance, the probable semantics and the almost-sure semantics have the same power. Thus:

Theorem 5. $\boldsymbol{PBA}^{>0} = \boldsymbol{PRA}^{>0} = \boldsymbol{PSA}^{>0} = \boldsymbol{PRA}^{=1} = \boldsymbol{PSA}^{=1}$

Remind that $\mathbf{PBA}^{=1}$ is a proper subclass of $\mathbf{PBA}^{>0}$. Thus, within the different types of probabilistic ω-automata, almost-sure PBA play a similar role as DBA within the class of (non)deterministic ω-automata.

The sketched transformation from probable PBA to 0/1-PRA relies on a certain powerset construction and can cause an exponential blow-up. Vice versa, given a probable PRA an equivalent probable PBA can be constructed using a similar construction as it is known for the transformation of an NRA into an equivalent NBA. If the given PRA $\mathcal{P}_R$ has ℓ acceptance pairs then the size of the constructed PBA is bounded by $O(\ell|\mathcal{P}_R|)$.

4 Decision Problems for PBA

For many applications of automata-like models, it is important to have (efficient) decision algorithms for some fundamental problems, like checking emptiness or language inclusion. For instance, the automata-based approach [19] for verifying ω-regular properties of a nondeterministic finite-state system relies on a reduction to the emptiness problem for NBA.

Given the undecidability results for almost all relevant decision problems for PFA (see e.g. [12]), it is clear that the emptiness problem and related problems are undecidable for PBA with the threshold semantics. Unfortunately this also holds for the probable semantics:

Theorem 6 (Undecidability of PBA with the probable semantics). *The following problems are undecidable:*

emptiness: *given a PBA $\mathcal{P}$, does $\mathcal{L}^{>0}(\mathcal{P}) = \emptyset$ hold?*
universality: *given a PBA $\mathcal{P}$ with the alphabet Σ, does $\mathcal{L}^{>0}(\mathcal{P}) = \Sigma^\omega$ hold?*
equivalence: *given two PBA $\mathcal{P}_1$ and $\mathcal{P}_2$, does $\mathcal{L}^{>0}(\mathcal{P}_1) = \mathcal{L}^{>0}(\mathcal{P}_2)$ hold?*

To prove undecidability of the emptiness problem, we provided in [2] a reduction from a variant of the emptiness problem for probabilistic finite automata (PFA) which has been shown to be undecidable [12]. Undecidability of the universality problem then follows by the effectiveness of complementation for PBA. Undecidability of the PBA-equivalence problem is an immediate consequence of the undecidability of the emptiness problem (just consider $\mathcal{P}_1 = \mathcal{P}_2$ and $\mathcal{P}_2$ a PBA for the empty language).

A consequence of Theorem 6 is that PBA are not appropriate for verification algorithms. Consider, e.g., a finite-state transition system $\mathcal{T}$ and suppose that a linear-time property p to be verified for $\mathcal{T}$ is specified by a PBA $\mathcal{P}$ in the sense that $\mathcal{L}(\mathcal{P})$ represents all infinite behaviors where property p holds. (Typically p is a language over some alphabet $\Sigma = 2^{\mathrm{AP}}$ where AP is a set of atomic propositions and the states in $\mathcal{T}$ are labeled with subsets of AP.) Then, the question whether all traces of $\mathcal{T}$ have property p is reducible to the universality problem for PBA and therefore undecidable. Similarly, the question whether $\mathcal{T}$ has at least one trace where p holds is reducible to the emptiness problem for PBA and therefore undecidable too. Another important consequence of Theorem 6 is that it yields the undecidability of the verification problem for partially observable Markov decision processes against ω-regular properties (see [2]).

Theorem 7 (Decidability of PBA with the almost-sure semantics). *The following problems are decidable:*

emptiness: *given a PBA $\mathcal{P}$, does $\mathcal{L}^{=1}(\mathcal{P}) = \emptyset$ hold?*
universality: *given a PBA $\mathcal{P}$ with the alphabet Σ, does $\mathcal{L}^{=1}(\mathcal{P}) = \Sigma^\omega$ hold?*

Let us briefly sketch how the emptiness problem for almost-sure PBA can be solved by a reduction to a certain variant of probabilistic reachability problems. Given an almost-sure $\mathcal{P} = (Q, \Sigma, \delta, \mu_0, F)$, then we may add an additional trap state p_F which can be entered from each accepting state in $\mathcal{P}$ with probability $1/2$. That is, we regard $\mathcal{P}' = (Q \cup \{p_F\}, \Sigma, \delta', \mu_0, \{p_F\})$ where $p_F \notin Q$ and for all $p \in F$, $q \in Q$ and $a \in \Sigma$ we have $\delta'(p_F, a, p_F) = 1$, $\delta'(p, a, p_F) = \frac{1}{2}$, $\delta'(p, a, q) = \frac{1}{2} \cdot \delta(p, a, q)$, and $\delta'(\cdot) = \delta(\cdot)$ in all remaining cases. Then, $\mathcal{L}^{=1}(\mathcal{P}) = \mathcal{L}^{=1}(\mathcal{P}')$ and the problem whether $\mathcal{L}^{=1}(\mathcal{P}')$ is empty can be solved by checking whether there exists an infinite word σ such that almost all runs for σ in $\mathcal{P}'$ will eventually enter state q_F. This problem is solvable by means of a certain powerset construction.

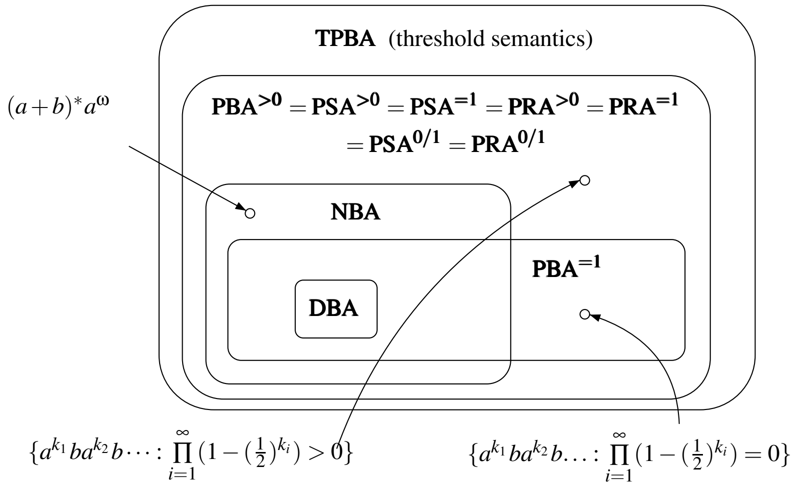

TPBA (threshold semantics)
$\mathbf{PBA}^{>0} = \mathbf{PSA}^{>0} = \mathbf{PSA}^{=1} = \mathbf{PRA}^{>0} = \mathbf{PRA}^{=1} = \mathbf{PSA}^{0/1} = \mathbf{PRA}^{0/1}$
NBA
$\mathbf{PBA}^{=1}$
DBA
$(a+b)^* a^\omega$
$\{a^{k_1} b a^{k_2} b \cdots : \prod_{i=1}^{\infty} (1 - (\frac{1}{2})^{k_i}) > 0\}$
$\{a^{k_1} b a^{k_2} b \ldots : \prod_{i=1}^{\infty} (1 - (\frac{1}{2})^{k_i}) = 0\}$

Fig. 5. Classes of languages recognizable by probabilistic ω-automata

5 Conclusion

We gave a summary of the fundamental properties of probabilistic acceptors for infinite words formalized by PBA, PRA or PSA. Figure 5 illustrates the presented results on the hierarchy of languages accepted by probabilistic ω-automata. In this paper, we mainly concentrated on the probable and almost-sure semantics and pointed out some major differences to nondeterministic (or alternating) ω-automata concerning the expressiveness, efficiency and decidability, which makes PBA interesting at least from a theoretical point of view.

The undecidability of the emptiness problem and related problems shows that PBA with the probable semantics are not adequate for algorithmic purposes, e.g., the verification of systems with nondeterministic behaviors. The situation changes if the system to be verified is purely probabilistic (i.e., modelled by a Markov chain). In this case some decidability results for the verification problem against PBA-specifications can be established [3]. In [2] we showed that PBA can be regarded as special instances of partially-observable Markov decision processes (POMDPs). Hence, all negative results for PBA (undecidability) carry over from PBA to POMDP. Vice versa, for many algorithmic problems for POMDPs, algorithmic solutions for PBA with the almost-sure semantics can be combined with standard algorithms for (fully observable) Markov decision processes to obtain an algorithm that solves the analogous problem for POMDPs.

In [4], special types of PBA have been considered in the context of probabilistic monitors. Topological and other interesting properties of PBA have been studied in the recent paper [5] in this volume.

Given the wide range of application areas of probabilistic finite automata, there might be various other applications of probabilistic ω-automata. For instance, the concept of probabilistic ω-automata is also related to partial-information games with ω-regular winning objectives [6] or could serve as starting point for studying quantum automata over infinite inputs, in the same way as PFA yield the basis for the definition of quantum finite automata [11,1]. For these reasons, we argue that the concept of probabilistic ω-automata is an interesting new research field with plenty of open questions (e.g., logical or algebraic characterizations of $\mathbf{PBA}^{>0}$ or $\mathbf{PBA}^{=1}$) that might lead to interesting applications.

References

1. Ambainis, A., Freivalds, R.: 1-way quantum finite automata: strengths, weaknesses and generalizations. In: Proc. of the 39th Symposium on Foundations of Computer Science (FOCS 1998). IEEE Computer Society Press, Los Alamitos (1998)
2. Baier, C., Bertrand, N., Größer, M.: On decision problems for probabilistic büchi automata. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 287–301. Springer, Heidelberg (2008)
3. Baier, C., Grösser, M.: Recognizing ω-regular languages with probabilistic automata. In: Proc. of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005), pp. 137–146. IEEE Computer Society Press, Los Alamitos (2005)
4. Chadha, R., Sistla, A.P., Viswanathan, M.: On the expressiveness and complexity of randomization in finite state monitors. In: Proc. of the 23rd IEEE Symposium on Logic in Computer Science (LICS 2008), pp. 18–29. IEEE Computer Society Press, Los Alamitos (2008)

5. Chadha, R., Sistla, A.P., Viswanathan, M.: Power of randomization in automata on infinite strings. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 229–243. Springer, Heidelberg (2009)
6. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for ω-regular games with imperfect information. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 287–302. Springer, Heidelberg (2006)
7. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
8. Freivalds, R.: Probabilistic two-way machines. In: Gruska, J., Chytil, M.P. (eds.) MFCS 1981. LNCS, vol. 118, pp. 33–45. Springer, Heidelberg (1981)
9. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
10. Größer, M.: Reduction Methods for Probabilistic Model Checking. PhD thesis, Technical University Dresden, Faculty for Computer Science (2008)
11. Kondacs, A., Watrous, J.: On the power of quantum finite state automata. In: Proc. of the 38th Symposium on Foundations of Computer Science (FOCS 1997), pp. 66–75. IEEE Computer Society Press, Los Alamitos (1997)
12. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and related stochastic optimization problems. Artificial Intelligence 147(1-2), 5–34 (2003)
13. Paz, A.: Some aspects of probabilistic automata. Information and Control 9 (1966)
14. Paz, A.: Introduction to probabilistic automata. Academic Press Inc., London (1971)
15. Rabin, M.O.: Probabilistic automata. Information and Control 6(3), 230–245 (1963)
16. Safra, S.: On the complexity of ω-automata. In: Proc. of the 29th Symposium on Foundations of Computer Science (FOCS 1988), pp. 319–327. IEEE Computer Society Press, Los Alamitos (1988)
17. Safra, S., Vardi, M.Y.: On ω-automata and temporal logic. In: Proc. of the 21st ACM Symposium on Theory of Computing (STOC 1989), pp. 127–137. ACM, New York (1989)
18. Thomas, W.: Languages, automata, and logic. Handbook of formal languages 3, 389–455 (1997)
19. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. of the 1st IEEE Symposium on Logic in Computer Science (LICS 1986), pp. 332–345. IEEE Computer Society Press, Los Alamitos (1986)

Bigraphical Categories

Robin Milner

University of Cambridge, The Computer Laboratory,
J J Thomson Avenue, Cambridge CB3 0FD, UK
and
University of Edinburgh, Informatics Forum,
10 Crichton Street, Edinburgh EH8 9AB, UK

Abstract. Bigraphs are a candidate model that aims to provide a theoretical platform for ubiquitous computing systems. This short paper summarises the categories, and the functors between them, that represent the structure of that theory.

Keywords: bigraph, category, process, ubiquitous computing.

1 Introduction

Bigraphs are an experimental model, aiming to provide a rigorous platform for designing, simulating and analysing ubiquitous informatic systems. Such artifacts will increasingly become embedded in our natural and artificial environments.

In a recently published book [1] I argue for the need for such a platform, and I propose bigraphs to meet that need. The book has a full bibliography, and on my website [2] the reader will find slides for a course of six lectures, notes that correlate the lectures with the book, a collection of exercises with solutions, and a commentary on the book that will grow with time, including corrections in order of their discovery.

This note assumes some familiarity with bigraphs, but its main purpose is to emphasize an important aspect of their theory: the essential part played by various forms of category. This aspect is treated to a certain extent in the book. By focussing upon it in this summary I hope to cast some of the theory in a vivid form, free of many technical details.

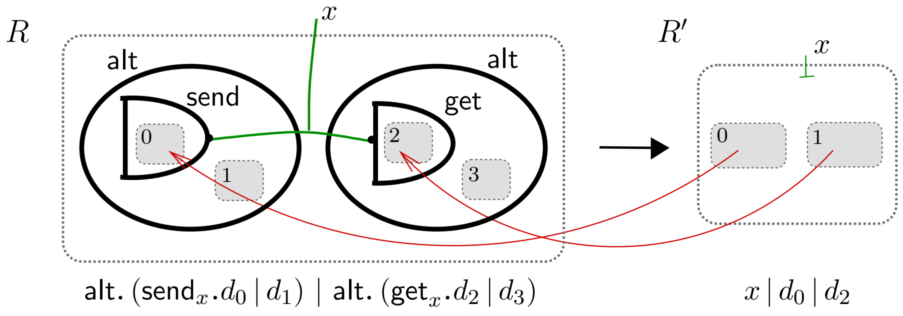
2 Preliminaries

To introduce bigraphs informally, it is useful to see how they encode CCS. Recall the familiar CCS reaction rule:

$$(\overline{x}.P + A) \mid (x.Q + B) \longrightarrow P \mid Q \ .$$

Here the parameters P and Q stand for arbitrary processes, and A and B for arbitrary alternations (essentially sums of prefix-guarded processes). The rule asserts that a communication may occur on the channel x, discarding all the alternatives for its two participants.

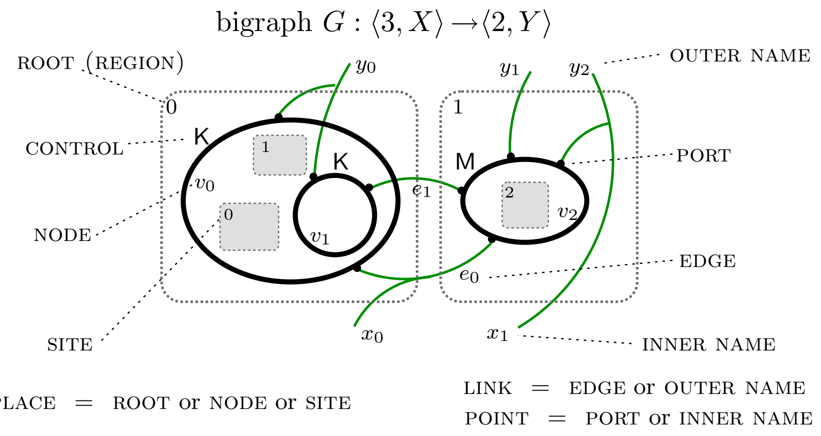
CCS and other process calculi have been embedded in the bigraph model, which thus provides them with uniform behavioural theory. Here, we recall how the CCS reaction rule is expressed bigraphically:


R
x
alt
send
alt
get
0
1
2
3
R'
x
0
1
$\mathsf{alt}.\,(\mathsf{send}_x.d_0 \mid d_1) \mid \mathsf{alt}.\,(\mathsf{get}_x.d_2 \mid d_3)$
$x \mid d_0 \mid d_2$

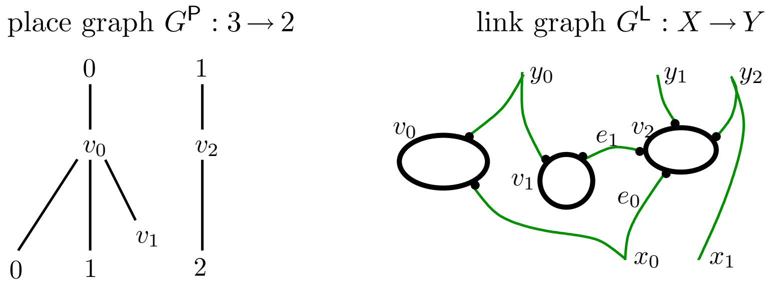
The bigraph R, the *redex*, has sites (grey boxes) corresponding to the four parameters. In the *reactum* R', the result of the reaction, two of these parameters (indicated by the back pointing arrows) survive. Under the diagram is the algebraic form of the rule, in which the tensor product $d = d_0 \otimes \cdots \otimes d_3$ represents the four parameters. In the algebraic form it is worth noting that the operations of parallel product (|) and nesting (.), considered primitive in process calculi, are actually *derived* from the operators of a symmetric monoidal category.

For those not familiar with bigraphs, this example should give an idea of what bigraphs are and how their dynamic reconfiguration can be defined. For the rest of the paper we focus upon the static structure of bigraphs, only mentioning dynamics when it imposes a requirement on that structure.

The following diagram is designed to illustrate the taxonomy of bigraphs, which are so called because each bigraph has a *placing* structure and a *linking* structure. G is an arrow in a certain kind of category, whose objects are *interfaces* $I = \langle m, X\rangle$, where m is a finite ordinal and X a finite name-set.


bigraph $G : \langle 3, X\rangle \to \langle 2, Y\rangle$
ROOT (REGION)
CONTROL
NODE
SITE
OUTER NAME
PORT
EDGE
INNER NAME
y_0
y_1
y_2
0
1
K
K
M
v_0
v_1
v_2
e_1
e_0
x_0
x_1
PLACE = ROOT or NODE or SITE
LINK = EDGE or OUTER NAME
POINT = PORT or INNER NAME

Here we have $G : I \to J$ with $I = \langle 3, X\rangle$ and $J = \langle 2, Y\rangle$. This reflects that G has three *sites* (0,1,2) and two *roots* or *regions* (0,1); the name sets $X = \{x_0, x_1\}$ and $Y = \{y_0, y_1, y_2\}$ are clearly seen. Each node has a *control* such as K, which determines its arity (number of ports). The placing of G is represented by nesting, and its linking by (hyper)*links* (outer names or edges) connecting *points* (inner names or ports); the links themselves are either *closed* (e.g. the edge e_0) or *open* (e.g. the outer name y_0). Such a structure is called a *bigraph* because it combines two *constituents* with the same node-set, here $\{v_0, v_1, v_2\}$. These constituents are called a *place graph* and a *link graph*. For G, they are denoted by $G^{\mathsf{P}} : 3 \to 2$ (a forest) and $G^{\mathsf{L}} : X \to Y$ (a map from points to links), and pictured thus:


place graph $G^{\mathsf{P}} : 3 \to 2$
0
1
v_0
v_2
v_1
0
1
2
link graph $G^{\mathsf{L}} : X \to Y$
y_0
y_1
y_2
v_0
e_1
v_2
v_1
e_0
x_0
x_1

We write the combination $G = \langle G^{\mathsf{P}}, G^{\mathsf{L}}\rangle$. Composition in the category $\mathrm{P{\small G}}(\mathcal{K})$ of place graphs over signature $\mathcal{K}$ is defined by coalescing each site of one with the corresponding root of the other and erasing their ordinals; similarly in the category $\mathrm{L{\small G}}(\mathcal{K})$ of link graphs, coalescing inner with outer names. (The identities are obvious.) Finally, composition in the category $\mathrm{B{\small G}}(\mathcal{K})$ of bigraphs is defined by

$$G \circ F \stackrel{\text{def}}{=} \langle G^{\mathsf{P}} \circ F^{\mathsf{P}}, G^{\mathsf{L}} \circ F^{\mathsf{L}}\rangle \ .$$

Observe that we are dealing with *abstract* bigraphs; that is, equivalence classes under bijection of nodes and edges. Thus, in forming a composition, we assume that the node- and edge-sets of the two components are distinct.

The tensor product of two interfaces $I_i = \langle m_i, X_i\rangle$ is defined by $I_0 \otimes I_1 \stackrel{\text{def}}{=} \langle m_0{+}m_1,\ X_0 \uplus X_1\rangle$, when X_0 and X_1 are disjoint. When the interface products are defined, the tensor product of both place and link graphs is defined simply by juxtaposition, and then the product of bigraphs is defined by combination of constituents.

Apart from the partiality of the product of link graphs we therefore have that all three categories are symmetric and (partial-)monoidal; for short, *spm* categories.

3 The Bigraphical Categories

Just as the structure of a bigraph can be usefully captured as a categorical arrow, so bigraphical theory can be revealingly structured by functors between a variety of categories. For the remainder of this short note we shall discuss the role played by the functors in the following diagram, in which every square commutes. All the categories are instances of a general class called *wide s-categories*, which we define later.

$$\begin{tikzcd}
\text{`}\textsc{Bg}(\Sigma) \arrow[r, "\textit{forget sorting}"] \arrow[d, "\textit{forget support}"'] & \text{`}\textsc{Bg}(\mathcal{K}) \arrow[r, "\textit{forget signature}"] \arrow[d] & \text{`}\textsc{Bg} \arrow[d] & & \\
\textsc{Bg}(\Sigma) \arrow[r] \arrow[d, "\textit{refine}"'] & \textsc{Bg}(\mathcal{K}) \arrow[r] \arrow[d] & \textsc{Bg} \arrow[d] \arrow[dr, "\textit{project}"] & & \\
\textsc{Bg}(\Sigma') \arrow[r] & \textsc{Bg}(\mathcal{K}') & \textsc{Lg} & \textsc{Pg} \arrow[r, "\textit{width}"] & \textsc{Ord}
\end{tikzcd}$$

We have already begun with $\textsc{Bg}(\mathcal{K})$, in the middle of this diagram. There is an obvious forgetful functor that forgets the signature; its target $\textsc{Bg}$ is just bigraphs without controls. Much theory can be done in $\textsc{Bg}$; it is then less cluttered, and is often obviously preserved by retrofitting the controls. Since a bigraph is a combination of a place graph and a link graph, there are two obvious functors projecting bigraphs onto their constituents. Later we shall discuss the width functor to $\textsc{Ord}$.

Moving to the left, Σ stands for a *sorting*, which – for a given signature $\mathcal{K}$ – is usually an enrichment of interfaces coupled with a constraint upon the admissible bigraphs over $\mathcal{K}$. This enriched and constrained category is denoted by $\textsc{Bg}(\Sigma)$. Hitherto, almost every application of bigraphs has involved a sorting. For example, in the simple case of CCS, the sorting Σ_{ccs} requires that the nesting of nodes should interleave send- and get-nodes with alt-nodes. In a built environment, say a building, one would naturally require no room to contain another room.

It is not clear what the general definition of sortings Σ should be, beyond requiring that for each sorting Σ there be a forgetful functor from $\textsc{Bg}(\Sigma)$ to $\textsc{Bg}(\mathcal{K})$ for some $\mathcal{K}$. Birkedal *et al* [3] have proposed the broad definition that it should consist of any functor that is both surjective on objects and faithful (i.e. injective on each homset). This has the merit that it allows *binding bigraphs*, in which certain links are required to be confined to a given place, to be expressed as a sorting. Binding bigraphs have been used to encode both the λ-calculus and the π-calculus.

In this note we are avoiding details of dynamics, but it is important that when $\textsc{Bg}(\Sigma)$ is equipped with dynamics then its reconfigurations preserve the sorting condition.

Having dealt with the middle row of the diagram, consider now the bottom row. Bigraphs may be used to model a system at different levels of detail. This recalls the natural definition of a homomorphism of algebras, in which an algebra $\mathcal{A}$ is refined to an algebra $\mathcal{A}'$ by representing each single operator of $\mathcal{A}$ by a compound operation built from the operators of $\mathcal{A}'$. Here, then, we would represent each K-ion of Σ (i.e. each elementary bigraph consisting of a single K-node containing zero or more sites) by a compound bigraph in $\mathrm{B}\textsc{g}(\Sigma')$[1].

When $\mathrm{B}\textsc{g}(\Sigma)$ and $\mathrm{B}\textsc{g}(\Sigma')$ are both equipped with dynamics, one expects that the latter will correctly represent the former. This 'respect' can be defined in a variety of ways; we avoid the details here.

We now come to the top row. Hitherto we have dealt with abstract bigraphs, where we do not distinguish bigraphs that differ only in a bijection between their nodes and edges. This can be expressed in another way: in an abstract bigraph we choose not to distinguish between two 'occurrences' of the same control K. However, there are situations in which we may wish to do so. For example, if the control A represents human agents and R represents rooms, we may wish to answer a question about the history of humans' movement in a built environment, such as "Has any agent entered the same room twice?". This question cannot be answered unless we can distinguish between different occurrences, or identities, of people and rooms, so that they can be tracked throughout any sequence of reactions.

Another need for node identity arises (again in dynamics) when we wish to ask "how many ways does the redex R of a reaction rule occur in a given bigraph G?", or even "does only a part of R occur within G, so that a context H is needed to supply the missing part, allowing $H \circ G$ can perform the reaction?". The latter has been crucial in defining the behavioural equivalence of bigraphical agents, since we want this equivalence to be a congruence – i.e. preserved by all contexts.

In other computational models, notably the λ-calculus, occurrences are often identified by the *labelling* or *tagging* of terms. Indeed one proof of the well-known Church-Rosser theorem, via the so-called 'parallel moves' lemma, rests upon tagging. In bigraphs, since we already have node-identities $v_0, v_1, \ldots$ (and edge-identities $e_0, e_1, \ldots$) it is natural to employ them for tagging.

The simplest – and a very effective – way to do this is to refine our notion of category by no longer equating bigraphs that differ only by a bijection of nodes and edges. The notion of *precategory* is standard; it is like a category except that composition of $F : a \to b$ and $G : b \to c$ may not always be defined. With this hint, we define an *s-category* to be an spm category except that each arrow F is assigned a finite *support set* $|F|$, drawn from an infinite repertoire $\mathcal{S}$ of *support elements*; then for $G \circ F$ or $F \otimes G$ to be defined we require that $|F| \cap |G| = \emptyset$, and then the construction has support $|F| \cup |G|$[2].

Let us say that a *concrete* bigraph is one whose support is its nodes and edges. It is remarkable that we thus obtain all the ability needed to handle occurrences. Also, for many purposes, s-categories behave just as spm categories. For example, there is a notion of functor between them. In the top row of our diagram of

[1] Ions typically have rank 1, i.e. a single site, but a sorting can give them any rank.
[2] There are some simple auxiliary conditions that we ignore here.

bigraphical functors, we distinguish our s-categories of concrete bigraphs by the tag in `Bg. All that we have said about forgetting sorting and signature applies as before; we also have functors (shown vertically) that forget support. Indeed, an spm category is just an s-category in which all support-sets are empty.

4 Activity

There is one final feature of our functor diagram to be explained: the role of Ord, the category of finite ordinals and maps between them. Recall that each finite ordinal m is the set of its predecessors, i.e. $m = \{0, 1, \ldots, m-1\}$. The objects in Pg are finite ordinals, and the functor $\mathsf{width} : \mathrm{P}\textsc{g} \to \mathrm{O}\textsc{rd}$ is defined as the identity on objects, while for any place graph $P : m \to n$ we take $\mathsf{width}(P)$ to be the map $\mathsf{root} : m \to n$, where $\mathsf{root}(i) \in n$ is the unique root (region) of P that contains the site $i \in m$.

In introducing dynamics to bigraphs, we wish to define not only *what* can happen, but *where* it can happen. A simple way to do this (there are more subtle ways) is to enrich the signature by declaring whether or not each control K is *active* – i.e. permits activity within any K-node. For example, the CCS control alt is not active; it forbids internal reactions (but, as we have seen, it can take part in activity and thereby activates its occupants). As another example, consider a building in a built environment. It normally permits activity,i.e. its control B is active. But a power-cut may cause its control to change its control to B', which is passive, preventing activity while the power is cut.

In studying behaviour, and behavioural equivalence, it is important to know where action can or cannot occur. The structure of place graphs is such that, for $G : \langle m, X\rangle \to \langle n, Y\rangle$ and F with outer face $\langle m, X\rangle$, if an action of F occurs in region $i \in m$, then we know it occurs in region $j \in n$ of $G \circ F$, where j is the unique root of G that lies above the site i.[3] This property of place graphs becomes essential in proving that behavioural equivalence is a congruence, i.e. preserved by context.

On the other hand, we wish our behavioural theory to apply to as broad a range of reactive systems as possible. We immediately ask: Can we develop a behavioural theory for s-categories in general, not just for bigraphical ones? It turns out that s-categories in general are not specific enough about locality. But let us define a *wide* s-category to be one for which there is a functor to Ord. This is more general than a bigraphical category (e.g. it has no notion of node) but has exactly the structure required to yield non-trivial behavioural congruences, which has always been a driving motive behind the bigraph model.

5 Conclusion

This short summary has attempted to reveal the importance of categorical notions in behavioural models, and in particular in bigraphs. Two points stand

[3] An action also be located in more that one place; this causes no difficulty.

out. First, categories can smoothly handle behaviour in a discrete space, which is becoming increasingly important in the applications of informatics. Second, the structure of the behavioural theory can rest firmly upon functors, which are employed to correlate aspects of the theory that are best expressed in different categories.

References

1. Milner, R.: The Space and Motion of Communicating Agents. Cambridge University Press, Cambridge (2009)
2. Milner, R.: `http://www.cl.cam.ac.uk/~rm135`
3. Birkedal, L., Debois, S., Hildebrandt, T.: Sortings for reactive systems. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 248–262. Springer, Heidelberg (2006)

BlenX
Static and Dynamic Semantics

Corrado Priami, Paola Quaglia, and Alessandro Romanel

CoSBi and Università di Trento, Italy

Abstract. We introduce a new programming language called BlenX. It is strongly inspired to process calculi and it is an evolution of Beta-binders. It has been specifically designed and implemented to model and simulate biological systems, but it is general enough to handle core computer science problems as well.

1 Introduction

Biological systems share many characteristics with distributed and mobile computer systems. Biological systems are characterized by millions of simultaneously interacting entities, by the absence of centralized points of control, by a spatial structure that resembles administrative domains, and by the movement in space of their components. The recent field of algorithmic systems biology [3] proposes the use of programming languages to model biological systems enhancing current modeling capabilities (richness of aspects that can be described as well as easiness, composability and reusability of models) [4]. The metaphor which inspires this idea is one where biological entities are represented as programs being executed simultaneously and the interaction of two entities is represented by the exchange of a message between the programs representing those entities in the model [6]. The biological entities involved in the biological process and the corresponding programs in the abstract model are in a 1:1 correspondence, thus coping by construction with the combinatorial explosion of variables needed in the mathematical approach to describe the whole set of states through which a single component can pass.

The metaphor above explicitly refers to concurrency. Indeed concurrency is endemic in nature, and we see this in examples ranging from atoms, molecules in living organisms, organisms themselves and populations to astronomy. If we are going to re-engineer artificial systems to match the efficiency, resilience, adaptability and robustness of natural systems, then concurrency must be a core design principle that, at the end of the day, will simplify the entire design and implementation process. Concurrency must not be considered a tool to improve the performance of sequential programming languages and architectures, which is the standard practice in most actual cases. Some programming languages that address concurrency as a core primitive issue and aim at modeling biological systems are emerging (see, e.g., [1,7]) from the field of process calculi. These concurrent programming languages are very promising for the establishment of a link between artificial concurrent programming and natural phenomena, thus contributing to the exposure of computer science to experimental, natural sciences. Furthermore, concurrent programming languages are suitable candidates for easily and efficiently expressing the mechanistic rules that propel algorithmic systems biology. The suitability of these languages is reinforced by their clean and formal definition that supports the verification of properties.

BlenX is a programming language designed explicitly to model biological systems. BlenX is an evolution of Beta-binders [5]. A program is a collection of boxes (representing biological components) with typed and dynamically varying interfaces running in parallel. The status of an interface (binder) can be free or complexed (when the binder is bound to the binder of another entity). Binders are associated with two types describing what can flow along the interface and the properties of the interfaces. Furthermore, there are primitives that can change the values associated to the binder to change the status and the properties of the interface.

The main step of computation for BlenX is the interaction between boxes or their binding/unbinding. Interaction is sensitivity-based: values associated to binders determine the possibility of interaction between two entities. We release the key-lock interaction mechanism based on the notion, implemented in all process calculi, of exact complementarity of channel names. This allows us to avoid any global policy on the usage of names for interaction between components, i.e. we do not need centralized authorities that decide how to name entities or interfaces. No other process calculi-based tool supports compatibility of channel names in the manner described above.

BlenX can also specify the dynamics of systems through events that are used to remove or inject entities from/into the system, to join two entities into a single one, or to split an entity in two entities.

Although BlenX has been designed to specifically address biological systems, its main features are useful also to model computer science systems. This is a consequence of being developed as an extension of process calculi that have already proved themselves of being a core technology to design, implement and verify concurrent systems.

Biological interactions are inherently stochastic, and the full-fledged version of BlenX is indeed stochastic. Here we focus on the deterministic (vs stochastic) basis of core BlenX, which is an enhancement of the language described in [1]. Below we present the syntax of BlenX (Sec. 2), its operational semantics (Sec. 3), and its static semantics (Sec. 4), which is related to the dynamic one by a soundness result.

2 Syntax and Notation

The syntax of BlenX is parametrized by three countable and pairwise disjoint sets: the set $\mathcal{N}$ of *names* (ranged over by $\mathtt{n}, \mathtt{m}, \mathtt{n}', \mathtt{m}', \ldots$), the set $\mathcal{X}$ of *variables* (ranged over by $\mathtt{x}, \mathtt{x}', \ldots$), and the set $\mathcal{BV}$ of *basic values* (ranged over by $\mathtt{v}, \mathtt{v}', \ldots$). We assume $\mathcal{BV} \supset \mathbb{B} \cup \{unit\}$, where $\mathbb{B} = \{true, false\}$ is the set of the basic boolean values. The metavariables $\mathtt{w}, \mathtt{w}', \ldots$ range over $\mathcal{N} \cup \mathcal{X}$.

Table 1. Bio-processes syntax

```
B    ::=  Nil │ I[P]n │ B||B
I    ::=  K(n,V) │ K(n,V)I
K    ::=  # │ #c
V    ::=  n │ v │ (V,V)
P    ::=  P|P │ rep <EXP>A.P │ G
G    ::=  nil │ <EXP>A.P │ G+G
EXP  ::=  n │ x │ v │ val(w) │ check(w,K) │ (EXP,EXP) │
          EXP.1 │ EXP.2
A    ::=  delay │ w?PAT │ w!EXP │ ch(w,EXP) │ new(n,EXP)
PAT  ::=  x │ (PAT,PAT)
```

A BlenX system, written SYS=$\langle\langle$B; E; ENV$\rangle\rangle$, is a triple made up of a bio-process B, a collection of events E, and an environment ENV.

Bio-processes are generated by the non-terminal symbol B of the grammar reported in Tab. 1. A bio-process can be either the deadlocked process (Nil), or a *box* (I[P]n), or the parallel composition of bio-processes (B||B). In I[P]n, I represents the *interface* of the box, i.e. its interaction capabilities, while P is its internal engine, and n is used as an identifier to address the box at hand.

An *interface* I is a non-empty string of *binders* of the form K(n,V), where K denotes the state of the binder, which can be either *free* (#) or *bound* (#c), the name n is the *subject* of the binder, and V is a *value* representing the structure of the interface. We sometimes use *active* as synomimous of free, and *complexed* as synonimous of bound. The value V of a binder can be either a name, or a basic value, or a pair of values. The subject n of the binder K(n,V) of the box K(n,V)I[P]n$'$ is a binding for the free occurrences of n in P. We write I $=$ I$_1$I$_2$ to mean that I is the interface given by the juxtaposition of I$_1$ and I$_2$. The metavariables I*, I$_1^*$, $\ldots$ stay for either an interface or the empty string, and the above notation for juxtaposition is extended to these metavariables in the natural way. Moreover, we use the functions subj(I) and val(I) to extract from I the set of its subjects and the set of its values, respectively.

Values are generated by V. They can be names, or basic values, or pairs of values. We call $\mathcal{V}$ the set of all these values, and let $\mathcal{V}$ be ranged over by V,V$'$,$\ldots$.

The non-terminal symbol P generates *processes*. A process can be either the parallel composition of two processes (P|P), or the replication of an action-guarded process (rep <EXP>A.P), or the deadlocked process nil, or an action-guarded process (<EXP>A.P), or the non-deterministic choice of guarded processes (G+G). Guards of the shape <EXP>A extend the usual notion of action prefix in process calculi. The intuition behind process <EXP>A.P is that the expression EXP is a conditional control over the execution of the action A. If

Table 2. Function $\mathcal{M}$ for pattern-matching

$$\mathcal{M}(a,b) = \begin{cases} \{\mathtt{V}/\mathtt{x}\} & \text{if } a = \mathtt{x} \text{ and } b = \mathtt{V} \\ \mathcal{M}(\mathtt{PAT}_1, \mathtt{V}_1) \uplus \mathcal{M}(\mathtt{PAT}_2, \mathtt{V}_2) & \text{if } a = (\mathtt{PAT}_1, \mathtt{PAT}_2) \text{ and } b = (\mathtt{V}_1, \mathtt{V}_2) \\ \perp & \text{otherwise} \end{cases}$$

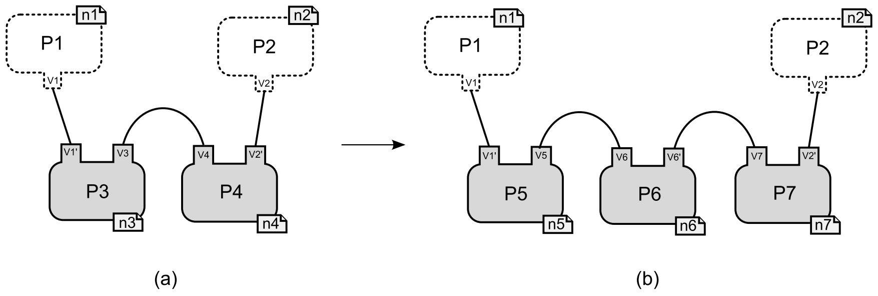


Fig. 1. Graphical representation of BlenX systems

EXP is a boolean expression and evaluates to *true*, then A can fire and P gets unblocked. Processes are ranged over by P, P′, ...

val(w) and check(w,K) are *expressions*, meaning respectively the value and the status of the binder with subject w. Pairs of expressions are expressions, as well as first and second projections of an expression (denoted EXP.1 and EXP.2, respectively). Static checks ensure that projection operators are only applied to pairs of expressions (see Tab. 9).

The actions that a process can perform are described by the syntactic category A. The action delay, which is best understood in the stochastic version of BlenX, here just represents a discrete passage of time; w?PAT and w!EXP denote, respectively, an input over w, and the output over w of the expression EXP; ch(w,EXP) is a directive to change into EXP the value associated with the binder named w; new(n,EXP) is a directive to expose a new binder with value EXP. In either <EXP>w?PAT.P and <EXP>new(n,EXP).P, w and n act as binders for the free occurrences of w and n in P. The free names of process P are denoted $\mathsf{fn}(P)$.

Patterns are either variables or pairs of patterns. As detailed in Sec. 3, the expression EXP is pattern-matched against PAT in the communications triggered by the complementary actions w?PAT and w!EXP. Pattern-matching is based on the definition of the function $\mathcal{M}$ reported in Tab. 2, where $\perp$ is used to represent failure of pattern-matching, and the symbol $\uplus$ denotes a special union operation over substitutions that is defined only for operands which agree on the variables in the intersection of their domains (e.g., $\{true/\mathtt{x}, true/\mathtt{x}'\}$ agrees with $\{true/\mathtt{x}\}$ but not with $\{false/\mathtt{x}\}$). It is intended that when $\mathcal{M}(\mathtt{PAT}_1, \mathtt{V}_1) \uplus \mathcal{M}(\mathtt{PAT}_2, \mathtt{V}_2)$ is not defined the pattern-matching of $(\mathtt{PAT}_1, \mathtt{PAT}_2)$ with $(\mathtt{V}_1, \mathtt{V}_2)$ fails. For example, $\mathcal{M}((\mathtt{x}, \mathtt{x}), (true, false)) = \perp$ just like $\mathcal{M}((\mathtt{x}, \mathtt{x}'), true)$. Below we use the metavariables $\sigma, \sigma', \ldots$ to denote successful applications of pattern-matching.

Table 3. Function $\mathcal{CB}$ for the description of the borders of complexes

$$\mathcal{CB}(\mathtt{Nil}, \mathtt{ENV}) = \emptyset$$

$$\mathcal{CB}(\mathtt{B||B'}, \mathtt{ENV}) = \mathcal{CB}(\mathtt{B}, \mathtt{ENV}) \cup \mathcal{CB}(\mathtt{B'}, \mathtt{ENV})$$

$$\mathcal{CB}(\mathtt{I[P]m}, \mathtt{ENV}) = \mathcal{CI}(\mathtt{I}, \mathtt{ENV}, \mathtt{m})$$

$$\mathcal{CI}(\mathtt{K(n,V)I}, \mathtt{ENV}, \mathtt{m}) = \mathcal{CI}(\mathtt{K(n,V)}, \mathtt{ENV}, \mathtt{m}) \cup \mathcal{CI}(\mathtt{I}, \mathtt{ENV}, \mathtt{m})$$

$$\mathcal{CI}(\mathtt{\#(n,V)}, \mathtt{ENV}, \mathtt{m}) = \emptyset$$

$$\mathcal{CI}(\mathtt{\#c(n,V)}, \mathtt{ENV}, \mathtt{m}) = \begin{cases} \mathtt{V@m} & \text{if } \nexists (a,b) \in \mathtt{ENV} \text{ such that } \mathtt{V@m} = a \text{ or } \mathtt{V@m} = b \\ \emptyset & \text{otherwise} \end{cases}$$

Boxes can be bound the one with the other through their interfaces to form complexes which are best thought of as graphs with boxes as nodes. Intuitively, the *environment* component `ENV` of the BlenX system $\langle\langle \mathtt{B;\ E;\ ENV} \rangle\rangle$ is used to record these bindings. In detail, `ENV` is a set of pairs of the shape $(\mathtt{V@n}, \mathtt{V'@n'})$ meaning that the two boxes addressed by `n` and $\mathtt{n'}$ are bound together through the interfaces with values `V` and $\mathtt{V'}$, respectively. As an example, Fig. 1(a) shows a complex formed by four boxes running in parallel which contain processes `P1`, `P2`, `P3`, `P4`, and are addressed, respectively, by `n1`, `n2`, `n3`, and `n4`. In the picture, each box has one or more complexed interfaces with associated values `V1`, `V2`, ... An environment representing the complex in Fig. 1(a) is given by $\mathtt{ENV}_1 = \{(\mathtt{V1@n1}, \mathtt{V1'@n3}), (\mathtt{V3@n3}, \mathtt{V4@n4}), (\mathtt{V2'@n4}, \mathtt{V2@n2})\}$. In what follows, we denote by *names*(`ENV`) the set of the names which are used as addresses in the environment `ENV`. For instance, $names(\mathtt{ENV}_1) = \{\mathtt{n1}, \mathtt{n2}, \mathtt{n3}, \mathtt{n4}\}$.

The *events* component `E` of the system $\langle\langle \mathtt{B;\ E;\ ENV} \rangle\rangle$ is a set of directives used to substitute bio-processes with other bio-processes. Since bio-processes may be parts of complexes, the implementation of these substitutions may force the modification of the structure of the complexes recorded in the environment `ENV`. To this end we use an abstraction that, with respect to the interpretation of environments in terms of graphs, amounts to state that a certain subgraph g_1 of a graph G_1 has to be substituted by the subgraph g_2 of the graph G_2. The abstraction is powerful enough to allow the encoding of a set of useful operations on the global systems (e.g., deleting a box, joining two boxes in one, injecting new boxes). Its implementation, however, requires some care in ensuring that the edges at the border of g_1 (those which could remain hangling, with no node at one of their tips) are properly connected to nodes of g_2. Given a bio-process `B` and an environment `ENV`, the function $\mathcal{CB}$ defined in Tab. 3 is used to collect the identity of interfaces of boxes in `B` which are involved in bindings outside `ENV`, i.e. outside the borders of `B`. For instance, referring to Fig. 1(a), if $\mathtt{B}$ is the bio-process given by the parallel composition of the boxes containing $\mathtt{P3}$ and $\mathtt{P4}$, then $\mathcal{CB}(\mathtt{B}, \{\mathtt{(V3@n3,V4@n4)}\}) = \{\mathtt{V1'@n3}, \mathtt{V2'@n4}\}$. In detail, the events component $\mathtt{E}$ of the system $\langle\langle \mathtt{B;\ E;\ ENV} \rangle\rangle$ is a set of phrases of the shape $\mathtt{sub(B_1,ENV_1,B_2,ENV_2)}$ where $\mathtt{sub(B_1,ENV_1,B_2,ENV_2)}$ intuitively drives the substitution of the bio-process $\mathtt{B_1}$, defined over the environment $\mathtt{ENV_1}$, with the bio-process $\mathtt{B_2}$, defined over the environment $\mathtt{ENV_2}$.

Below, the typical post-fixed notation $\{a/b\}$ is used to mean the substitution of the entity b with the entity a, with the usual conventions about renaming and α-conversion when the substitution is applied to processes. This notation is naturally extended to other domains, e.g. to sets. For example, $\mathtt{ENV}\{\mathtt{V@n/V@m}\}$ denotes the substitution of the occurrences of $\mathtt{V@m}$ with $\mathtt{V@n}$ in $\mathtt{ENV}$.

3 Dynamic Semantics

The dynamics of a system is formally specified by the reduction semantics reported in Tab. 6. It makes use of the structural congruence $\equiv$ over BlenX systems, whose definition is based on both a structural congruence over processes ($\equiv_p$) and a structural congruence over bio-processes ($\equiv_b$). The needed congruences are the smallest relations satisfying the laws in Tab. 4.

The laws defining $\equiv_p$ are the typical axioms used in process calculi, with $=_\alpha$ used to denote α-equivalence.

The first axiom for $\equiv_b$ serves two purposes. It declares that the actual ordering of binders within an interface is irrelevant, and states that the structural congruence of processes is reflected at the level of boxes. The second law for $\equiv_b$ is a sort of α-conversion axiom for boxes. It states that the subject of binders can be refreshed under the proviso that name clashes are avoided. The latest laws are the monoidal axioms for the parallel composition of boxes.

The first law for $\equiv$ states the possibility of refreshing names used as addresses. The second axiom lifts the congruence of bio-processes at the level of systems by also checking the equality of the sets used as events and as environment components in the two systems.

A few auxiliary functions are used in stating the operational semantics of BlenX. Function $[\![\_]\!]_{\_}$, which is adopted for the evaluation of expressions, is defined in Tab. 5 where, as above, we use $\perp$ to denote undefinedness. The definition of $[\![\_]\!]_{\_}$ is fairly usual, just notice that, due to expressions involving the definition of interfaces ($\mathtt{val(w)}$ and $\mathtt{check(w,K)}$), the evaluation is related to a specific interface.

The reduction relation describing the operational semantics of BlenX is defined by the axioms and rules collected in Tab. 6.

The very first rule (rd) states that a $\mathtt{delay}$ action can fire under the proviso that its guarding expression $\mathtt{EXP}$ evaluates to *true* in the interface of the containing box. The execution of the $\mathtt{delay}$ action leaves the environment $\mathtt{ENV}$ unaffected.

Rule (rc) says that three conditions have to be met for the firing of a $\mathtt{<EXP_1>ch(m,EXP_2)}$ prefix within a box with interface $\mathtt{I}$: the guarding expression $\mathtt{EXP_1}$ has to evaluate to *true* in $\mathtt{I}$, the argument expression $\mathtt{EXP_2}$ has to

Table 4. Structural congruence laws

$\mathtt{P}_1 \equiv_p \mathtt{P}_2$ provided $\mathtt{P}_1 =_\alpha \mathtt{P}_2$
$\mathtt{P|nil} \equiv_p \mathtt{P}$, $\quad \mathtt{P}_1\mathtt{|P}_2 \equiv_p \mathtt{P}_2\mathtt{|P}_1$, $\quad \mathtt{P}_1\mathtt{|(P}_2\mathtt{|P}_3\mathtt{)} \equiv_p \mathtt{(P}_1\mathtt{|P}_2\mathtt{)|P}_3$
$\mathtt{rep\ <EXP>A.P} \equiv_p \mathtt{<EXP>A.(P|rep\ <EXP>A.P)}$

$\mathtt{G+nil} \equiv_p \mathtt{G}$, $\quad \mathtt{G}_1\mathtt{+G}_2 \equiv_p \mathtt{G}_2\mathtt{+G}_1$, $\quad \mathtt{G}_1\mathtt{+(G}_2\mathtt{+G}_3\mathtt{)} \equiv_p \mathtt{(G}_1\mathtt{+G}_2\mathtt{)+G}_3$

$\mathtt{I}_1\mathtt{I}_2^*\mathtt{[P}_1\mathtt{]n} \equiv_b \mathtt{I}_2^*\mathtt{I}_1\mathtt{[P}_2\mathtt{]n}$ provided $\mathtt{P}_1 \equiv_p \mathtt{P}_2$
$\mathtt{B} \equiv_b \mathtt{B}'$ if
$(\mathtt{B} = \mathtt{K(n,V)I}^*\mathtt{[P]m}$ and $\mathtt{B}' = \mathtt{K(n',V)I}^*\mathtt{[P\{n'/n\}]m})$ or
$(\mathtt{B}' = \mathtt{K(n,V)I}^*\mathtt{[P]m}$ and $\mathtt{B} = \mathtt{K(n',V)I}^*\mathtt{[P\{n'/n\}]m})$
with $\mathtt{n}'$ fresh in $\mathtt{P}$ and in $subj(\mathtt{I})$.

$\mathtt{B||Nil} \equiv_b \mathtt{B}$, $\quad \mathtt{B}_1\mathtt{||B}_2 \equiv_b \mathtt{B}_2\mathtt{||B}_1$, $\quad \mathtt{B}_1\mathtt{||(B}_2\mathtt{||B}_3\mathtt{)} \equiv_b \mathtt{(B}_1\mathtt{||B}_2\mathtt{)||B}_3$

$\mathtt{SYS} \equiv \mathtt{SYS}'$ if
$(\mathtt{SYS} = \langle\langle \mathtt{I[P]n||B;\ E;\ ENV}\rangle\rangle$ and $\mathtt{SYS}' = \langle\langle \mathtt{I[P]m||B;\ E;\ ENV\{m/n\}}\rangle\rangle)$ or
$(\mathtt{SYS}' = \langle\langle \mathtt{I[P]n||B;\ E;\ ENV}\rangle\rangle$ and $\mathtt{SYS} = \langle\langle \mathtt{I[P]m||B;\ E;\ ENV\{m/n\}}\rangle\rangle)$
with $\mathtt{m}$ fresh in $\mathtt{B}$ and in $\mathtt{ENV}$
$\langle\langle \mathtt{B;\ E;\ ENV}\rangle\rangle \equiv \langle\langle \mathtt{B';\ E';\ ENV'}\rangle\rangle$
provided $\mathtt{B} \equiv_b \mathtt{B}'$, $\mathtt{E} = \mathtt{E}'$, $\mathtt{ENV} = \mathtt{ENV}'$

Table 5. Function $[\![\_]\!]_\_$ for the evaluation of expressions

$[\![\mathtt{n}]\!]_\mathtt{I} = \mathtt{n} \qquad [\![\mathtt{x}]\!]_\mathtt{I} = \mathtt{x} \qquad [\![\mathtt{v}]\!]_\mathtt{I} = \mathtt{v}$

$$[\![\mathtt{val(n)}]\!]_\mathtt{I} = \begin{cases} \mathtt{V} & \text{if } \mathtt{I} = \mathtt{I}_1^*\mathtt{K(n,V)I}_2^* \text{ for some } \mathtt{I}_1^*, \mathtt{K}, \mathtt{V}, \mathtt{I}_2^* \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\mathtt{check(n,K)}]\!]_\mathtt{I} = \begin{cases} \textit{true} & \text{if } \mathtt{I} = \mathtt{I}_1^*\mathtt{K(n,V)I}_2^* \text{ for some } \mathtt{I}_1^*, \mathtt{V}, \mathtt{I}_2^* \\ \textit{false} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{(EXP_1,EXP_2)}]\!]_\mathtt{I} = ([\![\mathtt{EXP_1}]\!]_\mathtt{I}, [\![\mathtt{EXP_2}]\!]_\mathtt{I})$$

$$[\![\mathtt{(EXP_1,EXP_2).1}]\!]_\mathtt{I} = [\![\mathtt{EXP_1}]\!]_\mathtt{I} \qquad [\![\mathtt{(EXP_1,EXP_2).2}]\!]_\mathtt{I} = [\![\mathtt{EXP_2}]\!]_\mathtt{I}$$

evaluate to a value $\mathtt{V}_1$, and $\mathtt{I}$ must have an interface named $\mathtt{m}$. Under the above hypotheses, and if $\mathtt{V}_1$ does not clash with the values of the other interfaces in $\mathtt{I}$, then the value of $\mathtt{m}$ is turned to $\mathtt{V}_1$. Since the interface $\mathtt{m}$ could be involved in a binding, the environment is consistently updated by possibly refreshing the previous value of $\mathtt{m}$ with $\mathtt{V}_1$. Here notice that the requirement about the freshness of $\mathtt{V}_1$ guarantess the freshness of $\mathtt{V}_1\mathtt{@n}$ in the updated environment.

Rule (rn) is used to add a new binder to a box. The prefix $\mathtt{<EXP_1>new(m,EXP_2)}$ can fire only if the value resulting from the evaluation of $\mathtt{EXP}_2$ is fresh in the

Table 6. BlenX operational semantics

(rd) $$\frac{[\![\mathtt{EXP}]\!]_{\mathtt{I}} = \textit{true}}{\langle\langle \mathtt{I[<EXP>delay.P+G|P_1]n;\ E;\ ENV}\rangle\rangle \rightarrow \langle\langle \mathtt{I[P|P_1]n;\ E;\ ENV}\rangle\rangle}$$

(rc) $$\frac{[\![\mathtt{EXP_1}]\!]_{\mathtt{I}} = \textit{true},\quad [\![\mathtt{EXP_2}]\!]_{\mathtt{I}} = \mathtt{V_1}}{\langle\langle \mathtt{I[<EXP_1>ch(m,EXP_2).P+G|P_1]n;\ E;\ ENV}\rangle\rangle \rightarrow \langle\langle \mathtt{I_1[P|P_1]n;\ E;\ ENV\{V_1@n/V@n\}}\rangle\rangle}$$

where $\mathtt{I} = \mathtt{K(m,V)I^*}$ and $\mathtt{I_1} = \mathtt{K(m,V_1)I^*}$ and provided $\mathtt{V_1} \notin \mathsf{val}(\mathtt{I^*})$

(rn) $$\frac{[\![\mathtt{EXP_1}]\!]_{\mathtt{I}} = \textit{true},\quad [\![\mathtt{EXP_2}]\!]_{\mathtt{I}} = \mathtt{V}}{\langle\langle \mathtt{I[<EXP_1>new(m,EXP_2).P+G|P_1]n;\ E;\ ENV}\rangle\rangle \rightarrow \langle\langle \mathtt{K(m',V)I[P\{m'/m\}|P_1]n;\ E;\ ENV}\rangle\rangle}$$

provided $\mathtt{m'}$ fresh in $(\mathtt{P|P_1})$ and $\mathtt{V} \notin \mathsf{val}(\mathtt{I})$

(ra) $$\frac{[\![\mathtt{EXP_1}]\!]_{\mathtt{I}} = \textit{true},\quad [\![\mathtt{EXP_2}]\!]_{\mathtt{I}} = \textit{true},\quad [\![\mathtt{EXP}]\!]_{\mathtt{I}} = \mathtt{V},\quad \mathcal{M}(\mathtt{PAT},\mathtt{V}) = \sigma}{\langle\langle \mathtt{I[P_1+G_1|P_2+G_2|P']n;\ E;\ ENV}\rangle\rangle \rightarrow \langle\langle \mathtt{I[P_1'|P_2'}\sigma\mathtt{|P']n;\ E;\ ENV}\rangle\rangle}$$

where $\mathtt{P_1} = \mathtt{<EXP_1>m!EXP.P_1'}$ and $\mathtt{P_2} = \mathtt{<EXP_2>m?PAT.P_2'}$

(rb) $$\frac{\alpha_b(\mathtt{V_1},\mathtt{V_2}) = \textit{true}}{\langle\langle \mathtt{B_1||B_2;\ E;\ ENV}\rangle\rangle \rightarrow \langle\langle \mathtt{B_1'||B_2';\ E;\ ENV} \cup \{\mathtt{(V_1@n_1,V_2@n_2)}\}\rangle\rangle}$$

where $\mathtt{B}_j\mathtt{=\#(m}_j\mathtt{,V}_j\mathtt{)I}_j^*\mathtt{[P}_j\mathtt{]n}_j$ and $\mathtt{B}_j'\mathtt{=\#c(m}_j\mathtt{,V}_j\mathtt{)I}_j^*\mathtt{[P}_j\mathtt{]n}_j$ for $j = 1, 2$

(ru) $$\frac{\alpha_u(\mathtt{V_1},\mathtt{V_2}) = \textit{true}}{\langle\langle \mathtt{B_1||B_2;\ E;\ ENV} \cup \{\mathtt{(V_1@n_1,V_2@n_2)}\}\rangle\rangle \rightarrow \langle\langle \mathtt{B_1'||B_2';\ E;\ ENV}\rangle\rangle}$$

where $\mathtt{B}_j\mathtt{=\#c(m}_j\mathtt{,V}_j\mathtt{)I}_j^*\mathtt{[P}_j\mathtt{]n}_j$ and $\mathtt{B}_j'\mathtt{=\#(m}_j\mathtt{,V}_j\mathtt{)I}_j^*\mathtt{[P}_j\mathtt{]n}_j$ for $j = 1, 2$

(ri) $$\frac{\alpha_i(\mathtt{V_1},\mathtt{V_2}) = \textit{true},\ [\![\mathtt{EXP_1}]\!]_{\mathtt{I_1}} = \textit{true},\ [\![\mathtt{EXP_2}]\!]_{\mathtt{I_2}} = \textit{true},\ [\![\mathtt{EXP}]\!]_{\mathtt{I_1}} = \mathtt{V},\ \mathcal{M}(\mathtt{PAT},\mathtt{V}) = \sigma}{\begin{array}{l}\langle\langle \mathtt{I_1[<EXP_1>n'!EXP.P_1+G_1|P_1']n||I_2[<EXP_2>m'?PAT.P_2+G_2|P_2']m;\ E;\ ENV}\rangle\rangle \rightarrow \\ \langle\langle \mathtt{I_1[P_1|P_1']n||I_2[P_2}\sigma\mathtt{|P_2']m;\ E;\ ENV}\rangle\rangle\end{array}}$$

provided $\mathtt{I_1} = \mathtt{K(n',V_1)I_1^*}$ and $\mathtt{I_2} = \mathtt{K(m',V_2)I_2^*}$ with

($\mathtt{K=\#c}$ and $(\mathtt{(V_1@n,V_2@m)}\in\mathtt{ENV}$ or $\mathtt{(V_2@m,V_1@n)}\in\mathtt{ENV})$) or

($\mathtt{K=\#}$ and $\alpha(\mathtt{V_1},\mathtt{V_2}) = (\textit{false}, \textit{false}, \textit{true})$)

(re) $$\frac{\textit{map}(\mathcal{CB}(\mathtt{B_1},\mathtt{ENV_1}),\mathcal{CB}(\mathtt{B_2},\mathtt{ENV_2})) = \sigma,\quad \textit{names}(\mathtt{ENV}) \cap \textit{names}(\mathtt{ENV_2}) = \emptyset}{\langle\langle \mathtt{B_1;\ E;\ ENV_1} \cup \mathtt{ENV}\rangle\rangle \rightarrow \langle\langle \mathtt{B_2;\ E;\ ENV_2} \cup \mathtt{ENV}\sigma\rangle\rangle}$$

where $\mathtt{sub(B_1,ENV_1,B_2,ENV_2)} \in \mathtt{E}$

(rr) $$\frac{\langle\langle \mathtt{B_1;\ E;\ ENV}\rangle\rangle \rightarrow \langle\langle \mathtt{B_2;E;ENV'}\rangle\rangle}{\langle\langle \mathtt{B_1||B;\ E;\ ENV}\rangle\rangle \rightarrow \langle\langle \mathtt{B_2||B;E;ENV'}\rangle\rangle}$$

(rs) $$\frac{\mathtt{SYS_1} \equiv \mathtt{SYS_1'},\quad \mathtt{SYS_1'} \rightarrow \mathtt{SYS_2'},\quad \mathtt{SYS_2'} \equiv \mathtt{SYS_2}}{\mathtt{SYS_1} \rightarrow \mathtt{SYS_2}}$$

interface of the box. If so, the name `m` declared in the `new` prefix is renamed to avoid clashes with the names of the residual process.

Intra-communication, i.e. communication between processes within the same box, is ruled by (ra). The pattern `PAT` carried by the output action is matched against the expression `EXP` argument of the input action, and the resulting substitution is applied to the residual of the receiving process.

Boxes can interact the one with the other in various ways: they can bind together, unbind, or just communicate. These interactions are based on the existence of an affinity function $\alpha : \mathtt{V} \times \mathtt{V} \rightarrow \mathbb{B}^3$ which returns a triple of boolean values representing the *binding*, *unbinding* and *inter-communication affinities* of the two argument values. We use $\alpha_b(\mathtt{V}, \mathtt{V}')$, $\alpha_u(\mathtt{V}, \mathtt{V}')$, and $\alpha_i(\mathtt{V}, \mathtt{V}')$ to mean, respectively, the first, the second, and the third projection of $\alpha(\mathtt{V}, \mathtt{V}')$.

Rules (rb) and (ru) describe the dynamics of binding and unbinding, respectively, which can only take place if the binding/unbinding affinity of the values of the involved interfaces is set to *true*. In both cases the modification of the binding state of the relevant interfaces is reflected in the interface markers, which are changed either from `#` to `#c` or the other way round. Also, the association $(\mathtt{V}_1@\mathtt{n}_1, \mathtt{V}_2@\mathtt{n}_2)$ recording the actual binding is either added to the environment or removed from it.

The third kind of interaction between boxes, called *inter-communication*, is ruled by (ri). This involves an input and an output action firable in two distinct boxes over binders with associated values $\mathtt{V}_1$ and $\mathtt{V}_2$. Information passes from the box containing the sending action to the box enclosing the receiving process. Here notice that inter-communication depends on the affinity of $\mathtt{V}_1$ and $\mathtt{V}_2$ rather than on the fact that input and output actions occur over exactly the same name. Indeed inter-communication is enabled only if $\alpha_i(\mathtt{V}_1, \mathtt{V}_2) = \mathit{true}$, and only under the proviso that either the two binders are already bound together or they are free and both $\alpha_b(\mathtt{V}_1, \mathtt{V}_2)$ and $\alpha_u(\mathtt{V}_1, \mathtt{V}_2)$ are set to *false*.

Given the system $\langle\langle \mathtt{B}_1;\ \mathtt{E};\ \mathtt{ENV}_1 \cup \mathtt{ENV} \rangle\rangle$, rule (re) defines reductions corresponding to the occurrence of *events* of the shape $\mathtt{sub(B_1,ENV_1,B_2,ENV_2)}$ when there is no clash between the names of `ENV` and those of $\mathtt{ENV}_2$. The application of the rule involves checking the possibility of substituting $\mathtt{B}_1$ and the portion of environment $\mathtt{ENV}_1$ with the bio-process $\mathtt{B}_2$ and the sub-environment $\mathtt{ENV}_2$. The function *map* defined in Tab. 7 serves this goal. Recall from Sec. 2 that the function $\mathcal{CB}(\mathtt{B}_j, \mathtt{ENV}_j)$ allows the collection of data about interfaces of boxes in $\mathtt{B}_j$ which are possibly involved in bindings outside $\mathtt{ENV}_j$. Function *map* is applied to $\mathcal{CB}(\mathtt{B}_1, \mathtt{ENV}_1)$ and $\mathcal{CB}(\mathtt{B}_2, \mathtt{ENV}_2)$ to return a consistent mapping from the hangling bindings of $\mathtt{B}_1$ over $\mathtt{ENV}_1$ to the hangling bindings of $\mathtt{B}_2$ over $\mathtt{ENV}_2$. When such a

Table 7. Function *map*

$$\mathit{map}(a,b) = \begin{cases} \emptyset & \text{if } a = \emptyset \text{ and } b = \emptyset \\ \{\mathtt{V@m}/\mathtt{V@n}\} \uplus \mathit{map}(S, S') & \text{if } \exists \mathtt{V} \text{ s.t. } a = \{\mathtt{V@n}\} \cup S \text{ and } b = \{\mathtt{V@m}\} \cup S' \\ \bot & \text{otherwise} \end{cases}$$

mapping exists, *map* returns a substitution that is applied to ENV to get a fully updated environment after the substitution of $\mathtt{B}_1$ with $\mathtt{B}_2$.

Fig. 1 actually describes a reduction step of a BlenX system which can be inferred by using (re). In fact, naming Bj the box containing Pj (j=1,. . .,7) and setting:

$$\begin{aligned}
\mathtt{ENV}_1 &= \{\mathtt{(V3@n3,V4@n4)}\}\\
\mathtt{ENV}_2 &= \{\mathtt{(V5@n5,V6@n6)}, \mathtt{(V6'@n6,V7@n7)}\}\\
\mathtt{E} &= \{\mathtt{sub(B3||B4,ENV_1,B5||B6||B7,ENV_2)}\}\\
\mathtt{ENV} &= \{\mathtt{(V1@n1,V1'@n3)}, \mathtt{(V2'@n4,V2@n2)}\}
\end{aligned}$$

by (re) we get:

$$\langle\langle \mathtt{B3||B4;\ E;\ ENV_1} \cup \mathtt{ENV}\rangle\rangle \rightarrow \langle\langle \mathtt{B5||B6||B7;\ E;\ ENV_2} \cup \mathtt{ENV}\sigma\rangle\rangle$$

where $\sigma = \{\mathtt{V1'@n5/V1'@n3}, \mathtt{V2'@n7/V2'@n4}\}$.

We conclude the description of the rules in Tab. 6 by just observing that, as usual in reduction semantics, the rules (rr) and (rs) are meant to lift reductions over parallel compositions and over structural re-shufflings.

4 Static Semantics

In this section we define a type system for BlenX. It guarantees that well-typed BlenX systems respect a number of constraints (e.g. complexes are described consistently, binders are used in the right way in inter-communications) that we will explain throughout this section.

We assume a set of *basic types* $\mathcal{BT}$ (with metavariables BT, BT′, ...) such that Unit and Bool belongs to $\mathcal{BT}$. Types are generated by the syntax reported in Tab. 8. Basic types are types, chan(T) is the type associated to a channel allowing communication of only type T values and binder(T,T′) is the type associated to a binder with form K(x:T,V:T′). Finally, T×T′ is the product. We call $\mathcal{T}$ the set of all these types and with $\overline{\mathcal{T}} \subset \mathcal{T}$ the set of all the types not containing in their structure binder types.

We assume $\mathcal{BV} = \bigcup_{\mathtt{BT}\in\mathcal{BT}} \mathcal{BV}_{\mathtt{BT}}$ and $\bigcap_{\mathtt{BT}\in\mathcal{BT}} \mathcal{BV}_{\mathtt{BT}} = \emptyset$ where with $\mathcal{BV}_{\mathtt{BT}}$ we indicate all the basic values of type BT.

Tab. 8 defines also three different signatures associated with bio-processes, interfaces and environments, respectively. We assume $N \subset \mathcal{N}$ and $C \subset \mathcal{V} \times \mathcal{N}$

Table 8. Definition of types and signatures

$$\mathtt{T} ::= \mathtt{BT} \;\big|\; \mathtt{chan(T)} \;\big|\; \mathtt{binder(T,T)} \;\big|\; \mathtt{T}\times\mathtt{T}$$

$$\mathtt{S} ::= (N, C) \;\big|\; (\Delta, C) \;\big|\; C$$

Table 9. Type-checking expressions

$$\text{(t1)}\ \frac{}{\emptyset \vdash_e \mathit{unit} : \mathtt{Unit}} \qquad \text{(t2)}\ \frac{\mathtt{v} \in \mathcal{BV}_{\mathtt{BT}}}{\emptyset \vdash_e \mathtt{v} : \mathtt{BT}} \qquad \text{(t3)}\ \frac{\Psi(\mathtt{w}) = \mathtt{T}}{\Psi \vdash_e \mathtt{w} : \mathtt{T}}$$

$$\text{(t4)}\ \frac{\Psi \vdash_e \mathtt{EXP} : \mathtt{T}_1 \times \mathtt{T}_2}{\Psi \vdash_e \mathtt{EXP.1} : \mathtt{T}_1} \qquad \text{(t5)}\ \frac{\Psi \vdash_e \mathtt{EXP} : \mathtt{T}_1 \times \mathtt{T}_2}{\Psi \vdash_e \mathtt{EXP.2} : \mathtt{T}_2} \qquad \text{(t6)}\ \frac{\Psi \vdash_e \mathtt{n} : \mathtt{binder(T_1,T_2)}}{\Psi \vdash_e \mathtt{val(n)} : \mathtt{T}_2}$$

$$\text{(t7)}\ \frac{\Psi \vdash_e \mathtt{n} : \mathtt{binder(T_1,T_2)}}{\Psi \vdash_e \mathtt{check(n,K)} : \mathtt{Bool}} \qquad \text{(t8)}\ \frac{\Psi \vdash_e \mathtt{EXP}_1 : \mathtt{T}_1, \quad \Psi \vdash_e \mathtt{EXP}_2 : \mathtt{T}_2}{\Psi \vdash_e (\mathtt{EXP}_1, \mathtt{EXP}_2) : \mathtt{T}_1 \times \mathtt{T}_2}$$

to be finite sets and Ψ to be a typing context. Typing contexts are finite maps from variables and names to types, denoted also as sets of variables and names bindings $\{\mathtt{w}_i : \mathtt{T}_i\}_{i \in I}$ where all the $\mathtt{w}_i$'s are distinct. We denote with $\emptyset$ the empty typing context and with Ψ/Ψ_1 a typing context result of the combination of Ψ and Ψ_1 such that $dom(\Psi/\Psi_1) = dom(\Psi_1) \cup (dom(\Psi) \setminus dom(\Psi_1))$; hence, $\Psi/\Psi_1(\mathtt{w}) = \Psi_1(\mathtt{w})$ if $\mathtt{w} \in dom(\Psi_1)$ and $\Psi(\mathtt{w})$ otherwise. We call $\mathcal{TC}$ the set of all these typing contexts and with $\overline{\mathcal{TC}} \subset \mathcal{TC}$ the set of typing contexts containing only bindings between names and channel types.

We explicitly type BlenX by associating types to binders definitions, hence modifying the interfaces syntax and the new action syntax:

$$\mathtt{I} ::= \mathtt{K(n{:}T_1,V{:}T_2)} \;\big|\; \mathtt{K(n{:}T_1,V{:}T_2)I} \qquad \mathtt{A} ::= \cdots \;\big|\; \mathtt{new(n{:}T_1,V{:}T_2)}$$

Moreover, since we want to guarantee that inter-communications happen only through binders having subjects of the same type, we modify the definition of the affinity function in the following way:

$$\overline{\alpha}(\mathtt{K(x{:}T_1,V{:}T_2)}, \mathtt{K'(x'{:}T'_1,V'{:}T'_2)}) = \begin{cases} \alpha(\mathtt{V},\mathtt{V'}) & \text{if } \mathtt{T}_1 = \mathtt{T}'_1 \text{ and} \\ & \mathtt{T}_1 \in \overline{\mathcal{T}} \\ (\mathit{false}, \mathit{false}, \mathit{false}) & \text{otherwise} \end{cases}$$

By using the affinity function $\overline{\alpha}$ instead of α we guarantee that the type system we define is sound w.r.t. the BlenX operational semantics.

Type judgments and rules for expressions are reported in Tab. 9. Judgment (t1) says that the basic value *unit* is well-typed in the empty context and has type Unit. Judgment (t2) instead says that any basic value in $\mathcal{BV}_{\mathtt{BT}}$ is well-typed in the empty context with type BT, while judgment (t3) says that any name or variable w in $\mathcal{N} \cup \mathcal{X}$ defined in a typing context Ψ is well-typed in Ψ with type $\Psi(\mathtt{w})$. Typing rules (t4) and (t5) states that given an expression EXP, well-typed in a context Ψ with product type $\mathtt{T}_1 \times \mathtt{T}_2$, the projections EXP.1 and EXP.2 are well-typed in Ψ and have types $\mathtt{T}_1$ and $\mathtt{T}_2$, respectively. Rules (t6) and (t7) say that given a name a, well-typed in a typing context Ψ with binder type $\mathtt{binder(T_1,T_2)}$, the operators val(a) and check(a,K) are well-typed in Ψ with

Table 10. Type-checking bio-processes and processes

$$(\mathsf{t9})\ \frac{}{\emptyset \vdash_b \mathtt{Nil} : (\emptyset, \emptyset)}$$

$$(\mathsf{t10})\ \frac{\Psi \vdash_b \mathtt{B}_1 : (N_1, C_1), \quad \Psi \vdash \mathtt{B}_2 : (N_2, C_2)}{\Psi \vdash \mathtt{B}_1 \,||\, \mathtt{B}_2 : (N_1 \cup N_2, C_1 \cup C_2)}, N_1 \cap N_2 = \emptyset$$

$$(\mathsf{t11})\ \frac{\emptyset \vdash_i \mathtt{I} : (\Psi_1, V), \quad \Psi/\Psi_1 \vdash_p \mathtt{P} : ok}{\Psi \vdash_b \mathtt{I[P]n} : (\{\mathtt{n}\}, \{(\mathtt{V}, \mathtt{n}) \mid \mathtt{V} \in V\})}$$

$$(\mathsf{t12})\ \frac{\Psi \vdash_e \mathtt{V} : \mathtt{T}_2}{\emptyset \vdash_i \mathtt{\#(n{:}T_1,V{:}T_2)} : (\{\mathtt{n} : \mathtt{binder(T_1,T_2)}\}, \emptyset)}\ \text{, provided } \mathtt{T}_2 \in \overline{\mathcal{T}}$$

$$(\mathsf{t13})\ \frac{\Psi \vdash_e \mathtt{V} : \mathtt{T}_2}{\emptyset \vdash_i \mathtt{\#c(n{:}T_1,V{:}T_2)} : (\{\mathtt{n} : \mathtt{binder(T_1,T_2)}\}, \{\mathtt{V}\})}\ \text{, provided } \mathtt{T}_2 \in \overline{\mathcal{T}}$$

$$(\mathsf{t14})\ \frac{\emptyset \vdash_i \mathtt{K(n{:}T_1,V{:}T_2)} : (\Psi_2, V_2), \quad \emptyset \vdash_i \mathtt{I} : (\Psi_1, V_1)}{\emptyset \vdash_i \mathtt{K(n{:}T_1,V{:}T_2)I} : (\Psi_1/\Psi_2, V_1 \cup V_2)}\ \text{, } \mathtt{n} \notin \mathsf{subj}(\mathtt{I}) \text{ and } \mathtt{V} \notin \mathsf{val}(\mathtt{I})$$

$$(\mathsf{t15})\ \frac{}{\emptyset \vdash_p \mathtt{nil} : ok} \qquad (\mathsf{t16})\ \frac{\Psi \vdash_e \mathtt{EXP} : \mathtt{Bool}, \quad \Psi \vdash_p \mathtt{A.P} : ok}{\mathtt{<EXP>A.P} : ok}$$

$$(\mathsf{t17})\ \frac{\Psi(\mathtt{w}) = \mathtt{T}_1, \quad \Psi \vdash_e \mathtt{EXP}' : \mathtt{T}, \quad \Psi \vdash_p \mathtt{P} : ok}{\Psi \vdash_p \mathtt{w!EXP'.P} : ok}\ \text{, } \mathtt{T}_1 = \mathtt{chan(T)} \text{ or } \mathtt{T}_1 = \mathtt{binder(T,T')}$$

$$(\mathsf{t18})\ \frac{\Psi(\mathtt{w}) = \mathtt{T}_1, \quad \overline{\mathcal{M}}(\mathtt{PAT}, \mathtt{T}) = \Psi_1, \quad \Psi/\Psi_1 \vdash_p \mathtt{P} : ok}{\Psi \vdash_p \mathtt{w?PAT.P} : ok}\ \text{, } \mathtt{T}_1 = \mathtt{chan(T)} \text{ or } \mathtt{T}_1 = \mathtt{binder(T,T')}$$

$$(\mathsf{t19})\ \frac{\Psi \vdash_p \mathtt{P} : ok}{\Psi \vdash_p \mathtt{delay.P} : ok} \qquad (\mathsf{t20})\ \frac{\Psi(\mathtt{w}) = \mathtt{binder(T,T')}, \quad \Psi \vdash_e \mathtt{EXP} : \mathtt{T}', \quad \Psi \vdash_p \mathtt{P} : ok}{\Psi \vdash_p \mathtt{ch(w,EXP).P} : ok}$$

$$(\mathsf{t21})\ \frac{\Psi \vdash_e \mathtt{EXP} : \mathtt{T}', \quad \Psi/\{\mathtt{n{:}binder(T,T')}\} \vdash_p \mathtt{P} : ok}{\Psi \vdash_p \mathtt{new(n{:}T,EXP{:}T').P} : ok}\ \text{, provided } \mathtt{T}' \in \overline{\mathcal{T}}$$

$$(\mathsf{t22})\ \frac{\Psi \vdash_e \mathtt{EXP} : \mathtt{Bool}, \quad \Psi \vdash_p \mathtt{A.P} : ok}{\mathtt{rep\ <EXP>A.P} : ok}$$

$$(\mathsf{t23})\ \frac{\Psi \vdash_p \mathtt{P}_1 : ok, \quad \Psi \vdash_p \mathtt{P}_2 : ok}{\mathtt{P}_1 | \mathtt{P}_2 : ok} \qquad (\mathsf{t24})\ \frac{\Psi \vdash_p \mathtt{G}_1 : ok, \quad \Psi \vdash_p \mathtt{G}_2 : ok}{\mathtt{G}_1 + \mathtt{G}_2 : ok}$$

types $\mathtt{T}_2$ and $\mathtt{Bool}$, respectively; this means that the two operators are applied correctly on names representing binder subjects. Rule (t8) states that given two expression $\mathtt{EXP}_1$ and $\mathtt{EXP}_2$, well-typed in a typing context Ψ with respectively types $\mathtt{T}_1$ and $\mathtt{T}_2$, the pair $(\mathtt{EXP}_1, \mathtt{EXP}_2)$ has product type $\mathtt{T}_1 \times \mathtt{T}_2$.

Type judgments and rules for bio-processes, interfaces and processes are reported in Tab. 10. A well-typed bio-process B is associated with a signature

(N, C), where N contains the indexes of all the boxes defined in $\mathtt{B}$ and C contains a pair $(\mathtt{V}, \mathtt{n})$ if and only if a binder with value $\mathtt{V}$ is declared as bound in a box with index $\mathtt{n}$. All the judgments and rules guarantee that no boxes with identical indexes are defined. Rule (t9) says that the deadlock box $\mathtt{Nil}$ is well-typed and has signature $(\emptyset, \emptyset)$ in any typing context. Rule (t10) states that given two bio-processes $\mathtt{B}_1$ and $\mathtt{B}_2$, well-typed in a typing context Ψ with respectively signatures (N_1, C_1) and (N_2, C_2) where $N_1 \cap N_2 = \emptyset$, the bio-process obtained by composing in parallel the two bio-processes is well-typed in Ψ with signature $(N_1 \cup N_2, C_1 \cup C_2)$. Notice that by controlling that the intersection of the two set of boxes indexes is empty we guarantee that no boxes in $\mathtt{B}_1$ and $\mathtt{B}_2$ have the same index. Rule (t11) controls the well-typedness of boxes. A box $\mathtt{I[P]n}$ well-typed in a typing context Ψ is associated with a signature $(\{\mathtt{n}\}, \{(\mathtt{V}, \mathtt{n}) \mid \mathtt{V} \in V\})$, where the first element is a set containing the index of the box and the second element contains the values $V \subset \mathcal{V}$ of the bound binders in $\mathtt{I}$, generated by the premises of the typing rule. The premises indeed control the well-typedness of the interface $\mathtt{I}$, generating a signature (Ψ_1, V), and the well-typedness of the process $\mathtt{P}$, which is verified in the typing context Ψ/Ψ_1, containing all the entries for the binder subjects.

The well-typedness of interfaces is verified by judgments and rules (t12-14). These rules verify that all the subjects and values of the binders composing an interface are distinct; generate a typing context containing the associations

Table 11. Type-checking environments, events and systems

$$\text{(t25)}\ \frac{}{\emptyset \vdash_{en} \emptyset : \emptyset} \qquad \text{(t26)}\ \frac{}{\emptyset \vdash_{en} (\mathtt{V_1@n,V_2@m}) : \{(\mathtt{V}_1, \mathtt{n}), (\mathtt{V}_2, \mathtt{m})\}}\ , \mathtt{n} \neq \mathtt{m}$$

$$\text{(t27)}\ \frac{\emptyset \vdash_{en} \mathtt{ENV}_1 : C_1, \quad \emptyset \vdash_{en} \mathtt{ENV}_2 : C_2}{\emptyset \vdash_{en} \mathtt{ENV}_1 \cup \mathtt{ENV}_2 : C_1 \cup C_2}\ , C_1 \cap C_2 = \emptyset$$

$$\text{(t28)}\ \frac{}{\emptyset \vdash_{ev} \emptyset : ok} \qquad \text{(t29)}\ \frac{\Psi \vdash_{ev} \mathtt{E}_1 : ok, \quad \Psi \vdash_{ev} \mathtt{E}_2 : ok}{\mathtt{E}_1 \cup \mathtt{E}_2 : ok}$$

$$\text{(t30)}\ \frac{\Psi \vdash_b \mathtt{B}_1 : (N_1, C_1), \quad \Psi \vdash_b \mathtt{B}_2 : (N_2, C_2), \quad \emptyset \vdash_{en} \mathtt{ENV}_1 : C_1', \quad \emptyset \vdash_{en} \mathtt{ENV}_2 : C_2'}{\Psi \vdash_{ev} \mathtt{sub(B_1,ENV_1,B_2,ENV_2)} : ok}$$

provided $C_1' \subseteq C_1,\ C_2' \subseteq C_2$ and

$$\forall\ \mathtt{V} \in \mathcal{V}.\ |\{(\mathtt{V}, \mathtt{n}) \in C_1 \setminus C_1' \mid \mathtt{n} \in \mathcal{N}\}| = |\{(\mathtt{V}, \mathtt{n}) \in C_2 \setminus C_2' \mid \mathtt{n} \in \mathcal{N}\}|$$

$$\text{(t31)}\ \frac{\Psi \vdash_b \mathtt{B} : (N, C), \quad \Psi \vdash_{ev} \mathtt{E} : ok, \quad \emptyset \vdash \mathtt{ENV} : C'}{\Psi \vdash \langle\langle \mathtt{B;\ E;\ ENV} \rangle\rangle : ok}\ , C = C'$$

between the binder subjects and the corresponding binder types; generate a set containing all the values of bound binders.

Rule (t15) says that the `nil` process is well-typed in any context. Rule (t16) says that an expression guarding a process `A.P` has to be always well-typed in a context Ψ with type `Bool`. Rules (t17) and (t18) control if outputs and inputs, respectively, are well-typed in a typing context Ψ. We have to guarantee that the output expression and the input pattern are consistent w.r.t. the type associated in Ψ to the communication channel; for input patterns this control is performed by the function $\overline{\mathcal{M}}$, a static matching function defined as follows:

$$\overline{\mathcal{M}}(a,b) = \begin{cases} \{\mathtt{x}:\mathtt{T}\} & \text{if } a = \mathtt{x} \text{ and } b = \mathtt{T} \\ \overline{\mathcal{M}}(\mathtt{PAT}_1,\mathtt{T}_1) \uplus \overline{\mathcal{M}}(\mathtt{PAT}_2,\mathtt{T}_2) & \text{if } a = (\mathtt{PAT}_1,\mathtt{PAT}_2) \text{ and } b = \mathtt{T}_1 \times \mathtt{T}_2 \\ \perp & \text{otherwise} \end{cases}$$

Obviously, for an input the well-typedness of its continuation process is verified in a typing context extended with the typing context result of the matching between the input pattern and the type of the input channel. Rule (t19) is straightforward, while rules (t20) and (t21) verify the well-typedness of actions change and new in a typing context Ψ. A change action `ch(w,EXP)` is well-typed in Ψ if `w` is a binder and the expression `EXP` is well-typed in Ψ with a type that corresponds to the type of the binder value. A new action `new(n:T,EXP:T′)` is well-typed in a typing context Ψ if the expression is well-typed in Ψ with the type `T′` explicitly annotated in the syntax of the action and if its continuation process is well-typed in Ψ extended with a new association of name `n` to a binder type `binder(T,T′)`. Rule (t22) is similar to rule (t16). Rules (t23) and (t24) say that operators `|` and `+` preserve well-typedness.

Type judgments and rules for environments, events and systems are in Tab. [11]. An environment is well-typed (t25-27) if it does not contain multiple `V@n` entries. This guarantees that the environment is not ambiguous in terms of complex bindings, i.e., we don't have the specification of multiple bindings on the same interface binder. Events are well-typed (t28-30) if all the substitutions are composed by well-typed bio-processes $\mathtt{B}_i$ and environments $\mathtt{ENV}_i$, with $i = 1, 2$, and the corresponding signatures (N_i, C_i) and C'_i guarantee that all the bindings specified in the environments are contained in the corresponding bio-processes and that the sets of pending binding values of $\mathtt{B}_1$ and $\mathtt{B}_2$ coincide. These two controls are verified by checking that $C'_i \subseteq C_i$ and that for each $\mathtt{V} \in \mathcal{V}$ we have that the cardinalities of sets $\{(\mathtt{V},\mathtt{n}) \in C_1 \setminus C'_1 \mid \mathtt{n} \in \mathcal{N}\}$ and $\{(\mathtt{V},\mathtt{n}) \in C_2 \setminus C'_2 \mid \mathtt{n} \in \mathcal{N}\}$ coincide, respectively. Notice that we do not require here equality because in events we can also specify bio-processes representing sub-complexes. Equality is instead a requisite in rule (t31) because we want to ensure all the bindings specified in the environments to be contained in the corresponding bio-processes and vice-versa.

We end the presentation of the type system by stating two results proving, respectively, that typing is invariant under structural congruence and sound with respect to the operational semantics.

Lemma 1. *Let SYS and SYS' be systems such that $\mathit{SYS} \equiv \mathit{SYS}_1$ and let $\Psi \in \overline{\mathcal{TC}}$ be a typing context. Then $\Psi \vdash \mathit{SYS} : ok$ iff $\Psi \vdash \mathit{SYS}_1 : ok$.*

Theorem 1. *Let $\Psi \in \overline{\mathcal{TC}}$ be a typing context. If $\Psi \vdash \mathit{SYS} : ok$ and $\mathit{SYS} \rightarrow \mathit{SYS}_1$ then $\Psi \vdash \mathit{SYS}_1 : ok$.*

5 Conclusions

The paper introduced a new programming language called BlenX. An operational semantics formally define the dynamics of BlenX programs. BlenX is also equipped with a type system that is invariant under structural congruence and sound with respect to the operational semantics. BlenX is an evolution of Beta-binders [5] and hence is strongly based on process calculi, from which it inherits the formal methods that allows verification and analysis of concurrent and distributed systems.

BlenX has been designed to model, simulate and analyse biological systems, but it is not an ad hoc or domain-specific language because it can handle also core computer science problems [2].

The main features of BlenX are

- typed and dynamically varying interfaces of boxes;
- sensitivity-based interaction decoupled from the complementarity of names of channels;
- one-to-one correspondence between the components of the system to be modeled and boxes specified in the BlenX system;
- description of complexes and dynamic generation of complexes as graphs and graph manipulation primitives;
- usage of events as further primitives.

We presented here just the core part of BlenX. There are also other features (see [1]), like the handling of functions within the expressions and the introduction of conditional events that can help defining and performing in-silico experiments, that we plan to introduce in the formal semantics in an extended version of this paper.

References

1. Dematté, L., Priami, C., Romanel, A.: The BlenX Language: A Tutorial. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 313–365. Springer, Heidelberg (2008)
2. Prandi, D., Priami, C., Quaglia, P.: Communicating by compatibility. JLAP 75, 167 (2008)
3. Priami, C.: Algorithmic systems biology. CACM 52(5), 80–88 (2009)
4. Priami, C., Quaglia, P.: Modeling the dynamics of bio-systems. Briefings in Bionformatics 5(3) (2004)

5. Priami, C., Quaglia, P.: Beta Binders for Biological Interactions. In: Danos, V., Schächter, V. (eds.) CMSB 2004. LNCS (LNBI), vol. 3082, pp. 20–33. Springer, Heidelberg (2005)
6. Regev, A., Shapiro, E.: Cells as computations. Nature 419, 343 (2002)
7. Welch, P.H., Barnes, F.R.M.: Communicating mobile processes: Introducing occam-pi. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 175–210. Springer, Heidelberg (2005)

Flow Policy Awareness for Distributed Mobile Code

Ana Almeida Matos

SQIG-Instituto de Telecomunicações and Instituto Superior Técnico de Lisboa

Abstract. In the context of global computing, information flow security must deal with the decentralized nature of security policies. This issue is particularly challenging when programs are given the flexibility to perform declassifying instructions. We point out potential unwanted behaviors that can arise in a context where such programs can migrate between computation domains with different security policies. We propose programming language techniques for tackling such unwanted behaviors, and prove soundness of those techniques at the global computation level.

1 Introduction

The new possibilities opened by global computing have brought information security issues to a new level of concern. Indeed, such possibilities can just as well be exploited by parties with hazardous intentions. Many attacks arise at the application level, and can be tackled by means of programming language design and analysis techniques, such as static analysis and proof carrying code. For instance, confidentiality can be violated by execution of programs that reveal secret information. This kind of program behavior can be controlled using *information flow* analyses [16], by detecting dependencies in programs that could encode flows of information from private to publicly available resources.

In information flow research, it has been a challenging problem to find an alternative to the classical non-interference property [9] that is flexible enough to allow for *declassification* to take place in a controlled manner [19]. So far, most solutions have been directed towards local computation scenarios, thus overlooking decentralization issues that are inherent to distributed settings. Indeed, enforcement of confidentiality in networks must deal with distributed security policies, since different *computation domains* (or *sites*) follow different security orientations. For example, migrating programs that were conceived to comply with certain flow policies don't necessarily respect those of the computational locations they might end up executing at. This problem seems to be beyond the grasp of single declassification constructs that can restrict by whom, when, what, or where in the program declassification can be performed [17], since now the question is: *in which context?*

In this paper we show that the issue of enabling and controlling flexible information flow policies in computations that can spread out over sites that are governed by different flow policies can be addressed at the programming language level. We propose to remove some of the burden of restricting declassification away from the declassification instruction itself, and transfer it to new program constructs that provide awareness about the flow policy of the context in which it is running. Given the appropriate tools to predict alternatives to the pieces of code that contain potentially forbidden declassification operations, it becomes realistic to write programs that can safely run under any flow policy.

Some security minded distributed network models have been proposed with the purpose of controlling the migration of code in between computation sites, such as by means of programmable domains [4] and type systems [14]. These ideas can be applied to the proof-carrying code model [15], since it consists of a particular instance of boundary transposition control that performs type checks to incoming code [10]. We propose to apply migration control techniques to the problem of controlling declassification by preventing programs from migrating to sites if they would potentially violate that site's flow policies. However, we fall short of technical mechanisms that would allow, on one hand, for a site to know what are the most flexible flow policies that a program sets up for its own executions; on another hand, for a program to know how flexible is the flow policy of the context in which it is running.

Setting. We follow the non-disclosure point of view of information flow analysis [1,3]. The *non-disclosure property* is a generalization of non-interference. It uses information provided by the program semantics describing which flow policies are valid at different points of the computation, to ensure that, at each step, all information flows comply with the valid flow policy. In order to enable local dynamic changes to the valid flow policy, the programming language may be enriched with a *flow declaration* construct (flow F in M) that simply declares the flow policy F that is valid in its scope M within the program. It is then easy to set up more flexible flow policy environments for delimited blocks of code, as for instance the part of a program that is executed by authenticated users:

$$(\text{if } \textit{authenticated} \text{ then } (\text{flow } F_{permissive} \text{ in } M) \text{ else } N)$$

This program declares that flows in M conform to a policy that is extended (made more permissive) by $F_{permissive}$ (the N branch is of course not given this flexibility). In other words, M may contain declassifications that comply to $F_{permissive}$.

Once the language is enriched with flow declarations (or any other means for expressing declassification), some mechanism for controlling its usage is desirable. This is particularly relevant in mobile code settings. For instance, a computation domain d might want to impose a limit to the flexibility of the flow declarations that it executes, and prevent incoming code from containing:

$$(\text{flow } F_{all\_is\_allowed} \text{ in } M)$$

In the above example, the flow declaration validates any information flow that might occur in M, regardless of what is considered acceptable by d. This motivates the notion of a domain's *allowed flow policy*, which represents the flow policy that should rule for all programs that are running at a certain domain. We can then define the notion of *confinement with respect to a flow policy* as a property of programs that can only perform steps that comply with that allowed flow policy. We will see that this property can be formalized by making use of the information about the declared flow policies that is provided by the semantics.

At the moment that a program is written, it might be hard to anticipate which flow policies will be imposed at execution time by the domains where the program will run. In a distributed context with code mobility, the problem becomes more acute, since the computation site might change *during* execution, along with the allowed flow policy with which the program must comply. In order to provide programs with some awareness regarding the flow policy that is ruling in the current computation domain, we introduce the *allowed-condition*, written (allowed F then M else N), that tests whether the flow policy F is allowed by the current domain and executes branches M or N accordingly. Programs can then offer alternative behaviors to be taken in case the domains they end up at do not allow declassifications of the kind they wished to perform:

$$(\text{allowed } F_{disclose\_secret} \text{ then } M \text{ else } plan\_B)$$

The allowed-condition brings no guarantees that the "*plan_B*" of the above program does not disclose just as much as the M branch. However, misbehaving programs can be rejected by the domains where they would like to execute, so its chances of being allowed to execute are increased by adequately "protecting" portions of code containing declassifications by appropriate allowed-conditions.

In the spirit of the proof carrying code model, domains can statically check incoming code against their own flow policies, ideally assisted by certificates that are carried by the program, and then decide upon whether those programs should be "let in". A certificate could consist of information about all the flow policies that are declared in the program and do *not* appear within the "allowed" branch of an allowed-condition. We call this flow policy the *declassification effect* of the program, and provide a type system for obtaining it. Then, while the program

$$(\text{allowed } F_1 \text{ then } M \text{ else } (\text{flow } F_2 \text{ in } N))$$

would have a declassification effect that includes F_2 – meaning that it should only be allowed to run in domains where F_2 is allowed –, the program

$$(\text{allowed } F \text{ then } (\text{flow } F \text{ in } M) \text{ else } N)$$

(assuming that M and N have no flow declarations) would have an empty declassification effect – meaning that it could be safely allowed to run in any domain.

In order to formalize these ideas, it is useful to consider a concrete distributed language with code mobility, where we use the declassification effect to control migration according to the following rule: programs can only migrate to a site if their declassification behaviors conform to that site's flow policy. We can then analyze the conditions under which the programs of this language comply with a network level version of the information flow and confinement properties.

We start by presenting the language and giving some intuitions (Section 2). Two main section follow, each presenting the security analysis for our information flow property of non-disclosure (Section 3) and for the new confinement property (Section 4). Finally we discuss related work and conclude (Section 5). Detailed proofs may be found in [2].

2 Language

We consider a distributed imperative higher-order λ-calculus with reference and thread creation, where we include flow policy declarations (for directly manipulating flow policies [1,3]) and the new allowed flow policy condition that branches according to whether a certain flow policy is allowed in the program's computing context. We also add a notion of computation domain, to which we associate an allowed flow policy, and a code migration primitive. Programs computing in different domains are subjected to different allowed flow policies, and their behavior is affected accordingly – this is the main novelty in this language.

Syntax. Security annotations and types are apparent in the syntax of the language, though they do not play any role in the operational semantics (they will be used at a later stage of the analysis). Security levels l, j, k are sets of principals, which are ranged over by $p, q \in \boldsymbol{Pri}$. They are associated to references (and reference creators), representing the set of principals that are allowed to read the information contained in each reference. We also decorate references with the type of the values that they can hold. The syntax of types τ, σ, θ is given in the next section. In the following we may omit reference subscripts whenever they are not relevant. A security level is also associated to each thread, and appears as a subscript of thread names. This level can be understood as the set of principals that are allowed to know about the location of the thread in the network. Flow policies A, F, G are binary relations over $\boldsymbol{Pri}$. A pair $(p, q) \in F$, most often written $p \prec q$, is to be understood as "information may flow from principal p to principal q". We denote, as usual, by F^* the reflexive and transitive closure of F.

The language of *threads* (defined in Fig. 1) is based on a call-by-value λ-calculus extended with the imperative constructs of ML, conditional branching and boolean values (here the $(\varrho x.W)$ construct provides for recursive values). Variables x, references a, b, c, threads m, n and domains d are drawn from the disjoint countable sets $\boldsymbol{Var}$ and $\boldsymbol{Ref}$, $\boldsymbol{Dom} \neq \emptyset$ and $\boldsymbol{Nam}$, respectively. Reference names can be created at runtime. The new feature is the allowed-condition (allowed F then N_t else N_f) which tests the flow policy F and branches to N_t or N_f accordingly. The flow declaration construct (flow F in M) extends the current flow policy F within M's scope. Also worth mentioning are the thread

$$\begin{array}{lll}
\textit{Variables} & x, y \in \boldsymbol{Var} & \textit{Thread Names} \quad m, n \in \boldsymbol{Nam} \\
\textit{Reference Names} & a, b, c \in \boldsymbol{Ref} & \textit{Domain Names} \quad d \in \boldsymbol{Dom} \\
\textit{Values} & V \in \boldsymbol{Val} & ::= () \mid x \mid a_{l,\theta} \mid (\lambda x.M) \mid tt \mid f\!f \\
\textit{Pseudo-values} & W \in \boldsymbol{Pse} & ::= V \mid (\varrho x.W) \\
\textit{Expressions} & M, N \in \boldsymbol{Exp} & ::= W \mid (M\ N) \mid (M; N) \mid (\text{if } M \text{ then } N_t \text{ else } N_f) \\
 & & \quad (\text{ref}_{l,\theta}\ M) \mid (!\ N) \mid (M := N) \mid (\text{thread}_l\ M) \mid (\text{goto } d) \mid \\
 & & \quad (\text{flow } F \text{ in } M) \mid \boldsymbol{(\text{allowed } F \text{ then } N_t \text{ else } N_f)} \\
\textit{Threads} & & ::= M^{m_j} \quad (\in \boldsymbol{Exp} \times \boldsymbol{Nam} \times 2^{\boldsymbol{Pri}})
\end{array}$$

Fig. 1. Syntax of threads

creator $(\text{thread}_l\ M)$, which spawns a concurrent thread M, to which a name and the security level l is given, and the migration construct $(\text{goto}\ d)$, which triggers the migration of the thread that executes it to the domain d.

Networks are flat juxtapositions of domains, each containing a store and a pool of threads, which are subjected to the flow policy of the domain. Threads run concurrently in *pools* $P : (\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \rightarrow \boldsymbol{Exp}$, which are mappings from decorated thread names to expressions (they can also be seen as sets of threads). *Stores* $S : (\boldsymbol{Ref} \times 2^{\boldsymbol{Pri}} \times \boldsymbol{Typ}) \rightarrow \boldsymbol{Val}$ map decorated reference names to values. To keep track of the locations of threads it suffices to maintain a mapping from thread names to domain names. This is the purpose of the *position-tracker* $T : (\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \rightarrow \boldsymbol{Dom}$, which is a mapping from a finite set of decorated thread names to domain names. The pool P containing all the threads in the network, the mapping T that keeps track of their positions, and the store S containing all the references in the network, form *configurations* $\langle P, T, S\rangle$, over which the evaluation relation is defined in the next subsection. The flow policies that are allowed by each domain are kept by the *policy-mapping* $W : \boldsymbol{Dom} \rightarrow 2^{\boldsymbol{Pri} \times \boldsymbol{Pri}}$ from domain names to flow policies, which is considered fixed in this model.

Operational Semantics. We now define the semantics of the language as a small step operational semantics on configurations. The call-by-value evaluation order can be conveniently specified by writing expressions using *evaluation contexts*. We write $\mathrm{E}[M]$ to denote an expression where the subexpression M is placed in the evaluation context E, obtained by replacing the occurrence of $[]$ in E by M.

$$\begin{array}{l} \textit{Evaluation Contexts}\ \mathrm{E} ::= [] \mid (\mathrm{E}\ N) \mid (V\ \mathrm{E}) \mid (\mathrm{E}; N) \mid (\mathrm{ref}_{l,\theta}\ \mathrm{E}) \mid (!\ \mathrm{E}) \mid \\ \quad (\mathrm{E} := N) \mid (V := \mathrm{E}) \mid (\text{if E then } N_t \text{ else } N_f) \mid \textbf{(flow } \boldsymbol{F} \textbf{ in E)} \mid \end{array}$$

We denote by $\lceil \mathrm{E} \rceil$ the flow policy that is enabled by the evaluation context E, and consists of the union of all the policies that are declared by E:

$$\begin{array}{c} \lceil [] \rceil = \emptyset, \qquad \lceil (\text{flow } F \text{ in E}) \rceil = F \cup \lceil \mathrm{E} \rceil, \\ \lceil \mathrm{E}'[\mathrm{E}] \rceil = \lceil \mathrm{E} \rceil,\ \textit{if}\ \mathrm{E}'\ \textit{does not contain flow declarations} \end{array}$$

We use some notations and conventions for defining transitions on configurations. Given a configuration $\langle P, T, S\rangle$, we call the pair $\langle T, S\rangle$ its *state*. We define $\mathrm{dom}(S)$ as the set of decorated reference names that are mapped by S; similarly, the sets $\mathrm{dom}(W)$, $\mathrm{dom}(P)$ and $\mathrm{dom}(T)$, are the sets of domains and decorated names of threads that are mapped by W, P and T. We say that a thread or reference name is fresh in T or S if it does not occur, with any subscript, in $\mathrm{dom}(T)$ or $\mathrm{dom}(S)$, respectively. We denote by $\mathrm{tn}(P)$ and $\mathrm{rn}(P)$ the set of decorated thread and reference names, respectively, that occur in the expressions of P (this notation is extended in the obvious way to expressions). We let $\mathrm{fv}(M)$ be the set of variables occurring free in M. We restrict our attention to well formed configurations $\langle P, T, S\rangle$ satisfying the following additional conditions: $\mathrm{rn}(P) \subseteq \mathrm{dom}(S)$; for any $a_{l,\theta} \in \mathrm{dom}(S)$ we have $\mathrm{rn}(S(a_{l,\theta})) \subseteq \mathrm{dom}(S)$ $\mathrm{dom}(P) \subseteq \mathrm{dom}(T)$; $\mathrm{tn}(\mathrm{dom}(S)) \subseteq \mathrm{dom}(T)$; all threads in a configuration have distinct names, and also all occurrences of a name in a configuration have the same subscripts. We denote by $\{x \mapsto W\}M$ the capture-avoiding substitution of

$$A \vdash \langle \mathrm{E}[((\lambda x.M)\ V)], S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \mathrm{E}[\{x \mapsto V\}M], S\rangle$$

$$A \vdash \langle \mathrm{E}[(\mathrm{if}\ tt\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f)], S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \mathrm{E}[N_t], S\rangle$$

$$A \vdash \langle \mathrm{E}[(\mathrm{if}\ f\!f\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f)], S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \mathrm{E}[N_f], S\rangle$$

$$\boldsymbol{A \vdash \langle \mathbf{E}[(\mathbf{allowed}\ F\ \mathbf{then}\ N_t\ \mathbf{else}\ N_f)], S\rangle \xrightarrow[\lceil \mathbf{E}\rceil]{} \langle \mathbf{E}[N_t], S\rangle,}\ \textit{where}\ \boldsymbol{F \subseteq A^*}$$

$$\boldsymbol{A \vdash \langle \mathbf{E}[(\mathbf{allowed}\ F\ \mathbf{then}\ N_t\ \mathbf{else}\ N_f)], S\rangle \xrightarrow[\lceil \mathbf{E}\rceil]{} \langle \mathbf{E}[N_f], S\rangle,}\ \textit{where}\ \boldsymbol{F \not\subseteq A^*}$$

$$A \vdash \langle \mathrm{E}[(V; N)], S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \mathrm{E}[N], S\rangle$$

$$A \vdash \langle \mathrm{E}[(\varrho x.W)], S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \mathrm{E}[(\{x \mapsto (\varrho x.W)\}\ W)], S\rangle$$

$$A \vdash \langle \mathrm{E}[(\mathrm{flow}\ F\ \mathrm{in}\ V)], S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \mathrm{E}[V], S\rangle$$

$$A \vdash \langle \mathrm{E}[(!\ a_{l,\theta})], S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \mathrm{E}[V], S\rangle,\ \textit{where}\ S(a_{l,\theta}) = V$$

$$A \vdash \langle \mathrm{E}[(a_{l,\theta} := V)], S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \mathrm{E}[()], [a_{l,\theta} := V]S\rangle$$

$$A \vdash \langle \mathrm{E}[(\mathrm{ref}_{l,\theta}\ V)], S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \mathrm{E}[a_{l,\theta}], [a_{l,\theta} := V]S\rangle,\ \textit{where } a \textit{ fresh in } S$$

$$W \vdash \langle \{\mathrm{E}[(\mathrm{thread}_l\ N)]^{m_j}\}, T, S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[()]^{m_j}, N^{n_k}\}, [n_k := T(m_j)]T, S\rangle,\ \textit{where } n \textit{ fresh in } T$$

$$\boldsymbol{W \vdash \langle \{\mathbf{E}[(\mathbf{goto}\ d)]^{m_j}\}, T, S\rangle \xrightarrow[\lceil \mathbf{E}\rceil]{} \langle \{\mathbf{E}[()]^{m_j}\}, [m_j := d]T, S\rangle,}\ \textit{if}\ \boldsymbol{\emptyset \vdash \mathbf{E}[()] : s, \tau}\ \textit{and}\ \boldsymbol{s \subseteq W(d)^*}$$

$$\frac{W(T(m_j)) \vdash \langle M, S\rangle \xrightarrow[F]{} \langle M', S'\rangle}{W \vdash \langle \{M^{m_j}\}, T, S\rangle \xrightarrow[F]{} \langle \{M'^{m_j}\}, T, S'\rangle} \qquad \frac{W \vdash \langle P, T, S\rangle \xrightarrow[F]{} \langle P', T', S'\rangle \quad \langle P \cup Q, T, S\rangle\ \textit{well formed}}{W \vdash \langle P \cup Q, T, S\rangle \xrightarrow[F]{} \langle P' \cup Q, T', S'\rangle}$$

Fig. 2. Operational semantics. See Fig. 4 on side condition of the migration rule.

W for the free occurrences of x in M. The operation of adding or updating the image of an object z to z' in a mapping Z is denoted $[z := z']Z$.

The transitions of our *small step semantics* are defined in Fig. 2. In the local computations rules, the '$A \vdash$' turnstile makes explicit the allowed flow policy A of the computation domain. The semantics of local evaluation is embedded in the distributed language by specifying the local flow policy A as $W(T(m_j))$, where $T(m_j)$ represents the location of the thread m_j. The last rule establishes that the execution of a pool of threads is compositional (up to the expected restriction on the choice of new names). The semantics of global computations introduces the rules for thread creation and for migration. The latter depends on the type system of Section 4, similarly to a rule in [10]. The condition represents the standard theoretical requirement of checking incoming code before allowing it to execute in a given machine. For now, it is enough to know that s represents an approximation of the flow policies that are used by the typed expression.

The labeled transition rules of our semantics are decorated with the flow policy declared by the evaluation context where they are performed. This label is added only for the purpose of the security analysis. In particular, the evaluation of (flow F in M) simply consists in the evaluation of M, annotated with a flow policy that comprises (in the sense of set inclusion) F. The lifespan of the flow declaration terminates when the expression M in its scope terminates.

The allowed flow policy A of a site represents a restriction on the flow policies that can be set up by programs running in that site. It determines the behavior of the allowed-condition (allowed F then N_t else N_f), which tests whether it is allowed to set up flow declarations enabling F in its "allowed" branch N_t. The alternative branch N_f is executed otherwise. A typical usage could be:

$$(\text{allowed } \{H \prec L\} \text{ then } (\text{flow } \{H \prec L\} \text{ in } (x_L := (!\ y_H))) \text{ else } \mathit{plan\_B}) \qquad (1)$$

The allowed flow policy is also used to determine whether or nota migration instruction may be consummated. The idea is that a thread can only migrate to another domain if it respects its allowed flow policy. E.g., the configuration

$$\langle \{\mathrm{E}[((\text{goto } d); (\text{flow } F \text{ in } M))]^{m_j}\}, T, S\rangle \qquad (2)$$

can only proceed if $W(d)$ allows for F; otherwise it gets stuck (this will become clear in Section 4). The flow declaration does not imply checks to the allowed flow policy of the site. Here we preserve the original semantics of the flow declaration [3] as a construct that does not change the behavior of programs, and leave the functionality of inspecting the allowed flow policy to the allowed-condition.

According to the chosen semantics, dereferencing and assigning to remote references can be done transparently. One may wonder whether it is correct to consider a system with a shared global state as distributed. We point out that in this model, the flow policies are distributed, while the behavior of a program fragment may differ on different machines. As an example, consider the thread

$$(\text{allowed } F \text{ then } (y_L := 1) \text{ else } (y_L := 2))^{m_j} \qquad (3)$$

running in a network $\langle P, T, S\rangle$ such that $W(d_1) = F_1$ and $W(d_2) = F_2$, where $F \subseteq F_2^*$ but $F \not\subseteq F_2^*$. The thread will perform different assignments depending on whether $T(m_j) = d_1$ or $T(m_j) = d_2$. In Section 3 we will see that their behavior is distinguishable by an information flow bisimulation relation. In other words, the network does exhibit a distributed behavior. For a study of a similar model with distributed and mobile references, see [1].

3 Information Flow Analysis

We start by briefly defining the underlying information flow policy that this work is based on, non-disclosure for networks (we refer the reader to [1] for further explanations). We will see that a new form of migration leaks appears due to the new allowed-condition primitive that was introduced in our language. We then present a type system for enforcing non-disclosure, and state its soundness.

Non-Disclosure for Networks. The study of confidentiality typically relies on a lattice of security levels [8], corresponding to security clearances that are associated to information containers in a programming language. Here, as in [3], we will use a pre-lattice (a preordered set with least upper-bound and a greatest lower-bound operations), a more general structure that is convenient for defining a dynamic flow relation that accounts for runtime changes in the flow policy of a program. More concretely, our security pre-lattices are derived from a lattice where security levels are sets of principals representing read-access rights, partially ordered by the reverse inclusion relation, which indicates allowed flows of information: if $l_1 \supseteq l_2$ then information in a reference a_{l_1} may be transferred to b_{l_2}, since the principals allowed to read this value from b are also allowed to read it from a. Flow policies, which are binary relations between principals, then represent additional directions in which information is allowed to flow. This leads to the underlying *preorder on security levels*, given by $l_1 \preceq_F l_2$ iff $(l_1 \uparrow_F) \supseteq (l_2 \uparrow_F)$, where the *$F$-upward closure* $l \uparrow_F$ of a security level l, defined as $\{q | \exists p \in l.\ pF^* q\}$, contains all the principals that are allowed by the policy F to read the contents of a reference labeled l. Notice that $\preceq_F$ extends $\supseteq$ in the sense that $\preceq_F$ contains $\supseteq$ and $\preceq_\emptyset = \supseteq$. We choose $l_1 \curlywedge_F l_2 = l_1 \cup l_2$ and $l_1 \curlyvee_F l_2 = (l_1 \uparrow_F) \cap (l_2 \uparrow_F)$ as meet and join operations, from which $\top = \emptyset$ and $\bot = \boldsymbol{Pri}$.

Equipped with a flow relation between security levels, we define the notion of low-equality between states with respect to a flow policy F and security level l as pointwise equality between the low part of the position tracker and of the store, i.e. $\langle T_1, S_1 \rangle =^{F,l} \langle T_2, S_2 \rangle \overset{\text{def}}{\Leftrightarrow} T_1 \upharpoonright^{F,l} = T_2 \upharpoonright^{F,l}$ and $S_1 \upharpoonright^{F,l} = S_2 \upharpoonright^{F,l}$, where $T \upharpoonright^{F,l} \overset{\text{def}}{=} \{(n_k, d) \mid (n_k, d) \in T \ \&\ k \preceq_F l\}$ and $S \upharpoonright^{F,l} \overset{\text{def}}{=} \{(a_{k,\theta}, V) \mid (a_{k,\theta}, V) \in S \ \&\ k \preceq_F l\}$. Intuitively, two states are "low-equal" if they have the same "low-domain", and if they give the same values to all objects (in this case, references and threads) that are labeled with "low" security levels.

Since we are in a concurrent setting, it is natural to formulate our information flow property in terms of a bisimulation [5]. Our bisimulation, which is based on the small-step semantics defined in Section 2, relates two pools of threads if they show the same behavior on the low part of two states. We denote by $\rightarrow^*$ the reflexive and transitive closure of the union of the transitions $\underset{F}{\rightarrow}$, for all F.

Definition 1 ($\approx_l$). *An l-bisimulation is a symmetric relation $\mathcal{R}$ on sets of threads such that, for all T_1, S_1, T_2, S_2:*

$$P_1 \ \mathcal{R}\ P_2 \textit{ and } W \vdash \langle P_1, T_1, S_1 \rangle \underset{F}{\rightarrow} \langle P_1', T_1', S_1' \rangle \textit{ and } \langle T_1, S_1 \rangle =^{F,l} \langle T_2, S_2 \rangle \textit{ implies}$$
$$\exists P_2', T_2', S_2' \ .\ W \vdash \langle P_2, T_2, S_2 \rangle \rightarrow^* \langle P_2', T_2', S_2' \rangle \textit{ and } \langle T_1', S_1' \rangle =^{\emptyset,l} \langle T_2', S_2' \rangle \textit{ and } P_1' \ \mathcal{R}\ P_2'$$
$$\textit{when: } (\text{dom}(S_1{}') - \text{dom}(S_1)) \cap \text{dom}(S_2) = \emptyset \textit{ and } (\text{dom}(T_1{}') - \text{dom}(T_1)) \cap \text{dom}(T_2) = \emptyset$$

The largest l-bisimulation is denoted by $\approx_l$.

Note that for any l, the set of pairs of named values is an l-bisimulation. Furthermore, the union of all l-bisimulations is the largest l-bisimulation.

Our bisimulation potentially relates more programs than one for Non-interference thanks to the stronger premise $S_1 =^{F,l} S_2$. By starting with pairs of memories that are low-equal "to a greater extent", i.e. that coincide in a larger portion of the memory, the condition on the behavior of the program P_2 becomes weaker.

Note that the relation $\approx_l$ is not reflexive. In fact, a program is bisimilar to itself, if the high part of the state never interferes with the low part, i.e., no security leak can occur. This motivates the definition of our security property:

Definition 2 (Non-disclosure for Networks). *A pool of threads P satisfies Non-disclosure for Networks if it satisfies $P \approx_l P$ for all security levels l.*

Intuitively, the above definition requires information flows occurring at any computation step that can be performed by some thread in a network, to comply with the flow policy that is declared by the context where the command is executed.

We are considering a simplistic memory model that is globally shared, thus avoiding *migration leaks* that derive from synchronization behaviors on memory

$$[\textsc{Nil}]\ \Gamma \vdash () : \mathsf{unit} \quad [\textsc{BoolT}]\ \Gamma \vdash tt : \mathsf{bool} \quad [\textsc{BoolF}]\ \Gamma \vdash \mathit{ff} : \mathsf{bool} \quad [\textsc{Loc}]\ \Gamma \vdash a_{l,\theta} : \theta\ \mathrm{ref}_l$$

$$[\textsc{Var}]\ \Gamma, x:\tau \vdash x : \tau \qquad [\textsc{Abs}]\ \frac{\Gamma, x:\tau \vdash^j_F M : s, \sigma}{\Gamma \vdash (\lambda x.M) : \tau \xrightarrow[F,j]{s} \sigma} \qquad [\textsc{Rec}]\ \frac{\Gamma, x:\tau \vdash^j_F W : s, \tau}{\Gamma \vdash (\varrho x.W) : \tau}$$

$$[\textsc{Ref}]\ \frac{\Gamma \vdash^j_F M : s, \theta \quad s.r, s.t \preceq_F l}{\Gamma \vdash^j_F (\mathrm{ref}_{l,\theta}\ M) : s \curlyvee \langle \bot, l, \bot\rangle, \theta\ \mathrm{ref}_l} \qquad [\textsc{Der}]\ \frac{\Gamma \vdash^j_F M : s, \theta\ \mathrm{ref}_l}{\Gamma \vdash^j_F (!\ M) : s \curlyvee \langle l, \top, \bot\rangle, \theta}$$

$$[\textsc{Ass}]\ \frac{\begin{array}{ll} \Gamma \vdash^j_F M : s, \theta\ \mathrm{ref}_l & s.t \preceq_F s'.w \\ \Gamma \vdash^j_F N : s', \theta & s.r, s'.r, s.t, s'.t \preceq_F l \end{array}}{\Gamma \vdash^j_F (M := N) : s \curlyvee s' \curlyvee \langle \bot, l, \bot\rangle, \mathsf{unit}} \qquad [\textsc{Seq}]\ \frac{\begin{array}{l} \Gamma \vdash^j_F M : s, \tau \\ \Gamma \vdash^j_F N : s', \sigma \end{array} \quad s.t \preceq_F s'.w}{\Gamma \vdash^j_F (M; N) : s \curlyvee s', \sigma}$$

$$[\textsc{Cond}]\ \frac{\Gamma \vdash^j_F M : s, \mathsf{bool} \quad \begin{array}{l} \Gamma \vdash^j_F N_t : s_t, \tau \\ \Gamma \vdash^j_F N_f : s_f, \tau \end{array} \quad s.r, s.t \preceq_F s_t.w, s_f.w}{\Gamma \vdash^j_F (\mathrm{if}\ M\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f) : s \curlyvee s_t \curlyvee s_f \curlyvee \langle \bot, \top, s.r\rangle, \tau}$$

$$[\textsc{Allow}]\ \frac{\begin{array}{l} \boldsymbol{\Gamma \vdash^j_F N_t : s_t, \tau} \\ \boldsymbol{\Gamma \vdash^j_F N_f : s_f, \tau} \end{array} \quad \boldsymbol{j \preceq_F s_t.w, s_f.w}}{\boldsymbol{\Gamma \vdash^j_F (\mathbf{allowed}\ F'\ \mathbf{then}\ N_t\ \mathbf{else}\ N_f) : s_t \curlyvee s_f \curlyvee \langle \bot, \top, j\rangle, \tau}}$$

$$[\textsc{Flow}]\ \frac{\Gamma \vdash^j_{F \cup F'} N : s, \tau}{\Gamma \vdash^j_F (\mathrm{flow}\ F'\ \mathrm{in}\ N) : s, \tau} \qquad [\textsc{App}]\ \frac{\begin{array}{ll} \Gamma \vdash^j_F M : s, \tau \xrightarrow[F,j]{s'} \sigma & s.t \preceq_F s''.w \\ \Gamma \vdash^j_F N : s'', \tau & s.r, s''.r, s.t, s''.t \preceq_F s'.w \end{array}}{\Gamma \vdash^j_F (M\ N) : s \curlyvee s' \curlyvee s'' \curlyvee \langle \bot, \top, s.r \curlyvee s''.r\rangle, \sigma}$$

$$[\textsc{Thr}]\ \frac{j \preceq_F l \qquad \Gamma \vdash^l_\emptyset M : s, \mathsf{unit}}{\Gamma \vdash^j_F (\mathrm{thread}_l\ M) : \langle \bot, j \curlywedge s.w, \bot\rangle, \mathsf{unit}} \qquad [\textsc{Mig}]\ \Gamma \vdash^j_F (\mathrm{goto}\ d) : \langle \bot, j, \bot\rangle, \mathsf{unit}$$

Fig. 3. Type and effect system for non-disclosure for networks

accesses [1]. However, in our setting, migration leaks can be encoded nonetheless. The idea is that a program can reveal information about the position of a thread in a network by testing the flow policy that is allowed by that site:

$$(\mathrm{if}\ !x_H\ \mathrm{then}\ (\mathrm{goto}\ d_1)\ \mathrm{else}\ (\mathrm{goto}\ d_2)); (\mathrm{allowed}\ F\ \mathrm{then}\ (y_L := 1)\ \mathrm{else}\ (y_L := 2)) \quad (4)$$

In this example, the thread migrates to domains d_1 or d_2 depending on the tested high value; then, if these domains have different allowed flow policies, different low-assignments are performed, thus revealing high level information. Therefore, the program is insecure with respect to non-disclosure for networks. The fact that migration issues that are typical of distributed settings appear in spite of the state being globally shared allows us to make the point that migration leaks are not specific to distributed memory models. In fact, they can occur whenever the semantics of a program fragment differs on different machines.

Type System. We now present a type and effect system that accepts programs that satisfy non-disclosure for networks, as defined in Section 3. The judgments of the type and effect system, presented in Fig. 3, have the form $\Gamma \vdash^j_F M : s, \tau$, meaning that the expression M is typable with type τ and security effect s in the typing context Γ: $\boldsymbol{Var} \rightarrow \boldsymbol{Typ}$, which assigns types to variables. The turnstile has two parameters: the flow policy F *declared by the context*, is the one that is valid in the evaluation context in which the expression M is typed, and contributes to the meaning of the flow relations between security levels; the security level j is the confidentiality level associated to the position in the network of the thread containing M. The security effect s is composed of three security levels: $s.r$ is the *reading effect*, an upper-bound on the security levels of the references that are read by M; $s.w$ is the *writing effect*, a lower bound on the references that are written by M; $s.t$ is the *termination effect*, an upper bound on the level of the references on which the termination of expression M might depend. The reading and termination levels are composed in a covariant way, whereas the writing level is contravariant. Types have the following syntax (t is a type variable):

$$\tau, \sigma, \theta \ \in \ \boldsymbol{Typ} \ ::= \ t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta \ \mathrm{ref}_l \mid \tau \xrightarrow[F,j]{s} \sigma$$

Typable expressions that reduce to a function that takes a parameter of type τ, that returns an expression of type σ, and with a *latent* [12] effect s, flow policy F and security level j have the function type $\tau \xrightarrow[F,j]{s} \sigma$.

We use a (join) pre-semilattice on security effects, that is obtained from the pointwise composition of the pre-lattices of the security effects. More precisely, $s \preceq_F s'$ iff $s.r \preceq_F s'.r$ & $s'.w \preceq_F s.w$ & $s.t \preceq_F s'.t$, and $s \curlyvee_F s'$ iff $\langle s.r \curlyvee_F s'.r, s.w \curlywedge_F s'.w, s.t \curlyvee_F s'.t \rangle$. Consequently, $\bot = \langle \boldsymbol{Pri}, \emptyset, \boldsymbol{Pri} \rangle$. We abbreviate $\Gamma \vdash^j_F M : \langle \bot, \top, \bot \rangle, \tau$ by $\Gamma \vdash M : \tau$ and we write $\boldsymbol{s} \curlyvee \boldsymbol{s'}$ when $\boldsymbol{s} \curlyvee_{\emptyset} \boldsymbol{s'}$.

Our type and effect system enforces compliance of all information flows with the current flow policy. This is achieved by imposing conditions of the kind "$\preceq_F$" in the premises of the rules, and by updating the security effects in the conclusions. Apart from the parameterization of the flow relation with the current flow policy, these are fairly standard in information flow type systems and enforce syntactic rules of the kind "no low writes should depend on high reads", both with respect to the values that are read, and to termination behaviors that might be derived. Notice that the Flow rule types the body of the flow declaration under a more permissive flow policy.

The extra conditions that deal with the migration leaks that are introduced by the allowed construct (see Example 4) deserve further attention: the security level j that is associated to each thread, and represents the confidentiality level of the position of the thread in the network, is used to update the writing effect in the thread creation and migration rules, as well as the termination effect in the allowed-condition rule; on the other hand, it is constrained not to "precede low writes" in rule Allow, and to be a lower bound of runtime threads in rule Thr. We refer the reader to [1] for further explanations on the remaining conditions.

One can prove that the proposed type system ensures non-disclosure, i.e. that it constrains the usage of the new constructs introduced in this language in order to prevent them from encoding information leaks. In fact, security of expressions with respect to Non-disclosure is guaranteed by the type system:

Theorem 1 (Soundness for Non-disclosure for Networks)
Consider a pool of threads P. If for all $M^{m_j} \in P$ there exist Γ, s and τ such that $\Gamma \vdash^j_\emptyset M : s, \tau$, then P satisfies the Non-disclosure for Networks policy.

Notice that our soundness result for non-disclosure is compositional, in the sense that it is enough to verify the typability of each thread separately in order to ensure non-disclosure for the whole network.

4 Confinement Analysis

We now formally define Operational Confinement for Networks, a new security property that specifies the restricted usage of declassification instructions. We present a simple type system for obtaining the declassification effect of programs, which can be used by the semantics of the language to control migration between sites. We prove the soundness of the proposed migration control mechanism.

Operational Confinement. Here we will deal with relations between flow policies, which leads us to define a (meet) pre-semilattice of flow policies. We introduce the *preorder on flow policies* $\preceq$, thus overloading the notation for the flow relations on security levels. The meaning of relating two flow policies as in $F_1 \preceq F_2$ is that F_1 is more permissive than F_2, in the sense that F_1 encodes all the information flows that are enabled by F_2, i.e. $F_1 \preceq F_2$ if and only if $F_2 \subseteq F_1^*$, where the meet operation is given by $F_1 \curlywedge F_2 = F_1 \cup F_2$. Consequently, $\top = \emptyset$.

The property of operational confinement is formulated abstractly for any distributed model whose semantics is decorated with the flow policies that are set up at each step. The decorated semantics conveys the required information for anticipating which flow policies are declared by a program at runtime. The property is set up on pairs that carry information about the location of each thread. The allowed flow policy of the current location of the thread is used to place a restriction on the flow policies that decorate the transitions, step-by-step.

Definition 3 (Operationally Confined Located Threads). *Given a fixed policy-mapping W, a set $\mathcal{C}$ of pairs $\langle d, M^{m_j}\rangle$ is a set of* operationally confined located threads *if the following holds for any $\langle d, M^{m_j}\rangle \in \mathcal{C}$, T and S s.t. $T(m_j) = d$:*

$$W \vdash \langle\{M^{m_j}\}, T, S\rangle \xrightarrow[F]{} \langle\{M'^{m_j}\}, T', S'\rangle \text{ implies } W(T(m_j)) \preceq F \text{ and } \langle T'(m_j), M'^{m_j}\rangle \in \mathcal{C} \text{ and}$$
$$W \vdash \langle\{M^{m_j}\}, T, S\rangle \xrightarrow[F]{} \langle\{M'^{m_j}, N'^{n_k}\}, T', S'\rangle \text{ implies } W(T(m_j)) \preceq F \text{ and}$$
$$\langle T'(m_j), M'^{m_j}\rangle, \langle T'(n_k), N^{n_k}\rangle \in \mathcal{C}$$

We say that a located thread is operationally confined if it belongs to the largest set of operationally confined threads (this set exists, analogously to Definition 1).

Operational confinement means that for every execution step that is performed by a program at a certain site, the declared flow policy always complies to that site's allowed flow policy. The above definition for individual threads leads to a notion of network confinement by obtaining, from a pool of threads P and its corresponding position-tracker T, the set $\text{pair}(P, T)$ defined as $\{\langle d, M^{m_j}\rangle \mid M^{m_j} \in P \text{ and } T(m_j) = d\}$. A network is then said to be operationally confined if all of the pairs of threads and their location are operationally confined.

Type System. We now present a simple type and effect system that constructs a declassification effect that can be used to enforce Confinement. The judgments are now a lighter version of those that were used in Section 3. They have the form $\Gamma \vdash M : s, \tau$, meaning that the expression M is typable with type τ and security effect s in the typing context $\Gamma : \boldsymbol{Var} \to \boldsymbol{Typ}$, which assigns types to variables. Here, s is the *declassification effect*: a lower bound to the flow policies

$$\frac{\Gamma \vdash N : s, \tau}{\Gamma \vdash (\text{flow } F' \text{ in } N) : \boldsymbol{s \cup F'}, \tau} \qquad \frac{\Gamma \vdash N_t : s_t, \tau \quad \Gamma \vdash N_f : s_f, \tau}{\Gamma \vdash (\text{allowed } F' \text{ then } N_t \text{ else } N_f) : \boldsymbol{s_t - F'} \curlywedge s_f, \tau}$$

Fig. 4. Fragment of the type and effect system for the declassification effect

that are declared in the typed expression, excluding those that are positively tested by an allowed-condition. Types are analogous to those of Section 3:

$$\tau, \sigma, \theta \in \boldsymbol{Typ} ::= t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta \text{ ref}_l \mid \tau \xrightarrow{s} \sigma$$

In Fig. 4 we only exhibit the typing rules that are relevant to the construction of the declassification effect. The omitted rules simplify those in Fig. 3, where typing judgments have no parameters, the updates of the security effects as well as all side conditions involving $\preceq_F$ are removed, and the meet operator $\curlywedge$ is used instead of $\curlyvee$ because the declassification effect is contravariant.

The declassification effect of the program is constructed as follows: The new effect is updated in rule Flow each time a flow declaration is performed, and "grows" as the declassification effects of subterms are met (by union) to form that of the parent command. However, in rule Allow, the declassification effect of the "allowed" branch is not used entirely: the part that is tested by the allowed-condition is omitted. The intuition is that the part that is removed (say, F) is of no concern since the allowed branch will only be executed if F is allowed.

As we have mentioned, the declassification effect should give information about the potential flow policy environments that are set up by the program. It is easy to see that the proposed type system provides a rather naive solution to this problem, since it does not take into consideration the migration instructions that appear in the code. This means that it might over-approximate the declassification effect by counting in flow declarations that are not relevant to the site where the program is arriving. Here we are not concerned with the precision of the type system, but rather with putting forward its idea. In spite of the simplicity of the type system, notice that it does incorporate the effort of "protecting" portions of code by means of allowed conditions, when building the declassification effect of a program. This is achieved by discarding information regarding the tested flow policy from the declassification effect of the "allowed" branch. As a result, programs containing flow declarations that are too permissive might still be authorized to execute in a certain domain, as long as they occur in the "allowed" branch of our new construct.

In order to ensure that the declassifications that are enabled by flow declarations never violate the allowed flow policy of the domain where they are performed, we will check that if a program has a declassification effect s, and if it performs an execution step while declaring a flow policy F, then F is stricter than s. A site can then trust that the declassifications performed by an incoming thread are not more permissive than what is declared in the type. Programs whose type cannot guarantee respect for the allowed flow policy of the site can be treated as insecure by that site – in our model, they are simply forbidden to enter. This migration control mechanism allows us to formulate a network level soundness result, guaranteeing that typable programs can roam over the network, never violating the allowed flow policy of the sites where they execute:

Theorem 2 (Soundness of Typing Confinement for Networks). *Consider a fixed policy-mapping W, a pool of threads P and position tracker T, such that for all $M^{m_j} \in P$ there exist Γ, s and τ satisfying $\Gamma \vdash M : s, \tau$ and $W(T(m_j)) \preceq s$. Then* $\text{pair}(P, T)$ *is a set of operationally confined located threads.*

The above result might seem somewhat surprising at first, given that in the typing rule of the allowed construct, the flow policy that is being tested is subtracted from the declassification effect of the allowed branch. Notice however that the allowed branch will only be taken if the tested flow policy is allowed (by G) in the first place. This means that the part of the declassification effect of the allowed branch that is omitted is known to be allowed by G. To illustrate this idea, consider again Example 1, which has an empty declassification effect; its transitions can however be decorated with the flow policy $\{H \prec L\}$ in case the first branch is taken, which in turn can only happen if $\{H \prec L\} \subseteq G^*$.

The proposed method for enforcing operational confinement combines a static analysis technique for obtaining the declassification effect of a program with a migration control technique that is built into the semantics of the language. It does not offer a safety result, guaranteeing that programs never "get stuck", as a thread can be blocked at the point of performing a migration instruction:

$$(\text{allowed } F \text{ then } ((\text{goto } d); (\text{flow } F \text{ in } M_1)) \text{ else } M_2) \tag{5}$$

According to the type system of Fig. 4, the declassification effect of the continuation $((\text{goto } d); (\text{flow } F \text{ in } M_1))$ includes F. This means that the migration instruction will be performed only in the case that the allowed flow policy of d allows for F; otherwise, the program will get stuck. We notice, however, that in order to avoid this situation, the program might have better been written

$$((\text{goto } d); (\text{allowed } F \text{ then } (\text{flow } F \text{ in } M_1) \text{ else } M_2))$$

so that the flow declaration of F would not contribute to weaken the declassification effect of the continuation $(\text{allowed } F \text{ then } (\text{flow } F \text{ in } M_1) \text{ else } M_2)$.

Here we follow the spirit of the proof carrying code scenario: When using the type system to construct a declassification effect, we provide a way to build a certificate that can be analyzed to conclude whether an incoming program should be allowed to execute or not. In case the certificate is not trusted, programs could also be statically checked to be "flow declaration safe".

5 Related Work and Conclusions

We have motivated the need for controlling the usage of declassification in a global computing context by pointing out that programs that were conceived to comply with certain flow policies might not respect those of the computational locations they could end up executing at. We have addressed this issue by studying techniques for ensuring that a thread can only migrate to a site if it complies to its *allowed flow policy*. More concretely, we have proposed a security property we call *confinement to a flow policy*, that ensures programs will respect the allowed flow policies of the domains where they compute. In order to assist the programmer to comply with confinement, we propose a language construct – the *allowed condition* – that tests the flexibility of the allowed flow policy imposed by the domain where it is currently located, and for programming alternative behaviors, should it end up in a site where its declassification operations are not permitted. The introduction of this construct in a language with code mobility must be done with care, since it can give rise to a new form of *migration leaks*, which we showed are not only the result of memory synchronization issues. We show how these migration leaks can be controlled by means of a type system that enforces the non-disclosure policy for networks. As to the enforcement of the confinement property, we propose a new form of security effect, the *declassification effect*, that holds information about relevant declassifying environments that can be established by a program. We show that it can be constructed by means of a type and effect system. This information is useful when setting up a migration control mechanism for controlling execution of programs at each site.

Among other few previous studies of information flow in distributed contexts [7,13,20], the only that considers declassification in a context with code mobility is [1]. The main difference regards the memory model, which in [1] is strictly distributed (i.e. accesses to remote references are restricted), while threads own references that they carry during migration; in that setting, memory synchronization issues also give rise to *migration leaks*. Here we avoid such synchronization issues, which we believe are orthogonal to the ideas presented here, by assuming transparent remote accesses to references. We can then show that migration leaks do not exclusively depend on the memory model. This point motivates a better understanding of migration leaks in global computations.

The literature regarding the subject of declassification is examined in [17]. Most of the overviewed approaches implicitly assume local settings, where the computation platform enforces a centralized policy, while the control of the usage of declassification operations are restricted to the declassifying operations themselves. Here we separate the control of declassification from its encoding, as both the allowed-construct and restricted version of migration are external to the flow declaration construct. Previous works on forms of dynamic flow policy testing consider settings where distribution and mobility are not explicitly dealt with. In [18] and [21], testing is performed over security labels, while the underlying security lattice remains fixed. Closer to ours is the work by Hicks *et al.* [11], where the global flow policy can be updated and tested. However, the language that is considered is local and sequential, and updates to the global flow policy are not meant as declassification operations. Furthermore, the security property does not deal with updates, but rather with what happens in between them. In [6] access control and declassification are combined to ensure that a program can only declassify information that it has the right to read.

The network model we studied in this paper assumes that the allowed flow policy of each domain is fixed. It would be interesting to generalize the model in order to account for dynamic changes in these policies. However, combining this more general scenario with the allowed-condition would lead to inconsistencies, since the policies could potentially change after the branch of the allowed-condition had been chosen. This motivates the study of other alternatives to the allowed-condition for inspecting the *current* allowed flow policy of the context.

Acknowledgments. I thank Gérard Boudol and Tamara Rezk for insightful discussions, as well as the anonymous reviewers who helped shape this paper. This work was partially supported by FCT via KLog PTDC/MAT/68723/2006.

References

1. Almeida Matos, A.: Typing Secure Information Flow: Declassification and Mobility. PhD thesis, École Nationale Supérieure des Mines de Paris (2006)
2. Almeida Matos, A.: Flow policy awareness for distributed mobile code (proofs). Technical report, Instituto Superior Técnico de Lisboa (2008)
3. Almeida Matos, A., Boudol, G.: On declassification and the non-disclosure policy. In: 18th IEEE Computer Security Foundations Workshop, pp. 226–240. IEEE Computer Society, Los Alamitos (2005)
4. Boudol, G.: A generic membrane model. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 208–222. Springer, Heidelberg (2005)
5. Boudol, G., Castellani, I.: Noninterference for concurrent programs and thread systems. Theoretical Computer Science 281(1-2), 109–130 (2002)
6. Boudol, G., Kolundzija, M.: Access Control and Declassification. In: Computer Network Security. CCIS, vol. 1, pp. 85–98. Springer, Heidelberg (2007)
7. Crafa, S., Bugliesi, M., Castagna, G.: Information flow security for boxed ambients. In: Sassone, V. (ed.) Workshop on Foundations of Wide Area Network Computing. ENTCS, vol. 66, pp. 76–97. Elsevier, Amsterdam (2002)
8. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
9. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symp. on Security and Privacy, pp. 11–20. IEEE Computer Society, Los Alamitos (1982)
10. Gorla, D., Hennessy, M., Sassone, V.: Security policies as membranes in systems for global computing. In: Foundations of Global Ubiquitous Computing, FGUC 2004. ENTCS, pp. 23–42. Elsevier, Amsterdam (2005)
11. Hicks, M., Tse, S., Hicks, B., Zdancewic, S.: Dynamic updating of information-flow policies. In: Workshop on Foundations of Comp. Security, pp. 7–18 (2005)
12. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: 15th ACM Symp. on Principles of Programming Languages, pp. 47–57. ACM Press, New York (1988)
13. Mantel, H., Sabelfeld, A.: A unifying approach to the security of distributed and multi-threaded programs. Journal of Computer Security 11(4), 615–676 (2003)
14. Martins, F., Vasconcelos, V.T.: History-based access control for distributed processes. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 98–115. Springer, Heidelberg (2005)
15. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pp. 106–119. ACM, New York (1997)
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., Sands, D.: Declassification: Dimensions and principles. Journal of Computer Security (2007) (to appear)
18. Tse, S., Zdancewic, S.: Run-time principals in information-flow type systems. In: IEEE 2004 Symposium on Security and Privacy, pp. 179–193. IEEE Computer Society Press, Los Alamitos (2004)
19. Zdancewic, S.: Challenges for information-flow security. In: 1st International Workshop on the Programming Language Interference and Dependence (2004)
20. Zdancewic, S., Zheng, L., Nystrom, N., Myers, A.: Secure program partitioning. ACM Transactions on Computer Systems 20(3), 283–328 (2002)
21. Zheng, L., Myers, A.: Dynamic security labels and noninterference. In: Proc. 2nd Workshop on Formal Aspects in Security and Trust, pp. 27–40. Springer, Heidelberg (2004)

Volume and Entropy of Regular Timed Languages: Discretization Approach[⋆]

Eugene Asarin[1] and Aldric Degorre[2]

[1] LIAFA, Université Paris Diderot / CNRS
case 7014, 75205 Paris Cedex 13, France
Eugene.Asarin@liafa.jussieu.fr
[2] VERIMAG,
Centre Equation, 2 av. de Vignate, 38610 Gières, France
Aldric.Degorre@imag.fr

Abstract. For timed languages, we define size measures: volume for languages with words having a fixed finite number of events, and entropy (growth rate) as asymptotic measure for an unbounded number of events. These measures can be used for quantitative comparison of languages, and the entropy can be viewed as the information contents of a timed language. For languages accepted by deterministic timed automata, we give exact formulas for volumes. Next, for a large class of timed languages accepted by non-Zeno timed automata, we devise a method to approximate the volumes and the entropy based on discretization. We give an information-theoretic interpretation of the entropy in terms of Kolmogorov complexity.

1 Introduction

Since early 90s timed automata and timed languages are extensively used for modelling and verification of real-time systems, and thoroughly explored from a theoretical standpoint. However, two important, and closely related, aspects have never been addressed: quantitative analysis of the size of these languages and of the information content of timed words. In this paper, we formalize and partially solve these problems for large subclasses of timed languages.

Recall that a timed word describes a behaviour of a system, taking into account delays between events. For example, $2a3.11b$ means that an event a happened 2 time units after the system start, and b happened 3.11 time units after a. A timed language, which is just a set of timed words, may represent all such potential behaviours. Our aim is to measure the size of such a language. For a fixed number n of events, we can consider the language as a subset of $\Sigma^n \times \mathbb{R}^n$ (that is of several copies of the space $\mathbb{R}^n$). A natural measure in this case is just the Euclidean volume V_n of this subset. When the number of events is not fixed, we can still consider for each n all the timed words with n events belonging to the language and their volume V_n. It turns out that in most cases V_n asymptotically behaves as $2^{n\mathcal{H}}$ for some constant $\mathcal{H}$ that we call the entropy of the language.

[⋆] Support from French ANR project AMAES is gratefully acknowledged.

The information-theoretic meaning of $\mathcal{H}$ can be stated as follows: for a small ε, if the delays are measured with a finite precision ε, then using the words of a language L with entropy $\mathcal{H}$ one can transmit $\mathcal{H} + \log(1/\varepsilon)$ bits of information per event (see Thms. 4-5 below for a formalization in terms of Kolmogorov complexity).

There can be several potential applications of these notions:

– The most direct one is capacity estimation for an information transmission channel or for a time-based information flow.
– When one overapproximates a timed language L_1 by a simpler timed language L_2, e.g. using some abstractions as in [1], it is important to assess the quality of the approximation. Comparing entropies of L_1 and L_2 provides such an assessment.
– In model-checking of timed systems it is often interesting to know the size of the set of all behaviours violating a property or of a subset of those presented as a counter-example by a verification tool.

In this paper, we explore, and partly solve the following problems: given a prefix-closed timed language accepted by a timed automaton, find the volume V_n of the set of accepted words of a given length n and the entropy $\mathcal{H}$ of the whole language.

Two Papers. In fact, we have developed two different and complementary approaches (discretization based and analytical) to the computation of volumes and entropy of timed languages. In this paper, we present for the first time the main definitions, exact formulas for volumes, and a method to approximate the volumes and the entropy using discretization. In [2] we propose methods of computation of the entropy based on functional analysis of positive integral operators. Our preprint [3] available on the Web presents both approaches with more detailed proofs.

Related Work. Our problems and techniques are partly inspired by works concerning the entropy of finite-state languages (cf. [4]). There the cardinality of the set L_n of all elements of length n of a prefix-closed regular language L also behaves as $2^{n\mathcal{H}}$ for some entropy $\mathcal{H}$. This entropy can be found as logarithm of the spectral radius of adjacency matrix of reachable states of $\mathcal{A}$.[1] In this paper, we reduce our problem by discretization to entropy computation for some discrete automata.

In [5,6] probabilities of some timed languages and densities in the clock space are computed. Our formulae for fixed-length volumes can be seen as specialization of these results to uniform measures. As for unbounded languages, they use stringent condition of full simultaneous reset of all the clocks at most every k steps, and under such a condition, they provide a finite stochastic class graph that allows computing various interesting probabilities. We use a much weaker

[1] This holds also for automata with multiplicities, see [4].

hypothesis (every clock to be reset at most every D steps, but these resets need not be simultaneous), and we obtain only the entropy.

In [7] probabilities of LTL properties of one-clock timed automata (over infinite timed words) are computed using Markov chains techniques. It would be interesting to try to adapt our methods to this kind of problems.

Last, our studies of Kolmogorov complexity of rational elements of timed languages, relating this complexity to the entropy of the language, remind of earlier works on complexity of rational approximations of continuous functions [8,9], and those relating complexity of trajectories to the entropy of dynamical systems [9,10].

Paper Organization. This paper is organized as follows. In Sect. 2, we define volumes of fixed-length timed languages and entropy of unbounded-length timed languages, identify some useful subclasses of automata and obtain exact formulas for the volumes of languages of deterministic timed automata. In Sect. 3, we devise a procedure that provides upper and lower bounds of the volumes and the entropy by discretization of the timed automaton. In the next Sect. 4, and using similar techniques, we give an interpretation of the entropy of timed regular languages in terms of Kolmogorov complexity. We conclude the paper by some final remarks in Sect. 5.

2 Problem Statement

2.1 Geometry, Volume and Entropy of Timed Languages

A *timed word* of length n over an alphabet Σ is a sequence $w = t_1 a_1 t_2 \dots t_n a_n$, where $a_i \in \Sigma, t_i \in \mathbb{R}$ and $0 \le t_i$ (notice that this definition rules out timed words ending by a time delay). Here t_i represents the delay between the events a_{i-1} and a_i. With such a timed word w of length n we associate its *untiming* $\eta(w) = a_1, \dots, a_n \in \Sigma^n$ (which is just a word), and its *timing* which is a point $\theta(w) = (t_1, \dots, t_n)$ in $\mathbb{R}^n$. A *timed language* L is a set of timed words. For a fixed n, we define the *n-volume* of L as follows:

$$V_n(L) = \sum_{v \in \Sigma^n} \mathrm{Vol}\{\theta(w) \mid w \in L, \eta(w) = v\},$$

where Vol stands for the standard Euclidean volume in $\mathbb{R}^n$. In other words, we sum up over all the possible untimings v of length n the volumes of the corresponding sets of delays in $\mathbb{R}^n$. In case of regular timed languages, these sets are polyhedral, and hence their volumes (finite or infinite) are well-defined.

We associate with every timed language a sequence of n-volumes V_n. We will show in Sect. 2.4 that, for languages of deterministic timed automata, V_n is a computable sequence of rational numbers. However, we would like to find a unique real number characterizing the asymptotic behaviour of V_n as $n \to \infty$. Typically, V_n depends approximately exponentially on n. We define the entropy of a language as the rate of this dependence.

Formally, for a timed language L we define its *entropy* as follows:

$$\mathcal{H}(L) = \limsup_{n\to\infty} \frac{\log_2 V_n}{n},$$

where lim sup stands for upper limit[2].

Remark 1. Many authors consider a slightly different kind of timed words: sequences $w = (a_1, d_1), \ldots, (a_n, d_n)$, where $a_i \in \Sigma, d_i \in \mathbb{R}$ and $0 \le d_1 \le \cdots \le d_n$, with d_i representing the date of the event a_i. This definition is in fact isomorphic to ours by a change of variables: $t_1 = d_1$ and $t_i = d_i - d_{i-1}$ for $i = 2..n$. It is important for us that this change of variables preserves the n-volume, since it is linear and its matrix has determinant 1. Therefore, choosing date (d_i) or delay (t_i) representation has no influence on language volumes (and entropy). Due to the authors' preferences (justified in [11]), delays will be used in the sequel.

2.2 Three Examples

To illustrate the problem of determining volume and entropy, consider the languages recognized by three timed automata [12] of Fig. 1. The third one resists naive analysis, and will be used to illustrate the discretization method at the end of Sect. 3.

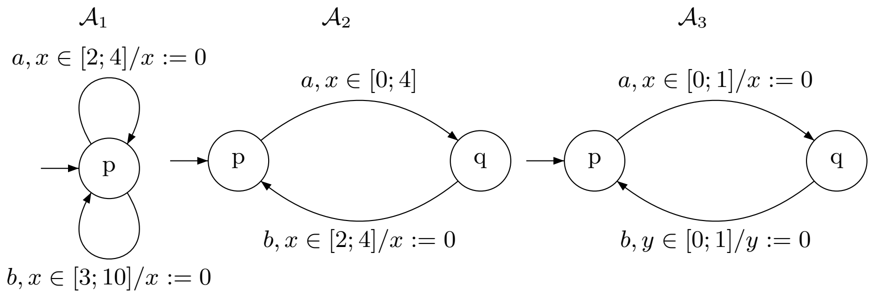


Fig. 1. Three simple timed automata $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$

Rectangles. Consider the timed language defined by the expression

$$L_1 = ([2;4]a + [3;10]b)^*,$$

recognized by $\mathcal{A}_1$ of Fig. 1.

For a given untiming $w \in \{a,b\}^n$ containing k letters a and $n-k$ letters b, the set of possible timings is a rectangle in $\mathbb{R}^n$ of a volume $2^k 7^{n-k}$ (notice that there are C_n^k such untimings). Summing up all the volumes, we obtain

$$V_n(L_1) = \sum_{k=0}^{n} C_n^k 2^k 7^{n-k} = (2+7)^n = 9^n,$$

and the entropy $\mathcal{H}(L_1) = \log 9 \approx 3.17$.

[2] In fact, due to Assumption A2 below, the languages we consider in the paper are prefix-closed, and lim sup is in fact a lim.

A Product of Trapezia. Consider the language defined by the automaton $\mathcal{A}_2$ on Fig. 1, that is containing words of the form $t_1 a s_1 b t_2 a s_2 b \dots t_k a s_k b$ such that $2 \le t_i + s_i \le 4$. Since we want prefix-closed languages, the last $s_k b$ can be omitted.

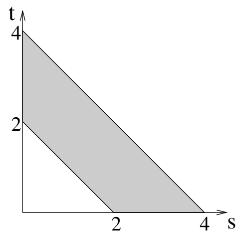


Fig. 2. Timings (t_i, s_i) for $\mathcal{A}_2$

For an even $n = 2k$ the only possible untiming is $(ab)^k$. The set of timings in $\mathbb{R}^{2k}$ is a Cartesian product of k trapezia $2 \le t_i + s_i \le 4$. The surface of each trapezium equals $S = 4^2/2 - 2^2/2 = 6$, and the volume $V_{2k}(L_2) = 6^k$. For an odd $n = 2k+1$ the same product of trapezia is combined with an interval $0 \le t_{k+1} \le 4$, hence the volume is $V_{2k+1}(L_2) = 6^k \cdot 4$. Thus the entropy $\mathcal{H}(L_2) = \log 6/2 \approx 1.29$.

Our Favourite Example. The language recognized by the automaton $\mathcal{A}_3$ on Fig. 1 contains the words of the form $t_1 a t_2 b t_3 a t_4 b \dots$ with $t_i + t_{i+1} \in [0; 1]$. Notice that the automaton has two clocks that are never reset together. The geometric form of possible untimings in $\mathbb{R}^n$ is defined by overlapping constraints $t_i + t_{i+1} \in [0; 1]$.

It is not so evident how to compute the volume of this polyhedron. A systematic method is described below in Sect. 2.4. An *ad hoc* solution would be to integrate 1 over the polyhedron, and to rewrite this multiple integral as an iterated one. The resulting formula for the volume is

$$V_n(L_3) = \int_0^1 dt_1 \int_0^{1-t_1} dt_2 \int_0^{1-t_2} dt_3 \dots \int_0^{1-t_{n-1}} dt_n.$$

This gives the sequence of volumes:

$$1; \frac{1}{2}; \frac{1}{3}; \frac{5}{24}; \frac{2}{15}; \frac{61}{720}; \frac{17}{315}; \frac{277}{8064}; \dots$$

In the sequel, we will also compute the entropy of this language.

2.3 Subclasses of Timed Automata

In the rest of the paper, we compute volumes and entropy for regular timed languages recognized by some subclasses of timed automata (TA). We assume that the reader is acquainted with timed automata; otherwise, we refer her or him to [12] for details. Here we only fix notations for components of timed automata and state several requirements they should satisfy. Thus a TA is a tuple $\mathcal{A} = (Q, \Sigma, C, \Delta, q_0)$. Its elements are respectively the set of locations, the alphabet, the set of clocks, the transition relation, and the initial location (we do not need to specify accepting states due to A2 below, neither we need any invariants). A generic state of $\mathcal{A}$ is a pair $(q, \mathbf{x})$ of a control location and a vector of clock values. A generic element of Δ is written as $\delta = (q, a, \mathfrak{g}, \mathfrak{r}, q')$ meaning a transition from q to q' with label a, guard $\mathfrak{g}$ and reset $\mathfrak{r}$. We spare the reader the definitions of a run of $\mathcal{A}$ and of its accepted language.

Several combinations of the following Assumptions will be used in the sequel:

A1. The automaton $\mathcal{A}$ is deterministic[3].
A2. All its states are accepting.
A3. Guards are rectangular (i.e. conjunctions of constraints $L_i \leq x_i \leq U_i$, strict inequalities are also allowed). Every guard upper bounds at least one clock.
A4. There exists a $D \in \mathbb{N}$ such that on every run segment of D transitions, every clock is reset at least once.
A5. There is no punctual guards, that is in any guard $L_i < U_i$.

Below we motivate and justify these choices:

A1: Most of known techniques to compute entropy of untimed regular languages work on deterministic automata. Indeed, these techniques count paths in the automaton, and only in the deterministic case their number coincides with the number of accepted words. The same is true for volumes in timed automata. Unfortunately, for timed automata, determinization is not always possible, and A1 is restrictive.
A2: Prefix-closed languages are natural in the entropy context, and somewhat easier to study. These languages constitute the natural model for the set of behaviours of causal systems.
A3: If a guard of a feasible transition is infinite, the volume becomes infinite. We conclude that A3 is unavoidable and almost not restrictive.
A4: We use this variant of non-Zenoness condition several times in our proofs and constructions. As the automaton of Fig. 3 shows, if we omit this assumption some anomalies can occur.

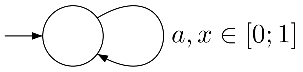


Fig. 3. An automaton without resets

The language of this automaton is

$$L = \{t_1 a \dots t_n a \mid 0 \leq \sum t_i \leq 1\},$$

and V_n is the volume of an n-dimensional simplex defined by the constraints $0 \leq \sum t_i \leq 1$, and $0 \leq t_i$. Hence $V_n = 1/n!$ which decreases faster than any exponent, which is too fine to be distinguished by our methods. Assumption A4 rules out such anomalies.

This assumption is also the most difficult to check. A possible way would be to explore all simple cycles in the region graph and to check that all of those reset every clock.

A5: While assumptions A1-A4 can be restrictive, we always can remove the transitions with punctual guards from any automaton, without changing the volumes V_n. Hence, A5 is not restrictive at all, as far as volumes are considered. In Sect. 4 we do not make this assumption.

[3] That is any two transitions with the same source and the same label have their guards disjoint.

2.4 Computing Volumes

Given a timed automaton $\mathcal{A}$ satisfying A1-A3, we want to compute n-volumes V_n of its language. In order to obtain recurrent equations on these volumes, we need to take into account all possible initial locations and clock configurations. For every state $(q, \mathbf{x})$, let $L(q, \mathbf{x})$ be the set of all the timed words corresponding to the runs of the automaton starting at this state, let $L_n(q, \mathbf{x})$ be its sublanguage consisting of its words of length n, and $v_n(q, \mathbf{x})$ the volume of this sublanguage. Hence, the quantity we are interested in, is a value of v_n in the initial state:

$$V_n = v_n(q_0, 0).$$

By definition of runs of a timed automaton, we obtain the following language equations:

$$\begin{aligned} L_0(q,\mathbf{x}) &= \varepsilon; \\ L_{k+1}(q,\mathbf{x}) &= \bigcup_{(q,a,\mathfrak{g},\mathfrak{r},q')\in\Delta} \bigcup_{\tau:\mathbf{x}+\tau\in\mathfrak{g}} \tau a L_k(q', \mathfrak{r}(\mathbf{x}+\tau)). \end{aligned}$$

Since the automaton is deterministic, the union over transitions (the first $\bigcup$ in the formula) is disjoint. Hence, it is easy to pass to volumes:

$$v_0(q,\mathbf{x}) = 1; \tag{1}$$

$$v_{k+1}(q,\mathbf{x}) = \sum_{(q,a,\mathfrak{g},\mathfrak{r},q')\in\Delta} \int_{\tau:\mathbf{x}+\tau\in\mathfrak{g}} v_k(q', \mathfrak{r}(\mathbf{x}+\tau))\, d\tau. \tag{2}$$

Remark that for a fixed location q, and within every clock region, as defined in [12], the integral over $\tau : \mathbf{x}+\tau \in \mathfrak{g}$ can be decomposed into several $\int_l^u$ with bounds l and u either constants or of the form $c - x_i$ with c an integer and x_i a clock variable.

These formulas lead to the following structural description of $v_n(q, \mathbf{x})$, which can be proved by a straightforward induction.

Lemma 1. *The function $v_n(q, \mathbf{x})$ restricted to a location q and a clock region can be expressed by a polynomial of degree n with rational coefficients in variables $\mathbf{x}$.*

Thus in order to compute the volume V_n one should find by symbolic integration polynomial functions $v_k(q, \mathbf{x})$ for $k = 0..n$, and finally compute $v_n(q_0, 0)$.

Theorem 1. *For a timed automaton $\mathcal{A}$ satisfying A1-A3, the volume V_n is a rational number, computable from $\mathcal{A}$ and n using the procedure described above.*

3 Discretization Approach

In this section, we use an approach to volume/entropy computation based on discretization of timed languages and timed automata. This discretization is strongly inspired by [13,14,15].

3.1 ε-Words and ε-Balls

We start with a couple of preliminary definitions. Take an $\varepsilon = 1/N > 0$. A timed word w is ε-timed if all the delays in this word are multiples of ε. Any ε-timed word w over an alphabet Σ can be written as $w = h_\varepsilon(v)$ for an untimed $v \in \Sigma \cup \{\tau\}$, where the morphism h_ε is defined as follows:

$$h_\varepsilon(a) = a \text{ for } a \in \Sigma, \qquad h_\varepsilon(\tau) = \varepsilon.$$

The discrete word v with ticks τ (standing for ε delays) represents in this way the ε-timed word w.

Example. Let $\varepsilon = 1/5$, then the timed word $0.6a0.4ba0.2a$ is ε-timed. Its representation is $\tau\tau\tau a\tau\tau ba\tau a$.

The notions of ε-timed words and their representation can be ported straightforwardly to languages.

For a timed word $w = t_1a_1t_2a_2\dots t_na_n$ we introduce its North-East ε-neighbourhood like this:

$$\mathcal{B}_\varepsilon^{NE}(w) = \{s_1a_1s_2a_2\dots s_na_n \mid \forall i\,(s_i \in [t_i; t_i + \varepsilon])\}\,.$$

For a language L, we define its NE-neighbourhood elementwise:

$$\mathcal{B}_\varepsilon^{NE}(L) = \bigcup_{w\in L} \mathcal{B}_\varepsilon^{NE}(w). \tag{3}$$

The next simple lemma will play a key role in our algorithm (here $\#L$ stands for the cardinality of L).

Lemma 2. *Let L be some finite set of timed words of length n. Then*

$$\mathrm{Vol}(\mathcal{B}_\varepsilon^{NE}(L)) \le \varepsilon^n \#L.$$

If, moreover, L is ε-timed, then

$$\mathrm{Vol}(\mathcal{B}_\varepsilon^{NE}(L)) = \varepsilon^n \#L.$$

Proof. Notice that for a timed word w of a length n the set $\mathcal{B}_\varepsilon^{NE}(w)$ is a hypercube of edge ε (in the delay space), and of volume ε^n. Notice also that neighbourhoods of different ε-timed words are almost disjoint: the interior of their intersections are empty. With these two remarks, the two statements are immediate from (3). □

3.2 Discretizing Timed Languages and Automata

Suppose now that we have a timed language L recognized by a timed automaton $\mathcal{A}$ satisfying A2-A5 and we want to compute its entropy (or just the volumes V_n).

Take an $\varepsilon = 1/N > 0$. We will build two ε-timed languages L_- and L_+ that under- and over-approximate L in the following sense:

$$\mathcal{B}_\varepsilon^{NE}(L_-) \subset L \subset \mathcal{B}_\varepsilon^{NE}(L_+). \tag{4}$$

The recipe is like this. Take the timed automaton $\mathcal{A}$ accepting L. Discrete automata A_+^ε and A_-^ε can be constructed in two stages. First, we build counter automata C_+^ε and C_-^ε. They have the same states as $\mathcal{A}$, but instead of every clock x they have a counter c_x (roughly representing x/ε). For every state add a self-loop labelled by τ and incrementing all the counters. Replace any reset of x by a reset of c_x. Whenever $\mathcal{A}$ has a guard $x \in [l; u]$ (or $x \in (l; u)$, or some other interval), the counter automaton C_+^ε has a guard $c_x \in [l/\varepsilon \dot{-} D; u/\varepsilon - 1]$ (always the closed interval) instead, where D is as in assumption A4. At the same time C_-^ε has a guard $c_x \in [l/\varepsilon; u/\varepsilon - D]$. Automata C_+^ε and C_-^ε with bounded counters can be easily transformed into finite-state ones A_+^ε and A_-^ε .

Lemma 3. *Languages $L_+ = h_\varepsilon(L(A_+^\varepsilon))$ and $L_- = h_\varepsilon(L(A_-^\varepsilon))$ have the required property (4).*

Proof (sketch).

Inclusion $\mathcal{B}_\varepsilon^{NE}(L_-) \subset L$. Let a discrete word $u \in L(A_-^\varepsilon)$, let $v = h_\varepsilon(u)$ be its ε-timed version, and let $w \in \mathcal{B}_\varepsilon^{NE}(v)$. We have to prove that $w \in L$. Notice first that $L(A_-^\varepsilon) = L(C_-^\varepsilon)$ and hence u is accepted by C_-^ε. Mimic the run of C_-^ε on u, but replace every τ by an ε duration, thus, a run of $\mathcal{A}$ on v can be obtained. Moreover, in this run every guard $x \in [l, u]$ is respected with a security margin: in fact, a stronger guard $x \in [l, u - D\varepsilon]$ is respected. Now one can mimic the same run of $\mathcal{A}$ on w. By definition of the neighbourhood, for any delay t_i in u the corresponding delay t_i' in w belongs to $[t_i, t_i + \varepsilon]$. Clock values are always sums of several (up to D) consecutive delays. Whenever a narrow guard $x \in [l, u - D\varepsilon]$ is respected on v, its "normal" version $x' \in [l, u]$ is respected on w. Hence, the run of $\mathcal{A}$ on w obtained in this way respects all the guards, and thus $\mathcal{A}$ accepts w. We deduce that $w \in L$. □

Inclusion $L \subset \mathcal{B}_\varepsilon^{NE}(L_+)$. First, we define an approximation function on $\mathbb{R}_+$ as follows:

$$\underline{t} = \begin{cases} 0 & \text{if} \quad t = 0 \\ t - \varepsilon & \text{if} \quad t/\varepsilon \in \mathbb{N}_+ \\ \varepsilon \lfloor t/\varepsilon \rfloor & \quad \text{otherwise.} \end{cases}$$

Clearly, $\underline{t}$ is always a multiple of ε and belongs to $[t - \varepsilon, t)$ with the only exception that $\underline{0} = 0$.

Now we can proceed with the proof. Let $w = t_1 a_1 \dots t_n a_n \in L$. We define its ε-timed approximation v by approximating all the delays: $v = \underline{t_1} a_1 \dots \underline{t_n} a_n$. By construction $w \in \mathcal{B}_\varepsilon^{NE}(v)$. The run of $\mathcal{A}$ on w respects all the guards $x \in [l; u]$. Notice that the clock value of x on this run is a sum of several (up to D) consecutive t_i. If we try to run $\mathcal{A}$ over the approximating word v, the value x' of the same clock at the same transition would be a multiple of ε and it would belong to $[x - D\varepsilon; x)$. Hence $x' \in [l \dot{-} D\varepsilon, u - \varepsilon]$.

By definition of C_+ this means that the word $u = h_\varepsilon^{-1}(v)$ is accepted by this counter automaton. Hence $v \in L_+$.

Let us summarize: for any $w \in L$ we have constructed $v \in L_+$ such that $w \in \mathcal{B}_\varepsilon^{NE}(v)$. This concludes the proof. □

3.3 Counting Discrete Words

Once the automata A_+^ε and A_-^ε constructed, we can count the number of words with n events and its asymptotic behaviour using the following simple result.

Lemma 4. *Given an automaton $\mathcal{B}$ over an alphabet $\{\tau\} \cup \Sigma$, let*

$$L_n = L(\mathcal{B}) \cap (\tau^* \Sigma)^n .$$

Then (1) $\#L_n$ is computable; and (2) $\lim_{n\to\infty}(\log \#L_n/n) = \log \rho_\mathcal{B}$ with $\rho_\mathcal{B}$ a computable algebraic real number.

Proof. We proceed in three stages. First, we determinize $\mathcal{B}$ and remove all the useless states (unreachable from the initial state). These transformations yield an automaton $\mathcal{D}$ accepting the same language, and hence having the same cardinalities $\#L_n$. Since the automaton is deterministic, to every word in L_n corresponds a unique accepting path with n events from Σ and terminating with such an event.

Next, we eliminate the tick transitions τ. As we are counting paths, we obtain an automaton without silent (τ) transitions, but with multiplicities representing the number of realizations of every transition. More precisely, the procedure is as follows. Let $\mathcal{D} = (Q, \{\tau\} \cup \Sigma, \delta, q_0)$. We build an automaton with multiplicities $\mathcal{E} = (Q, \{e\}, \Delta, q_0)$ over one-letter alphabet. For every $p, q \in Q$ the multiplicity of the transition $p \to q$ in $\mathcal{E}$ equals the number of paths from p to q in $\mathcal{D}$ over words from $\tau^*\Sigma$. A straightforward induction over n shows that the number of paths in $\mathcal{D}$ with n non-tick events equals the number of n-step paths in $\mathcal{E}$ (with multiplicities).

Let M be the adjacency matrix with multiplicities of $\mathcal{E}$. It is well known (and easy to see) that the $\#L(n)$ (that is the number of n-paths) can be found as the sum of the first line of the matrix M_-^n. This allows computing $\#L(n)$. Moreover, using Perron-Frobenius theorem we obtain that $\#L(n) \sim \rho^n$ where ρ is the spectral radius of M, the greatest (in absolute value) real root λ of the integer characteristic polynomial $\det(M - \lambda I)$. □

3.4 From Discretizations to Volumes

As soon as we know how to compute the cardinalities of under- and over- approximating languages $\#L_-(n)$ and $\#L_+(n)$ and their growth rates ρ_- and ρ_+, we can deduce the following estimates solving our problems.

Theorem 2. *For a timed automaton $\mathcal{A}$ satisfying A2-A5, the n-volumes of its language satisfy the estimates:*

$$\#L_-(n) \cdot \varepsilon^n \le V_n \le \#L_+(n) \cdot \varepsilon^n .$$

Proof. In inclusions (4) take the volumes of the three terms, and use Lemma 2. □

Theorem 3. *For a timed automaton $\mathcal{A}$ satisfying A2-A5, the entropy of its language satisfies the estimates:*

$$\log(\varepsilon\rho_-) \le \mathcal{H}(L(\mathcal{A})) \le \log(\varepsilon\rho_+).$$

Proof. Just use the previous result, take the logarithm, divide by n and pass to the limit. □

We summarize the algorithm in Table 1.

Table 1. Discretization algorithm: bounding $\mathcal{H}$

1. Choose an $\varepsilon = 1/N$.
2. Build the counter automata C_-^ε and C_+^ε.
3. Transform them into finite automata A_-^ε and A_+^ε.
4. Eliminate τ transitions introducing multiplicities.
5. Obtain adjacency matrices M_- and M_+.
6. Compute their spectral radii ρ_- and ρ_+.
7. Conclude that $\mathcal{H} \in [\log \varepsilon\rho_-; \log \varepsilon\rho_+]$.

This theorem can be used to estimate the entropy. However, it can also be read in a converse direction: the cardinality of L restricted to n events and discretized with quantum ε is close to $2^{\mathcal{H}n}/\varepsilon^n$. Hence, we can encode $\mathcal{H} - \log\varepsilon$ bits of information per event. These information-theoretic considerations are made more explicit in Sect. 4 below.

A Case Study. Consider the example $L_3 = \{t_1at_2bt_3at_4b\cdots \mid t_i+t_{i+1} \in [0;1]\}$ from Sect. 2.2. We need two clocks to recognize this language, and they are never reset together. We choose $\varepsilon = 0.05$ and build the automata on Fig. 4 according to the recipe (the discrete ones A_+ and A_- are too big to fit on the figure).

We transform $C_-^{0.05}$ and $C_+^{0.05}$, into A_+ and A_-, eliminate silent transitions and unreachable states, and compute spectral radii of adjacency matrices (their

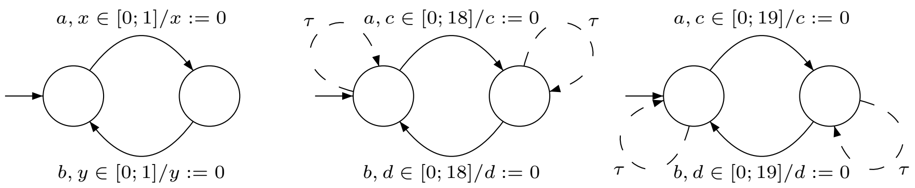


Fig. 4. A two-clock timed automaton $\mathcal{A}_3$ and its approximations $C_-^{0.05}$ and $C_+^{0.05}$. All τ-transitions increment counters c and d.

sizes are 38x38 and 40x40): $\#L_-^{0.05}(n) \sim 12.41^n$, $\#L_+^{0.05}(n) \sim 13.05^n$. Hence $12.41^n \cdot 0.05^n \leq V_n \leq 13.05^n \cdot 0.05^n$, and the entropy

$$\mathcal{H} \in [\log 0.62; \log 0.653] \subset (-0.69; -0.61).$$

Taking a smaller $\varepsilon = 0.01$ provides a better estimate for the entropy:

$$\mathcal{H} \in [\log 0.6334; \log 0.63981] \subset (-0.659; -0.644).$$

We prove in [2,3] that the true value of the entropy is $\mathcal{H} = \log(2/\pi) \approx \log 0.6366 \approx -0.6515$.

4 Kolmogorov Complexity of Timed Words

To interpret the results above in terms of information content of timed words we state, using similar techniques, some estimates of Kolmogorov complexity of timed words. Recall first the basic definition from [16], see also [17]. Given a partial computable function (decoding method) $f : \{0;1\}^* \times B \to A$, a description of an element $x \in A$ knowing $y \in B$ is a word w such that $f(w, y) = x$. The Kolmogorov complexity of x knowing y, denoted $K_f(x|y)$ is the length of the shortest description. According to Kolmogorov-Solomonoff theorem, there exists the best (universal) decoding method providing shorter descriptions (up to an additive constant) than any other method. The complexity $K(x|y)$ with respect to this universal method represents the quantity of information in x knowing y.

Coming back to timed words and languages, remark that a timed word within a "simple" timed language can involve rational delays of a very high complexity, or even uncomputable real delays. For this reason we consider timed words with finite precision ε. For a timed word w and $\varepsilon = 1/N$ we say that a timed word v is a rational ε-approximation of w if all delays in v are rational and $w \in \mathcal{B}_\varepsilon^{NE}(v)$[4].

Theorem 4. *Let $\mathcal{A}$ be a timed automaton satisfying A2-A4, L its language, $\mathcal{H}$ its entropy. For any rational $\alpha, \varepsilon > 0$, and any $n \in \mathbb{N}$ large enough there exists a timed word $w \in L$ of length n such that the Kolmogorov complexity of all the rational ε-approximations v of the word w is lower bounded as follows*

$$K(v|n, \varepsilon) \geq n(\mathcal{H} + \log 1/\varepsilon - \alpha). \tag{5}$$

Proof. By definition of the entropy, for n large enough

$$V_n > 2^{n(\mathcal{H}-\alpha)}.$$

Consider the set S of all timed words v violating the lower bound (5)

$$S = \{v \mid K(v|n, \varepsilon) \leq n(\mathcal{H} + \log(1/\varepsilon) - \alpha)\}.$$

[4] In this section, we use such South-West approximations v for technical simplicity only.

The cardinality of S can be bounded as follows:

$$\#S \leq 2^{n(\mathcal{H}+\log(1/\varepsilon)-\alpha)} = 2^{n(\mathcal{H}-\alpha)}/\varepsilon^n.$$

Applying Lemma 2 we obtain

$$\mathrm{Vol}(\mathcal{B}_\varepsilon^{NE}(S)) \leq \varepsilon^n \#S \leq 2^{n(\mathcal{H}-\alpha)} < V_n.$$

We deduce that the set L_n of timed words from L of length n cannot be included into $\mathcal{B}_\varepsilon^{NE}(S)$. Thus, there exists a word $w \in L_n \setminus \mathcal{B}_\varepsilon^{NE}(S)$. By construction, it cannot be approximated by any low-complexity word with precision ε. □

Theorem 5. *Let $\mathcal{A}$ be a timed automaton satisfying A2-A4, L its language, $\alpha > 0$ a rational number. Consider a "bloated" automaton $\mathcal{A}'$ which is like $\mathcal{A}$, but in all the guards each constraint $x \in [l, u]$ is replaced by $x \in [l \dot{-} \alpha, u+\alpha]$. Let $\mathcal{H}'$ be the entropy of its language. Then the following holds for any $\varepsilon = 1/N \in (0; \alpha/D)$, and any n large enough.*

For any timed word $w \in L$ of length n there exists its ε-approximation v with Kolmogorov complexity upper bounded as follows:

$$K(v|n, \varepsilon) \leq n(\mathcal{H}' + \log 1/\varepsilon + \alpha).$$

Proof. Denote the language of $\mathcal{A}'$ by L', the set of words of length n in this language by L'_n and its n-volume by V'_n. We remark that for n large enough

$$V'_n < 2^{n(\mathcal{H}'+\alpha/2)}.$$

Let now $w = t_1 a_1 \dots t_n a_n$ in L_n. We construct its rational ε-approximation as in Lemma 3: $v = \underline{t_1} a_1 \dots \underline{t_n} a_n$. To find an upper bound for the complexity of v we notice that $v \in U$, where U is the set of all ε-timed words u of n letters such that $\mathcal{B}_\varepsilon^{NE}(u) \subset L'_n$. Applying Lemma 2 to the set U we obtain the bound

$$\#U \leq V'_n/\varepsilon^n < 2^{n(\mathcal{H}'+\alpha/2)}/\varepsilon^n.$$

Hence, in order to encode v (knowing n and ε) it suffices to give its number in a lexicographical order of U, and

$$K(v|n, \varepsilon) \leq \log \#U + c \leq n(\mathcal{H}' + \log 1/\varepsilon + \alpha/2) + c \leq n(\mathcal{H}' + \log 1/\varepsilon + \alpha)$$

for n large enough. □

Two theorems above provide close upper and lower bounds for complexity of ε-approximations of elements of a timed language.

However, the following example shows that because we removed Assumption A5, in some cases these bounds do not match and $\mathcal{H}'$ can possibly not converge towards $\mathcal{H}$ when α becomes small.

Example 1. Consider the automaton of Fig. 5. For this example, the state q does not contribute to the volume, and $\mathcal{H} = \log 3$. But whenever we bloat the guards, both states become "usable" and, for the bloated automaton $\mathcal{H}' \approx \log 5$. As for Kolmogorov complexity, for ε-approximations of words from the sublanguage $1b([0; 5]a)^*$ it behaves as $n(\log 5 + \log(1/\varepsilon))$. Thus, for this bothering example, the complexity matches $\mathcal{H}'$ rather than $\mathcal{H}$.

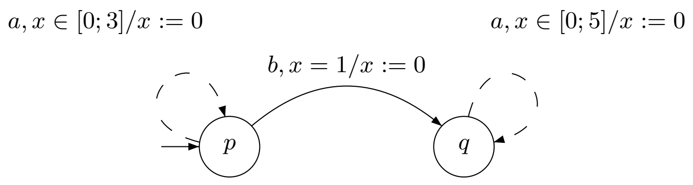


Fig. 5. A pathological automaton

5 Conclusions and Further Work

In this paper, we have defined size characteristics of timed languages: volume and entropy, and suggested a procedure to compute their approximations. Research in this direction is very recent, and many questions need to be studied. We are planning to explore practical feasibility of the procedure described here and compare to the techniques from [2]. We will explore potential applications mentioned in the introduction.

Many theoretical questions still require exploration. It would be interesting to estimate the gap between our upper and lower bounds for the entropy (we believe that this gap tends to 0 for strongly connected automata) and establish entropy computability. We would be happy to remove some of Assumptions A1-A5, in particular non-Zenoness. Kolmogorov complexity estimates can be improved, in particular, as shows Example 1, it could be more suitable to use another variant of entropy, perhaps $\mathcal{H}^+ = \max \mathcal{H}_q$, where the entropy is maximized with respect to initial states q. Extending results to probabilistic timed automata is another option. Our entropy represents the amount of information per timed event. It would be interesting to find the amount of information per time unit. Another research direction is to associate a dynamical system (a subshift) to a timed language and to explore entropy of this dynamical system.

Acknowledgment. The authors are thankful to Oded Maler for motivating discussions and valuable comments on the manuscript.

References

1. Ben Salah, R., Bozga, M., Maler, O.: On timed components and their abstraction. In: SAVCBS 2007, pp. 63–71. ACM, New York (2007)
2. Asarin, E., Degorre, A.: Volume and entropy of regular timed languages: Analytic approach. In: FORMATS 2009. LNCS. Springer, Heidelberg (to appear, 2009)
3. Asarin, E., Degorre, A.: Volume and entropy of regular timed languages. Preprint (2009), `http://hal.archives-ouvertes.fr/hal-00369812/`
4. Lind, D., Marcus, B.: An introduction to symbolic dynamics and coding. Cambridge University Press, Cambridge (1995)
5. Bucci, G., Piovosi, R., Sassoli, L., Vicario, E.: Introducing probability within state class analysis of dense-time-dependent systems. In: QEST 2005, pp. 13–22. IEEE Computer Society, Los Alamitos (2005)

6. Sassoli, L., Vicario, E.: Close form derivation of state-density functions over dbm domains in the analysis of non-Markovian models. In: QEST 2007, pp. 59–68. IEEE Computer Society, Los Alamitos (2007)
7. Bertrand, N., Bouyer, P., Brihaye, T., Markey, N.: Quantitative model-checking of one-clock timed automata under probabilistic semantics. In: QEST 2008, pp. 55–64. IEEE Computer Society, Los Alamitos (2008)
8. Asarin, E., Pokrovskii, A.: Use of the Kolmogorov complexity in analyzing control system dynamics. Automation and Remote Control (1), 25–33 (1986)
9. Rojas, C.: Computability and information in models of randomness and chaos. Mathematical Structures in Computer Science 18(2), 291–307 (2008)
10. Brudno, A.: Entropy and the complexity of the trajectories of a dynamical system. Trans. Moscow Math. Soc. 44, 127–151 (1983)
11. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. Journal of the ACM 49, 172–206 (2002)
12. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
13. Asarin, E., Maler, O., Pnueli, A.: On discretization of delays in timed automata and digital circuits. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 470–484. Springer, Heidelberg (1998)
14. Henzinger, T.A., Manna, Z., Pnueli, A.: What good are digital clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
15. Göllü, A., Puri, A., Varaiya, P.: Discretization of timed automata. In: CDC 1994, vol. 1, pp. 957–958 (1994)
16. Kolmogorov, A.: Three approaches to the quantitative definition of information. Problems of Information Transmission 1(1), 1–7 (1965)
17. Li, M., Vitányi, P.: An introduction to Kolmogorov complexity and its applications, 3rd edn. Springer, Heidelberg (2008)

A Logical Interpretation of the λ-Calculus into the π-Calculus, Preserving Spine Reduction and Types

Steffen van Bakel and Maria Grazia Vigliotti

Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK
svb@doc.ic.ac.uk, mgv98@doc.ic.ac.uk

Abstract. We define a new, output-based encoding of the λ-calculus into the asynchronous π-calculus – enriched with pairing – that has its origin in mathematical logic, and show that this encoding respects one-step spine-reduction up to substitution, and that normal substitution is respected up to similarity. We will also show that it fully encodes lazy reduction of closed terms, in that term-substitution as well as each reduction step are modelled up to similarity. We then define a notion of type assignment for the π-calculus that uses the type constructor $\rightarrow$, and show that all Curry-assignable types are preserved by the encoding.

Introduction

In this paper we present a new encoding of terms of Church's λ-calculus [7,11] into Milner's π-calculus [18] that respects reduction, and define a new notion of type assignment for π so that processes will become witnesses for the provable formulae.

Sangiorgi [21] states good reasons for obtaining an expressive encoding:

– From the process calculi point of view, to gain deeper insight into its theory.
– From the λ-calculus point of view, to provide the opportunity to study λ-terms in different contexts than the sequential one.
– The λ-calculus is a model for functional language programming; these languages have never been very efficient, and one way of improving efficiency is to use parallel implementation.

So therefore, it is important to understand which relation is held between the two paradigms. Research in the direction of encodings of λ-terms was started by Milner in [18]; he defined an input-based encoding, and showed that the interpretation of closed λ-terms respects *lazy* reduction up to substitution. Milner also defined another input-based encoding that respects *call-by-value* reduction up to substitution, but the latter had fewer followers. An input-based interpretation of the λ-calculus into the π-calculus has also been studied by Sangiorgi [23], but in the context of the higher-order π-calculus, by Honda *et al.* [17] with a rich type theory, and by Thielecke [25] in the context of continuation passing style programming languages.

For many years, it seemed that the first and final word on the encoding of the λ-calculus has been said by Milner; in fact, Milner's encoding has set a milestone in the comparison of the two paradigms, and all the above mentioned systems present variants of Milner's encoding. Sangiorgi [20] says:

“ *It seems established that* [Milner’s encoding] *is canonical, i.e. it is the ‘best’ or ‘simplest’ encoding of the lazy λ-calculus into π-calculus. Nevertheless, one has to think carefully about it – in particular at the encoding of application – to understand that it really does work.* ”

We present in this paper a *conceptually different* encoding, that not only respects lazy reduction, but also the (larger) notion of spine reduction, and is easier to understand. Essentially following the structure of Milner’s proof, central to our approach is the interpretation of the *explicit substitution* version of spine reduction, which allows us to establish a clear connection between term-substitution in the λ-calculus, and the simulation of this operation in the π-calculus via channel-name passing.

We also investigate our interpretation in relation to the type system presented in [4]. This system provides a logical view to the π-calculus, where π-processes can be witnesses of formulae that are provable (within the implicative fragment) in classical logic, as was shown in [4]. That system is different from standard type systems for π as it does not contain any channel information, and in that it expresses implication. We show that our encoding preserves types assignable to λ-terms in Curry’s system. Through this type preservation result we show that our encoding also respects the *functional* behaviour of terms, as expressed via assignable types. In this way we establish a deeper relationship between sequential/applicative and concurrent paradigms.

The results on the type system that we present here determines the choice of π-calculus used for the encoding: we use the asynchronous π-calculus enriched with *pairs* of names [1]. In principle, our encoding could be adapted to the synchronous monadic π-calculus, however we would not be able to achieve the preservation of assignable types. Our encoding takes inspiration from, but it is a much improved version of, the encoding of λ-terms in to the sequent calculus $\mathcal{X}$ [5,6] – a first variant was defined by Urban [26,27]; $\mathcal{X}$ is a sequent calculus that enjoys the Curry-Howard correspondence for Gentzen’s LK [13] – and the encoding of $\mathcal{X}$ into π-calculus as defined in [4].

Our work not only sheds new light on the connection between sequential and concurrent computation but also established a firm link between logic and process calculi. The relation between process calculi and classical logic as first reported on in [4] is an interesting and very promising area of research (similar attempts we made in the context of natural deduction [17], and linear logic [9]). A preliminary interpretation of λ-terms in to the π-calculus was shown in [4]; in this paper we improve the interpretation and strengthen operational correspondence results, by establishing the relationship between the π-calculus ad explicit substitution, and by considering spine reduction.

In summary, the main achievements of this paper are:

– an output-based encoding of the λ-calculus into the asynchronous π-calculus with pairing is defined that preserves spine reduction with explicit substitution for all terms up to contextual equivalence, and, by inclusion, for lazy reduction with explicit substitution;
– our encoding also respects implicit substitution, and respects lazy reduction for closed terms up to simulation;
– our encoding preserves assignable Curry types for λ-terms, with respect to the context assignment system for π from [4].

Paper outline. In Sec. 1, we repeat the definition of the asynchronous π-calculus with pairing, and in Sec. 2 that of the λ-calculus, where we present the notion of explicit spine reduction '$\rightarrow_{\rm xs}$' which takes a central role in this paper; in Sec. 3 we also briefly discuss Milner's interpretation result for the lazy λ-calculus. Then, in Sec. 4, we will define an encoding where terms are interpreted under *output* rather than *input* (as in Milner's), and show that $\rightarrow_{\rm xs}$ is respected by our interpretation; we will also show a simulation result for full β-reduction. Finally, in Sec. 5, we give a notion of (type) context assignment on processes in π, and show that our interpretation preserves types. In fact, this result is the main motivation for our interpretation, which is therefore *logical*.

1 The Asynchronous π-Calculus with Pairing

The notion of asynchronous π-calculus that we consider in this paper is the one used also in [1,4], and is different from other systems studied in the literature [15] in a number of aspects: we add pairing, and introduce the *let*-construct to deal with inputs of pairs of names that get distributed. The main reason for the addition of pairing [1] lies in the fact that we want to preserve implicate type assignment. The π-calculus is an input-output calculus, where terms have not just more than one input, but also more than one output. This is similar to what we find in Gentzen's LK, where right-introduction of the arrow is represented by

$$(\Rightarrow R): \quad \frac{\Gamma \vdash_{\rm LK} \Delta}{\Gamma' \vdash_{\rm LK} A \Rightarrow B, \Delta'} \; (\Gamma = \Gamma', A \;\&\; \Delta = B, \Delta')$$

where Γ and Δ are multi-sets of formulae. Notice that only *one* of the possible formulae is selected from the right context, and *two* formulae are selected in *one* step; when searching for a Curry-Howard correspondence, this will have to be reflected in the (syntactic) witness of the proof. So if we want to model this in π, i.e. want to express function construction (abstraction), we need to bind *two* free names, one as name for the input of the function, and the other as name for its output. We can express that a process P acts as a function *only* when fixing (binding) *both* an input *and* an output *simultaneously*, i.e. in *one* step; we use pairing exactly for this: interfaces for functions are modelled by sending and receiving *pairs* of names.

Below, we will use '$\circ$' for the generic variable, and introduce a structure over names, such that not only names but also pairs of names can be sent (but not a pair of pairs); this way a channel may pass along either a name or a pair of names.

Definition 1. *Channel names* and *data* are defined by:

$$a,b,c,d,x,y,z \quad \textit{names} \qquad\qquad p ::= a \mid \langle a,b\rangle \quad \textit{data}$$

Notice that pairing is *not* recursive. Processes are defined by:

$$\begin{array}{rlllll}
P,Q ::= & 0 & \textit{Nil} & & & \\
\mid & P \mid Q & \textit{Composition} & \mid & a(x).P & \textit{Input} \\
\mid & !\,P & \textit{Replication} & \mid & \overline{a}\langle p\rangle & \textit{(Asynchronous) Output} \\
\mid & (\nu a)\,P & \textit{Restriction} & \mid & \textit{let}\,\langle x,y\rangle = z \textit{ in } P & \textit{Let construct}
\end{array}$$

We abbreviate $a(x).\,\textit{let}\,\langle y,z\rangle = x \textit{ in } P$ by $a(\langle y,z\rangle).\,P$, and $(\nu m)\,(\nu n)\,P$ by $(\nu mn)\,P$.

A (process) context is simply a term with a hole $[\cdot]$.

Definition 2 (Congruence). The structural congruence is the smallest equivalence relation closed under contexts defined by the following rules:

$$\begin{array}{rclrcl}
P \mid \mathbf{0} & \equiv & P & (\nu m)\,(\nu n)\,P & \equiv & (\nu n)\,(\nu m)\,P \\
P \mid Q & \equiv & Q \mid P & (\nu n)\,(P \mid Q) & \equiv & P \mid (\nu n)\,Q \quad \textit{if } n \notin \textit{fn}(P) \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & !\,P & \equiv & P \mid !\,P \\
(\nu n)\,\mathbf{0} & \equiv & \mathbf{0} & \textit{let}\,\langle x,y\rangle = \langle a,b\rangle\,\textit{in}\,R & \equiv & R[a/x,b/y]
\end{array}$$

Definition 3. The *reduction relation* over the processes of the π-calculus is defined by following (elementary) rules:

$$\begin{array}{rl}
(\textit{synchronisation}) : & \overline{a}\langle b\rangle \mid a(x).\,Q \;\rightarrow_\pi\; Q[b/x] \\
(\textit{hiding}) : & P \rightarrow_\pi P' \;\Rightarrow\; (\nu n)\,P \rightarrow_\pi (\nu n)\,P' \\
(\textit{composition}) : & P \rightarrow_\pi P' \;\Rightarrow\; P \mid Q \rightarrow_\pi P' \mid Q \\
(\textit{congruence}) : & P \equiv Q \;\&\; Q \rightarrow_\pi Q' \;\&\; Q' \equiv P' \;\Rightarrow\; P \rightarrow_\pi P'
\end{array}$$

We write $\rightarrow_\pi^*$ for the reflexive and transitive closure of this relation.

Notice that $\overline{a}\langle\langle b,c\rangle\rangle \mid a(\langle x,y\rangle).\,Q \;\rightarrow_\pi\; Q[b/x,c/y]$.

Definition 4. 1. We write $P\downarrow n$ (P *outputs on* n) if $P \equiv (\nu b_1 \ldots b_m)\,(\overline{n}\langle p\rangle \mid Q)$ for some Q, where $n \neq b_1 \ldots b_m$.
2. We write $P \Downarrow n$ (P *will output on* n) if there exists Q such that $P \rightarrow_\pi^* Q$ and $Q\downarrow n$.
3. We write $P \sim_{\mathsf{C}} Q$ (and call P and Q *contextually equivalent*) if, for all contexts $\mathsf{C}[\cdot]$, and for all n, $\mathsf{C}[P] \Downarrow n \iff \mathsf{C}[Q] \Downarrow n$.

Definition 5 ([16]). *Barbed contextual simularity* is the largest relation $\overset{\pi}{\preceq}$ such that $P \overset{\pi}{\preceq} Q$ implies:

– for each name n, if $P\downarrow n$ then $Q \Downarrow n$;
– for any context $\mathsf{C}[\cdot]$, if $\mathsf{C}[P] \rightarrow_\pi P'$, then for some Q', $\mathsf{C}[Q] \rightarrow_\pi^* Q'$ and $P' \overset{\pi}{\preceq} Q'$.

2 The Lambda Calculus (and Variants Thereof)

We assume the reader to be familiar with the λ-calculus; we just repeat the definition of the relevant notions.

Definition 6 (Lambda terms and β-contraction [7]).

1. The set Λ of λ*-terms* is defined by the grammar:
$$M,N ::= x \mid \lambda x.M \mid MN$$
2. The reduction relation $\rightarrow_\beta$ is defined by the rules:
$$(\lambda x.M)N \rightarrow M[N/x] \qquad M \rightarrow N \;\Rightarrow\; \left\{\begin{array}{l} ML \rightarrow NL \\ LM \rightarrow LN \\ \lambda x.M \rightarrow \lambda x.N \end{array}\right.$$
3. *Lazy*[1] reduction [2] $\rightarrow_{\mathrm{L}}$ is defined by limiting the reduction relation to:
$$(\lambda x.M)N \rightarrow M[N/x] \qquad M \rightarrow N \;\Rightarrow\; ML \rightarrow NL$$

[1] This reduction relation is sometimes also known as 'Call-by-Name'; since this is an overloaded concept, we stick to the terminology 'lazy'; the definition here is the one used in [18].

4. We define *spine* reduction[2] $\rightarrow_{\text{S}}$ by limiting reduction to:

$$(\lambda x.M)N \rightarrow M[N/x] \qquad M \rightarrow N \;\Rightarrow\; \begin{cases} ML \rightarrow NL \\ \lambda x.M \rightarrow \lambda x.N \end{cases}$$

Notice that spine reduction is aptly named, since all reductions take place on the spine of the λ-tree (see [8]): searching for a redex, starting from the root, we can walk 'down' and turn 'left', but not turn 'right', so stay on the spine of the tree. This notion of reduction is shown to be head-normalising in [8]; in fact, the normal forms for spine reduction are exactly the head-normal forms for normal reduction [28].

Example 7. Spine reduction encompasses lazy reduction:

$$(\lambda x.(\lambda y.M)N)L \rightarrow_{\text{S}} \begin{cases} (\lambda x.M[N/y])L \\ ((\lambda y.M)N)[L/x] \end{cases}$$

whereas only $(\lambda x.(\lambda y.M)N)L \rightarrow_{\text{L}} ((\lambda y.M)N)[L/x]$.

In view of the importance of substitution also in Milner's result (see Thm. 14), rather than directly interpreting the spine calculus λS, in this paper we will treat λ**x**S, a version with explicit substitution, *à la* Bloo and Rose's calculus λ**x** [10].

Definition 8 (λxS). The syntax of λ**x**S is an extension of that of the λ-calculus:

$$M,N ::= x \mid \lambda x.M \mid MN \mid M\langle x = N\rangle$$

The explicit variant $\rightarrow_{\mathbf{x}\text{S}}$ of spine reduction is now defined as follows. We take the basic rules:

$$(\lambda x.M)N \rightarrow M\langle x = N\rangle \qquad M \rightarrow N \;\Rightarrow\; \begin{cases} ML \rightarrow NL \\ \lambda x.M \rightarrow \lambda x.N \end{cases}$$

(notice the use of $\langle x = N\rangle$ rather than $[N/x]$ in the first rule). The 'propagation rules' for substitution are defined as:

$$M\langle x = N\rangle \rightarrow M\;(x \notin fv(M)) \qquad (\lambda y.M)\langle x = N\rangle \rightarrow \lambda y.(M\langle x = N\rangle)$$
$$(xM_1 \cdots M_n)\langle x = N\rangle \rightarrow (NM_1 \cdots M_n)\langle x = N\rangle$$

Remark 9. We deviate above from the original definition of reduction $\rightarrow_{\mathbf{x}}$ in [10], which included the rules:

$$(ML)\langle x = N\rangle \rightarrow M\langle x = N\rangle L\langle x = N\rangle \qquad x\langle x = N\rangle \rightarrow N$$
$$(\lambda y.M)\langle x = N\rangle \rightarrow \lambda y.(M\langle x = N\rangle) \qquad y\langle x = N\rangle \rightarrow y$$

Since spine reduction focusses on the head of a term, we postpone the substitution on other parts, and only 'work' on the head.[3]

It is easy to show that $M \rightarrow_{\mathbf{x}\text{S}} N$ implies $M \rightarrow_{\mathbf{x}} N$, and that if $M \rightarrow_{\mathbf{x}\text{S}} N$, then there exists a pure term L (not containing $\langle \cdot = \cdot \rangle$) such that $N \rightarrow_{\mathbf{x}} L$, and in this reduction only the substitution rules of λ**x** are applied. Since spine reduction reduces a term M to head-normal form, if it exists, this implies that also $\rightarrow_{\mathbf{x}\text{S}}$ reduces to head-normal form, albeit with perhaps some substitutions still pending.

[2] In [14], essentially following [8], spine reduction is defined by "*just contracting redexes that are on the spine*"; *head* spine reduction is mentioned, but not defined, in [24].

[3] This appears to be the implicit approach of [18] (see Lem. 4.5, case 3).

Example 10. Some terms leave substitutions after reducing: $(\lambda z.yz)N \to_{\mathrm{xs}} yz\langle z = N\rangle$.
We can reduce $(\lambda x.(\lambda z.(\lambda y.M)x))N$ in two different ways:

$$\begin{array}{ll@{\qquad}ll}
(\lambda x.(\lambda z.(\lambda y.M)x))N & \to_{\mathrm{xs}} & (\lambda x.(\lambda z.(\lambda y.M)x))N & \to_{\mathrm{xs}} \\
(\lambda z.(\lambda y.M)x)\langle x = N\rangle & \to_{\mathrm{xs}} & (\lambda x.(\lambda z.(M\langle y = x\rangle)))N & \to_{\mathrm{xs}} \\
\lambda z.((\lambda y.M)x)\langle x = N\rangle & \to_{\mathrm{xs}} & \lambda z.(M\langle y = x\rangle)\langle x = N\rangle & \to_{\mathrm{xs}} \\
\lambda z.(M\langle y = x\rangle\langle x = N\rangle) & & \lambda z.(M\langle y = x\rangle\langle x = N\rangle) &
\end{array}$$

3 Milner's Input-Based Lazy Encoding

Milner defines an encoding of the λ-calculus into the (synchronous, monadic) π-calculus [18][4], and shows some correctness results. This encoding is inspired by the normal semantics of λ-terms, which states for abstraction:

$$[\![\lambda x.M]\!]^{\mathcal{M}}_{\xi} = G(\lambda\!\!\lambda d \in \mathcal{M}.[\![M]\!]^{\mathcal{M}}_{\xi(d/x)})$$

Here the body of the abstraction is interpreted in the updated valuation, where now also x is mapped to d, an arbitrary element of the domain. So, also in the encoding, instead of executing $M[N/x]$, M is executed in an environment that binds N to the variable x; this leads to:

Definition 11 (Milner's interpretation [18]). Input-based encoding of λ-terms into the π-calculus is defined by:

$$\begin{array}{rcll}
[\![x]\!]\,a & \triangleq & \overline{x}\langle a\rangle & x \neq a \\
[\![\lambda x.M]\!]\,a & \triangleq & a(x).\,a(b).\,[\![M]\!]\,b & b\,\textit{fresh} \\
[\![MN]\!]\,a & \triangleq & (\nu c)\,([\![M]\!]\,c \mid (\nu z)\,(\overline{c}\langle z\rangle.\,\overline{c}\langle a\rangle.\,[\![z := N]\!]\,)) & c,z\,\textit{fresh} \\
[\![x := M]\!] & \triangleq & !\,x(w).\,[\![M]\!]\,w & w\,\textit{fresh}
\end{array}$$

(Milner calls $[\![x := M]\!]$ an "*environment entry*"; it could be omitted from the definition above, but is of use separately.) Here a is the link along which $[\![M]\!]$ receives its argument; this is used to communicate with the interpretation of the argument, as made clear in the third case, were the input channel of the left-hand term is used to send the name over on which the right-hand term will receive its input.

Milner's initial approach has since become standard, and is also used in [20,21,22,17]; in fact, as mentioned in the introduction, Sangiorgi considers it canonical [20].

Notice that both the body of the abstraction (M) and the argument in an application (N) get positioned under an input, and that therefore reductions inside these subterms cannot be modelled, and the simulation via the encoding is limited to lazy reduction[5].

Example 12. Using $[\![\cdot]\!]\,\cdot$, the interpretation of a β-redex (only) reduces as follows:

$$\begin{array}{lcl}
[\![(\lambda x.M)N]\!]\,a & \triangleq & \\
(\nu c)\,(c(x).\,c(b).\,[\![M]\!]\,b \mid (\nu z)\,(\overline{c}\langle z\rangle.\,\overline{c}\langle a\rangle.\,[\![z := N]\!]\,)) & \to^{*}_{\pi} & \\
(\nu z)\,([\![M[z/x]]\!]\,a \mid [\![z := N]\!]\,) & = & (z \notin [\![M]\!]\,b) \\
(\nu x)\,([\![M]\!]\,a \mid [\![x := N]\!]\,) \quad \triangleq \; (\nu x)\,([\![M]\!]\,a \mid !\,x(w).\,[\![N]\!]\,w) & &
\end{array}$$

[4] [18] also deals with Call-By-Value, which we will not consider here.

[5] It is possible to improve on this result by extending the notion of reduction or congruence on π, by adding, for example, $P \to_{\pi} Q \Rightarrow x(v).\,P \to_{\pi} x(v).\,Q$, but that is not our intention here; we aim just to compare our result with Milner's.

Now reduction can continue in (the interpretation of) M, but not in N that is still guarded by the input on x, which will not be used until the evaluation of $[\![M]\!]\,a$ reaches the point where output is generated over x.

In fact, Milner shows that all interpretations of closed λ-terms reduce to terms of this shape (Thm. 14). Notice that the result $(\nu x)\,([\![M]\!]\,a \mid [\![x := N]\!]\,)$ is not the same as $[\![M[N/x]]\!]\,a$, and also does not reduce to that term, as illustrated by the following:

Example 13. $[\![(\lambda x.xx)(\lambda y.y)]\!]\,a \quad \stackrel{\Delta}{=}$

$$\begin{array}{ll}
(\nu c)\,(c(x).c(b).[\![xx]\!]\,b \mid (\nu z)\,(\overline{c}\langle z\rangle.\overline{c}\langle a\rangle.[\![z := \lambda y.y]\!]\,)) & \to_\pi^* \\
(\nu z)\,([\![zz]\!]\,a \mid [\![z := \lambda y.y]\!]\,) & \stackrel{\Delta}{=} \\
(\nu z)\,((\nu c)\,(\overline{z}\langle c\rangle \mid (\nu z_1)\,(\overline{c}\langle z_1\rangle.\overline{c}\langle a\rangle.[\![z_1 := z]\!]\,)) \mid [\![z := \lambda y.y]\!]\,) & \equiv \\
(\nu zc)\,(\overline{z}\langle c\rangle \mid (\nu z_1)\,(\overline{c}\langle z_1\rangle.\overline{c}\langle a\rangle.[\![z_1 := z]\!]\,) & \\
\qquad\qquad \mid z(w).[\![\lambda y.y]\!]\,w \mid [\![z := \lambda y.y]\!]\,) & \to_\pi \\
(\nu zc)\,((\nu z_1)\,(\overline{c}\langle z_1\rangle.\overline{c}\langle a\rangle.[\![z_1 := z]\!]\,) \mid [\![\lambda y.y]\!]\,c \mid [\![z := \lambda y.y]\!]\,) & \equiv \\
(\nu zcz_1)\,(\overline{c}\langle z_1\rangle.\overline{c}\langle a\rangle.[\![z_1 := z]\!] \mid c(y).c(b).\overline{y}\langle b\rangle \mid [\![z := \lambda y.y]\!]\,) & \to_\pi^* \\
(\nu zz_1)\,([\![z_1 := z]\!] \mid \overline{z_1}\langle a\rangle \mid [\![z := \lambda y.y]\!]\,) & \to_\pi \\
(\nu zz_1)\,(\overline{z}\langle a\rangle \mid [\![z_1 := z]\!] \mid [\![z := \lambda y.y]\!]\,) & \equiv \\
(\nu zc)\,(\overline{z}\langle a\rangle \mid z(w).[\![\lambda y.y]\!]\,w \mid [\![z := \lambda y.y]\!]\,) & \to_\pi \\
(\nu z)\,([\![\lambda y.y]\!]\,a \mid [\![z := \lambda y.y]\!]\,) & \equiv [\![\lambda y.y]\!]\,a
\end{array}$$

Notice that we executed the only possible communications, and that in the reduction path no term corresponds to $(\nu c)\,([\![\lambda y.y]\!]\,c \mid (\nu z)\,(\overline{c}\langle z\rangle.\overline{c}\langle a\rangle.[\![z := \lambda y.y]\!]\,))$ (i.e. $[\![(\lambda y.y)(\lambda y.y)]\!]\,a$). Of course reducing the term $[\![(\lambda y.y)(\lambda y.y)]\!]\,a$ will also yield $[\![\lambda y.y]\!]\,a$, but we can only show that $[\![(\lambda x.xx)(\lambda y.y)]\!]\,a$ and $[\![(\lambda y.y)(\lambda y.y)]\!]\,a$ have a *common reduct*, not that the first reduces to the second.

Milner states the correctness for his interpretation with respect to lazy reduction as:

Theorem 14 ([18]). *For all closed λ-terms M, either:*

1. *$M \to_{\mathrm{L}} \lambda y.R[\overrightarrow{N/x}]$, and $[\![M]\!]\,u \to_\pi (\nu\vec{x})\,([\![\lambda y.R]\!]\,u \mid [\![\overrightarrow{x := N}]\!]\,)$, or*
2. *both M and $[\![M]\!]\,u$ diverge.*

It is worthwhile to note that, although not mentioned in [18], in the proof of this result Milner treats the substitution as *explicit*, not as *implicit*. In fact, although stated with implicit substitution, Milner's result does not show that lazy reduction (with implicit substitution) is fully modelled, but only 'up to substitution'; as shown in Ex. 13, it is impossible to reduce the encoding of $(\lambda x.xx)(\lambda y.y)$ to that of $(\lambda y.y)(\lambda y.y)$.

We quickly found that we (also) could not model implicit substitution, and reverted to *explicit* substitution; however, $(\lambda x.xx)(\lambda y.y) \to_{\mathbf{xs}} ((\lambda y.y)x)\langle x = (\lambda y.y)\rangle$, and we *can* show that the encoding of $(\lambda x.xx)(\lambda y.y)$ runs to that of $((\lambda y.y)x)\langle x = (\lambda y.y)\rangle$, as shown in Ex. 23.

4 A Logical, Output-Based Encoding of λ-Terms

In this section, we will show that it is possible to deviate from Milner's original encoding, and actually making a gain in the process. Inspired by the relation between natural deduction and the sequent calculus [13], interpreting terms under *output* rather than under *input*, and using the π-calculus with pairing, we can define a different encoding of the λ-calculus into the π-calculus that preserves not just lazy reduction, but also the larger notion of spine reduction.

Our encoding follows from – but is an improvement of – the concatenation of the encoding of the λ-calculus into $\mathcal{X}$ (which established a link between natural deduction and the sequent calculus), defined in [6], and the interpretation of $\mathcal{X}$ into the π-calculus as defined in [4]. The main objective of our encoding is to show the preservation of type assignment; pairing is used in order to be able to effectively represent arrow types.

The idea behind our encoding originates from the observation that, in the lambda calculus, all input is named, but output is anonymous. Input (i.e. a *variable*) is named to serve as a destination for the substitution; output need not be named, since all terms have only one result (represented by the term itself), which is used *in sito*[6]. Translating into the (multi-output) π-calculus, this locality property no longer holds; we need to specify the destination of a term, by naming its output: this is what the encoding does.

We explicitly convert '*an output sent on a is to be received as input on b*' via '$a(\circ).\overline{b}\langle\circ\rangle$' (call a *forwarder* in [16]), which for convenience is abbreviated into $a{=}b$.

Definition 15 (Output-based interpretation). The mapping $[\![\cdot]\!]\cdot$ is defined by[7]:

$$\begin{array}{rcll}
[\![x]\!]a & \triangleq & x(\circ).\overline{a}\langle\circ\rangle & x \neq a \\
[\![\lambda x.M]\!]a & \triangleq & (\nu xb)\,([\![M]\!]b \mid \overline{a}\langle\langle x,b\rangle\rangle) & b\,\textit{fresh} \\
[\![MN]\!]a & \triangleq & (\nu c)\,([\![M]\!]c \mid c(\langle b,d\rangle).(!\,[\![N]\!]b \mid d{=}a)) & b,c,d\,\textit{fresh}
\end{array}$$

In particular: • we see a variable x as an input channel, and we need to retransmit its input to the output channel a that we interpret it under;

- for an abstraction $\lambda x.M$, we give the name b to the output of M; that M has input x and output b gets sent out over a, which is the name of $\lambda x.M$, so that a process that wants to call on this functionality, knows which channel to send the input to, and on which channel to pick up the result[8];
- for an application MN, the output of M, transmitted over c, is received as a pair $\langle b,d\rangle$ of input-output names in the right-hand side; the received input b name is used as output name for N, enabling the simulation of substitution, and the received output name d gets redirected to the output of the application a.

Notice that only one replication is used, on the argument in an application; this corresponds, as usual, to the implementation of the (distributive) substitution on λ-terms.

[6] In terms of *continuations*, the continuation of a term is not mentioned, since it is the current.

[7] We could have defined our encoding directly in the standard π-calculus:

$$\begin{array}{rcl}
[\![x]\!]'a & \triangleq & !\,x(\circ).\overline{a}\langle\circ\rangle \\
[\![\lambda x.M]\!]'a & \triangleq & (\nu xb)\,([\![M]\!]'b \mid \overline{a}\langle x\rangle.\overline{a}\langle b\rangle) \\
[\![MN]\!]'a & \triangleq & (\nu c)\,([\![M]\!]'c \mid c(b).c(d).(!\,[\![N]\!]'b \mid d{=}a))
\end{array}$$

without losing any of the reduction results for our encoding, but this has additional replication, and is less suited for type assignment (see Sect. 5).

[8] This view of computation is exactly that of the calculus $\mathcal{X}$.

Also, every $\llbracket N \rrbracket a$ is a process that outputs on a non-hidden name a (albeit perhaps not actively, as in the third case, where it will not be activated until input is received on the channel c, in which case it is used to output the data received in on the channel d that is passed as a parameter), and that this output is unique, in the sense that a is the only output channel, is only used once, and for output only. The structure of the encoding of application corresponds, in fact, to how Gentzen encodes *modus ponens* in the sequent calculus [13]: see [6], Thm. 4.8, and the proof of Thm. 31 below.

Example 16. 1. $\llbracket (\lambda y.P)Q \rrbracket a \quad \triangleq$
$(\nu c)\,((\nu y b_1)\,(\llbracket P \rrbracket b_1 \mid \overline{c}\langle\langle y,b_1\rangle\rangle) \mid c(\langle b,d\rangle).\,(!\,\llbracket Q \rrbracket b \mid d{=}a)) \rightarrow_\pi$
$(\nu y b)\,(\llbracket P \rrbracket b \mid !\,\llbracket Q \rrbracket y \mid b{=}a)$

In short, the encoding of the redex $(\lambda y.P)Q$ will yield a communication that receives on the input channel called y in the interpretation of P, and the interpretation of Q being output on y.

2. $\llbracket \lambda x.x \rrbracket a \triangleq (\nu x b)\,(x(\circ).\,\overline{b}\langle\circ\rangle \mid \overline{a}\langle\langle x,b\rangle\rangle)$.
3. $\llbracket xN \rrbracket a = (\nu c)\,(\llbracket x \rrbracket c \mid c(\langle b,d\rangle).\,(!\,\llbracket N \rrbracket b \mid d{=}a)) \succeq^{\pi} x(\langle b,d\rangle).\,(!\,\llbracket N \rrbracket b \mid d{=}a)$
This term correctly expresses that computation is halted until on x we send the input-output interface of a function, which will then communicate with the output channel of N as its input channel.

Notice that the second term cannot input; it is easy to check that this is true for the the interpretation of all closed λ-terms, since we can show the following:

Property 17. $fn(\llbracket M \rrbracket a) = fv(M) \cup \{a\}$.

The following result, which states that we can safely rename the (hidden) output of an encoded λ-term, is needed below:

Lemma 18. $(\nu a)\,(a{=}e \mid \llbracket N \rrbracket a) \sim_{\mathsf{c}} \llbracket N \rrbracket e$.

Using this result, we can show that

$$(\nu x b)\,(\llbracket M \rrbracket b \mid !\,\llbracket N \rrbracket x \mid b{=}a) \sim_{\mathsf{c}} (\nu x)\,(\llbracket M \rrbracket a \mid !\,\llbracket N \rrbracket x)$$

Following on from Ex. 16, we can now define

Definition 19. We extend our interpretation to $\lambda\mathbf{x}$S, adding

$$\llbracket M\langle x{=}N\rangle \rrbracket a = (\nu x)\,(\llbracket M \rrbracket a \mid !\,\llbracket N \rrbracket x)$$

As in [18,21,23], we can show a reduction preservation result; however, not by just restricting to (explicit) lazy reduction, but to the larger system for explicit spine reduction for the λ-calculus. Notice that, essentially following Milner, by using the reduction relation $\rightarrow_{\mathrm{xs}}$, we show that our interpretation respects reduction upto substitution; however, we do not require the terms to be closed:

Theorem 20 **($\llbracket\cdot\rrbracket\cdot$ preserves $\rightarrow_{\mathrm{xs}}$).** *If $M \rightarrow_{\mathrm{xs}} N$, then $\llbracket M \rrbracket a \sim_{\mathsf{c}} \llbracket N \rrbracket a$.*

So, perhaps contrary to expectation, since abstraction is not encoded using *input*, we can without problem model a reduction under a λ-abstraction. Moreover, the only extra property we use is the renaming of the output under which λ-terms are encoded (Lem. 18).

Let us illustrate this result via a concrete example.

Example 21.

$$\begin{array}{lll}
\llbracket(\lambda x.(\lambda z.(\lambda y.M)x))N\rrbracket a & \triangleq,\equiv & \\
(\nu cxb_1)\,(\llbracket(\lambda z.(\lambda y.M)x)\rrbracket b_1 \mid \overline{c}\langle\langle x,b_1\rangle\rangle \mid c(\langle b,d\rangle).\,(!\,\llbracket N\rrbracket b \mid d{=}a)) & \rightarrow_\pi & \\
(\nu xb_1)\,(\llbracket(\lambda z.(\lambda y.M)x)\rrbracket b_1 \mid !\,\llbracket N\rrbracket x \mid b_1{=}a) & \triangleq,\equiv & \\
(\nu xb_1zb_2c_1yb_3)\,(\llbracket M\rrbracket b_3 \mid \overline{c_1}\langle\langle y,b_3\rangle\rangle \mid & & \\
\qquad\qquad c_1(\langle b,d\rangle).\,(!\,\llbracket x\rrbracket b \mid d{=}b_2) \mid \overline{b_1}\langle\langle z,b_2\rangle\rangle \mid !\,\llbracket N\rrbracket x \mid b_1{=}a) & \rightarrow_\pi & (c_1) \\
(\nu xb_1zb_2yb_3)\,(\llbracket M\rrbracket b_3 \mid !\,\llbracket x\rrbracket y \mid b_3{=}b_2 \mid \overline{b_1}\langle\langle z,b_2\rangle\rangle \mid !\,\llbracket N\rrbracket x \mid b_1{=}a) & \rightarrow_\pi & (b_1) \\
(\nu xzb_2yb_3)\,(\llbracket M\rrbracket b_3 \mid !\,\llbracket x\rrbracket y \mid b_3{=}b_2 \mid \overline{a}\langle\langle z,b_2\rangle\rangle \mid !\,\llbracket N\rrbracket x) & \sim_{\mathsf{c}} & (18) \\
(\nu xzb_2y)\,(\llbracket M\rrbracket b_2 \mid !\,\llbracket x\rrbracket y \mid \overline{a}\langle\langle z,b_2\rangle\rangle \mid !\,\llbracket N\rrbracket x) & \equiv & \\
(\nu zb)\,((\nu x)\,((\nu y)\,(\llbracket M\rrbracket b \mid !\,\llbracket x\rrbracket y) \mid !\,\llbracket N\rrbracket x) \mid \overline{a}\langle\langle z,b\rangle\rangle) & \triangleq & \\
\llbracket\lambda z.M\langle y=x\rangle\,\langle x=N\rangle\rrbracket a & &
\end{array}$$

In the proof of Thm. 20, in only two places do we perform a renaming via Lem. 18, so use that $(\nu a)\,(a{=}e \mid \llbracket N\rrbracket a) \sim_{\mathsf{c}} \llbracket N\rrbracket e$. In both cases, the term we rename occurs at the head, and – assuming the reduction terminates – either N reduces to an abstraction $\lambda z.N'$, and then (wlog)

$$\begin{array}{ll}
(\nu a)\,(a{=}e \mid \llbracket N\rrbracket a) & \rightarrow_\pi \\
(\nu a)\,(a{=}e \mid (\nu zb)\,(\llbracket N'\rrbracket b \mid \overline{a}\langle\langle z,b\rangle\rangle)) & \equiv \\
(\nu azb)\,(a{=}e \mid \llbracket N'\rrbracket b \mid \overline{a}\langle\langle z,b\rangle\rangle) & \rightarrow_\pi \\
(\nu zb)\,(\llbracket N'\rrbracket b \mid \overline{e}\langle\langle z,b\rangle\rangle) &
\end{array}$$

or N reduces to a variable, and the renaming need not be executed. So when performing a reduction on $\llbracket M\rrbracket a$, where M has a normal form with respect to $\rightarrow_{\mathrm{xs}}$ (a head-normal form with respect to $\rightarrow_\beta$), these renamings can be postponed.

Corollary 22. If M has a normal form N (wrt $\rightarrow_{\mathrm{xs}}$), then $\llbracket M\rrbracket a \rightarrow_\pi \llbracket N\rrbracket a$.

Example 23. By Thm. 20, $\llbracket(\lambda x.xx)(\lambda y.y)\rrbracket a \sim_{\mathsf{c}} (\nu x)\,(\llbracket xx\rrbracket a \mid !\,\llbracket\lambda y.y\rrbracket x)$; but we *can* run the π-process without using the equivalence relation:

$$\begin{array}{lll}
\llbracket(\lambda x.xx)(\lambda y.y)\rrbracket a & \triangleq,\equiv & \\
(\nu cxb_1)\,(\llbracket xx\rrbracket b_1 \mid \overline{c}\langle\langle x,b_1\rangle\rangle \mid c(\langle b,d\rangle).\,(!\,\llbracket\lambda y.y\rrbracket b \mid d{=}a)) & \rightarrow_\pi & \\
(\nu xb_1)\,(\llbracket xx\rrbracket b_1 \mid !\,\llbracket\lambda y.y\rrbracket x \mid b_1{=}a) & \triangleq,\equiv & \\
(\nu xb_1)\,(\llbracket xx\rrbracket b_1 \mid (\nu yb)\,(y(\circ).\overline{b}\langle\circ\rangle \mid \overline{x}\langle\langle y,b\rangle\rangle) \mid !\,\llbracket\lambda y.y\rrbracket x \mid b_1{=}a) & \triangleq,\equiv & \\
(\nu xb_1c_1yb)\,(x(\circ).\overline{c_1}\langle\circ\rangle \mid c_1(\langle b_2,d\rangle).\,(!\,\llbracket x\rrbracket b_2 \mid d{=}b_1) & & \\
\qquad\qquad \mid \llbracket y\rrbracket b \mid \overline{x}\langle\langle y,b\rangle\rangle \mid !\,\llbracket\lambda y.y\rrbracket x \mid b_1{=}a) & \rightarrow_\pi & (x) \\
(\nu xb_1c_1yb)\,(\overline{c_1}\langle\langle y,b\rangle\rangle \mid c_1(\langle b_2,d\rangle).\,(!\,\llbracket x\rrbracket b_2 \mid d{=}b_1) & & \\
\qquad\qquad \mid \llbracket y\rrbracket b \mid !\,\llbracket\lambda y.y\rrbracket x \mid b_1{=}a) & \rightarrow_\pi & (c_1) \\
(\nu xb_1yb)\,(!\,\llbracket x\rrbracket y \mid b{=}b_1 \mid \llbracket y\rrbracket b \mid !\,\llbracket\lambda y.y\rrbracket x \mid b_1{=}a) & \equiv & (\alpha) \\
(\nu xb_1yb)\,(!\,\llbracket x\rrbracket y \mid b{=}b_1 \mid \llbracket y\rrbracket b & & \\
\qquad\qquad \mid (\nu zb_2)\,(\llbracket z\rrbracket b_2 \mid \overline{x}\langle\langle z,b_2\rangle\rangle) \mid !\,\llbracket\lambda y.y\rrbracket x \mid b_1{=}a) & \equiv & \\
(\nu xb_1ybzb_2)\,(x(\circ).\overline{y}\langle\circ\rangle \mid !\,\llbracket x\rrbracket y \mid b{=}b_1 \mid \llbracket y\rrbracket b & & \\
\qquad\qquad \mid \llbracket z\rrbracket b_2 \mid \overline{x}\langle\langle z,b_2\rangle\rangle \mid !\,\llbracket\lambda y.y\rrbracket x \mid b_1{=}a) & \rightarrow_\pi^* & (x,y,b,b_1) \\
(\nu xyzb_2)\,(!\,\llbracket x\rrbracket y \mid \overline{a}\langle\langle z,b_2\rangle\rangle \mid \llbracket z\rrbracket b_2 \mid !\,\llbracket\lambda y.y\rrbracket x) & \equiv & \\
(\nu zb)\,(\llbracket z\rrbracket b \mid \overline{a}\langle\langle z,b\rangle\rangle) & \triangleq & \llbracket\lambda z.z\rrbracket a
\end{array}$$

Notice that we performed the two substitutions without resorting to the renaming of outputs of encoded λ-terms, and that all those renamings take place at the end.

Notice also that, because the encoding implements a limited notion of substitution, the reduction does *not* run past

$$(\nu c)\,(\llbracket \lambda y.y \rrbracket c \mid c(\langle b,d\rangle).(!\llbracket \lambda y.y \rrbracket b \mid d{=}a)) \triangleq \llbracket (\lambda y.y)(\lambda y.y) \rrbracket a.$$

The only expression that gets close is that in the sixth line, which corresponds to

$$(\nu x b_1 c_1 y b)\,(\llbracket \lambda y.y \rrbracket c_1 \mid c_1(\langle b_2,d\rangle).(!\,\llbracket x \rrbracket b_2 \mid d{=}b_1) \mid !\,\llbracket \lambda y.y \rrbracket x \mid b_1{=}a)$$

which is (up to renaming) $\llbracket ((\lambda y.y)x)\langle x = (\lambda y.y)\rangle \rrbracket a$. In fact, this is as expected, since:

$$(\lambda x.xx)(\lambda y.y) \to_{\text{xs}} xx\langle x = (\lambda y.y)\rangle \to_{\text{xs}} ((\lambda y.y)x)\langle x = (\lambda y.y)\rangle$$

We can also show that $(\nu x)\,(\llbracket M \rrbracket a \mid !\,\llbracket N \rrbracket x)$ represents (implicit) term substitution successfully, at least for closed N.

Theorem 24. *For N a closed λ-term, and M any λ-term:*

$$(\nu x)\,(\llbracket M \rrbracket a \mid !\,\llbracket N \rrbracket x) \succeq^{\pi} \llbracket M[N/x] \rrbracket a.$$

In [21], a similar result is shown, but for the *higher-order* π-calculus, where substitution is a primitive operation, so the encoding is easier.

That we had to limit the contraction of redexes $(\lambda x.M)N$ to those where N is closed, might seem a strong restriction. However, notice that Milner's simulation result is stated, not just for lazy reduction - which is a weaker notion than spine reduction - but also only for closed λ-terms. Consider a reduction $M \to_{\text{L}} M'$, where M is closed, then $M = (\lambda x.M_1)NM_2 \cdots M_n$; since M is closed, so is N, so our result is as strong as need be in the context of Milner's result.

Corollary 25. If $M \to_{\text{L}} N$, and M is a closed λ-term, then $\llbracket M \rrbracket a \succeq^{\pi} \llbracket N \rrbracket a$.

5 Context Assignment

The π-calculus is equipped with a rich type theory [23]: from the basic type system for counting the arity of channels to sophisticated linear types in [17], which studies a relation between Call-by-Value $\lambda\mu$ and a linear π-calculus. Linearisation is used to be able to achieve processes that are functions, by allowing output over one channel name only, in a λ-calculus, natural deduction style. Moreover, the encoding presented in [17] is type dependent, in that, for each term, different π-processes are assigned, depending on the original type; this makes the encoding quite cumbersome.

The type system presented in this section differs quite drastically from the standard type system presented in [23]: here input and output channels essentially have the type of the data they are sending or receiving, and are separated by the type system by putting all inputs with their types on the left of the sequent, and the outputs on the right. In our system, types give a logical view to the π-calculus rather than an abstract specification on how channels should behave. Our encoding is very simple and intuitive by interpreting the cut operationally as a communication. The idea of giving a computational interpretation of the cut as a communication primitive is also used by [3] and [9]. In both papers, only a small fragment of Linear Logic was considered, and the encoding between proofs and π-calculus was left rather implicit.

In this section, we define a notion of context assignment for processes in π that describes the '*input-output interface*' of a process, by assigning a left-context, containing the types for the input channels, and a right-context, containing the types for the output channels; it was first presented in [4]. This notion is different from others in that it assigns to channels the type of the input or output that is sent over the channel.

Context assignment was defined in [4] to establish preservation of assignable types under the interpretation of the sequent calculus $\mathcal{X}$, as presented in [6], into the π-calculus. As for the notion of type assignment on $\mathcal{X}$ terms, in the typing judgements we write names used for input on the left and names used for output on the right; this implies that, if a name is both used to send and to receive, it will appear on both sides, with the same type. Since $\mathcal{X}$ offers a natural presentation of the classical propositional calculus with implication, and enjoys the Curry-Howard isomorphism for the implicative fragment of Gentzen's system LK [12], this implies that the notion of context assignment as defined below is *classical* (i.e. not intuitionistic) in nature.

We now repeat the definition of (simple) type assignment; we first define types and contexts.

Definition 26 (Types and Contexts)

1. The set of types is defined by the grammar: $A, B ::= \varphi \mid A{\rightarrow}B$, where φ is a basic type of which there are infinitely many.
2. An *input context* Γ is a mapping from names to types, denoted as a finite set of *statements* $a{:}A$, such that the *subject* of the statements (a) are distinct. We write Γ_1, Γ_2 to mean the *compatible union* of Γ_1 and Γ_2 (if Γ_1 contains $a{:}A_1$ and Γ_2 contains $a{:}A_2$, then $A_1 = A_2$), and write $\Gamma, a{:}A$ for $\Gamma, \{a{:}A\}$.
3. *Output* contexts Δ, and the notions Δ_1, Δ_2, and $n{:}A, \Delta$ are defined in a similar way.
4. If $n{:}A \in \Gamma$ and $n{:}B \in \Delta$, then $A = B$.

Definition 27 ((Classical) Context Assignment). Context assignment for the π-calculus with pairing is defined by the following sequent system:

$$(0) : \frac{}{0 : \Gamma \vdash_\pi \Delta} \qquad (\mathit{pair\text{-}out}) : \frac{}{\overline{a}\langle\langle b, c\rangle\rangle : \Gamma, b{:}A \vdash_\pi a{:}A{\rightarrow}B, c{:}B, \Delta}$$

$$(!) : \frac{P : \Gamma \vdash_\pi \Delta}{!\,P : \Gamma \vdash_\pi \Delta} \qquad (\mathit{out}) : \frac{}{\overline{a}\langle b\rangle : \Gamma, b{:}A \vdash_\pi a{:}A, b{:}A, \Delta}\ (a \neq b)$$

$$(\nu) : \frac{P : \Gamma, a{:}A \vdash_\pi a{:}A, \Delta}{(\nu a)\, P : \Gamma \vdash_\pi \Delta} \qquad (\mathit{let}) : \frac{P : \Gamma, y{:}B \vdash_\pi x{:}A, \Delta}{\mathit{let}\,\langle x, y\rangle = z\ \mathit{in}\ P : \Gamma, z{:}A{\rightarrow}B \vdash_\pi \Delta}$$

$$(|) : \frac{P : \Gamma \vdash_\pi \Delta \quad Q : \Gamma \vdash_\pi \Delta}{P \mid Q : \Gamma \vdash_\pi \Delta} \qquad (\mathit{in}) : \frac{P : \Gamma, x{:}A \vdash_\pi x{:}A.\Delta}{a(x).\,P : \Gamma, a{:}A \vdash_\pi \Delta}$$

The side-condition on rule (out) is there to block the derivation of $\overline{a}\langle a\rangle : \;\vdash_\pi a{:}A$.

Example 28. Although we have no rule $(\mathit{pair\text{-}in})$, it is admissible, since we can derive

$$\cfrac{\cfrac{P : \Gamma, y{:}B \vdash_\pi x{:}A, \Delta}{\mathit{let}\,\langle x, y\rangle = z\ \mathit{in}\ P : \Gamma, z{:}A{\rightarrow}B \vdash_\pi \Delta}}{a(z).\,\mathit{let}\,\langle x, y\rangle = z\ \mathit{in}\ P : \Gamma, a{:}A{\rightarrow}B \vdash_\pi \Delta}$$

This notion of type assignment does not (directly) relate back to the logical calculus LK. For example, rules $(\mid)$ and $(!)$ do not change the contexts, so do not correspond to any rule in LK, not even to a $\lambda\mu$-style [19] activation step; moreover, rule (ν) just removes a formula.

The *weakening* rule is admissible:

$$(W): \frac{P : \Gamma \vdash_\pi \Delta}{P : \Gamma' \vdash_\pi \Delta'} \; (\Gamma' \supseteq \Gamma, \Delta' \supseteq \Delta)$$

This result allows us to be a little less precise when we construct derivations, and allow for rules to join contexts, by using, for example, the rule

$$(\mid): \frac{P : \Gamma_1 \vdash_\pi \Delta_1 \quad Q : \Gamma_2 \vdash_\pi \Delta_2}{P \mid Q : \Gamma_1, \Gamma_2 \vdash_\pi \Delta_1, \Delta_2}$$

so switching, without any scruples, to multiplicative style, whenever convenient.

We have a soundness (witness reduction) result for our notion of type assignment for π as shown in [4].

Theorem 29 (Witness reduction [4]). *If* $P : \Gamma \vdash_\pi \Delta$ *and* $P \rightarrow_\pi Q$, *then* $Q : \Gamma \vdash_\pi \Delta$.

We will now show that our interpretation preserves types assignable to lambda terms using Curry's system, which is defined as follows:

Definition 30 (Curry type assignment for the λ-calculus).

$$(Ax): \frac{}{\Gamma, x{:}A \vdash_\lambda x : A} \qquad (\rightarrow I): \frac{\Gamma, x{:}A \vdash_\lambda M : B}{\Gamma \vdash_\lambda \lambda x.M : A \rightarrow B}$$

$$(\rightarrow E): \frac{\Gamma \vdash_\lambda M : A \rightarrow B \quad \Gamma \vdash_\lambda N : A}{\Gamma \vdash_\lambda MN : B}$$

We can now show that typeability is preserved by $\llbracket \cdot \rrbracket \cdot$:

Theorem 31. *If* $\Gamma \vdash_\lambda M : A$, *then* $\llbracket M \rrbracket a : \Gamma \vdash_\pi a{:}A$.

Proof. By induction on the structure of derivations in $\vdash_\lambda$; notice that we use implicit weakening.

(Ax) : Then $M = x$, and $\Gamma = \Gamma', x{:}A$. Notice that $x(\circ).\overline{a}\langle\circ\rangle = \llbracket x \rrbracket a$, and that

$$\frac{\dfrac{}{\overline{a}\langle\circ\rangle : \Gamma, \circ{:}A \vdash_\pi a{:}A, \circ{:}A}}{x(\circ).\overline{a}\langle\circ\rangle : \Gamma', x{:}A \vdash_\pi a{:}A}$$

$(\rightarrow I)$: Then $M = \lambda x.N$, $A = C \rightarrow D$, and $\Gamma, x{:}C \vdash_\lambda N : D$. Then, by induction, a derivation $\mathcal{D} :: \llbracket N \rrbracket b : \Gamma, x{:}C \vdash_\pi b{:}D$ exists, and we can construct:

$$\frac{\dfrac{\dfrac{\dfrac{\mathcal{D}}{\llbracket N \rrbracket b : \Gamma, x{:}C \vdash_\pi b{:}D} \quad \dfrac{}{\overline{a}\langle\langle x,b\rangle\rangle : x{:}C \vdash_\pi a{:}C \rightarrow D, b{:}D}}{\llbracket N \rrbracket b \mid \overline{a}\langle\langle x,b\rangle\rangle : \Gamma, x{:}C \vdash_\pi a{:}C \rightarrow D, b{:}D}}{(\nu b)(\llbracket N \rrbracket b \mid \overline{a}\langle\langle x,b\rangle\rangle) : \Gamma, x{:}C \vdash_\pi a{:}C \rightarrow D}}{(\nu xb)\,(\llbracket N \rrbracket b \mid \overline{a}\langle\langle x,b\rangle\rangle) : \Gamma \vdash_\pi a{:}C \rightarrow D}$$

Notice that $(\nu xb)\,(\llbracket N \rrbracket b \mid \overline{a}\langle\langle x,b\rangle\rangle) = \llbracket \lambda x.N \rrbracket a$.

$(\rightarrow E)$: Then $M = PQ$, and there exists B such that $\Gamma \vdash_\lambda P : B{\rightarrow}A$ and $\Gamma \vdash_\lambda Q : B$. By induction, there exist $\mathcal{D}_1 :: [\![P]\!]c : \Gamma \vdash_\pi c{:}B{\rightarrow}A$ and $\mathcal{D}_2 :: [\![Q]\!]b : \Gamma \vdash_\pi b{:}B$, and we can construct:

$$\dfrac{\dfrac{\mathcal{D}_1}{[\![P]\!]c : \Gamma \vdash_\pi c{:}B{\rightarrow}A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}_2}{[\![Q]\!]b : \Gamma \vdash_\pi b{:}B}}{!\,[\![Q]\!]b : \Gamma \vdash_\pi b{:}B} \qquad \dfrac{\dfrac{}{\overline{a}\langle\circ\rangle : \Gamma, \circ{:}A \vdash_\pi a{:}A, \circ{:}A, \Delta}}{d{=}a : d{:}A \vdash_\pi a{:}A}}{!\,[\![Q]\!]b \mid d{=}a : \Gamma, d{:}A \vdash_\pi b{:}B, a{:}A}}{c(\langle b,d\rangle)\,.\,(!\,[\![Q]\!]b \mid d{=}a) : \Gamma, c{:}B{\rightarrow}A \vdash_\pi a{:}A}}{\dfrac{[\![P]\!]c \mid c(\langle b,d\rangle)\,.\,(!\,[\![Q]\!]b) \mid d{=}a) : \Gamma, c{:}B{\rightarrow}A \vdash_\pi c{:}B{\rightarrow}A, a{:}A}{(\nu c)\,([\![P]\!]c \mid c(\langle b,d\rangle)\,.\,(!\,[\![Q]\!]b \mid d{=}a)) : \Gamma \vdash_\pi a{:}A}}$$

and $[\![PQ]\!]a = (\nu c)\,([\![P]\!]c \mid c(\langle b,d\rangle)\,.\,(!\,[\![Q]\!]b \mid d{=}a))$. □

Notice that although, in the above proof, we are only interested in showing results with *one* typed output (conclusion) – after all, we are interpreting the typed λ-calculus, an intuitionistic system – we need the classical, multi-conclusion character of our type assignment system for π to achieve this result.

6 Conclusions and Future Work

We have found a new, simple and intuitive encoding of λ-terms in π that respects our definition of explicit spine reduction, is similar with normal reduction, and encompasses Milner's lazy reduction on closed terms. We have shown that, for our context assignment system that uses the type constructor $\rightarrow$ for π and is based on classical logic, assignable types for λ-terms are preserved by our interpretation as typeable π-processes. We managed this without having to linearise the calculus as done in [17].

The classical sequent calculus $\mathcal{X}$ has two natural, dual notions of sub-reduction, called Call-by-Name and Call-by-Value; we will investigate if the interpretation of these systems in to the π-calculus gives natural notions of CBN of CBV reduction on π-processes, and if this enables CBN or CBV logical encodings of the λ-calculus.

Acknowledgements

We would like to thank Jan Willem Klop, Fer-Jan de Vries, and Vincent van Oostrom for their willingness to instruct, and Martin Berger for useful comments.

References

1. Abadi, M., Gordon, A.: A Calculus for Cryptographic Protocols: The Spi Calculus. In: CC&CS 1997, pp. 36–47 (1997)
2. Abramsky, S.: The lazy lambda calculus. In: Research topics in functional programming, pp. 65–116 (1990)
3. Abramsky, S.: Proofs as Processes. TCS 135(1), 5–9 (1994)

4. van Bakel, S., Cardelli, L., Vigliotti, M.G.: From X to π; Representing the Classical Sequent Calculus in π-calculus. In: CL&C 2008 (2008)
5. van Bakel, S., Lengrand, S., Lescanne, P.: The language formula_image: Circuits, computations and classical logic. In: Coppo, M., Lodi, E., Pinna, G.M. (eds.) ICTCS 2005. LNCS, vol. 3701, pp. 81–96. Springer, Heidelberg (2005)
6. van Bakel, S., Lescanne, P.: Computation with Classical Sequents. MSCS 18, 555–609 (2008)
7. Barendregt, H.: The Lambda Calculus: its Syntax and Semantics. North-Holland, Amsterdam (1984)
8. Barendregt, H.P., Kennaway, R., Klop, J.W., Sleep, M.R.: Needed Reduction and Spine Strategies for the Lambda Calculus. I&C 75(3), 191–231 (1987)
9. Bellin, G., Scott, P.J.: On the pi-Calculus and Linear Logic. TCS 135(1), 11–65 (1994)
10. Bloo, R., Rose, K.H.: Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection. In: CSN 1995 – Computer Science in the Netherlands, pp. 62–72 (1995)
11. Church, A.: A note on the entscheidungsproblem. JSL 1(1), 40–41 (1936)
12. Gentzen, G.: Investigations into logical deduction. In: Szabo, M.E. (ed.) The Collected Papers of Gerhard Gentzen, p. 68, 1935. North Holland, Amsterdam (1969)
13. Gentzen, G.: Untersuchungen über das Logische Schliessen. Mathematische Zeitschrift 39, 176–210, 405–431 (1935)
14. Goubault-Larrecq, J.: A Few Remarks on SKInT. RR-3475, INRIA Rocquencourt (1998)
15. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
16. Honda, K., Yoshida, N.: On the Reduction-based Process Semantics. TCS 151, 437–486 (1995)
17. Honda, K., Yoshida, N., Berger, M.: Control in the π-Calculus. In: CW 2004 (2004)
18. Milner, R.: Function as processes. MSCS 2(2), 269–310 (1992)
19. Parigot, M.: An algorithmic interpretation of classical natural deduction. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 190–201. Springer, Heidelberg (1992)
20. Sangiorgi, D.: Expressing Mobility in Process Algebra: First Order and Higher order Paradigms. PhD thesis, Edinburgh University (1992)
21. Sangiorgi, D.: An Investigation into Functions as Processes. In: Main, M.G., Melton, A.C., Mislove, M.W., Schmidt, D., Brookes, S.D. (eds.) MFPS 1993. LNCS, vol. 802, pp. 143–159. Springer, Heidelberg (1994)
22. Sangiorgi, D.: Lazy functions and processes. RR2515, INRIA, Sophia-Antipolis (1995)
23. Sangiorgi, D., Walker, D.: The Pi-Calculus. Cambridge University Press, Cambridge (2003)
24. Sestoft, P.: Standard ML on the Web server. Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Denmark (1996)
25. Thielecke, H.: Categorical Structure of Continuation Passing Style. PhD thesis, University of Edinburgh (1997)
26. Urban, C.: Classical Logic and Computation. PhD thesis, University of Cambridge (2000)
27. Urban, C., Bierman, G.M.: Strong normalisation of cut-elimination in classical logic. FI 45(1,2), 123–155 (2001)
28. de Vries, F.-J.: Böhm trees, bisimulations and observations in lambda calculus. In: FLP 1997, pp. 230–245 (1997)

Encoding Asynchronous Interactions Using Open Petri Nets[⋆]

Paolo Baldan[1], Filippo Bonchi[2,3], and Fabio Gadducci[3]

[1] Dipartimento di Matematica Pura e Applicata, Università di Padova
[2] Centrum voor Wiskunde en Informatica, Amsterdam
[3] Dipartimento di Informatica, Università di Pisa

Abstract. We present an encoding for (bound) processes of the asynchronous CCS with replication into open Petri nets: ordinary Petri nets equipped with a distinguished set of *open* places. The standard token game of nets models the reduction semantics of the calculus; the exchange of tokens on open places models the interactions between processes and their environment. The encoding preserves strong and weak CCS asynchronous bisimilarities: it thus represents a relevant step in establishing a precise correspondence between asynchronous calculi and (open) Petri nets. The work is intended as fostering the technology transfer between these formalisms: as an example, we discuss how some results on expressiveness can be transferred from the calculus to nets and back.

Keywords: Asynchronous calculi, bisimilarity, decidability, open Petri nets.

1 Introduction

Distributed systems often rely on asynchronous communication, where the operation of sending messages is non-blocking: a process may send a message without any agreement with the receiver, and continue its execution while the message travels to destination. After the introduction of the *asynchronous π-calculus* [1, 2], many process calculi have been proposed that embody some asynchronous communication mechanism (see [3–5], among others). Due to the asymmetry between sending and receiving, behavioural equivalences for asynchronous systems (see e.g. [4, 6–9]) exhibit subtle differences with respect to their synchronous counterparts. Indeed, since sending is non-blocking, an external observer (interacting with a system by message exchanges) cannot know if a message has been received and thus message reception is considered, to some extent, unobservable.

In this paper we aim at establishing a formal correspondence between asynchronous calculi and Petri nets [10]. Perhaps due to their longevity, Petri nets are the best known and most widely used formalism for the visual specification of distributed systems. Besides being used in countless verification tools, suitable

[⋆] Partly supported by the EU FP6-IST IP 16004 SEnSOria and carried out during the second author's tenure of an ERCIM "Alain Bensoussa" Fellowship Programme.

net instances have been successfully adopted as a specification domain in many areas: for their relevance, we mention web services and workflow nets [11].

Petri nets exhibit both synchronous and asynchronous features in their behaviour. On the one hand, a transition having more than one place in its pre-set can be seen as a synchronisation point: it can be executed only if all the needed tokens are available in its pre-set at the same time. On the other hand, a token is first produced by a transition and it then remains available until another transition consumes it, in an asynchronous fashion.

The correspondence between synchronous calculi and Petri nets has been thoroughly investigated (see e.g. [12–16]). Typically, a set of operations is identified on a class of nets, and used for a denotational mapping of the calculus. The archetypal example is maybe the *simple calculus of nets* [13]: each occurrence of a CCS prefix in a process is mapped into a net transition, labelled by the name of the corresponding channel; and basic transitions are used for the parallel and non-deterministic operators. The resulting encodings are often quite syntactical, and forced to restrict their attention to the finite fragment of a calculus. More set-theoretical encodings, basically mapping each process onto a net whose places represent its sub-processes, whilst transitions simulate the control flow of the process, were considered [17]. The dichotomy is denoted [18] as *label* vs *location* oriented encoding, where it is argued that nets with inhibitor arcs should be considered for calculi equipped with both sums and restrictions.

Here we focus on the relation between *asynchronous* calculi and Petri nets. We propose an approach to process encoding which differs from those outlined above, as it relies on *reduction semantics* [19, 20] and *open nets* [21–23]. Open nets are ordinary P/T Petri nets equipped with a set of *open places*, i.e., a set of places visible from the environment: a net may then interact with the environment by exchanging tokens on these places. Open nets are closely related to previous approaches to reactivity and compositionality for Petri nets (see, e.g., [24–27], to mention a few), which, interestingly enough, have been often inspired by the search of encodings of calculi into nets or, more generally, by the investigation of the relation between Petri nets and process calculi.

Specifically, we encode an asynchronous variant of CCS [7] into open nets, in such a way that each process reduction corresponds to a transition firing in the net, and vice versa. The key idea is to exploit openness of places in order to account for name restriction. The free names of a process correspond to the open places of the associated net, and message exchanging between a process and the environment on a channel corresponds to token exchanging on open places. As the set of places in a net is fixed, the encoding applies to *bound* processes, i.e., processes where no restriction operator occurs under the scope of a replication (thus avoiding the generation of an unbounded number of restricted names).

Summarizing, the main features of our encoding are

1. it preserves the structural congruence of processes,
2. a bijective relation holds between process reductions and net firings,

3. the interaction between processes and environment is naturally modeled by the built-in interaction mechanism of open nets, thus formalising the fact that net interaction on open places is eminently *asynchronous*;
4. it preserves and reflects both strong and weak asynchronous bisimilarity.

As far as we know, the latter is the first result of this kind, raising the correspondence between reductions and firing steps up-to the level of asynchronous observational equivalences. Furthermore, it seems noteworthy that, while we consider the asynchronous equivalences for the calculus, the equivalences for open nets exploit the standard (either weak or strong) bisimulation game.

We believe that our encoding and its properties establish the fundamental correspondence between asynchronous calculi and open nets, thus paving the way for a fruitful "technology transfer" between the two formalisms.

In this paper, we exploit the encoding of (bound) asynchronous CCS processes into open nets in order to answer some questions about expressiveness of (fragments of) the two models. In an independent work, the recent paper [16], building upon [28], offers some results concerning the expressive power of restriction and its interplay with replication in synchronous calculi. Here we prove that analogous results can be given for asynchronous calculi. We first show that for bound asynchronous CCS processes strong and weak asynchronous bisimilarities (as well as many other behavioural equivalences) are undecidable. Exploiting the encoding, we immediately obtain that these bisimilarities are undecidable also for open nets. This fact falls outside the known undecidability of bisimilarity for Petri nets [29], as we only observe the interaction with the environment: internal transitions are indistinguishable for strong equivalences and unobservable for weak equivalences (e.g., all standard, closed Petri nets are weakly bisimilar in our setting). In the other direction, using the fact that reachability is decidable for Petri nets, through the encoding we prove that reachability and convergence are decidable for bound asynchronous CCS (which, thus, is not Turing powerful).

As mentioned before, in the study of the relation between Petri nets and process calculi, asynchronous calculi has received less attention than synchronous ones. In general terms, most of the proposals we are aware of put their emphasis on the preservation of the operational behaviour, while behavioural equivalences are seldom studied. This is e.g. the pattern followed in [30], which considers possible encodings of the join calculus, where communication is asynchronous, into Petri nets. In particular, the fragment of the join calculus with no name passing and process generation is shown to correspond to ordinary P/T Petri nets, while, in order to encode wider classes of join processes, high-level Petri nets, ranging from coloured nets to dynamic nets must be considered. The encoding share some ideas with ours, e.g., the fact that Petri net places are partitioned into public and private places, even if it does not tackle the relations between process and net behavioural equivalences. Some related work has been done in the direction of encoding several brands of coordination languages, where processes communicate through shared dataspaces, as Petri nets. The papers [30, 31] exploit the encoding to compare the expressiveness of Linda-like calculi with various communication primitives. In [32] an encoding of KLAIM, a Linda-like language with primitives for mobile computing, into high-level Petri nets is provided. The long-term goal there is to reuse in the context of KLAIM the techniques available for Petri net verification. Concrete results in this direction are obtained in [33], where finite control π-calculus processes are encoded as safe Petri nets and verified using an unfolding-based technique.

The paper is structured as follows. In Section 2 we recall the syntax and the reduction semantics of asynchronous CCS, further presenting the strong and weak (barbed) bisimilarities for the calculus. Section 3 recalls open nets and their equivalences. Section 4 is the core of the paper: it presents the encoding from bound processes of asynchronous CCS into open nets, proving that the encoding preserves and reflects the operational as well as the observational semantics of the calculus. Section 5 discusses some expressiveness issues for the considered models, taking advantage from the encoding. Finally, Section 6 draws some conclusions and provides pointers to future works.

2 Asynchronous CCS

Differently from synchronous calculi, where messages are simultaneously sent and received, in asynchronous communication the messages are sent and travel through some media until they reach destination. Thus sending is not blocking (i.e., a process may send even if the receiver is not ready to receive), while receiving is (processes must wait until a message becomes available). Observations reflect the asymmetry: since sending is non-blocking, receiving is unobservable.

This section introduces asynchronous CCS as a fragment of asynchronous π-calculus (with no name passing). We adopt the presentation in [6] that allows non-deterministic choice for input prefixes (a feature missing in [4, 7]).

Definition 1 (processes). *Let $\mathcal{N}$ be a set of* names*, ranged over by $a, b, c, \ldots$, and $\tau \notin \mathcal{N}$. A* process *P is a term generated by the (mutually recursive) syntax*

$$P ::= M,\ \bar{a},\ (\nu a)P,\ P_1 \mid P_2,\ !_a.P \qquad M ::= \mathsf{0},\ \mu.P,\ M_1 + M_2$$

for μ ranging over $\{\tau\} \cup \mathcal{N}$. We let $P, Q, R, \ldots$ range over the set Proc of processes, and $M, N, O \ldots$ range over the set Sum of summations.

The main difference with standard CCS [34] is the absence of output prefixes. The occurrence of an unguarded $\bar{a}$ indicates a message that is available on some communication media named a, and it disappears whenever it is received.

We assume the standard definitions for the set of free names of a process P, denoted by $fn(P)$. Similarly for α-convertibility w.r.t. the *restriction* operators $(\nu a)P$: the name a is restricted in P, and it can be freely α-converted. *Structural equivalence* ($\equiv$) is the smallest congruence induced by the axioms in Figure 1. The behaviour of a process P is then described as a relation over processes up to $\equiv$, obtained by closing a set of rules under structural congruence.

Definition 2 (reduction semantics). *The* reduction relation *for processes is the relation $R_A \subseteq Proc \times Proc$ inductively defined by the following set of rules*

$$a.P + M \mid \bar{a} \to P \qquad \tau.P + M \to P \qquad !_a.P \mid \bar{a} \to P \mid !_a.P$$

$$P \mid Q = Q \mid P \qquad P \mid (Q \mid R) = (P \mid Q) \mid R \qquad P \mid \mathsf{0} = P$$
$$M{+}N = N{+}M \qquad M{+}(N{+}O) = (M{+}N){+}O \qquad M{+}\mathsf{0} = M \qquad N{+}N = N$$
$$(\nu a)(\nu b)P = (\nu b)(\nu a)P \qquad (\nu a)(P \mid Q) = P \mid (\nu a)Q \ \text{ for } a \notin \mathit{fn}(P) \qquad (\nu a)\mathsf{0} = \mathsf{0}$$
$$(\nu a)(M + \mu.P) = M + \mu.(\nu a)P \ \text{ for } a \notin \mathit{fn}(M + \mu.\mathsf{0})$$

Fig. 1. The set of structural axioms

$$\frac{P \to Q}{(\nu a)P \to (\nu a)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R}$$

and closed under $\equiv$*, where* $P \to Q$ *means that* $\langle P, Q\rangle \in R_A$*. As usual, we let* $\Rightarrow$ *denote the reflexive and transitive closure of* $\to$*.*

The first rule denotes the reception of a message, possibly occurring inside a non-deterministic context: the process $a.P$ is ready to receive a message along channel a; it then receives message $\bar{a}$ and proceeds as P. The second rule represents an internal computation, while on the third rule the replication of a process P occurs after a message is received on a. The latter rules state the closure of the reduction relation w.r.t. the operators of restriction and parallel composition.

A difference w.r.t. the standard syntax of the asynchronous calculus proposed in [6] is the use of guarded input replication $!_a.P$ instead of pure replication $!M$ (see, e.g., [35] which shows that for the synchronous π-calculus, this restriction does not affect the expressiveness of the calculus). Since we plan to later use our encoding to study the concurrency properties of asynchronous interactions, this choice appears more reasonable. Indeed, unguarded replication has an (unrealistic) infinitely branching behaviour when considering concurrent semantics: just think of process $!\tau.\bar{a}$. We also remark that, at the price of a slight complication of the presentation, the results in the paper could be easily extended to a calculus with replication for guarded sums, i.e., allowing for terms of the kind $!\Sigma_{i \in I} a_i.P_i$.

As for the structural axioms, we added sum idempotency and an axiom schema for distributing the restriction under the sum: neither of them is changing the reduction relation, whilst they simplify our encoding of processes into nets.

2.1 Behavioural Equivalences

The main difference with the synchronous calculus lies in the notion of *observation*. Since sending messages is non-blocking, an external observer can just send messages to a system without knowing if they will be received or not. For this reason receiving should not be observable and thus *barbs*, i.e., basic observations on processes, take into account only outputs.

Definition 3 (barb). *Let* P *be a process. We say that* P satisfies the strong barb $\bar{a}$*, denoted* $P \downarrow \bar{a}$*, if there exists a process* Q *such that* $P \equiv \bar{a} \mid Q$*.*

Similarly, P satisfies the weak barb $\bar{a}$*, denoted* $P \Downarrow \bar{a}$*, if* $P \Rightarrow Q$ *and* $Q \downarrow \bar{a}$*.*

Now, strong and weak barbed bisimulation can be defined as in the synchronous case [20], but taking into account only output barbs.

Definition 4 (barbed bisimulation). *A symmetric relation $R \subseteq Proc \times Proc$ is a strong barbed bisimulation if whenever $(P, Q) \in R$ then*

1. *if $P \downarrow \bar{a}$ then $Q \downarrow \bar{a}$,*
2. *if $P \rightarrow P'$ then $Q \rightarrow Q'$ and $(P', Q') \in R$.*

Strong barbed bisimilarity $\sim$ is the largest strong barbed bisimulation.

Weak barbed bisimulation and weak barbed bisimilarity $\approx$ are defined analogously by replacing $\downarrow \bar{a}$ with $\Downarrow \bar{a}$ and $\rightarrow$ with $\Rightarrow$.

Strong (weak) barbed bisimilarities are not congruences. Indeed, $a.\bar{b} \sim \mathtt{0}$ (and $a.\bar{b} \approx \mathtt{0}$), since neither process can perform any transition, but when inserted into the context $- \mid \bar{a}$, the former can perform a transition, while the latter cannot. Behavioural equivalences which are congruences are obtained as follows.

Definition 5 (barbed equivalence). *Let P, Q be processes. They are* strongly barbed equivalent*, denoted $P \sim_b Q$, if $P \mid S \sim Q \mid S$ for all processes S.*

Similarly, they are weakly barbed equivalent*, denoted $P \approx_b Q$, if $P \mid S \approx Q \mid S$ for all processes S.*

An alternative characterization of barbed equivalence considers *output transitions* and the closure w.r.t. the parallel composition with outputs in the bisimulation game. *Strong* and *weak output transitions* are defined as follows: we respectively write $P \xrightarrow{\bar{a}} Q$ if $P \equiv \bar{a} \mid Q$, and $P \overset{\bar{a}}{\Rightarrow} Q$ if $P \Rightarrow P' \xrightarrow{\bar{a}} Q' \Rightarrow Q$.

Definition 6 (1-bisimulation). *A symmetric relation $R \subseteq Proc \times Proc$ is a strong 1-bisimulation if whenever $(P, Q) \in R$ then*

1. *$\forall a \in \mathcal{N}. (P \mid \bar{a}, Q \mid \bar{a}) \in R$,*
2. *if $P \rightarrow P'$ then $Q \rightarrow Q'$ and $(P', Q') \in R$,*
3. *if $P \xrightarrow{\bar{a}} P'$ then $Q \xrightarrow{\bar{a}} Q'$ and $(P', Q') \in R$.*

Strong 1-bisimilarity $\sim_1$ is the largest strong 1-bisimulation.

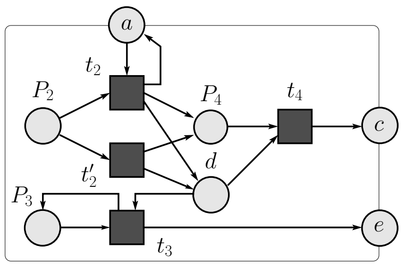
Weak 1-bisimulation and weak 1-bisimilarity $\approx_1$ are defined analogously by replacing $\rightarrow$ with $\Rightarrow$ and $\xrightarrow{\bar{a}}$ with $\overset{\bar{a}}{\Rightarrow}$.

Proposition 1 ([6]). *$\sim_b = \sim_1$ and $\approx_b = \approx_1$.*

Example 1. Consider the processes $P = (\nu d)(!_d.\bar{e} \mid (a.(\bar{a} \mid \bar{d} \mid d.\bar{c}) + \tau.(\bar{d} \mid d.\bar{c})))$ and $Q = (\nu d)(\tau.(d.\bar{c} \mid d.\bar{e} \mid \bar{d}))$. It is not difficult to see that $P \sim_1 Q$. First consider their internal steps: $P \rightarrow (\nu d)(!_d.\bar{e} \mid \bar{d} \mid d.\bar{c})$, while $Q \rightarrow (\nu d)(d.\bar{c} \mid d.\bar{e} \mid \bar{d})$. Notice that $(\nu d)(!_d.\bar{e} \mid \bar{d} \mid d.\bar{c}) \sim_1 (\nu d)(d.\bar{c} \mid d.\bar{e} \mid \bar{d})$, since after one execution the replication operator is stuck.

The process P can also receive on the channel a. Instead of observing the input transitions $\xrightarrow{a}$, in the 1-bisimulation game this behaviour is revealed plugging the processes into $- \mid \bar{a}$. The process $P \mid \bar{a}$ can choose one of the two branches of $+$, but in any case it performs an internal transition becoming $(\nu d)(!_d.\bar{e} \mid \bar{a} \mid \bar{d} \mid d.\bar{c})$. On the other hand, $Q \mid \bar{a}$ performs an internal transition to $(\nu d)(d.\bar{c} \mid d.\bar{e} \mid \bar{d} \mid \bar{a})$, and clearly the resulting states are 1-bisimilar.

Furthermore, consider the process $a.\bar{a}$: it is one of the idiosyncratic features of the asynchronous communication that the equivalence $a.\bar{a} \approx_1 \mathtt{0}$ holds.



$$P = (\nu d)(\underbrace{!_d.\bar{e}}_{P_3} \mid \underbrace{\overbrace{(\underbrace{a.(\bar{a} \mid \bar{d} \mid \overbrace{d.\bar{c}}^{P_4})}_{M} + \underbrace{\tau.(\bar{d} \mid \overbrace{d.\bar{c}}^{P_4})}_{M'})}^{P_1}}_{P_2})$$

$$t_2 = (P_2, M) \qquad t_3 = P_3$$
$$t'_2 = (P_2, M') \qquad t_4 = (P_4, P_4)$$

Fig. 2. An open net encoding a process P

3 Open Nets

Differently from process calculi, standard Petri nets do not exhibit an interactive behaviour, i.e., they are intended to model concurrent systems considered as *standalone*. This section reviews *open nets*, i.e., ordinary P/T nets with a distinguished set of *open places*: they represent the interfaces through which the environment interacts with a net, by putting and removing tokens (see [21–23])[1].

Given a set X, let $X^{\oplus}$ denote the free commutative monoid over X. An element $m \in X^{\oplus}$, called a *multiset* over X, is often viewed as a function from X to $\mathbb{N}$ (the set of natural numbers) that associates a multiplicity with every element of X. We write $m_1 \subseteq m_2$ if $\forall x \in X$, $m_1(x) \leq m_2(x)$. If $m_1 \subseteq m_2$, the multiset $m_2 \ominus m_1$ is defined as $\forall x \in X\ m_2 \ominus m_1(x) = m_2(x) - m_1(x)$. The symbol 0 denotes the empty multiset.

Definition 7 (Open P/T Petri net). *An* open (P/T Petri) net *is a tuple* $N = (S, T, {}^{\bullet}(.), (.)^{\bullet}, O)$ *where* S *is the set of places,* T *is the set of transitions,* ${}^{\bullet}(.), (.)^{\bullet} : T \rightarrow S^{\oplus}$ *are functions mapping each transition to its pre- and post-set, and* $O \subseteq S$ *is the set of* open places.

Example 2. Figure 2 shows an open net: as usual, circles represent places and rectangles transitions. Arrows from places to transitions represent function ${}^{\bullet}(.)$, while arrows from transitions to places represent $(.)^{\bullet}$. An open net is enclosed in a box and open places are on the border of such a box. Additionally, any open place has a name which is placed inside the corresponding circle: in this example these are chosen from the set $\mathcal{N}$. Also transitions and closed places are provided with an identifier, yet positioned outside of the corresponding circle or square: their precise meaning, as well as an explanation of the process on the right, is provided later on. The open place identified by a and the closed place identified by P_2 form e.g. the pre-set of transition t_2; its post-set is formed by a, P_4 and d.

Given an open net N, we consider the set of *interactions* (ranged over by i) $\mathcal{I}_N = \{s^+, s^- \mid s \in O\}$. The set of *labels* (ranged over by l) consists in $\{\tau\} \uplus \mathcal{I}_N$.

[1] Differently from [21], yet with no loss of generality, we do not distinguish between input and output open places, tailoring equivalences accordingly.

Table 1. Operational Semantics of open nets

$$(\textsc{tr})\ \frac{t \in T}{{}^\bullet t \oplus c \xrightarrow{\tau} t^\bullet \oplus c} \qquad (\textsc{in})\ \frac{s \in O}{m \xrightarrow{s^+} m \oplus s} \qquad (\textsc{out})\ \frac{s \in O}{m \xrightarrow{s^-} m \ominus s}$$

The operational semantics of open nets is expressed by the rules on Table 1, where we write ${}^\bullet t$ and $t^\bullet$ instead of ${}^\bullet(t)$ and $(t)^\bullet$. The rule (TR) is the standard rule of P/T nets (seen as multiset rewriting) modelling internal transitions. The other two rules model interactions with the environment: in any moment a token can be inserted in (rule (IN)) or removed from (rule (OUT)) an open place.

Weak transitions are defined as usual, i.e., $\overset{\tau}{\Rightarrow}$ denotes the reflexive and transitive closure of $\xrightarrow{\tau}$ and $\overset{i}{\Rightarrow}$ denotes $\overset{\tau}{\Rightarrow}\xrightarrow{i}\overset{\tau}{\Rightarrow}$.

Definition 8 (Strong and weak bisimilarity). *Let N_1, N_2 be open nets with the same interface i.e., $O_1 = O_2$. A* strong bisimulation *between N_1 and N_2 is a relation over markings $\mathcal{R} \subseteq S_1^\oplus \times S_2^\oplus$ such that if $(m_1, m_2) \in \mathcal{R}$ then*

– *if $m_1 \xrightarrow{l} m_1'$ in N_1 then $m_2 \xrightarrow{l} m_2'$ in N_2 and $(m_1', m_2') \in \mathcal{R}$;*
– *if $m_2 \xrightarrow{l} m_2'$ in N_2 then $m_1 \xrightarrow{l} m_1'$ in N_1 and $(m_1', m_2') \in \mathcal{R}$.*

Two markings $m_1 \in S_1^\oplus$ and $m_2 \in S_2^\oplus$ are bisimilar, written $m_1 \sim m_2$, if $(m_1, m_2) \in \mathcal{R}$ for some strong bisimulation $\mathcal{R}$.

Weak bisimilarity $\approx$ *is defined analogously by replacing strong transitions $\xrightarrow{l}$ by weak transitions $\overset{l}{\Rightarrow}$.*

In order to ease the intuition behind net bisimilarity, nets must be thought of as black boxes, where only the interfaces (i.e., the open places) are visible. Two nets are weak bisimilar if they cannot be distinguished by an external observer that may only insert and remove tokens in open places. For strong bisimilarity the observer can also see the occurrence of internal transitions.

4 From Processes to Nets

Our encoding is restricted to *bound processes*, i.e., processes where restrictions never occur under replications. Any bound process is structurally equivalent to a process of the shape $(\nu X)P$, for P a restriction-free process and $X \subseteq fn(P)$. In the following, we implicitly assume that bound processes are always in this shape. Moreover, we call *basic processes* those bound processes described by the following syntax (where P must be restriction-free)

$$B ::= \bar{a},\ \ !_a.P,\ \ M$$

Observe that any restriction-free process is just the parallel composition of several basic processes. In our net encoding, basic processes become places, marked with a number of tokens equal to the number of occurrences of the corresponding basic process in the parallel composition. Hereafter, we use $[P]$ to denote the equivalence class of process P w.r.t. structural equivalence.

$$\begin{array}{llll} Pl(P_1 \mid P_2) = Pl(P_1) \cup Pl(P_2) & & Pl(\bar{a}) & = \{a\} \\ Pl(!_a.P) & = \{a\} \cup \{[!_a.P]\} \cup Pl(P) & Pl_s(a.P) & = \{a\} \cup Pl(P) \\ Pl(M) & = \begin{cases} \emptyset & \text{if} M \equiv 0 \\ \{[M]\} \cup Pl_s(M) & \text{otherwise} \end{cases} & \begin{array}{l} Pl_s(\tau.P) \\ Pl_s(M_1 + M_2) \end{array} & \begin{array}{l} = Pl(P) \\ = Pl_s(M_1) \cup Pl_s(M_2) \end{array} \end{array}$$

Fig. 3. The place functions

Definition 9 (places). *The (mutually recursive) functions $Pl(P)$ and $Pl_s(M)$, associating to each restriction-free process and summation, respectively, a set of places, are defined by structural induction according to the rules in Figure 3.*

The set of places associated with a restriction-free process P thus consists in the set of names of P together with the set of equivalence classes of basic sub-processes of P of shape $!_a.P$ and M, for $M \not\equiv 0$. Notice that a basic process $\bar{a}$ is simply encoded by a token in the place corresponding to name a. We use $\mathbf{m}(P)$ to denote the *marking* associated with a restriction-free process P, which, assuming $\mathbf{m}(0)$ to be the empty marking, is defined by

$$\mathbf{m}(P_1 \mid P_2) = \mathbf{m}(P_1) \oplus \mathbf{m}(P_2) \qquad \mathbf{m}(\bar{a}) = a \qquad \mathbf{m}(M) = [M] \qquad \mathbf{m}(!_a.P) = [!_a.P]$$

Definition 10 (transitions, pre- and post-conditions). *Let M be a summation. The set of atoms of M, denoted $Atom(M)$, is inductively defined as:*

$$Atom(0) = \emptyset,\ Atom(\mu.P) = \{[\mu.P]\},\ Atom(M_1 + M_2) = Atom(M_1) \cup Atom(M_2)$$

for $\mu \in \mathcal{N} \cup \{\tau\}$. Now, let P be a restriction-free process. The set of transitions of P, denoted $T(P)$, is inductively defined as:

$$\begin{array}{ll} T(0) = \emptyset & T(P_1 \mid P_2) = T(P_1) \cup T(P_2) \\ T(\bar{a}) = \emptyset & T(!_a.P) = \{[!_a.P]\} \cup T(P) \\ \multicolumn{2}{c}{T(M) = (\{[M]\} \times Atom(M)) \cup \bigcup_{[\mu.P] \in Atom(M)} T(P)} \end{array}$$

The pre- and post-conditions ${}^\bullet(.)_P, (.)^\bullet_P : T(P) \to Pl(P)^\oplus$ are defined as follows.

$${}^\bullet t_P = \begin{cases} \{[M]\} \oplus fn(\mu.0) & \text{if } t = \langle [M], [\mu.P] \rangle \\ \{[!_a.P]\} \oplus \{a\} & \text{if } t = [!_a.P] \end{cases} \quad t^\bullet_P = \begin{cases} \mathbf{m}(P) & \text{if } t = \langle [M], [\mu.P] \rangle \\ \{[!_a.P]\} \oplus \mathbf{m}(P) & \text{if } t = [!_a.P] \end{cases}$$

Intuitively, transitions mimic the control flow of a process, passing the token between its sequential components (its basic processes).

We next introduce the net encoding for bound processes. In order to get a full correspondence between behavioural equivalences, we need to encode our processes parametrically w.r.t. a set of names Γ, as usually necessary in graphical encodings of process calculi (based e.g. on DPO rewriting [36] or bigraphs [37]).

Definition 11. *Let P be a restriction-free process and X, Γ disjoint sets of names such that $fn((\nu X)P) \subseteq \Gamma$. Then, the open net $[\![(\nu X)P]\!]_\Gamma$ is the tuple*

$$(Pl(P) \cup \Gamma, T(P), {}^\bullet(.)_P, (.)^\bullet_P, \Gamma)$$

First we take the net associated with P, consisting of those places and transitions as given in Definitions 9 and 10. Then the set of places is extended with those names belonging to Γ (the assumption $\Gamma \cap X = \emptyset$ avoids that by chance a restricted name in X is overlapped). Finally, we take as open those places corresponding to a name in Γ (by hypothesis this includes the free names of $(\nu X)P$).

Example 3. The net in Example 2 is the encoding $[\![P]\!]_\Gamma$ of the process P in Example 1, with $\Gamma = \{a, c, e\}$. The places identified by a, c, and e correspond to the free names of the process, while place d corresponds to the restricted name d. The remaining places correspond to (equivalence classes of) the basic sub-processes of P: for example, the place P_2 corresponds to the sub-process $[a.(\bar{a} \mid \bar{d} \mid d.\bar{c}) + \tau.(\bar{d} \mid d.\bar{c})]$, while the place P_4 to $[d.\bar{c}]$. The transition t_2 encodes the pair (P_2, M), for M the atom $[a.(\bar{a} \mid \bar{d} \mid d.\bar{c})]$. It may fire in presence of a token in a and P_2: it roughly represents the reduction $P_2 \mid \bar{a} \rightarrow \bar{a} \mid \bar{d} \mid d.\bar{c}$.

We close this section by establishing a first correspondence result.

Proposition 2. *Let P, Q be bound processes and Γ a set of names such that $fn(P) \cup fn(Q) \subseteq \Gamma$. Then $P \equiv Q$ iff the open nets $[\![P]\!]_\Gamma$ and $[\![Q]\!]_\Gamma$ are isomorphic.*

4.1 Relating Asynchronous CCS and Open Nets

This section shows that our encoding preserves and reflects process reductions, as well as strong and weak behavioural equivalences. In order to state these results, we must define a correspondence between the set of processes reachable from P, hereafter denoted by $reach(P)$, and markings over the net $[\![P]\!]_\Gamma$. For this, we need a technical lemma concerning reductions for bound processes.

Lemma 1. *Let P be a restriction-free process and X a set of names. Then,*

1. *$(\nu X)P \rightarrow Q$ iff $Q \equiv (\nu X)Q_1$ and $P \rightarrow Q_1$;*
2. *if $P \rightarrow Q$ then $Q \equiv B_1 \mid \ldots \mid B_n$, for B_i's basic sub-processes of P.*

The above lemma tells us that each process Q reachable from a bound process $(\nu X)P$ can be seen as a (possibly empty) marking over the net $[\![(\nu X)P]\!]_\Gamma$. In fact, the set of places of $[\![(\nu X)P]\!]_\Gamma$ contains all the basic sub-processes of P.

Definition 12. *Let P be a bound process and X, Γ disjoint sets of names such that $P \equiv (\nu X)P_1$ (for P_1 restriction-free) and $fn(P) \subseteq \Gamma$. The function $\mathbf{m}_\Gamma^{X,P_1}$: $reach(P) \rightarrow (Pl(P_1) \cup \Gamma)^\oplus$ maps any process $Q \equiv (\nu X)Q_1$ (for Q_1 restriction-free) reachable from P into the marking $\mathbf{m}(Q_1)$ over the net $[\![P]\!]_\Gamma$.*

Example 4. Recall $P = (\nu d)P_1$, for $P_1 = !_d.\bar{e} \mid (a.(\bar{a} \mid \bar{d} \mid d.\bar{c}) + \tau.(\bar{d} \mid d.\bar{c}))$, from Example 1. Let $X = \{d\}$ and $\Gamma = \{a, c, e\}$. The function $\mathbf{m}_\Gamma^{X,P_1}$ maps the processes reachable from P into markings of $[\![P]\!]_\Gamma$, that is, the net in Figure 2. E.g., P is mapped to $P_2 \oplus P_3$; $(\nu d)(!_d.\bar{e} \mid \bar{d} \mid d.\bar{c})$ is mapped to $P_3 \oplus d \oplus P_4$; $(\nu d)(!_d.\bar{e} \mid \bar{c})$ is mapped to $P_3 \oplus c$ and $(\nu d)(\bar{e} \mid d.\bar{c})$ to $e \oplus P_4$.

Once established that any process reachable from a bound process P identifies a marking in the net $[\![P]\!]_\Gamma$, we can state the main correspondence results.

Theorem 1. *Let P be a bound process, and X, Γ disjoint sets of names such that $P \equiv (\nu X)P_1$ (for P_1 restriction-free) and $fn(P) \subseteq \Gamma$. Moreover, let Q be a process reachable from P. Then,*

1. *if $Q \to R$ then $\mathbf{m}_\Gamma^{X,P_1}(Q) \xrightarrow{\tau} \mathbf{m}_\Gamma^{X,P_1}(R)$ in $[\![P]\!]_\Gamma$;*
2. *if $\mathbf{m}_\Gamma^{X,P_1}(Q) \xrightarrow{\tau} m$ in $[\![P]\!]_\Gamma$, then $Q \to R$ for $m = \mathbf{m}_\Gamma^{X,P_1}(R)$.*

The result establishes a bijection between the reductions performed by any process Q reachable from P, and the firings in $[\![P]\!]_\Gamma$ from the marking $\mathbf{m}_\Gamma^{X,P_1}(Q)$, for any restriction-free P_1 such that $P \equiv (\nu X)P_1$.

Such a bijection can then be lifted to a fundamental correspondence between the observational semantics in the two formalisms.

Theorem 2. *Let P, Q be bound processes and X, Y, Γ sets of names (with $X \cap \Gamma = Y \cap \Gamma = \emptyset$) such that $P \equiv (\nu X)P_1$ and $Q \equiv (\nu Y)Q_1$ (for P_1, Q_1 restriction-free) and $fn(P) \cup fn(Q) \subseteq \Gamma$. Then,*

1. *$P \sim_1 Q$ iff $\mathbf{m}_\Gamma^{X,P_1}(P) \sim \mathbf{m}_\Gamma^{Y,Q_1}(Q)$;*
2. *$P \approx_1 Q$ iff $\mathbf{m}_\Gamma^{X,P_1}(P) \approx \mathbf{m}_\Gamma^{Y,Q_1}(Q)$.*

Please note that the markings $\mathbf{m}_\Gamma^{X,P_1}(P)$ and $\mathbf{m}_\Gamma^{X,Q_1}(Q)$ live in the open nets $[\![P]\!]_\Gamma$ and $[\![Q]\!]_\Gamma$, respectively.

Example 5. Consider the net on the left of Figure 4: it corresponds to $[\![Q]\!]_\Gamma$, for $\Gamma = \{a, c, e\}$, $Q = (\nu d)Q_1$ and $Q_1 = \tau.(d.\bar{c} \mid d.\bar{e} \mid \bar{d})$ as in Example 1. Note the presence of the isolated place a: it says that name a does not occur in Q. Now, the left-most place represents the (equivalence class of the) sub-process Q_1; the left-most transition, identified by t, the pair (Q_1, Q_1); and its firing is the execution of the internal transition $Q_1 \to d.\bar{c} \mid d.\bar{e} \mid \bar{d}$, putting a token on the three places representing the restricted name d and the basic sub-processes $Q_2 = d.\bar{c}$ and $Q_3 = d.\bar{e}$.

It is easy to see that $\mathbf{m}_\Gamma^{X,P_1}(P) = P_2 \oplus P_3$ in $[\![P]\!]_\Gamma$ is strongly bisimilar to the marking $\mathbf{m}_\Gamma^{X,Q_1}(Q) = Q_1$ in $[\![Q]\!]_\Gamma$: they clearly induce the same internal behaviour; moreover when the environment inserts a token into a, $P_2 \oplus P_3 \oplus a$ may fire also transition t_2 producing $a \oplus P_4 \oplus d \oplus P_3$, but this is equivalent to the firing of the other internal transition t_2'. From this, it follows that $P \sim_b Q$.

Consider now the net in the right of Figure 4: it corresponds to the encoding of $[\![a.\bar{a}]\!]_{\{a\}}$. It is weakly bisimilar to the encoding $[\![\mathsf{0}]\!]_{\{a\}}$, represented by a net having only an open place, identified by a. In fact, in both cases the only observable action is either placing or removing a token on the open place a, while the execution of the transition $(a.\bar{a}, a.\bar{a})$, consuming and producing a, is unobservable from the environment. It thus holds that $a.\bar{a} \approx_b \mathsf{0}$.

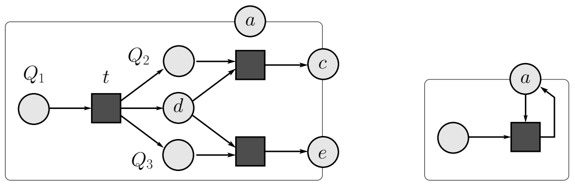


Fig. 4. The net encodings $[\![Q]\!]_\Gamma$ for $\Gamma = \{a, c, e\}$ (left) and $[\![a.\bar{a}]\!]_{\{a\}}$ (right)

5 Technology Transfer on Expressiveness

In this section we discuss some expressiveness issues for asynchronous CCS and open nets, taking advantage from the encoding presented before.

More specifically, we first show that strong and weak bisimilarity of bound processes of asynchronous CCS are undecidable, thus answering a question faced for the synchronous case in the recent [16]. By using Theorem 2, we can deduce that the same result holds for open nets, a fact which was previously unknown.

On the other hand, using the fact that reachability and convergence are decidable for Petri nets, through the encoding we can prove that the same holds for bound asynchronous CCS (which, thus, is not Turing powerful).

5.1 Undecidability of Bisimilarity

The undecidability of bisimilarity for bound CCS processes is proved by reduction to the halting problem for Minsky's two-register machines, adapting a proof technique originally proposed in [29].

Definition 13 (two-register machine). *A* two-register machine *is a triple $\langle r_1, r_2, P\rangle$ where r_1 and r_2 are two registers which can hold any natural number, and the program $P = I_1 \dots I_s$ consists of a sequence of instructions. An* instruction I_i *can be one of the following: for $x \in \{r_1, r_2\}$, $j, k \in \{1, \dots, s+1\}$*

- *$s(x, j)$: increment the value of register x and jump to instruction I_j*
- *$zd(x, j, k)$: if x is zero, then jump to I_j else decrement x and jump to I_k*

The execution of the program starts from instruction I_1, with $r_1 = 0$, $r_2 = 0$ and possibly terminate when the $(s+1)$st instruction is executed.

The idea consists in defining, for any two-register machine program P, a bound process $\gamma(P)$ which "non-deterministically simulates" the computations of the program (starting with null registers). Some "wrongful" computations are possible in the process $\gamma(P)$, not corresponding to a correct computation of the program P (due to the absence of zero-tests in the considered fragment of CCS). Still, this can be used to prove undecidability of (strong and weak) bisimilarity. In fact, given P, a second process $\gamma'(P)$ can be built such that $\gamma(P) \sim \gamma'(P)$ iff program P does not terminate. Therefore, deciding bisimilarity for bound CCS processes would allow to decide the termination of two-register machines. As two-register machines are Turing powerful, we conclude.

Theorem 3. *Strong (weak) 1-bisimilarity of bound processes is undecidable.*

From the properties of the encoding (Theorem 2) we can deduce the same undecidability results for open nets.

Corollary 1. *Strong (weak) bisimilarity is undecidable for open nets.*

In the same way, one can easily prove that various other behavioural equivalences, including failure equivalence (the notion of equivalence considered in [16]) and language equivalence, are undecidable.

5.2 Decidability of Reachability and Convergence

It is immediate to see that for open nets the reachability problem, i.e., the problem of determining whether a given marking is reachable by means of a firing sequence starting from the initial marking, is decidable. In fact, reachability in an open net N can be reduced to reachability in a standard P/T net obtained from N by adding, for any open place s, two transitions t_s^- and t_s^+ which, respectively, freely remove and add tokens from s, i.e., ${}^\bullet t_s^- = {t_s^+}^\bullet = s$ and ${t_s^-}^\bullet = {}^\bullet t_s^+ = 0$.

By exploiting the encoding, we may transfer the result to bound asynchronous CCS, showing that reachability in an open environment, providing any needed message, is decidable.

Proposition 3. *Let P, Q be bound processes. Then the problem of establishing whether there exists an environment R, consisting of the (possibly empty) parallel composition of output messages, such that $P|R \Rightarrow Q$ is decidable.*

In particular, the problem $P \Rightarrow Q$ is decidable. In fact, if $X = \mathit{fn}(P) \cup \mathit{fn}(Q)$, it is easy to see that $P \Rightarrow Q$ iff $(\nu X)P \Rightarrow (\nu X)Q$, and this is in turn equivalent to the existence of a suitable process R such that $R|(\nu X)P \Rightarrow (\nu X)Q$.

Another property which is often considered when studying the expressiveness of process calculi is convergence, i.e., the existence of a terminating computation. We recall such notion below, according to [28].

Definition 14 (convergence). *A process P is called* convergent *if there exists Q such that $P \Rightarrow Q \not\rightarrow$.*

Convergence is clearly decidable for an open net N, as it can be reduced to the existence of a deadlock in the standard P/T net obtained from N by closing all the open places, and this property is known to be decidable for P/T nets [38].

As a consequence the same holds for bound asynchronous CCS.

Proposition 4. *Convergence is decidable for bound processes.*

The paper [16] shows that, in the synchronous case, adding priorities to the language radically changes the situation: bound CCS becomes Turing complete and convergence is thus undecidable. It is easy to show that the same applies to the asynchronous case. The fact that adding priorities makes bound asynchronous CCS Turing complete can be proved by noting that using priorities the encoding of two-counter machines into bound asynchronous CCS can be made deterministic. This is not surprising as, on the Petri net side, priorities are strictly connected to inhibitor arcs, which make Petri nets Turing powerful [39].

6 Conclusions and Further Work

We believe that the relevance of our paper lies in establishing the fundamental correspondence between asynchronous calculi and open nets, as stated by the theorems of Section 4. Indeed, even if our presentation has been tailored over a variant of standard CCS, we feel confident that it can be generalized to other asynchronous calculi as well, at least to those based on a primitive notion of communication, i.e., without either value or name passing. As suggested by the work in [15, 30, 32], the generalisation to calculi with value or name passing looks feasible if one considers more expressive variants of Petri nets, ranging from high-level to reconfigurable/dynamic Petri nets.

We consider such a correspondence quite enlightening, since most of the encodings we are aware of focus on the preservation of some variants of reachability or of the operational behaviour [30–32, 40], while ours allow to establish a correspondence at the observational level.

As a remark, note that the encoding of CCS processes into open nets could be defined in a compositional way, either via a more syntactical presentation (thus losing the preservation of structural congruence) or by the exploiting the composition operation available for open nets. The latter would require to view open nets as cospans (with complex interfaces) in a suitable category [23, 36]. In order to keep the presentation simpler we adopted a direct definition.

We believe that the tight connection between Petri nets and asynchronous calculi allows for a fruitful "technology transfer". We started by showing the undecidability of bisimilarity for bound processes which, through the encoding, is used to prove undecidability of bisimilarity for open nets (where all internal transitions are considered indistinguishable in the strong case and unobservable in the weak case), a previously unknown fact. Analogously, decidability of reachability and convergence for open nets is transferred, through the encoding, to bound asynchronous CCS processes.

We are currently investigating the concurrent semantics for asynchronous CCS. The idea is to consider the *step* semantics for our nets, i.e., where many transitions may fire simultaneously, and then try to distill an adequate equivalence for bound process. Indeed, our initial results are quite encouraging. On the longer run, our hope would then be to lift these equivalences to richer calculi, such as the paradigmatic asynchronous π-calculus [6] and to different behavioural equivalences, including, e.g., failure and testing equivalences [7].

Acknowledgments. We are grateful to Barbara König for enlightening discussions on a preliminary version of this paper and to the anonymous reviewers for their inspiring comments.

References

1. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)

2. Boudol, G.: Asynchrony and the π-calculus. Technical Report 1702, INRIA, Sophia Antipolis (1992)
3. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Trans. Software Eng. 24(5), 315–330 (1998)
4. Castellani, I., Hennessy, M.: Testing theories for asynchronous languages. In: Arvind, V., Sarukkai, S. (eds.) FST TCS 1998. LNCS, vol. 1530, pp. 90–102. Springer, Heidelberg (1998)
5. Ferrari, G., Guanciale, R., Strollo, D.: Event based service coordination over dynamic and heterogeneous networks. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 453–458. Springer, Heidelberg (2006)
6. Amadio, R., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. TCS 195(2), 291–324 (1998)
7. Boreale, M., De Nicola, R., Pugliese, R.: Asynchronous observations of processes. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, pp. 95–109. Springer, Heidelberg (1998)
8. Boreale, M., De Nicola, R., Pugliese, R.: A theory of "may" testing for asynchronous languages. In: Thomas, W. (ed.) FOSSACS 1999. LNCS, vol. 1578, pp. 165–179. Springer, Heidelberg (1999)
9. Rathke, J., Sobocinski, P.: Making the unobservable, unobservable. In: ICE 2008. ENTCS. Elsevier, Amsterdam (2009) (to appear)
10. Reisig, W.: Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (1985)
11. van der Aalst, W.: Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype". BPTrends 3(5), 1–11 (2005)
12. Goltz, U.: CCS and Petri nets. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 334–357. Springer, Heidelberg (1990)
13. Gorrieri, R., Montanari, U.: SCONE: A simple calculus of nets. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 2–31. Springer, Heidelberg (1990)
14. Busi, N., Gorrieri, R.: A Petri net semantics for pi-calculus. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 145–159. Springer, Heidelberg (1995)
15. Devillers, R., Klaudel, H., Koutny, M.: A compositional Petri net translation of general pi-calculus terms. Formal Asp. Comput. 20(4-5), 429–450 (2008)
16. Aranda, J., Valencia, F., Versari, C.: On the expressive power of restriction and priorities in CCS with replication. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 242–256. Springer, Heidelberg (2009)
17. Olderog, E.: Nets, terms and formulas. Cambridge University Press, Cambridge (1991)
18. Busi, N., Gorrieri, R.: Distributed semantics for the π-calculus based on Petri nets with inhibitor arcs. Journal of Logic and Algebraic Programming 78, 138–162 (2009)
19. Berry, G., Boudol, G.: The chemical abstract machine. TCS 96, 217–248 (1992)
20. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
21. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open Petri nets based on deterministic processes. Mathematical Structures in Computer Science 15(1), 1–35 (2004)
22. Milner, R.: Bigraphs for Petri nets. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 686–701. Springer, Heidelberg (2004)

23. Sassone, V., Sobociński, P.: A congruence for Petri nets. In: Ehrig, H., Padberg, J., Rozenberg, G. (eds.) PNGT 2004. ENTCS, vol. 127, pp. 107–120. Elsevier, Amsterdam (2005)
24. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. LNCS, vol. 625. Springer, Heidelberg (1992)
25. Nielsen, M., Priese, L., Sassone, V.: Characterizing behavioural congruences for Petri nets. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 175–189. Springer, Heidelberg (1995)
26. Koutny, M., Esparza, J., Best, E.: Operational semantics for the Petri box calculus. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 210–225. Springer, Heidelberg (1994)
27. Kindler, E.: A compositional partial order semantics for Petri net components. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 235–252. Springer, Heidelberg (1997)
28. Busi, N., Gabbrielli, M., Zavattaro, G.: Comparing recursion, replication, and iteration in process calculi. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 307–319. Springer, Heidelberg (2004)
29. Jancar, P.: Undecidability of bisimilarity for Petri nets and some related problems. TCS 148(2), 281–301 (1995)
30. Buscemi, M., Sassone, V.: High-level Petri nets as type theories in the join calculus. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 104–120. Springer, Heidelberg (2001)
31. Busi, N., Zavattaro, G.: A process algebraic view of shared dataspace coordination. J. Log. Algebr. Program. 75(1), 52–85 (2008)
32. Devillers, R., Klaudel, H., Koutny, M.: A Petri net semantics of a simple process algebra for mobility. In: Baeten, J., Phillips, I. (eds.) EXPRESS 2005. ENTCS, vol. 154, pp. 71–94. Elsevier, Amsterdam (2006)
33. Meyer, R., Khomenko, V., Strazny, T.: A practical approach to verification of mobile systems using net unfoldings. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 327–347. Springer, Heidelberg (2008)
34. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
35. Sangiorgi, D.: On the bisimulation proof method. Mathematical Structures in Computer Science 8(5), 447–479 (1998)
36. Gadducci, F.: Graph rewriting and the π-calculus. Mathematical Structures in Computer Science 17, 1–31 (2007)
37. Milner, R.: Pure bigraphs: Structure and dynamics. Information and Computation 204, 60–122 (2006)
38. Esparza, J., Nielsen, M.: Decidability issues for Petri nets - a survey. Journal Inform. Process. Cybernet. EIK 30(3), 143–160 (1994)
39. Agerwala, T., Flynn, M.: Comments on capabilities, limitations and “correctness” of Petri nets. Computer Architecture News 4(2), 81–86 (1973)
40. Busi, N., Zavattaro, G.: Expired data collection in shared dataspaces. TCS 3(298), 529–556 (2003)

Distributed Asynchronous Automata

Nicolas Baudru

Laboratoire d'Informatique Fondamentale de Marseille — eRISCS group
Aix-Marseille Universités, 163, avenue de Luminy, F-13288 Marseille Cedex 9, France
nicolas.baudru@lif.univ-mrs.fr

Abstract. Asynchronous automata are a model of communication processes with a distributed control structure, global initializations and global accepting conditions. The well-known theorem of Zielonka states that they recognize exactly the class of regular Mazurkiewicz trace languages. In this paper we study the particular case of *distributed* asynchronous automata, which require that the initializations and the accepting conditions are distributed as well: every process chooses an initial *local* state and stops in a final *local* state independently from each other. We characterize effectively the regular trace languages recognized by these automata. Also, we present an original algorithm to build, if it is possible, a non-deterministic distributed asynchronous automaton that recognizes a given regular trace language. Surprisingly, this algorithm yields a new construction for the more general problem of the synthesis of asynchronous automata from regular trace languages that subsumes all existing ones in terms of space complexity.

Introduction

Asynchronous automata [17] modelize concurrent systems that use a mechanism based on shared variables to communicate. They consist of a set of processes with a distributed control structure, global initializations and global accepting conditions. During an execution the processes synchronize on shared variables, which are called simply actions in our setting: all actions a are associated with a subset of processes which agree jointly on a move on reading a. On the other hand, the theory of Mazurkiewicz traces [4] provides mathematical tools for the formal study of concurrent systems. In this theory, the actions of a concurrent system are equipped with an independent relation between actions that do not share any process.

One of the major contributions in the theory of Mazurkiewicz traces characterizes regular trace languages by means of asynchronous automata [17]. This result, known as Zielonka's theorem, and related techniques are fundamental tools in concurrency theory. For instance they are useful to compare the expressive power of classical models of concurrency such as Petri nets, asynchronous systems, and concurrent automata [12,16]. These methods have been also adapted to the construction of communicating finite-state machines from regular sets of message sequence charts [1,7,8].

For twenty years, several constructive proofs of Zielonka's theorem have been developed. All these constructions, which build asynchronous automata from regular trace languages, are quite involved and yield an exponential explosion of the number of states in each process [6,13]. To our knowledge, the complexity of this problem is still unknown, and it was asked in [5] whether a simpler construction could be designed.

In this paper, we are interested in the particular case of *distributed* asynchronous automata (DAA for short). In a DAA, the initializations and the accepting conditions are also distributed: each process chooses a local initial state and stops in a local final state independently from other processes. We introduce them as an interesting tool to develop alternative proofs of Zielonka's result. In particular, we present a technique based on simple compositions/decompositions of DAAs that results in the construction of a "small" non-deterministic asynchronous automaton from any regular trace language given by means of a trace automaton $\mathcal{A}$. The size of each process built by our method is then polynomial in the size of $\mathcal{A}$ and only exponential in the number of processes and actions of the system. So our method reduces significantly the explosion of the number of states in each process

Contrary to asynchronous automata, non-deterministic DAAs and deterministic DAAs do not recognize the same class of trace languages. Moreover, some regular trace languages are recognized by no DAA. Therefore we characterize the trace languages that correspond precisely to the behaviours of non-deterministic DAAs: these are the ones that satisfy the condition of *distributivity*. We explain how to verify whether or not a regular trace language is distributed. If so, we show that our method gives directly a DAA recognizing this language. However it is still an open question to decide whether a regular trace language is recognizable by a deterministic DAA .

Overview of the Paper. The basic notions and definitions are presented in Section 1. Section 2 introduces the model of distributed asynchronous automata and a short comparison with the classical asynchronous automata. In particular, we show that these two models are not equivalent. In Section 3, a tool is introduced to compose hierarchically DAAs into a *high-level* DAA. The key results presented in this section will be used all along the paper. Then we define in Section 4 the concept of *roadmaps* and their associated *located* trace languages. Our first main result, Theorem 4.4, states that any located trace language is recognized by some DAA. We explain how to build this DAA by using high-level DAAs and present a first complexity result about our construction at the end of this section. Also, Theorem 4.4 yields a new construction for the synthesis of classical asynchronous automata that subsumes all existing ones in terms of complexity. This construction and a comparison with related works are presented in Section 5. Finally, Section 6 contains the second main result of the paper: Theorem 6.5 shows that distributed regular trace languages correspond precisely to the behaviours of non-deterministic distributed asynchronous automata.

1 Background

1.1 Mazurkiewicz Traces

In this paper we fix a finite alphabet Σ whose elements are called *actions*. A *word* u over Σ is a sequence of actions of Σ; the empty word is denoted by ε. The *alphabet* $\mathrm{alph}(u)$ of a word $u \in \Sigma^\star$ consists of all actions that appear in u. It is inductively defined by $\mathrm{alph}(\varepsilon) = \emptyset$ and $\mathrm{alph}(ua) = \mathrm{alph}(u) \cup \{a\}$ for all $u \in \Sigma^\star$ and all $a \in \Sigma$.

In addition to Σ, we fix an *independence relation* I over Σ, that is, a binary relation $\mathrm{I} \subseteq \Sigma \times \Sigma$ that is irreflexive and symmetric. We also define the *dependence relation* D as the complementary relation of I: $\mathrm{D} = \Sigma \times \Sigma \setminus \mathrm{I}$. The *trace equivalence* $\sim$ associated with the *independence alphabet* (Σ, I) is the least congruence over the free monoid $\Sigma^\star$ such that $ab \sim ba$ for all pairs of independent actions $a\mathrm{I}b$. For a word $u \in \Sigma^\star$, the *(Mazurkiewicz) trace* $[u]$ collects all words that are equivalent to u: $[u] = \{v \in \Sigma^\star \mid v \sim u\}$. We extend this notation to sets of words: for all $L \subseteq \Sigma^\star$, $[L] = \{v \in \Sigma^\star \mid \exists u \in L : v \sim u\}$. Finally a set of words L is called a *trace language* if $[L] = L$.

1.2 Trace Automata

An *automaton* $\mathcal{A}$ is a quadruple $(Q, \rightarrow, I, F)$ where Q is a finite set of states, $\rightarrow \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is a set of transitions, and $I, F \subseteq Q$ are respectively a subset of initial states and a subset of final states. We write $q \xrightarrow{a} q'$ to denote $(q, a, q') \in \rightarrow$. Then the automaton $\mathcal{A}$ is *deterministic* if it satisfies the three next conditions: I is a singleton; if $q \xrightarrow{\varepsilon} q'$ then $q = q'$; if $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$ then $q' = q''$.

An *execution* of $\mathcal{A}$ of length $n \geq 1$ is a sequence of transitions $(q_i \xrightarrow{a_i} q_i')_{i \in [1,n]}$ such that $q_i' = q_{i+1}$ for each $i \in [1, n-1]$. An execution of length 0 is simply a state of $\mathcal{A}$. For any word $u \in \Sigma^\star$, we write $q \xrightarrow{u} q'$ if there is some execution $(q_i \xrightarrow{a_i} q_i')_{i \in [1,n]}$ such that $q = q_1$, $q' = q_n'$ and $u = a_1 \cdots a_n$. The language $L(\mathcal{A})$ accepted by an automaton $\mathcal{A}$ consists of all words $u \in \Sigma^\star$ such that $q \xrightarrow{u} q'$ for some $q \in I$ and some $q' \in F$. A set of words $L \subseteq \Sigma^\star$ is *regular* if it is accepted by some automaton.

Definition 1.1. *An automaton $\mathcal{A} = (Q, \rightarrow, I, F)$ is called a* trace automaton *if for all states $q, q' \in Q$ and all words $u, v \in \Sigma^\star$ such that $u \sim v$, $q \xrightarrow{u} q'$ implies $q \xrightarrow{v} q'$.*

Clearly the language accepted by a trace automaton is a regular trace language. Conversely for any regular trace language L the minimal deterministic automaton that accepts L is a trace automaton.

1.3 Synthesis of Asynchronous Automata

We present next the model of asynchronous automata [17]. At first we introduce some additional notations. A finite family $\delta = (\Sigma_p)_{p \in \mathcal{P}}$ of subsets of Σ is called a *distribution* of (Σ, I) if we have $\mathrm{D} = \bigcup_{p \in \mathcal{P}} (\Sigma_p \times \Sigma_p)$. We fix an arbitrary distribution $\delta = (\Sigma_p)_{p \in \mathcal{P}}$ in the rest of this paper and call *processes* the elements of $\mathcal{P}$. The *location* $\mathrm{Loc}(a)$ of an action $a \in \Sigma$ consists of all processes $p \in \mathcal{P}$ such that $a \in \Sigma_p$. We extend this notation to set of actions $T \subseteq \Sigma$ and to words $u \in \Sigma^\star$ in a natural way: $\mathrm{Loc}(T) = \bigcup_{a \in T} \mathrm{Loc}(a)$ and $\mathrm{Loc}(u) = \mathrm{Loc}(\mathrm{alph}(u))$.

Definition 1.2. *An* asynchronous automaton *(*AA *for short) $\mathcal{S}$ consists of*

- *a family of local states $(Q_p)_{p \in \mathcal{P}}$;*
- *a set of initial global states $I \subseteq \prod_{p \in \mathcal{P}} Q_p$;*
- *a set of final global states $F \subseteq \prod_{p \in \mathcal{P}} Q_p$;*
- *a family of mute transitions $(\tau_p)_{p \in \mathcal{P}}$ where $\tau_p \subseteq Q_p \times Q_p$ for all $p \in \mathcal{P}$;*
- *and a family of transition relations $(\partial_a)_{a \in \Sigma}$ where for all $a \in \Sigma$,*

$$\partial_a \subseteq \prod_{p \in \mathrm{Loc}(a)} Q_p \times \prod_{p \in \mathrm{Loc}(a)} Q_p$$

We stress here that the mute transitions have been introduced to simplify the different constructions presented throughout this paper. They can be removed without loss of generality and without increasing the number of local states. For each process $p \in \mathcal{P}$, we define $\tau_p^\star$ as the reflexive and transitive closure of τ_p: for all $q_0, q \in Q_p$, $(q_0, q) \in \tau_p^\star$ iff $q_0 = q$ or else there exists a sequence of states $q_1, \ldots, q_n$ such that $(q_i, q_{i+1}) \in \tau_p$ for all $0 \leq i < n$ and $q_n = q$.

The *global automaton* $\mathcal{A}_\mathcal{S}$ of $\mathcal{S}$ is the automaton $(Q, \rightarrow, I, F)$ where Q is the set of global states $\prod_{p\in\mathcal{P}} Q_p$ and the set of global transitions $\rightarrow \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is defined by the two next rules: for all $a \in \Sigma$, $(x_p)_{p\in\mathcal{P}} \xrightarrow{a} (y_p)_{p\in\mathcal{P}}$ if $x_p = y_p$ for all $p \notin \mathrm{Loc}(a)$ and $((x_p)_{p\in\mathrm{Loc}(a)}, (y_p)_{p\in\mathrm{Loc}(a)}) \in \partial_a$; $(x_p)_{p\in\mathcal{P}} \xrightarrow{\varepsilon} (y_p)_{p\in\mathcal{P}}$ if there is some $p \in \mathcal{P}$ such $(x_p, y_p) \in \tau_p$ and $x_k = y_k$ for all $k \neq p$. For convenience, an execution of the global automaton $\mathcal{A}_\mathcal{S}$ of $\mathcal{S}$ is simply called an execution of $\mathcal{S}$.

The language recognized by an asynchronous automaton $\mathcal{S}$ is simply the language of its global automaton $\mathcal{A}_\mathcal{S}$: $L(\mathcal{S}) = L(\mathcal{A}_\mathcal{S})$. Notice that $\mathcal{A}_\mathcal{S}$ is a trace automaton. Then $L(\mathcal{S})$ is a regular trace language. In [17], Zielonka shows that any regular trace language is recognized by an asynchronous automaton. Furthermore, this asynchronous automaton is *deterministic*, which means its global automaton is deterministic.

Theorem 1.3 (Zielonka [17]). *For all trace automata $\mathcal{A}$ there exists a deterministic asynchronous automaton $\mathcal{S}$ such that $L(\mathcal{S}) = L(\mathcal{A})$.*

2 Distributed Asynchronous Automata

Distributed asynchronous automata (DAA for short) differ slightly from asynchronous automata since we consider a *local* definition of initial and final states rather than a *global* one. We will show in Example 2.2 that the two models are not semantically equivalent: some regular trace languages are not recognizable by distributed asynchronous automata.

Definition 2.1. *An asynchronous automaton $((Q_p)_{p\in\mathcal{P}}, I, F, (\tau_p)_{p\in\mathcal{P}}, (\partial_a)_{a\in\Sigma})$ is* distributed *if there exist two families $(I_p)_{p\in\mathcal{P}}$ and $(F_p)_{p\in\mathcal{P}}$ such that $I_p, F_p \subseteq Q_p$ for all $p \in \mathcal{P}$, $I = \prod_{p\in\mathcal{P}} I_p$ and $F = \prod_{p\in\mathcal{P}} F_p$.*

For convenience we will denote a DAA $\mathcal{H}$ as $((Q_p, I_p, F_p, \tau_p)_{p\in\mathcal{P}}, (\partial_a)_{a\in\Sigma})$ where I_p and F_p represent the set of initial local states and the set of final local states of the process p, respectively. For next purpose, we also define the notion of *local executions* of $\mathcal{H}$. Let p be a process. Then a local execution of $\mathcal{H}$ for p is a sequence $(q_i, a_i, q_i')_{i\in[1,n]}$ of tuples of $Q_p \times (\Sigma_p \cup \{\varepsilon\}) \times Q_p$ that satisfies the following statements: $q_1 \in I_p$; $q_n' \in F_p$; $q_i' = q_{i+1}$ for all $i \in [1, n-1]$; and for all $i \in [1, n]$, either $(q_i, q_i') \in \tau_p$ and $a_i = \varepsilon$, or there is $((x_k)_{k\in\mathcal{P}}, (y_k)_{k\in\mathcal{P}}) \in \partial_{a_i}$ with $x_p = q_i$ and $y_p = q_i'$. For any word $u \in \Sigma_p^\star$, we write $q \overset{u}{\rightsquigarrow}_p q'$ if there is some local execution $(q_i, a_i, q_i')_{i\in[1,n]}$ for p such that $q_1 = q$, $q_n' = q'$ and $a_1 \cdots a_n = u$.

Example 2.2. Consider the regular trace language $L = [aab] \cup [abb]$ over the alphabet $\{a, b\}$ with aIb. L is recognized by the asynchronous automaton that consists of two processes p_a and p_b with $\Sigma_{p_a} = \{a\}$, $\Sigma_{p_b} = \{b\}$, $Q_{p_a} = \{q_a, q_a', q_a''\}$,

$Q_{p_b} = \{q_b, q'_b, q''_b\}$, $I = \{(q_a, q'_b), (q'_a, q_b)\}$, $F = \{(q''_a, q''_b)\}$, $\partial_a = \{(q_a, q'_a), (q'_a, q''_a)\}$ and $\partial_b = \{(q_b, q'_b), (q'_b, q''_b)\}$. However L is recognized by no DAA. Indeed, suppose for the sake of contradiction that such a DAA $\mathcal{H}$ exists, that is $L(\mathcal{H}) = L$. Then $\mathcal{H}$ consists of two processes p_a and p_b. Since $L = [aab] \cup [abb]$ and aIb, p_a is able to locally perform a and aa, and p_b is able to locally perform b and bb. Because initial and final states are local in a DAA, the process p_a carries out a or aa independently of the choice of the process p_b. Then the words ab and $aabb$ belong to $L(\mathcal{H})$. This contradicts $L(\mathcal{H}) = L$.

This example shows that some regular trace languages are not recognizable by DAAs. Since DAAs are a particular case of AAs, and because Theorem 1.3 states that any regular trace language is recognized by AAs, we conclude that DAAs are less expressive than AAs. Noteworthy the expressive power of asynchronous automata does not change if only the initializations or else only the accepting conditions are distributed.

Another interesting remark follows from Theorem 1.3. Let L be a regular trace language. Then L is recognized by some AA $\mathcal{S} = ((Q_p)_{p\in\mathcal{P}}, I_\mathcal{S}, F_\mathcal{S}, (\tau_p)_{p\in\mathcal{P}}, (\partial_a)_{a\in\Sigma})$. For each pair $(\imath, f)$ of initial and final states of $I_\mathcal{S} \times F_\mathcal{S}$, we can build another AA $\mathcal{S}_{\imath,f}$ that is identical to $\mathcal{S}$ except that the set of initial states and the set of final states are restricted to the singletons $\{\imath\}$ and $\{f\}$, respectively. Clearly, these new AAs are DAAs as well. Then the next corollary holds:

Corollary 2.3. *Let L be a regular trace language. There exists a finite set of* DAA*s $\mathcal{H}_1, \ldots, \mathcal{H}_n$ such that $L = \bigcup_{i\in[1,n]} L(\mathcal{H}_i)$.*

Thus, any regular trace language is the finite union of the languages of DAAs. The expressive power of DAAs will be studied in Section 6.

3 High-Level Distributed Asynchronous Automata

In this section, we introduce a natural operation to compose DAAs similar to HMSC [8]. We show that the language of the DAA associated to a high-level DAA can be expressed in terms of the languages of its DAA components under some assumptions. As example, we show that some natural and intuitive operations over DAAs like concatenation and choice can be seen as particular high-level DAAs. The different results of this section constitute the heart of this paper. They will be used in Sections 4 and 6.

A *high-level* DAA $\mathcal{G}$ is a structure (V, E, I, F, Ψ) where V is a finite and nonempty set of vertices, $E \subseteq V \times V$ is a set of edges and $I, F \subseteq V$ are respectively a set of initial vertices and a set of final vertices. The function Ψ maps all vertices $v \in V$ to a DAA $\Psi(v)$, which is denoted by $((Q_{v,p}, I_{v,p}, F_{v,p}, \tau_{v,p})_{p\in\mathcal{P}}, (\partial_{v,a})_{a\in\Sigma})$. Moreover, we require that the sets of states $Q_{v,p}$ and $Q_{v',p}$ are disjoint for all $v, v' \in V$ and all $p \in \mathcal{P}$. A *path* π of $\mathcal{G}$ is a sequence $(v_0, \ldots, v_n)$ such that $v_0 \in I$, $v_n \in F$ and $(v_{i-1}, v_i) \in E$ for $i \in [1, n]$. We denote by $\Pi_\mathcal{G}$ the set of all paths of $\mathcal{G}$. We say that $\mathcal{G}$ has *no self-loop* if $(v, w) \in E \Rightarrow v \neq w$. This assumption will be useful for Proposition 3.5 to avoid some synchronization problems as presented in Remark 3.6.

Now we define the DAA $\langle\mathcal{G}\rangle$ associated with a high-level DAA $\mathcal{G}$. Roughly speaking, $\langle\mathcal{G}\rangle$ consists of putting the DAAs $\Psi(v)$ all together, and adding mute transitions from their final local states to their initial local states in accordance with the edges of $\mathcal{G}$.

Definition 3.1. *The* DAA $\langle\mathcal{G}\rangle = ((Q_p, I_p, F_p, \tau_p)_{p\in\mathcal{P}}, (\partial_a)_{a\in\Sigma})$ *associated with a high-level* DAA $\mathcal{G} = (V, E, I, F, \Psi)$ *is defined by*

- $Q_p = \bigcup_{v\in V} Q_{v,p}$ *for each process* $p \in \mathcal{P}$,
- $I_p = \bigcup_{v\in I} I_{v,p}$ *for each process* $p \in \mathcal{P}$,
- $F_p = \bigcup_{v\in F} F_{v,p}$ *for each process* $p \in \mathcal{P}$,
- $\partial_a = \bigcup_{v\in V} \partial_{v,a}$ *for each action* $a \in \Sigma$,
- $\tau_p = \bigcup_{v\in V} \tau_{v,p} \cup \bigcup_{(v,w)\in E}(F_{v,p} \times I_{w,p})$ *for each process* $p \in \mathcal{P}$.

To illustrate this definition, we pay attention to the particular case of a high-level DAA with two vertices and one edge, which leads to the definition of the concatenation of two DAAs. The related Proposition 3.3 will be useful for the proof of Proposition 3.5.

Definition 3.2. *Let* $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *be two* DAA*s and let* $\mathcal{G}$ *be the high-level distributed asynchronous automaton* (V, E, I, F, Ψ) *where* $V = \{v_1, v_2\}$, $(v_1, v_2) \in E$, $I = \{v_1\}$, $F = \{v_2\}$, $\Psi(v_1) = \mathcal{H}_1$ *and* $\Psi(v_2) = \mathcal{H}_2$. *Then the* concatenation $\mathcal{H}_1 \odot \mathcal{H}_2$ *is* $\langle\mathcal{G}\rangle$.

Intuitively, $\mathcal{H}_1 \odot \mathcal{H}_2$ behaves as follows: each process p of $\mathcal{H}_1 \odot \mathcal{H}_2$ starts to behave like the process p of $\mathcal{H}_1$ until it reaches a final local state of $\mathcal{H}_1$. Thereafter it behaves like the process p of $\mathcal{H}_2$. So, it should be clear that the following proposition holds:

Proposition 3.3. $L(\mathcal{H}_1 \odot \mathcal{H}_2)$ *is the trace language* $[L(\mathcal{H}_1) \cdot L(\mathcal{H}_2)]$.

We come to the key result of the paper. The next proposition shows that, under some assumptions, the language of the DAA associated with a high-level DAA $\mathcal{G}$ can be expressed in terms of the languages of the DAA components of $\mathcal{G}$. This result requires that $\mathcal{G}$ has no self-loop and relies on the synchronization condition defined below. The latter uses the notion of local executions introduced in Section 2. We recall that local executions always start in an initial local state and end in a final local state.

Definition 3.4. *Let* $T \subseteq \Sigma$. *A* DAA $\mathcal{H}$ *is* T-synchronizable *if for all processes* $p \in \mathcal{P}$ *and all local executions* $q \overset{u}{\rightsquigarrow}_p q'$ *of* $\mathcal{H}$, *we have* $(\Sigma_p \cap T) = \mathrm{alph}(u)$.

For a distributed asynchronous automaton $\mathcal{H}$, the T-synchronization condition will imply the next fact: if the dependence graph (T, D) is connected, then all processes of $\mathrm{Loc}(T)$ always synchronize with each other along *any* execution of $\mathcal{H}$ that leads its processes from local initial states to final ones. Thus, whenever we compose several T-synchronizable DAAs in a high-level DAA $\mathcal{G}$, all processes of $\mathrm{Loc}(T)$ are forced to travel along the same sequence of DAA components. Consequently, the behaviours of the associated DAA $\langle\mathcal{G}\rangle$ can easily be characterized as unions and compositions of the languages of the DAA components.

Proposition 3.5. *Let* $\mathcal{G}$ *be a high-level* DAA *with* Ψ *as labelling function and* $T \subseteq \Sigma$. *If* $\mathcal{G}$ *has no self-loop,* Ψ *maps all vertices of* $\mathcal{G}$ *to* T*-synchronizable* DAA*s and the dependence graph* (T, D) *is connected, then* $\langle\mathcal{G}\rangle$ *is a* T*-synchronizable* DAA *that recognizes*

$$L(\langle\mathcal{G}\rangle) = \bigcup_{(v_0,\ldots,v_n)\in\Pi_{\mathcal{G}}} [L(\Psi(v_0)) \cdot \ldots \cdot L(\Psi(v_n))] \ .$$

Proof. Let $\mathcal{G} = (V, E, I, F, \Psi)$ be a high-level DAA. First, $\langle\mathcal{G}\rangle$ is T-synchronizable because any process has to travel along at least one T-synchronizable DAA component to reach a final local state from an initial local state. Then the processes of $\mathcal{P} \setminus \mathrm{Loc}(T)$ take part in mute transitions of $\langle\mathcal{G}\rangle$ only. For this reason, we suppose without loss of generality than $\mathcal{P} = \mathrm{Loc}(T)$. It is not hard to prove that $\bigcup_{(v_0,\ldots,v_n)\in\Pi_{\mathcal{G}}}[L(\Psi(v_0))\cdot\ldots\cdot L(\Psi(v_n))] \subseteq L(\langle\mathcal{G}\rangle)$. So, we prove only the backward inclusion. From now on, we fix an execution s of $\langle\mathcal{G}\rangle$ that leads the global automaton $\mathcal{A}_{\langle\mathcal{G}\rangle}$ from some global initial state to some global final state and we let u the word yielded by s. Then $u \in L(\langle\mathcal{G}\rangle)$.

We start with some additional notations. Let p be any process of $\mathrm{Loc}(T)$. By construction of $\langle\mathcal{G}\rangle$, along s, there are some vertices $v_1, \ldots, v_n \in V^\star$ such that: p starts from some initial state $\imath_1 \in I_{v_1,p}$ from which it reaches some state $f_1 \in F_{v_1,p}$ *by using exclusively* $\tau_{v_1,p}$ *or* $(\partial_{v_1,a})_{a\in\Sigma_p}$ *transitions*; then an added τ_p-transition leads p from f_1 to some state $\imath_2 \in I_{v_2,p}$ from which it reaches some $f_2 \in F_{v_2,p}$ *by using exclusively* $\tau_{v_2,p}$ *or* $(\partial_{v_2,a})_{a\in\Sigma_p}$ *transitions*; and so on, until p reaches a local final state of $F_{v_n,p}$. We denote by $\nu_p(s)$ this sequence of vertices $(v_1, \ldots, v_n)$ relied on p along s. In addition, for all $1 \leq i \leq n$ and all $a \in T$, we denote by $|s,p|_{i,a}$ the number of $\partial_{v_i,a}$ transitions in which p takes part when it goes from $\imath_i$ to f_i along s.

Two remarks are useful for the rest of this proof. Let $p \in \mathcal{P}$ and $\nu_p(s) = (v_1, \ldots, v_n)$. First, it should be clear that $|s,p|_{i,a} > 0$ for all $a \in \Sigma_p \cap T$ and all $i \in [1,n]$ because all the $\Psi(v_i)$ are T-synchronizable. Secondly, two consecutive vertices of $\nu_p(s)$ are always distinct because there is no self-loop in $\mathcal{G}$: $v_i \neq v_{i+1}$ for all $1 \leq i < n$.

Let $a \in T$ and $p, k \in \mathrm{Loc}(a)$. Let $\nu_p(s) = (v_1, \ldots, v_n)$ and $\nu_k(s) = (w_1, \ldots, w_m)$ with $n \leq m$. We prove by contradiction that the following property (P) holds:

(P) for all $i \in [1,n]$, $v_i = w_i$ and $|s,p|_{i,a} = |s,k|_{i,a}$

Suppose that (P) fails. Then we denote by i the least integer of $[1,n]$ such that $v_i \neq w_i$ or $|s,p|_{i,a} \neq |s,k|_{i,a}$. Let $c = 1 + \sum_{j=1}^{i-1} |s,p|_{j,a}$. By hypothesis, $c = 1 + \sum_{j=1}^{i-1} |s,k|_{j,a}$ as well. This means that the c-th a-transition in which p takes part results from the transition relation $\partial_{v_i,a}$ whereas the c-th a-transition in which k takes part results from the transition relation $\partial_{w_i,a}$. This implies that $v_i = w_i$. Since we have supposed that (P) fails for i, we should have $|s,p|_{i,a} \neq |s,k|_{i,a}$. We prove it is impossible. Assume that $|s,p|_{i,a} < |s,k|_{i,a}$ and let $c' = c + |s,p|_{i,a}$. If $i = n$ then p takes part in $c' - 1$ a-transitions along s whereas k takes part in at least c' a-transitions, which is impossible because p and k must synchronize on all a-transitions. So $i < n$. However in this case the c'-th transition in which p takes part results from the transition relation $\partial_{v_{i+1},a}$ whereas the c'-th transition in which k takes part results from the transition relation $\partial_{w_i,a}$. Hence $v_i = w_i = v_{i+1}$, which is impossible because there is no self-loop in $\mathcal{G}$.

Since (P) holds for each $a \in T$ and since each process of $\mathrm{Loc}(a)$ takes part in the same number of a-transitions along s, we deduce that for all $p, k \in \mathrm{Loc}(a)$, $\nu_p(s) = \nu_k(s)$. Then, step by step, considering all actions of T together with similar arguments, we conclude that for all processes p, k of $\mathrm{Loc}(T)$ we have $\nu_k(s) = \nu_p(s)$ because (T, D) is connected. Consequently s is an execution of the DAA $\Psi(v_1) \odot \ldots \odot \Psi(v_n)$ as well. To conclude, Proposition 3.5 implies that $u \in [L(\Psi(v_1) \cdot \ldots \cdot L(\Psi(v_n)]$. ∎

Remark 3.6. We stress here the importance of the absence of self-loop in $\mathcal{G}$. Consider the $\{a, c\}$-synchronizable DAA $\mathcal{H}$ that consists of two processes p_1 and p_2 such that: $\Sigma_{p_1} = \{a, c\}$ and $\Sigma_{p_2} = \{c\}$; $Q_{p_i} = \{q_i, q_i'\}$, $I_{p_i} = \{q_i\}$, $F_{p_i} = \{q_i'\}$ and $\tau_{p_i} = \emptyset$ for all $i \in \{1,2\}$; $((q_1,q_2),(q_1,q_2')) \in \partial_c$ and $(q_1,q_1') \in \partial_a$. Clearly $\mathcal{H}$ accepts only the word ca. Now, consider the high-level DAA $\mathcal{G}$ that consists of a single vertex v, which is both initial and final, a single edge from v to v and a labelling function Ψ that maps v to $\mathcal{H}$. Then $\langle\mathcal{G}\rangle$ accepts the word cca, so that $L(\langle\mathcal{G}\rangle)$ cannot be expressed in terms of a rational expression that uses exclusively $L(\mathcal{H})$. The problem with the presence of self-loops is the following. In spite of the T-synchronization condition of $\mathcal{H}$, some processes (p_2 in our example) can travel along the DAA $\mathcal{H}$ several times, without the other processes (p_1 in our example) being informed of it. This kind of problem justifies why we require that a high-level DAA has no self-loop.

To illustrate Proposition 3.5, we introduce a second natural operation based on high-level DAA. Let $\mathcal{H}_1$ and $\mathcal{H}_2$ be two DAAs. The choice operation $\mathcal{H}_1 \oplus \mathcal{H}_2$ yields a DAA where each process p of $\mathcal{H}_1 \oplus \mathcal{H}_2$ can choose, independently from the other processes, to behave either like the process p of $\mathcal{H}_1$ or else like the process p of $\mathcal{H}_2$. Formally:

Definition 3.7. *The* choice $\mathcal{H}_1 \oplus \mathcal{H}_2$ *of* $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *is the* DAA $\langle\mathcal{G}\rangle$ *where* $\mathcal{G}$ *consists of exactly two vertices, which are both initial and final, and no edges. As for the labelling function, it maps these vertices to* $\mathcal{H}_1$ *and* $\mathcal{H}_2$.

In this definition, $\mathcal{G}$ has no self-loop. So Proposition 3.5 can be applied as follows:

Corollary 3.8. *Let* $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *be two* T*-synchronizable* DAA*s. If the dependence graph* (T,D) *is connected, then* $\mathcal{H}_1 \oplus \mathcal{H}_2$ *is* T*-synchronizable and recognizes* $L(\mathcal{H}_1) \cup L(\mathcal{H}_2)$.

4 Located Trace Languages

In this section we fix a trace automaton $\mathcal{A} = (Q, \rightarrow, I, F)$ over (Σ, I). We show in Theorem 4.4 that any *located* trace language of $\mathcal{A}$ (Definition 4.1) corresponds to the language of a synchronizable DAA.

An analogy can be drawn between our technique of proof and the technique used by McNaughton and Yamada in [10] in the framework of the free monoid. In this paper, the authors define languages $R_{i,j}^K$ that correspond to the set of words that label a path from the state $i \in Q$ to the state $j \in Q$ and that use the states in $K \subseteq Q$ as intermediate states. Then they use a recursive algorithm on the size of K to build rational expressions that correspond to the languages $R_{i,j}^K$. Here we have adapted this method for the trace monoid (Σ, I), that is, to care of concurrency. The defined languages are called *located trace languages*. They are based on *roadmaps* instead of intermediate states. Then we use a recursive algorithm on the size n of the roadmaps to build, by using high-level DAAs, the DAAs that correspond to the located trace languages on roadmaps of size n.

4.1 Roadmaps and Located Trace Languages

From now on, we fix a total order $\sqsubseteq$ over the actions of Σ. This order is naturally extended to sets of actions: for $T_1, T_2 \subseteq \Sigma$, $T_1 \sqsubseteq T_2$ if the least action in T_1 is smaller than the least action in T_2 w.r.t. $\sqsubseteq$.

Let $T \subseteq \Sigma$. We denote by $\#\mathrm{cc}(T)$ the number of connected components of the dependence graph (T, D) and by $\mathrm{cc}(T)$ the sequence $(T_1, \ldots, T_{\#\mathrm{cc}(T)})$ of the $\#\mathrm{cc}(T)$

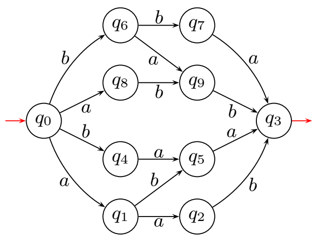


Fig. 1. $L(\mathcal{A})$ is recognized by no DAA

connected components of (T, D) sorted according to $\sqsubseteq$: $T_i \sqsubseteq T_{i+1}$ for all $1 \leq i < \#\mathrm{cc}(T)$. Note that $\#\mathrm{cc}(T) = 1$ if and only if (T, D) is connected.

Definition 4.1. *A* roadmap *r of the trace automaton $\mathcal{A}$ consists of a subset of actions $T \subseteq \Sigma$ together with a vector $\boldsymbol{q}$ of $\#\mathrm{cc}(T) + 1$ states of $\mathcal{A}$.*

The located trace language *$L_r(\mathcal{A})$ on a roadmap $r = (T, \boldsymbol{q})$ of $\mathcal{A}$ with $\mathrm{cc}(T) = (T_1, \ldots, T_n)$ and $\boldsymbol{q} = (q_1, \ldots, q_{n+1})$ comprises all the words contained in any trace $[u_1 \cdots u_n]$ such that $\mathrm{alph}(u_i) = T_i$ and $q_i \xrightarrow{u_i} q_{i+1}$ for all $1 \leq i \leq n$,* i.e.*:*

$$L_r(\mathcal{A}) = [\{u_1 \cdots u_n \in \Sigma^\star \mid \forall i \in [1, n], \mathrm{alph}(u_i) = T_i \wedge q_i \xrightarrow{u_i} q'_{i+1}\}].$$

Given a roadmap $r = (T, \boldsymbol{q})$, $\mathrm{start}(r)$ is the first component of $\boldsymbol{q}$, $\mathrm{stop}(r)$ is the last component of $\boldsymbol{q}$ and $\mathrm{alph}(r)$ is T. Then we denote by $\mathcal{R}(\mathcal{A})$ the set of all roadmaps of $\mathcal{A}$, and by $\mathcal{R}_f(\mathcal{A})$ the subset of roadmaps r of $\mathcal{R}(\mathcal{A})$ such that $\mathrm{start}(r) \in I$, $\mathrm{stop}(r) \in F$ and $L_r(\mathcal{A}) \neq \emptyset$.

Example 4.2. Consider the automaton $\mathcal{A}$ over $\{a, b\}$ with aIb and $a \sqsubseteq b$ depicted in Fig. 1, in which q_0 and q_3 are respectively the single initial state and the single final state. There are exactly two roadmaps r in $\mathcal{R}_f(\mathcal{A})$, that is, such that $\mathrm{start}(r) = q_0$, $\mathrm{stop}(r) = q_3$ and $L_r(\mathcal{A}) \neq \emptyset$: $r_1 = (\{a, b\}, q_0, q_2, q_3)$ and $r_2 = (\{a, b\}, q_0, q_8, q_3)$. The corresponding located trace languages are $L_{r_1}(\mathcal{A}) = [aab]$ and $L_{r_2}(\mathcal{A}) = [abb]$. We can check that $\mathcal{A}$ describes the trace language $[aab] \cup [abb]$ of Example 2.2.

As illustrated by Example 4.2, any regular trace language can be represented as the finite union of all the located trace languages on roadmaps of $\mathcal{R}_f(\mathcal{A})$:

Proposition 4.3. $L(\mathcal{A}) = \bigcup_{r \in \mathcal{R}_f(\mathcal{A})} L_r(\mathcal{A})$.

Now we turn to our first main result. The latter states that any located trace language is recognized by a DAA. Its constructive proof is detailed in Section 4.2.

Theorem 4.4. *Let $\mathcal{A}$ be a trace automaton and r be a roadmap of $\mathcal{A}$. There exists a* $\mathrm{alph}(r)$*-synchronizable* DAA *that recognizes $L_r(\mathcal{A})$.*

Note that Proposition 4.3 together with Theorem 4.4 give another proof of Corollary 2.3: any regular trace language can be seen as a finite union of languages of DAAs.

4.2 Constructive Proof of Theorem 4.4

We devote this subsection to the proof of Theorem 4.4. Our proof is constructive, that is, given a located trace language $L_r(\mathcal{A})$ of $\mathcal{A}$, we build a $\mathrm{alph}(r)$-synchronizable DAA and show that it recognizes L. To do this, we use an induction over the size of $\mathrm{alph}(r)$.

The base case is simple. Let r be a roadmap with $\mathrm{alph}(r) = \emptyset$. Either $L_r(\mathcal{A})$ is the empty language. Then it is recognized by the $\emptyset$-synchronized DAA where each process consists of one initial and non-final state and no transition. Or else $L_r(\mathcal{A})$ consists of the empty word ε only. So, it is recognized by the $\emptyset$-synchronized DAA where each process consists of one state, both initial and final, and all transition relations are empty.

We distinguish two inductive cases according to whether the dependence graph $(\mathrm{alph}(r), \mathrm{D})$ is connected or not.

Inductive Case 1. Let r be a roadmap of $\mathcal{R}(\mathcal{A})$ such that $\mathrm{alph}(r) \neq \emptyset$ and $(\mathrm{alph}(r), \mathrm{D})$ is unconnected. Let $r = (T, \boldsymbol{q})$ with $\mathrm{cc}(T) = (T_1, \ldots, T_n)$ and $\boldsymbol{q} = (q_1, \ldots q_{n+1})$. Then we set $r_i = (T_i, (q_i, q_{i+1}))$ for all $1 \leq i \leq n$. We can easily check that for all $1 \leq i \leq n$, r_i is a roadmap of $\mathcal{R}(\mathcal{A})$ and that $L_r(\mathcal{A}) = [L_{r_1}(\mathcal{A}) \cdot \ldots \cdot L_{r_n}(\mathcal{A})]$. By inductive hypothesis, for every $L_{r_i}(\mathcal{A})$ there exists a $\mathrm{alph}(r_i)$-synchronizable DAA $\mathcal{H}_i$ such that $L(\mathcal{H}_i) = L_{r_i}(\mathcal{A})$. Then Theorem 4.4 follows from Lemma 4.5. Note that Proposition 3.3 implies immediately that $L(\mathcal{H}_1 \odot \cdots \odot \mathcal{H}_n) = L_r(\mathcal{A})$.

Lemma 4.5. *$\mathcal{H}_1 \odot \cdots \odot \mathcal{H}_n$ is $\mathrm{alph}(r)$-synchronizable and recognizes $L_r(\mathcal{A})$.*

Inductive Case 2. Let r be a roadmap with $\mathrm{alph}(r) = T \neq \emptyset$ such that $(\mathrm{alph}(r), \mathrm{D})$ is connected. We denote by S_r the set of all roadmaps s such that $\mathrm{alph}(s) \subsetneq T$ and $\mathrm{stop}(s) = \mathrm{stop}(r)$. For $s \in S_r$, $\bar{s}$ denotes the roadmap $(T, \mathrm{start}(r), \mathrm{start}(s))$ and R_s consists of all the words u in $L_{\bar{s}}(\mathcal{A})$ that are also contained in some trace $[u_1 a_1 \cdots u_n a_n]$ such that $n \in \mathbb{N}$, $a_i \in T$ and $\mathrm{alph}(u_i) = T \setminus \{a_i\}$ for all $1 \leq i \leq n$: $R_s = L_{\bar{s}}(\mathcal{A}) \cap [\{u_1 a_1 \cdots u_n a_n \in \Sigma^\star \mid n \in \mathbb{N} \wedge \forall i \in [1, n], (a_i \in T \wedge \mathrm{alph}(u_i) \in T \setminus \{a_i\})\}]$. Clearly the following Lemma holds:

Lemma 4.6. $L_r(\mathcal{A}) = \bigcup_{s \in S_r} [R_s \cdot L_s(\mathcal{A})]$.

Then we express each R_s as a high-level DAA $\mathcal{G}_s = (V_s, E_s, I_s, F_s)$ defined below:

- $V_s = V \times \{0, 1\}$ where V is the set of all roadmaps v of $\mathcal{R}(\mathcal{A})$ such that $\mathrm{alph}(v) = T \setminus \{a_v\}$ for some action $a_v \in T$.
- for all $(v, i) \in V_s$, $(v, i) \in I_s$ if $\mathrm{start}(v) = \mathrm{start}(r)$.
- for all $(v, i) \in V_s$, $(v, i) \in F_s$ if $\mathrm{stop}(v) \xrightarrow{a_v} \mathrm{start}(s)$ is a transition of $\mathcal{A}$.
- for all $(v, i), (w, j) \in V_s$, $((v, i), (w, j)) \in E_s$ if $\mathrm{stop}(v) \xrightarrow{a_v} \mathrm{start}(w)$ is a transition of $\mathcal{A}$ and $i \neq j$.

It remains to define the labelling function Ψ_s. Let (v, i) be a vertex of $\mathcal{G}_s$. By definition of $\mathcal{G}_s$, $\mathrm{alph}(v)$ is the set $T \setminus \{a_v\}$ for some action $a_v \in T$. Clearly, the singleton $\{a_v\}$ is recognized by the $\{a_v\}$-synchronizable DAA $\mathcal{H}_{a_v}$ that consists of one state (both initial and final) for each process $p \notin \mathrm{Loc}(a_v)$, two states q_p (which is initial) and q'_p (which is final) for all processes $p \in \mathrm{Loc}(a_v)$, and one unique transition

$((q_p)_{p\in \mathrm{Loc}(a_v)}, (q'_p)_{p\in \mathrm{Loc}(a_v)}) \in \partial_{a_v}$. Then, using the inductive hypothesis on the located trace language $L_v(\mathcal{A})$, there is a $\mathrm{alph}(v)$-synchronizable DAA $\mathcal{H}_v$ that recognizes $L_v(\mathcal{A})$. Together with Proposition 3.3, we get a T-synchronizable DAA $\mathcal{H}_v \odot \mathcal{H}_{a_v}$ that recognizes $[L_v(\mathcal{A}) \cdot \{a_v\}]$. So, we set $\Psi_s(v,i) = \mathcal{H}_v \odot \mathcal{H}_{a_v}$.

Remark 4.7. At this step of the proof, we have the following facts: $\mathcal{G}_s$ as no self-loop because of the definition of E_s; each $\Psi(v,i)$ is a T-synchronizable DAA; we have assumed that the dependence graph (T, D) is connected. Then, all the hypotheses are satisfied to apply Proposition 3.5 on $\mathcal{G}_s$, which is useful to get the next result.

Lemma 4.8. *$\langle \mathcal{G}_s \rangle$ is $\mathrm{alph}(r)$-synchronizable and recognizes R_s.*

We come to the last step of the proof. First we recall that $L_r(\mathcal{A})$ describes the trace language $\bigcup_{s\in S_r}[R_s \cdot L_s(\mathcal{A})]$ (Lemma 4.6). Furthermore, for any $s \in S_r$, R_s is recognized by the T-synchronizable DAA $\langle \mathcal{G}_s \rangle$ (Lemma 4.8), and there is some $\mathrm{alph}(s)$-synchronizable DAA $\mathcal{H}_s$ that recognizes $L_s(\mathcal{A})$ (by the inductive hypothesis). Then Proposition 3.3 and Corollary 3.8 yield the next lemma from which Theorem 4.4 results.

Lemma 4.9. *$\oplus_{s\in S_r}(\langle \mathcal{G}_s \rangle \odot \mathcal{H}_s)$ is $\mathrm{alph}(r)$-synchronizable and recognizes $L_r(\mathcal{A})$.*

4.3 Complexity Analysis of the Construction

Theorem 4.4 shows that any located trace language is the language of some DAA. We explained in Subsection 4.2 how to build inductively this DAA. Here we analyse the complexity of our construction. In the next proposition, k denotes the number of processes in $\mathcal{P}$ and q denotes the number of states of the trace automaton $\mathcal{A}$.

Proposition 4.10. *Let $\mathcal{A}$ be a trace automaton, r be a roadmap of $\mathcal{A}$ with $|\mathrm{alph}(r)| = n$ and $\mathcal{H}_r$ be the* DAA *built by the algorithm described at Subsection 4.2. Then the number of local states in each process p of $\mathcal{H}_r$ is in $2^{O(n^2)}$ for q fixed, and $O(q^{2nk})$ for n fixed.*

Proof. For all roadmaps r of $\mathcal{A}$, $\mathcal{H}_r$ denotes the DAA inductively built by the construction presented along Subsection 4.2. We write $q_{r,p}$ to refer to the number of local states in the process p of $\mathcal{H}_r$. Then, $c_p(n)$ corresponds to the size of the biggest process p among all the DAAs $\mathcal{H}_r$ such that $|\mathrm{alph}(r)| \leq n$: $c_p(n) = \max\{q_{r,p} \mid r \in \mathcal{R}(\mathcal{A}) \wedge |\mathrm{alph}(r)| \leq n\}$. Clearly we have $c_p(n_1) \geq c_p(n_2)$ as soon as $n_1 \geq n_2$. We will compute an upper bound for $c_p(n)$. From now on, we fix a roadmap $r = (T, \boldsymbol{q}) \in \mathcal{R}(\mathcal{A})$ such that $|T| = n$ and a process $p \in \mathcal{P}$. We proceed in two cases according to whether the dependence graph (T, D) is connected.

Suppose that (T, D) is unconnected and let $\mathrm{cc}(T) = (T_1, \ldots, T_l)$. Clearly the number l of connected components of (T, D) is less than $\min\{n, k\}$. By Lemma 4.5, $\mathcal{H}_r = \mathcal{H}_{r_1} \odot \cdots \odot \mathcal{H}_{r_l}$ where each r_i is some roadmap with $\mathrm{alph}(r_i) = T_i \subsetneq T$. Therefore, we have $q_{r,p} \leq nc_p(n-1)$.

Suppose now that (T, D) is connected. By Lemma 4.9, $\mathcal{H}_r = \oplus_{s\in S_r}(\langle \mathcal{G}_s \rangle \odot \mathcal{H}_s)$. Let $s \in S_r$. Then $\mathrm{alph}(s) \subsetneq T$, which implies that $|\mathrm{alph}(s)| < n$. Consequently, $c_p(|\mathrm{alph}(s)|) \leq c_p(n-1)$. We now compute the number of local states in the process p of $\langle \mathcal{G}_s \rangle$. The latter is built from the graph $\mathcal{G}_s = (V_s, E_s, I_s, F_s, \Psi_s)$ where

$V_s = V \times \{0,1\}$ and V consists of all roadmaps v such that $\mathrm{alph}(v) = T \setminus \{a_v\}$ for some $a_v \in T$. Then V_s contains at most $2nq^k$ vertices. Each vertex v of $\mathcal{G}_s$ is labelled by the DAA $\mathcal{H}_v \odot \mathcal{H}_{a_v}$. The number of local states in the process p of $\mathcal{H}_v$ is at most $c_p(n-1)$ (by inductive hypothesis) and the one of $\mathcal{H}_{a_v}$ is at most 2. Consequently the number of local states in the process p of $\mathcal{H}_v \odot \mathcal{H}_{a_v}$ is at most $c_p(n-1)+2$ and the one of $\langle \mathcal{G}_s \rangle$ is at most $2nq^k(c_p(n-1)+2)$. Finally, we can compute the number of local states in the process p of $\mathcal{H}_r$: since the number of roadmaps in S_r is less than $2^n q^k$, we have that $q_{r,p} \leq 2^n q^k(2nq^k(c_p(n-1)+2)+c_p(n-1))$, that is, $q_{r,p} < n2^{n+3}q^{2k}c_p(n-1)$.

In both cases, for all roadmaps $r \in \mathcal{R}(\mathcal{A})$ such that $|\mathrm{alph}(r)| = n$, the number $q_{r,p}$ of local states in the process p of $\mathcal{H}_r$ is at most $n2^{n+3}q^{2k}c_p(n-1)$. In other words, $c_p(n) < n2^{n+3}q^{2k}c_p(n-1)$. Since $c_p(0)=1$, we get $c_p(n) < n!2^{(n+4)(n+3)/2}q^{2kn}$. ■

5 Back to Zielonka's Theorem

Zielonka's theorem presented in Section 1 states that any regular trace automaton is recognized by an asynchronous automaton (AA). However, all known constructions of asynchronous automata from regular trace languages are quite involved and yield an exponential state explosion. In this section, we discuss how to apply Proposition 4.3 and Theorem 4.4 to get immediately a new algorithm for the construction of *non-deterministic* asynchronous automata that reduces significantly the state explosion.

Indeed, Theorem 4.4 yields for every $r \in \mathcal{R}_f(\mathcal{A})$ a DAA $\mathcal{H}_r$ that recognizes $L_r(\mathcal{A})$. We denote by $I_{r,p}$ and $F_{r,p}$ the set of initial local states and the set of final local states of the process p of $\mathcal{H}_r$, respectively. Now, consider the asynchronous automaton $\mathcal{S}$ that is identical to $\oplus_{r\in\mathcal{R}_f(\mathcal{A})}\mathcal{H}_r$ except that the set of *global* initial states is $I_\mathcal{S} = \bigcup_{r\in\mathcal{R}_f(\mathcal{A})}(\prod_{p\in\mathcal{P}} I_{r,p})$ and the set of *global* final states is $F_\mathcal{S} = \bigcup_{r\in\mathcal{R}_f(\mathcal{A})}(\prod_{p\in\mathcal{P}} F_{r,p})$. Then, by Proposition 4.3, it should be clear that $\mathcal{S}$ recognizes $L(\mathcal{A})$. This leads us to the next corollary where the complexity result follows from Proposition 4.10 together with the fact that $\mathcal{R}_f(\mathcal{A})$ contains at most $2^{|\Sigma|}|Q|^{|\mathcal{P}|}$ roadmaps.

Corollary 5.1. *Let $\mathcal{A}$ be a trace automaton over (Σ, I) with Q as set of states. There exists an asynchronous automaton $\mathcal{S}$ that recognizes $L(\mathcal{A})$ such that the number of local states in each process is in $2^{O(|\Sigma|^2)}$ for $|Q|$ fixed, and $O(|Q|^{2\cdot|\Sigma|\cdot|\mathcal{P}|+1})$ for $|\Sigma|$ fixed.*

Comparison with Existing Approaches. We mention the approaches that lead to deterministic AAs first. In [13] a complexity analysis of Zielonka's construction [17] is detailed. The number of local states $|Q_p|$ built by Zielonka's technique for each process $p \in \mathcal{P}$ is $|Q_k| \leq 2^{O(2^{|\mathcal{P}|}|Q|\log(|Q|))}$. The simplified construction by Cori et al. in [3] also suffers from this exponential state-explosion [4]. More recently in [6], Genest and Muscholl improve the construction of [17]. To do this, they add to this construction the concept of zones to reduce the amount of information to store in each process. This new algorithm, that yields a deterministic AA still, is thus exponential in $|Q|$ and $|\mathcal{P}|$.

As for the non-deterministic approaches, the construction of Pighizzini [15] build non-deterministic AAs from particular rational expressions. This simpler approach gives

AAs whose number of local states in each process is exponential in the length of the rational expression. Another construction of non-deterministic asynchronous automata is presented in [2]. In this paper the number of local states built for each process is polynomial in $|Q|$ and double-exponential in $|\Sigma|$. In comparison, our construction builds non-deterministic AAs that are also polynomial in $|Q|$, but only exponential in $|\Sigma|$.

6 Expressive Power of Distributed Asynchronous Automata

In Corollary 2.3, we have shown that any regular trace language can be expressed as a finite union of languages of DAAs. However, as illustrated in Example 2.2, some of them are the language of no DAA. In this section, we characterize effectively in Theorem 6.5 those that correspond to the behaviours of non-deterministic DAAs: these are the ones that are *distributed*. Interestingly, for any given distributed regular trace language, the construction presented in Subsection 4.2 yields directly a DAA recognizing it.

Let L be a regular trace language and T be a nonempty subset of Σ. A word u is a *T-independent prefix* of L if there exists a word $uv \in L$ such that $\text{alph}(u) = T$ and $\text{Loc}(T) \cap \text{Loc}(v) = \emptyset$. We denote by $L\|T$ the set of all T-independent prefixes of L. Then $L\|T$ is a regular trace language. Note that, for a DAA $\mathcal{H}$ recognizing L, the set $L\|T$ represents all the words u with $\text{alph}(u) = T$ that can lead all the processes of $\text{Loc}(T)$ to final local states, independently from the behaviours of the other processes.

The next definition characterizes the trace languages that are recognizable by DAAs.

Definition 6.1. *Let L be a regular trace language over (Σ, I). L is* distributed *if one of the two following conditions holds for all nonempty $T \subseteq \Sigma$ with $\text{cc}(T) = (T_1, \ldots, T_n)$:*

C1: $\exists p \notin \text{Loc}(T), \forall v \in L : p \in \text{Loc}(v)$;
C2: $\prod_{i\in[1,n]} L\|T_i \subseteq L$.

Let L be a regular trace language that is not distributed. Then Conditions $C1$ and $C2$ fail for some subset of actions T with $\text{cc}(T) = (T_1, \ldots T_n)$. Suppose that there exists a DAA $\mathcal{H}$ that recognizes L. Since $C2$ fails for T there are words $u_1 \in L\|T_1, \ldots, u_n \in L\|T_n$ such that $u = u_1 \cdots u_n \notin L$. For each $u_i \in L\|T_i$, the processes of $\text{Loc}(T_i)$ can perform u_i and reach final local states. Then, all together, the processes of $\text{Loc}(T)$ can perform the word u and reach final local states, and this because $\text{Loc}(T_i)$ and $\text{Loc}(T_j)$ are disjoint for all $i \neq j$, . Since Condition $C1$ fails for T, the other processes can reach final local states without producing any action. In conclusion, u belongs to $L(\mathcal{H})$ while it doesn't belong to L, which contradicts that $\mathcal{H}$ recognizes L. This leads immediately to the following lemma:

Lemma 6.2. *The language recognized by a* DAA *is distributed.*

Example 6.3. Consider again the regular trace language $L = [aab] \cup [abb]$ of Example 2.2 that is recognized by no DAA. Now heed of the independent prefixes of L: $L\|\{a\} = \{a, aa\}$, $L\|\{b\} = \{b, bb\}$ and $L\|\{ab\} = L$. We can see that $ab \in L\|\{a\} \cdot L\|\{b\}$ while $ab \notin L$, which means that Condition $C2$ fails for $T = \{a, b\}$. Moreover, it is clear that Condition $C1$ fails for T as well. Then L is not distributed.

On the other hand, the next lemma shows that any distributed regular trace language is the language of some *non-deterministic* high-level DAA. Recall here that Theorem 4.4 states that any located trace language is recognized by a $\mathrm{alph}(r)$-synchronizable DAA.

Lemma 6.4. *Let $\mathcal{A}$ be a trace automaton. For all $r \in \mathcal{R}_f(\mathcal{A})$, we denote by $\mathcal{H}_r$ the $\mathrm{alph}(r)$-synchronizable* DAA *that recognizes the located trace language $L_r(\mathcal{A})$. If $L(\mathcal{A})$ is distributed, then $\oplus_{r\in\mathcal{R}_f(\mathcal{A})}\mathcal{H}_r$ recognizes $L(\mathcal{A})$.*

Proof. In this proof, we denote $\oplus_{r\in\mathcal{R}_f(\mathcal{A})}\mathcal{H}_r$ by $\mathcal{H}$. It is not hard to prove that $L(\mathcal{A})$ is included in $L(\mathcal{H})$. So we prove only the backward inclusion. Let $u \in L(\mathcal{H})$, $T = \mathrm{alph}(u)$ and $\mathrm{cc}(T) = (T_1, \ldots, T_n)$. By definition of $\mathrm{cc}(T)$, $\mathrm{Loc}(T_i) \cap \mathrm{Loc}(T_j) = \emptyset$ for all $1 \leq i < j \leq n$. Since $\mathrm{alph}(u) = T$, u is equivalent (w.r.t. $\sim$) to some word $u_1 \cdots u_n$ where $\mathrm{alph}(u_i) = T_i$ for each $i \in [1, n]$. Moreover $u_1 \cdots u_n$ belongs to $L(\mathcal{H})$ as well, because the latter is a trace language. So, there is an execution s of $\mathcal{H}$ that yields $u_1 \cdots u_n$ and leads all processes of $\mathcal{P}$ from initial states to final states.

Now consider any process $p \notin \mathrm{Loc}(T)$. Then, along s, p takes part in mute transitions only. By construction of $\mathcal{H}$ and because each DAA $\mathcal{H}_r$ is $\mathrm{alph}(r)$-synchronizable, the only way for p to do only mute transitions is to start in an initial local state of some DAA component $\mathcal{H}_{r_p}$ with $p \notin \mathrm{Loc}(\mathrm{alph}(r_p))$. Moreover the definition of $\mathcal{R}_f(\mathcal{A})$ gives that $L(\mathcal{H}_{r_p}) = L_{r_p}(\mathcal{A}) \neq \emptyset$. Then there exists some $v_p \in L_{r_p}(\mathcal{A})$ such that $p \notin \mathrm{Loc}(v_p)$. Since similar facts hold for all processes that are not in $\mathrm{Loc}(T)$ and $L_{r_p}(\mathcal{A}) \subseteq L(\mathcal{A})$, Condition $C1$ of Def. 6.1 fails for T. However we have supposed that $L(\mathcal{A})$ is distributed. Then Condition $C2$ holds for T, that is $\prod_{i\in[1,n]} L(\mathcal{A})\|T_i \subseteq L(\mathcal{A})$.

Let $i \in [1, n]$. We know that $\mathrm{alph}(u_i) = T_i$, (T_i, D) is connected (by def. of $\mathrm{cc}(T)$) and each DAA component $\mathcal{H}_r$ of $\mathcal{H}$ is $\mathrm{alph}(r)$-synchronizable. Then, along s, all processes of $\mathrm{Loc}(T_i)$ (those that take part in u_i) are forced to travel all together some DAA component $\mathcal{H}_{r_i}$ such that T_i appears in $\mathrm{cc}(\mathrm{alph}(r_i))$. Furthermore, $L(\mathcal{H}_{r_i}) = L_{r_i}(\mathcal{A})$ and the definition of $\mathcal{R}_f(\mathcal{A})$ gives that $L_{r_i}(\mathcal{A}) \neq \emptyset$. So, there is some word v_i such that $u_i v_i \in L_{r_i}(\mathcal{A})$ and $\mathrm{Loc}(v_i) \cap \mathrm{Loc}(T_i) = \emptyset$. In other words, $u_i \in L_{r_i}(\mathcal{A})\|T_i$. By Prop. 4.3, $L(\mathcal{A}) = \bigcup_{r\in\mathcal{R}_f(\mathcal{A})} L_r(\mathcal{A})$. It follows that $u_i \in L(\mathcal{A})\|T_i$.

To conclude, we have just shown that $\prod_{i\in[1,n]} L(\mathcal{A})\|T_i \subseteq L(\mathcal{A})$ and $u_i \in L(\mathcal{A})\|T_i$ for all $i \in [1, n]$. In consequence, $u_1 \cdots u_n$ belongs to $L(\mathcal{A})$. Since $L(\mathcal{A})$ is a trace language, u belongs to $L(\mathcal{A})$ as well. ■

We come to our main result that follows immediately from the two previous lemmas. Again, for each roadmap r in $\mathcal{R}_f(\mathcal{A})$, $\mathcal{H}_r$ refers to the $\mathrm{alph}(r)$-synchronizable DAA that recognizes the located trace language $L_r(\mathcal{A})$.

Theorem 6.5. *Let L be a regular trace language. The next statements are equivalent:*

1. *L is recognized by some* DAA *;*
2. *L is distributed;*
3. *$L = L(\oplus_{r\in\mathcal{R}_f(\mathcal{A})}\mathcal{H}_r)$ for all trace automata $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*
4. *$L = L(\oplus_{r\in\mathcal{R}_f(\mathcal{A})}\mathcal{H}_r)$ for some trace automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*

As corollary we can check effectively whether a given regular trace language is distributed. For instance it suffices to check whether $L(\oplus_{r\in\mathcal{R}_f(\mathcal{A})}\mathcal{H}_r) = L(\mathcal{A})$ for the minimal trace automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$.

Discussion

In this paper, we have paid attention to the particular case of non-deterministic distributed asynchronous automata. We have shown that they correspond to the regular trace languages that are distributed. Interestingly, we can verify whether or not a language is distributed. However we do not know the complexity of this problem, yet.

Another question is to determine how to verify whether a regular trace language is recognized by a *deterministic* DAA. Indeed, the class of regular trace languages recognized by deterministic DAAs is different from the one recognized by non-deterministic DAAs. For instance, consider the regular trace language $L = [c^\star a] \cup [c^\star b]$ over $\{a, b, c\}$ where $\mathrm{Loc}(a) = \{i\}$, $\mathrm{Loc}(b) = \{j\}$ and $\mathrm{Loc}(c) = \{i, j\}$. It is not hard to see that L is distributed. Then Theorem 6.5 ensures that L is recognized by some DAA. However, no deterministic DAA recognizes L.

Acknowledgment. The author would like to thank Rémi Morin for fruitful discussions and comments. We are grateful to the referees for their constructive criticism.

References

1. Baudru, N., Morin, R.: Safe Implementability of Regular Message Sequence Charts Specifications. In: Proc. of the ACIS 4th Int. Conf. SNDP, pp. 210–217 (2003)
2. Baudru, N., Morin, R.: Unfolding synthesis of asynchronous automata. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 46–57. Springer, Heidelberg (2006)
3. Cori, R., Métivier, Y., Zielonka, W.: Asynchronous mappings and asynchronous cellular automata. Inform. and Comput. 106, 159–202 (1993)
4. Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific, Singapore (1995)
5. Diekert, V., Muscholl, A.: Construction of asynchronous automata, ch. 5(4) (1995)
6. Genest, B., Muscholl, A.: Constructing Exponential-size Deterministic Zielonka Automata. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 565–576. Springer, Heidelberg (2006)
7. Genest, B., Kuske, D., Muscholl, A.: A Kleene theorem and model checking algorithms for existentially bounded communicating automata. I&C 204, 902–956 (2006)
8. Henriksen, J.G., Mukund, M., Narayan Kumar, K., Sohoni, M., Thiagarajan, P.S.: A Theory of Regular MSC Languages. I&C 202, 1–138 (2005)
9. Klarlund, N., Mukund, M., Sohoni, M.: Determinizing Asynchronous Automata. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 130–141. Springer, Heidelberg (1994)
10. McNaughton, R., Yamada, H.: Regular Expressions and State Graphs for Automata J. Symbolic Logic 32, 390–391 (1967)
11. Métivier, Y.: An algorithm for computing asynchronous automata in the case of acyclic non-commutation graph. In: Ottmann, T. (ed.) ICALP 1987. LNCS, vol. 267, pp. 226–236. Springer, Heidelberg (1987)
12. Morin, R.: Concurrent Automata vs. Asynchronous Systems. In: Jedrzejowicz, J., Szepietowski, A. (eds.) MFCS 2005. LNCS, vol. 3618, pp. 686–698. Springer, Heidelberg (2005)

13. Mukund, M., Sohoni, M.: Gossiping, Asynchronous Automata and Zielonka's Theorem Report TCS-94-2, SPIC Science Foundation Madras, India (1994)
14. Muscholl, A.: On the complementation of Büchi asynchronous cellular automata. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 142–153. Springer, Heidelberg (1994)
15. Pighizzini, G.: Synthesis of Nondeterministic Asynchronous Automata. Algebra, Logic and Applications 5, 109–126 (1993)
16. Thiagarajan, P.S.: Regular Event Structures and Finite Petri Nets: A Conjecture. In: Brauer, W., Ehrig, H., Karhumäki, J., Salomaa, A. (eds.) Formal and Natural Computing. LNCS, vol. 2300, pp. 244–256. Springer, Heidelberg (2002)
17. Zielonka, W.: Notes on finite asynchronous automata. RAIRO, Theoretical Informatics and Applications 21, 99–135 (1987)

Algebra for Infinite Forests with an Application to the Temporal Logic EF[⋆]

Mikołaj Bojańczyk and Tomasz Idziaszek

University of Warsaw, Poland
{bojan,idziaszek}@mimuw.edu.pl

Abstract. We define an extension of forest algebra for ω-forests. We show how the standard algebraic notions (free object, syntactic algebra, morphisms, etc.) extend to the infinite case. To prove its usefulness, we use the framework to get an effective characterization of the ω-forest languages that are definable in the temporal logic that uses the operator EF (exists finally).

1 Introduction

The goal of this paper is to explore an algebraic approach to infinite trees. We have decided to take a two-pronged approach:

– Develop a concept of forest algebra for infinite trees, extending to infinite trees the forest algebra defined in [8].
– Use the algebra to get an effective characterization for some logic (that is, an algorithm that decides which regular languages can be defined in the logic).

A good effective characterization benefits the algebra. Effective characterizations are usually difficult problems, and require insight into to the structure of the underlying algebra. We expected that as a byproduct of an effective characterization, we would discover what are the important ingredients of the algebra.

A good algebra benefits effective characterizations. A good algebra makes proofs easier and statements more elegant. We expected that an effective characterization would be a good test for the quality of an algebraic approach. In the previously studied cases of (infinite and finite) words and finite trees, some of the best work on algebra was devoted to effective characterizations.

We hope the reader will find that these expectations have been fulfilled.

Why the logic EF? What tree logic should we try to characterize? Since we are only beginning to explore the algebra for infinite trees, it is a good idea to start with some logic that is very well understood for finite trees. This is why for our case study we chose the temporal logic EF. For finite trees, this was one of the first nontrivial tree logic to get an effective characterization, for binary trees in [7], and for unranked trees [8]. Moreover, when stated in algebraic terms – as

⋆ Work partially funded by the Polish government grant no. N206 008 32/0810.

in [8] – this characterization is simple: there are two identities $h+g=g+h$ and $vh=vh+h$ (these will be explained later in the paper).

We were also curious how some properties of the logic EF would extend from finite trees to infinite trees. For instance, for finite trees, a language can be defined in the logic EF if and only if it is closed under EF-bisimulation (a notion of bisimulation that uses the descendant relation instead of the child relation). What about infinite trees? (We prove that this is not the case.) Another example: for finite trees, a key proof technique is induction on the size of a tree. What about infinite trees? (Our solution is to use only regular trees, and do the induction on the number of distinct subtrees.)

Our approach to developing an algebra. Suppose you want to develop a new kind of algebra. The algebra should be given by a certain number of operations and a set of axioms that these operations should satisfy. For instance, in the case of finite words, there is one operation, concatenation, and one axiom, associativity (such a structure, of course, is called a semigroup). Given a finite alphabet A, the set of all nonempty words A^+ is simply the free semigroup. Regular languages are those that are recognized by morphisms from the free semigroup into a finite semigroup.

This approach was used in [8] to define forest algebra, an algebraic framework for finite unranked trees. The idea was to develop operations and axioms such that the free object would contain all trees. One idea was to have a two-sorted algebra, where one sort described forests (sequences of unranked trees), and the other sort described contexts (forests with a hole). Forest algebra has been successfully applied in a number of effective characterizations, including fragments of first-order logic [6,5] and temporal logics [3], see [4] for a survey. An important open problem is to find an effective characterization of first-order logic with the descendant relation (first-order with the child relation was effectively characterized in [1]).

When developing an algebraic framework for infinite words (and even worse, infinite trees), we run into a problem. For an alphabet A with at least two letters, the set A^ω of all infinite words is uncountable. On the other hand, the free object will be countable, as long as the number of operations is countable. There are two solutions to this problem: either have an uncountable number of operations, or have a free object that is different from A^ω. The first approach is called an ω-semigroup (see the monograph [10]). The second approach is called a Wilke semigroup [11]. Like in forest algebra, a Wilke semigroup is a two-sorted object. The axioms and operations are designed so that the free object will have all finite words on the first sort, and all ultimately periodic words on the second sort. Why is it possible to ignore words that are not ultimately periodic? The reason is that any ω-regular language $L \subseteq A^\omega$ is uniquely defined by the ultimately periodic words that it contains. In this sense, a morphism from the free Wilke semigroup into a finite Wilke semigroup contains all the information about an ω-regular language.

Our approach to infinite trees combines forest algebra and Wilke semigroups. As in forest algebra, we have two sorts: forests and contexts. Both the forests and the contexts can contain infinite paths, although the hole in a context has to be at finite depth (since there is no such thing as a hole at infinite depth). As in a Wilke semigroup, the free object does not contain all forests or all contexts, but only contain the regular ones (a forest or context is regular if it has a finite number of nonisomorphic subtrees, which is the tree equivalent of ultimately periodic words).

Organization of the paper. In Section 2 we present the basic concepts, such as trees, forests, contexts and automata. The algebra is defined in Section 3. In Section 4, we define the logic EF and present a characterization, which says that a language can be defined in EF if and only if: (a) it is invariant under EF-bisimulation; and (b) its syntactic algebra satisfies a certain identity. There are three tasks: prove the characterization, decide condition (a), and decide condition (b). Each of these tasks is nontrivial and requires original ideas. Due to a lack of space, the algorithms for deciding (a) and (b) are relegated to the appendix. We end the paper with a conclusions section. Apart from the usual ideas for future work, we try to outline the limitations of our approach.

2 Preliminaries

2.1 Trees and Contexts

This paper mainly studies forests, which are ordered sequences of trees. Forests and trees can have both infinite and finite maximal paths, but each node must have finitely many siblings. Formally, a forest over finite alphabet A is a partial map $t : \mathbb{N}^+ \to A$ whose domain (the set of nodes) is closed under nonempty prefixes, and such that for each $x \in \mathbb{N}^*$, the set $\{i : x \cdot i \in \mathrm{dom}(t)\}$ is a finite prefix of $\mathbb{N}$. We use letters s, t for forests. An empty forest is denoted by 0. We use the terms root (there may be several, these are nodes in $\mathbb{N}$), leaf, child, ancestor and descendant in the standard way. A *tree* is an forest with one root. If t is a forest and x is a node, we write $t|_x$ for the subtree of t rooted in x, defined as $t|_x(y) = t(xy)$.

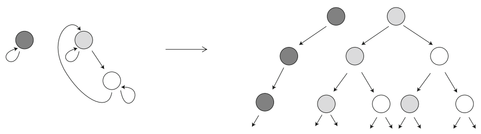
A *context* is a forest with a hole. Formally, a context over an alphabet A is a forest over the alphabet $A \cup \{\square\}$ where the label $\square$, called the hole, occurs exactly once and in a leaf. We use letters p, q to denote contexts. A context is called *guarded* if the hole is not a root.

Operations on forests and contexts. We define two types of operations on forests and contexts: a *(horizontal) concatenation operation*, written additively, and a *(vertical) composition operation*, written multiplicatively. In general, neither concatenation nor composition is commutative.

What objects can be concatenated? We can concatenate two forests s, t, the result is a forest $s + t$ that has as many roots as s and t combined. (We do not assume that concatenation is commutative, so $s + t$ need not be the same as $t + s$.) Since contexts can be interpreted as forests with a hole label, we can also concatenate a forest s with a context p, with the being result is a context $s + p$. There is also the symmetric concatenation $p + s$. In other words, we can concatenate anything with anything, as long as it is not two contexts (otherwise we could get two holes).

What objects can be composed? We can compose a context p with a forest t, the result is a forest pt obtained by replacing the hole of p with the forest t. For instance, if p is a context with a in the root and a hole below, written as $a\square$, and t is a forest (and also tree) with a single node labeled b, written as b, then $p(t+t) = a(b + b)$ is a tree with the root labeled a and two children with label b. We can also compose a context p with another context q, the resulting context pq satisfies $pqt = p(qt)$ for all forests t. We cannot compose a forest t with a context p, or another forest s, since t has no hole.

Regular forests and recursion schemes. A forest is called *regular* if it has finitely many distinct subtrees. Regular forests are important for two reasons: a) they can be represented in a finite way; and b) regular languages are uniquely determined by the regular forests they contain. One way of representing a regular forest is as a forest with backward loops, as in the picture below.



The formal definition of forests with backward loops is presented below, under the name of recursion schemes. Let $Z = Z_H \cup Z_V$ be a set of *label variables*. The set Z_H represents forest-sorted variables and the set Z_V represents context-sorted variables. Let Y be a set of *recursion variables*. *Recursion terms* are built in the following way:

1. 0 is a recursion term.
2. If $\tau_1, \ldots, \tau_n$ are recursion terms, then so is $\tau_1 + \cdots + \tau_n$.
3. Every forest-sorted label variable $z \in Z_H$ is a recursion term.
4. If $z \in Z_V$ is a context-sorted label variable and τ is a recursion term, then $z\tau$ is a recursion term.
5. Every recursion variable $y \in Y$ is a recursion term.
6. If $y \in Y$ a recursion variable and τ is a recursion term where y is guarded, then $\nu y.\tau$ is a recursion term. We say a recursion variable y is guarded in a recursion term τ if there is no decomposition $\tau = \tau_1 + y + \tau_2$.

A *recursion scheme* is a recursion term without free recursion variables, i.e. a recursion term where every recursion variable y occurs in as a subterm of a term $\nu y.\tau$. We denote recursion schemes and terms using letters τ, σ. We also assume that each recursion variable is bound at most once, to avoid discussing scope of variables.

Let η be a function (called a *valuation*) that maps forest-sorted label variables to forests and context-sorted label variables to guarded contexts. We define $\text{unfold}_\tau[\eta]$ to be the (possibly infinite) forest obtained by replacing the label z with their values $\eta(z)$, and unfolding the loops involving the recursion variables. The formal definition is in the appendix. If the recursion scheme uses only m forest-sorted variables $z_1, \ldots, z_m$ and n context-sorted variables $z'_1, \ldots, z'_n$ (we call this an (m, n)-ary recursion scheme), then only the values of η on these variables are relevant. In such a case, we will interpret unfold_τ as a function from tuples of m forests and n contexts into forests.

For instance, suppose that z' is a context-sorted variable and z_1, z_2 are forest sorted variables. For $\tau = z'z_1 + z_2$, $\text{unfold}_\tau(t_1, t_2, p) = pt_1 + t_2$, and if $\tau = \nu y.z'z'y$ then $\text{unfold}_\tau(p)$ is the infinite forest $ppp\cdots$. Note that the notation $\text{unfold}_\tau(t_1, t_2, p)$ uses an implicit order on the label variables.

Note 1. Why only allow guarded contexts as inputs for the unfolding? For the same reason as restricting the binding $\nu y.\tau$ to terms τ where y is guarded. Take, for instance a recursion scheme $\tau = \nu y.zy$. What should the result of $\text{unfold}_\tau(a + \Box)$ be, for the unguarded context $a + \Box$? We could say that this is the a forest $a + a + \cdots$ that is infinite to the right. But then, in a similar way, we could generate the forest $\cdots + a + a$ that is infinite to the left. What would happen after concatenating the two forests? In order to avoid such problems, we only allow contexts where the hole is not in the root. Another solution would be to suppose that the order on siblings is an arbitrary linear ordering.

Lemma 2.1. *Regular forests are exactly the unfoldings of recursion schemes.*

2.2 Automata for Unranked Infinite Trees

A (nondeterministic parity) *forest automaton* over an alphabet A is given by a set of states Q equipped with a monoid structure, a transition relation $\delta \subseteq Q \times A \times Q$, an initial state $q_I \in Q$ and a parity condition $\Omega : Q \to \{0, \ldots, k\}$. We use additive notation $+$ for the monoid operation in Q, and we write 0 for the neutral element.

A *run* of this automaton over a forest t is a labeling $\rho : \text{dom}(t) \to Q$ of forest nodes with states such that for any node x with children $x_1, \ldots, x_n$,

$$(\rho(x_1) + \rho(x_2) + \cdots + \rho(x_n), t(x), \rho(x)) \in \delta.$$

Note that if x is a leaf, then the above implies $(0, t(x), \rho(x)) \in \delta$.

A run is *accepting* if for every infinite path $\pi \subseteq \text{dom}(t)$, the highest value of $\Omega(q)$ is even among those states q which appear infinitely often on the path π. The *value* of a run over a forest t is the obtained by adding, using $+$, all the states assigned to roots of the forest. A forest is *accepted* if it has an accepting run whose value is the initial state q_I. The set of trees accepted by an automaton is called the *regular* language *recognized* by the automaton.

Theorem 2.2
Languages recognized by forest automata are closed under boolean operations and projection. Every nonempty forest automaton accepts some regular forest.

Two consequences of the above theorem are that forest automata have the same expressive power as the logic MSO, and that a regular forest language is determined by the regular forests it contains. We can also transfer complexity results from automata over binary trees to forest automata.

3 Forest Algebra

In this section we define ω-forest algebra, and prove some of its basic properties.

Usually, when defining an algebraic object, such as a semigroup, monoid, Wilke semigroup, or forest algebra, one gives the operations and axioms. Once these operations and axioms are given, a set of generators (the alphabet) determines the free object (e.g. nonempty words in the case of semigroups). Here, we use the reverse approach. We begin by defining the free object. Then, we choose the operations and axioms of ω-forest algebra so that we get this free object.

Let A be an alphabet. Below, we define a structure $A^{\triangle}$. The idea is that $A^{\triangle}$ will turn out to be the free ω-forest algebra generated by A. The structure $A^{\triangle}$ is two-sorted, i.e. it has two domains H_A and V_A. The first domain H_A, called the *forest sort*, consists of all (regular) forests over the alphabet A. The second domain V_A, called the *context sort* consists of all (regular) guarded contexts over the alphabet A. From now on, when writing forest, tree or context, we mean a regular forest, regular tree, or regular guarded context, respectively.

What are the operations? There are eight *basic operations*, as well as infinitely many *recursion operations.*

Basic operations. There is a constant $0 \in H_A$ and seven binary operations

$$\begin{array}{rcl} s,t \in H_A & \mapsto & s+t \in H_A \\ p,q \in V_A & \mapsto & pq \in V_A \\ p \in V_A, s \in H_A & \mapsto & ps \in H_A \\ p \in V_A, s \in H_A & \mapsto & p+s \in V_A \\ p \in V_A, s \in H_A & \mapsto & s+p \in V_A \\ p \in V_A, s \in H_A & \mapsto & p(\Box + s) \in V_A \\ p \in V_A, s \in H_A & \mapsto & p(s+\Box) \in V_A \end{array}$$

If we had all contexts, instead of only guarded contexts, in the context sort, we could replace the last four operations by two unary operations $s \mapsto s + \Box$ and $s \mapsto \Box + s$. However, having unguarded contexts would cause problems for the recursion operations.

Recursion operations. For each (m,n)-ary recursion scheme τ, there is an $(m+n)$-ary operation

$$\begin{array}{c} s_1,\ldots,s_m \in H_A \\ p_1,\ldots,p_n \in V_A \end{array} \quad \mapsto \quad \text{unfold}_\tau(s_1,\ldots,s_m,p_1,\ldots,p_n) \in H_A.$$

We use p^∞ as syntactic sugar for $\text{unfold}_{\nu y.zy}(p)$.

Generators. The operations are designed so that every forest and context over alphabet A can be generated from single letters in A. It is important however, that the alphabet, when treated as a generator for $A^{\triangle}$, is considered as part of the context sort.

More formally, for an alphabet A, we write $A\Box$ for the set of contexts $\{a\Box : a \in A\}$. By Lemma 2.1, the domain H_A is generated by $A\Box \subseteq V_A$. It is also easy to see that every context in V_A is also generated by this set, it suffices to construct the path to the hole in the context and generate all remaining subtrees. Therefore $A^{\triangle}$ is generated by $A\Box$.

Definition of ω-forest algebra. We are now ready to define what an ω-forest algebra is. It is a two sorted structure (H, V). The operations are the same as in each structure $A^{\triangle}$: eight basic operations and infinitely many recursion operations. The axioms that are required in an ω-forest algebra are described in the appendix. These axioms are designed so as to make the following theorem true. Homomorphisms (also called morphisms here) are defined in the appendix.

Theorem 3.1
Let A be a finite alphabet, and let (H, V) be an ω-forest algebra. Any function $f : A\Box \to V$ uniquely extends to a morphism $\alpha : A^{\triangle} \to (H, V)$.

Recognizing Languages with an ω-Forest Algebra

A set L of A-forests is said to be *recognized* by a surjective morphism $\alpha : A^{\triangle} \to (H, V)$ onto a finite ω-forest algebra (H, V)f membership $t \in L$ depends only on the value $\alpha(t)$. The morphism α, and also the target ω-forest algebra (H, V), are said to recognize L.

The next important concept is that of a syntactic ω-forest algebra of a forest language L. This is going to be an ω-forest algebra that recognizes the language, and one that is optimal (in the sense of 3.3) among those that do.

Let L be a forest language over an alphabet A. We associate with a forest language L two equivalence relations (à la Myhill-Nerode) on the free ω-forest algebra $A^{\triangle}$. The first equivalence, on contexts is defined as follows. Two contexts p, q are called *L-equivalent* if for every forest-valued term ϕ over the signature of ω-forest algebra, any valuation $\eta : X \to A^{\triangle}$ of the variables in the term, and any context-sorted variable x, either both or none of the forests

$$\phi[\eta[x \mapsto p]] \qquad \text{and} \qquad \phi[\eta[x \mapsto q]]$$

belong to L. Roughly speaking, the context p can be replaced by the context q inside any regular forest, without affecting membership in the language L. The notion of L-equivalence for forest s, t is defined similarly. We write $\sim_L$ for L-equivalence. Using universal algebra, it is not difficult to show:

Lemma 3.2. *L-equivalence, as a two-sorted equivalence relation, is a congruence with respect to the operations of the ω-forest algebra $A^{\triangle}$.*

The *syntactic algebra* of a forest language L is the quotient (H_L, V_L) of $A^{\triangle}$ with respect to L-equivalence, where the horizontal semigroup H_L consists of equivalence classes of forests over A, while the vertical semigroup V_L consists of equivalence classes of contexts over A. The syntactic algebra is an ω-forest algebra thanks to Lemma 3.2. The *syntactic morphism* α_L assigns to every element of $A^{\triangle}$ its equivalence class under L-equivalence. The following proposition shows that the syntactic algebra has the properties required of a syntactic object.

Proposition 3.3. A forest language L over A is recognized by its syntactic morphism α_L. Moreover, any morphism $\alpha : A^{\triangle} \to (H, V)$ that recognizes L can be extended by a morphism $\beta : (H, V) \to (H_L, V_L)$ so that $\beta \circ \alpha = \alpha_L$.

Note that in general the syntactic ω-forest algebra may be infinite. However, Proposition 3.3 shows that if a forest language is recognized by some finite forest algebra, then its syntactic forest algebra must also be finite. In the appendix we show that all regular forest languages have finite ω-forest algebras, which can furthermore be effectively calculated (since there are infinitely many operations, we also specify what it means to calculate an ω-forest algebra).

We use different notation depending on whether we are dealing with the free algebra, or with a (usually finite) algebra recognizing a language. In the first case, we use letters s, t for elements of H and p, q, r for elements of V, since these are "real" forests and contexts. In the second case, we will use letters f, g, h for elements of H and u, v, w for elements of V, and call them forest types and context types respectively.

4 EF for Infinite Trees

In this section we present the main technical contribution of this paper, which is an effective characterization of the forest and tree languages that can be defined in the temporal logic EF. We begin by defining the logic EF. Fix an alphabet A.

- Every letter $a \in A$ is an EF formula, which is true in trees with root label a.
- EF formulas admit boolean operations, including negation.
- If φ is an EF formula, then $\mathsf{EF}\varphi$ is an EF formula, which is true in trees that have a proper subtree where φ is true. EF stands for Exists Finally.

A number of operators can be introduced as syntactic sugar:

$$\mathsf{AG}\varphi = \neg\mathsf{EF}\neg\varphi, \qquad \mathsf{AG}^*\varphi = \varphi \wedge \mathsf{AG}\varphi, \qquad \mathsf{EF}^*\varphi = \varphi \vee \mathsf{EF}\varphi.$$

Since we deal with forest languages in this paper, we will also want to define forest languages using the logic. It is clear which forests should satisfy the formula EF^*a (some node in the forest has label a). It is less clear which forests should satisfy $\mathsf{EF}a$ (only non-root nodes are considered?), and even less clear which forests should satisfy a (which root node should have label a?). We will only use boolean combinations of formulas of the first kind to describe forests. That is, a *forest EF formula* is a boolean combination of formulas of the form $\mathsf{EF}^*\varphi$.

For finite forests, the logic EF was characterized in [8]. The result was:

Theorem 4.1
Let L be a language of finite forests. There is a forest formula of EF that is equivalent, over finite forests, to L if and only if the syntactic forest algebra (H_L, V_L) of L satisfies the identities

$$vh = vh + h \qquad \textit{for } h \in H_L, v \in V_L, \tag{1}$$

$$h + g = g + h \qquad \textit{for } g, h \in H_L. \tag{2}$$

A corollary to the above theorem is that, for finite forests, definability in EF is equivalent to invariance under EF-bisimulation. This is because two finite trees that are EF-bisimilar can be rewritten into each other using the identities (1) and (2).

Our goal in this paper is to test ω-forest algebra by extending Theorem 4.1 to infinite forests. There are nontrivial properties of infinite forests that can be expressed in EF. Consider for example the forest formula $\mathsf{AG}^*(a \wedge \mathsf{EF}a)$. This formula says that all nodes have label a and at least one descendant. Any forest that satisfies this formula is bisimilar to the tree $(a\Box)^\infty$. In this paper, we will be interested in a weaker form of bisimilarity (where more forests are bisimilar), which we will call EF-bisimilarity, and define below.

EF game. We define a game, called the *EF game*, which is used to test the similarity of two forests under forest formulas of EF. The name EF comes from the logic, but also, conveniently, is an abbreviation for Ehrenfeucht-Fraïssé.

Let t_0, t_1 be forests. The EF game over t_0 and t_1 is played by two players: Spoiler and Duplicator. The game proceeds in rounds. At the beginning of each round, the state in the game is a pair of forests, (t_0, t_1). A round is played as follows. First Spoiler selects one of the forests t_i $(i = 0, 1)$ and its subtree s_i, possibly a root subtree. Then Duplicator selects a subtree s_{1-i} in the other tree t_{1-i}. If the root labels a_0, a_1 of s_0, s_1 are different, then Spoiler wins the whole game. Otherwise the round is finished, and a new round is played with the state updated to (r_0, r_1) where the forest r_i is obtained from the tree s_i by removing the root node, i.e $s_i = a_i r_i$.

This game is designed to reflect the structure of EF formulas, so the following theorem, which is proved by induction on m, should not come as a surprise.

Fact 4.2. Spoiler wins the m-round EF game on forests t_0 and t_1 if and only if there is a forest EF formula of EF-nesting depth m that is true in t_0 but not t_1.

We will also be interested in the case when the game is played for an infinite number of rounds. If Duplicator can survive for infinitely many rounds in the game on t_0 and t_1, then we say that the forests t_0 and t_1 are *EF-bisimilar.* A forest language L is called *invariant under EF-bisimulation* if it is impossible to find two forests $t_0 \in L$ and $t_1 \notin L$ that are EF-bisimilar.

Thanks to Fact 4.2, we know that any language defined by a forest formula of EF is invariant under EF-bisimulation. Unlike for finite forests, the converse does not hold[1]. Consider, for instance the language "all finite forests". This language is invariant under EF-bisimulation, but it cannot be defined using a forest formula of EF, as will follow from our main result, stated below.

Theorem 4.3 (Main Theorem: Characterization of EF)
A forest language L can be defined by a forest formula of EF if and only if

- *It is invariant under EF-bisimulation;*
- *Its syntactic ω-forest algebra (H_L, V_L) satisfies the identity*

$$v^{\omega}h = (v + v^{\omega}h)^{\infty} \qquad \textit{for all } h \in H_L, v \in V_L. \tag{3}$$

Recall that we have defined the ∞ exponent as syntactic sugar for unfolding the context infinitely many times. What about the ω exponent in the identity? In the syntactic algebra, V_L is a finite monoid (this is proved in the appendix). As in any finite monoid, there is a number $n \in \mathbb{N}$ such that v^n is an idempotent, for any $v \in V_L$. This number is written as ω. Note that identity (3) is not satisfied by the language "all finite forests". It suffices to take v to be the image, in the syntactic algebra, of the forest $a\Box$. In this case, the left side of (3) corresponds to a finite forest, and the right side corresponds to an infinite forest.

In the appendixwe show that the two conditions (invariance and the identity) are necessary for definability in EF. The proof is fairly standard. The more interesting part is that the two conditions are sufficient, this is done in following section.

Corollary 4.4 (of Theorem 4.3). The following problem is decidable. The input is a forest automaton. The question is: can the language recognized by the forest automaton be defined by a forest formula of EF.

Proof
In appendix we show how to decide property (3) (actually, we show more: how to decide any identity). In appendix we show how to decide invariance under EF-bisimulation. An ExpTime lower bound holds already for deterministic automata over finite trees, see [7], which can be easily adapted to this setting. Our algorithm for (3) is in ExpTime, but we do not know if invariance under EF-bisimulation can also be tested in ExpTime (our algorithm is doubly exponential). $\Box$

Note 2. Theorem 4.3 speaks of forest languages defined by forest EF formulas. What about tree languages, defined by normal EF formulas? The latter can be reduced to the former. The reason, not difficult to prove, is that a tree language L can be defined by a normal formula of EF if and only if for each label $a \in A$, the forest language $\{t : at \in L\}$ is definable by a forest formula of EF. A tree version of Corollary 4.4 can also be readily obtained.

[1] In the appendix, we show a weaker form of the converse. Namely, for any fixed regular forest t, the set of forests that are EF-bisimilar to t can be defined in EF.

Note 3. In Theorem 4.3, invariance under EF-bisimulation is used as a property of the language. We could have a different statement, where invariance is a property of the syntactic algebra (e.g. all languages recognized by the algebra are invariant under EF-bisimulation). The different statement would be better suited towards variety theory, à la Eilenberg [9]. We intend to develop such a theory.

Invariance under EF-bisimulation and (3) are sufficient

We now show the more difficult part of Theorem 4.3. Fix a surjective morphism $\alpha : A^{\triangle} \to (H, V)$ and suppose that the target ω-forest algebra satisfies condition (3) and that the morphism is invariant under EF-bisimulation (any two EF-bisimilar forests have the same image). For a forest t, we use the name *type of* h for the value $\alpha(h)$. We will show that for any $h \in H$, the language L_h of forests of type h is definable by a forest formula of EF. This shows that the two conditions in Theorem 4.3 are sufficient for definability in EF, by taking α to be the syntactic morphism of L.

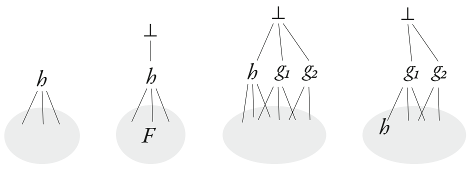
The proof is by induction on the size of H. The induction base, when H has only one element, is trivial. In this case all forests have the same type, and the appropriate formula is *true*.

We now proceed to the induction step. We say that an element $h \in H$ is *reachable* from $g \in H$ if there is some $v \in V$ with $h = vg$.

Lemma 4.5. *The reachability relation is transitive and antisymmetric.*

We say that an element $h \in H$ is *minimal* if it is reachable from all $g \in H$. (The name minimal, instead of maximal, is traditional in algebra. The idea is that the set of elements reachable from h is minimal.) There is at least one minimal element, since for every $v \in V$ an element $v(h_1 + \cdots + h_n)$ is reachable from each h_i. Since reachability is antisymmetric, this minimal element is unique, and we will denote it using the symbol $\perp$. An element $h \neq \perp$ is called *subminimal* if the elements reachable from h are a subset of $\{h, \perp\}$.

Recall that our goal is to give, for any $h \in H$, a formula of EF that defines the set L_h of forests with type h. Fix then some $h \in H$. We consider four cases (shown on the picture):



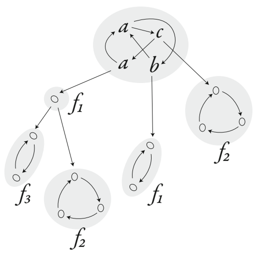
1. h is minimal.
2. h is subminimal, and there is exactly one subminimal element.
3. h is subminimal, but there are at least two subminimal elements.
4. h is neither minimal nor subminimal.

Note that the first case follows from the remaining three, since a forest has type $\perp$ if and only if it has none of the other types. Therefore the formula for $h = \perp$ is obtained by negating the disjunction of the formulas for the other types. Cases 3 and 4 are treated in the appendix. The more interesting case is 2, which is treated below.

Case 2. We now consider the case when h is a unique subminimal element.

Let F be the set of all elements different from h, from which h is reachable. In other words, F is the set of all elements from H beside h and $\perp$. Thanks to cases 3 and 4, for every $f \in F$ we have a formula φ_f that defines the set L_f of forests with type f. We write φ_F for the disjunction $\bigvee_{f \in F} \varphi_f$.

Let t be a forest. We say two nodes x, y of the forest are *in the same component* if the subtree $t|_x$ is a subtree of the subtree $t|_y$ and vice versa. Each regular forest has finitely many components. There are two kinds of component: *connected components*, which contain infinitely many nodes, and *singleton components*, which contain a single node. Since any two nodes in the same component are EF-bisimilar (i.e. their subtrees are EF-bisimilar), we conclude that two nodes in the same component have the same type. Therefore, we can speak of the type of a component. A tree is called *prime* if it has exactly one component with a type outside F. Note that the component with a type outside F must necessarily be the root component (the one that contains the root), since no type from F is reachable from types outside F. Depending on the kind of the root component, a prime tree is called a *connected prime* or *singleton prime*.



The *profile* of a prime tree t is a pair in $P(F) \times (A \cup P(A))$ defined as follows. On the first coordinate, we store the set $G \subseteq F$ of types of components with a type in F. On the second coordinate, we store the labels that appear in the root component. If the tree is connected prime, this is a set $B \subseteq A$ of labels (possibly containing a single label), and if the tree is singleton prime, this is a single label $b \in A$. In the first case, the profile is called a *connected profile*, and in the second case the profile is called a *singleton profile*. The picture shows a connected prime tree with connected profile $(\{f_1, f_2, f_3\}, \{a, b, c\})$.

In the appendix, we show that all prime trees with the same profile have the same type. Therefore, it is meaningful to define the set $prof_h$ of profiles of prime trees of type h.

Proposition 4.6. Let π be a profile. There is an EF formula φ_π such that
- Any prime tree with profile π satisfies φ_π.
- Any regular forest satisfying φ_π has type h if $\pi \in prof_h$ and $\perp$ otherwise.

The above proposition is shown in the appendix. We now turn our attention from prime trees to arbitrary forests. Let t be a forest. The formula which says when the type is h works differently, depending on whether t has a prime subtree or not. This case distinction can be done in EF, since not having a prime subtree is expressed by the formula $\mathsf{AG}^* \varphi_F$.

There is no prime subtree. We write $\sum\{g_1, \ldots, g_n\}$ for $g_1 + \cdots + g_n$. By invariance under EF-bisimulation, this value does not depend on the order or multiplicity of elements in the set. Suppose $\sum G = \perp$ and t has subtrees with each type in G. Thanks to (1), the type of t satisfies $\alpha(t) + \sum G = \alpha(t)$, and hence $\alpha(t) = \perp$.

Therefore, a condition necessary for having type h is

$$\neg \bigvee_{G \subseteq F,\ \sum G = \bot} \ \bigwedge_{g \in G} \mathsf{EF}^* \varphi_g. \tag{4}$$

By the same reasoning, a condition necessary for having type h is

$$\bigvee_{G \subseteq F,\ \sum G = h} \ \bigwedge_{g \in G} \mathsf{EF}^* \varphi_g. \tag{5}$$

It is not difficult to show that conditions (4) and (5) are also sufficient for a forest with no prime subtrees to have type h.

There is a prime subtree. As previously, a forest t with type h cannot satisfy (4). What other conditions are necessary? It is forbidden to have a subtree with type $\bot$. Thanks to Proposition 4.6, t must satisfy:

$$\neg \bigvee_{\pi \notin prof_h} \mathsf{EF}^* \varphi_\pi. \tag{6}$$

Since t has a prime subtree, its type is either h or $\bot$. Suppose that t has a subtree with type $f \in F$ such that $f + h = \bot$. By (1), we would have $\alpha(t) + f = \alpha(t)$, which implies that the type of t is $\bot$. Therefore, t must satisfy

$$\bigwedge_{f \in F,\ f + h = \bot} \neg \mathsf{EF}^* \varphi_f. \tag{7}$$

Let us define C to be the labels that preserve h, i.e. the labels $a \in A$ such that $\alpha(a)h = h$. If a forest has type h then it cannot have a subtree as where $a \notin C$ and s has type h or $\bot$. This is stated by the formula:

$$\mathsf{AG}^* \bigwedge_{c \notin C} (c \rightarrow \mathsf{AG}\varphi_F). \tag{8}$$

As we have seen, conditions (4) and (6)–(8) are necessary for a forest with a prime subtree t having type h. In the following lemma, we show that the conditions are also sufficient. This completes the analysis of Case 2 in the proof of Theorem 4.3, since it gives a formula that characterizes the set L_h of forests whose type is the unique subminimal element h.

Lemma 4.7. *A forest with a prime subtree has type h if it satisfies conditions* (4) *and* (6)*–*(8)*.*

5 Concluding Remarks

We have presented an algebra for infinite trees, and used it to get an effective characterization for the logic EF. In the process, we developed techniques for dealing with the algebra, such as algorithms deciding identities, or a kind of

Green's relation (the reachability relation). It seems that an important role is played by what we call connected components of a regular forest.

There are some natural ideas for future work. These include developing a variety theory, or finding effective characterizations for other logics. Apart from logics that have been considered for finite trees, there are some new interesting logics for infinite trees, which do not have counterparts for finite trees. These include weak monadic-second order logic, or fragments of the μ-calculus with bounded alternation.

However, since we are only beginning to study the algebra for infinite trees, it is important to know if we have the "right" framework (or at least one of the "right" frameworks). Below we discuss some shortcomings of ω-forest algebra, and comment on some alternative approaches.

Shortcomings of ω-forest algebra. A jarring problem is that we have an infinite number of operations and, consequently, an infinite number of axioms. This poses all sorts of problems.

It is difficult to present an algebra. One cannot, as with a finite number of operations, simply list the multiplication tables. Our solution, as presented in the appendix, is to give an algorithm that inputs the name of the operation and produces its multiplication table. In particular, this algorithm can be used to test validity of identities, since any identity involves a finite number of operations.

It is difficult to prove that something is an ω-forest algebra, since there are infinitely many axioms to check. Our solution is to define algebras as homomorphic images of the free algebra, which guarantees that the axioms hold. We give an algorithm that computes the syntactic algebra of a regular forest language.

We have proved that any regular language is recognized by a finite ω-forest algebra. A significant shortcoming is that we have not proved the converse. We do not, however, need the converse for effective characterizations, as demonstrated by our proof for EF. The effective characterization begins with a regular language, and tests properties of its syntactic algebra (therefore, algebras that recognize non-regular languages, if they exist, are never used).

An important advantage of using algebra is that properties can be stated in terms of identities. Do we have this advantage in ω-forest algebra? The answer is mixed, as witnessed by Theorem 4.3. One of the conditions is an identity, namely (3). However, for the other condition, invariance under EF-bisimulation, we were unable to come up with an identity (or a finite set of identities). This contrasts the case of finite trees, where invariance under EF-bisimulation is characterized by two identities. Below we describe an idea on to modify the algebra to solve this problem.

A richer algebra? In preliminary work, we have tried to modify the algebra. In the modified algebra, the context sort is richer, since contexts are allowed to have multiple occurrences of the hole (we still allow only one type of hole). This abundance of contexts changes the interpretation of identities, since the context variables quantify over a larger set. Preliminary results indicate that invariance under EF-bisimulation is described by the identities (1) and (2) – but with the new interpretation – as well as the following two identities:

$$(v(u+w))^\infty = (vuw)^\infty, \qquad (vuw)^\infty = (vwu)^\infty.$$

We intend to explore this richer algebra in more detail. However, allowing a richer context sort comes at a cost. First, it seems that the size of the context sort is not singly exponential, as here, but doubly exponential. Second, there are forest languages definable in first-order logic that are not aperiodic, i.e. do not satisfy $v^\omega = v^{\omega+1}$.

Where is the Ramsey theorem? An important tool in algebra for infinite words is the Ramsey theorem, which implies the following fact: for every morphism $\alpha : A^* \to S$ into a finite monoid, every word $w \in A^\omega$ has a factorization $w = w_0 w_1 w_2 \cdots$ such that all the words $w_1, w_2, \ldots$ have the same image under α. This result allows one to extend a morphism into a Wilke algebra from ultimately periodic words to arbitrary words.

It would be very nice to have a similar theorem for trees. This question has been independently investigated by Blumensath [2], who also provides an algebraic framework for infinite trees. Contrary to our original expectations, we discovered that a Ramsey theorem for trees is not needed to provide effective characterizations.

Acknowledgement. We thank Wojciech Czerwiński for many helpful discussions.

References

1. Benedikt, M., Segoufin, L.: Regular tree languages definable in FO. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 327–339. Springer, Heidelberg (2005); A revised version, correcting an error from the conference paper, www.lsv.ens-cachan.fr/~segoufin/Papers/
2. Blumensath, A.: Recognisability for algebras of infinite trees (unpublished, 2009)
3. Bojańczyk, M.: Two-way unary temporal logic over trees. In: Logic in Computer Science, pp. 121–130 (2007)
4. Bojańczyk, M.: Effective characterizations of tree logics. In: PODS, pp. 53–66 (2008)
5. Bojańczyk, M., Segoufin, L.: Tree languages defined in first-order logic with one quantifier alternation. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 233–245. Springer, Heidelberg (2008)
6. Bojańczyk, M., Segoufin, L., Straubing, H.: Piecewise testable tree languages. In: Logic in Computer Science, pp. 442–451 (2008)
7. Bojańczyk, M., Walukiewicz, I.: Characterizing EF and EX tree logics. Theoretical Computer Science 358(2-3), 255–273 (2006)
8. Bojańczyk, M., Walukiewicz, I.: Forest algebras. In: Automata and Logic: History and Perspectives, pp. 107–132. Amsterdam University Press (2007)
9. Eilenberg, S.: Automata, Languages and Machines, vol. B. Academic Press, New York (1976)
10. Perrin, D., Pin, J.-É.: Infinite Words. Elsevier, Amsterdam (2004)
11. Wilke, T.: An algebraic theory for languages of finite and infinite words. Inf. J. Alg. Comput. 3, 447–489 (1993)

Deriving Syntax and Axioms for Quantitative Regular Behaviours*

Filippo Bonchi[2], Marcello Bonsangue[1,2], Jan Rutten[2,3], and Alexandra Silva[2]

[1] LIACS - Leiden University
[2] Centrum voor Wiskunde en Informatica (CWI)
[3] Vrije Universiteit Amsterdam (VUA)

Abstract. We present a systematic way to generate (1) languages of (generalised) regular expressions, and (2) sound and complete axiomatizations thereof, for a wide variety of quantitative systems. Our quantitative systems include weighted versions of automata and transition systems, in which transitions are assigned a value in a monoid that represents cost, duration, probability, etc. Such systems are represented as coalgebras and (1) and (2) above are derived in a modular fashion from the underlying (functor) type of these coalgebras.

In previous work, we applied a similar approach to a class of systems (without weights) that generalizes both the results of Kleene (on rational languages and DFA's) and Milner (on regular behaviours and finite LTS's), and includes many other systems such as Mealy and Moore machines.

In the present paper, we extend this framework to deal with quantitative systems. As a consequence, our results now include languages and axiomatizations, both existing and new ones, for many different kinds of probabilistic systems.

1 Introduction

Kleene's Theorem [22] gives a fundamental correspondence between *regular expressions* and *deterministic finite automata* (DFA's): each regular expression denotes a language that can be recognized by a DFA and, conversely, the language accepted by a DFA can be specified by a regular expression. Languages denoted by regular expressions are called *regular*. Two regular expressions are (language) equivalent if they denote the same regular language. Salomaa [32] presented a sound and complete axiomatization (later refined by Kozen in [23]) for proving the equivalence of regular expressions.

The above programme was applied by Milner in [26] to process behaviours and labelled transition systems (LTS's). Milner introduced a set of expressions for finite LTS's and proved an analogue of Kleene's Theorem: each expression denotes the behaviour of a finite LTS and, conversely, the behaviour of a finite LTS can be specified by an expression. Milner also provided an axiomatization for his expressions, with the property that two expressions are provably equivalent if and only if they are bisimilar.

Coalgebras provide a general framework for the study of dynamical systems such as DFA's and LTS's. For a functor $G\colon \mathbf{Set} \to \mathbf{Set}$, a G-coalgebra or G-system is a pair

* This work was carried out during the first author's tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. The fourth author is partially supported by the Fundação para a Ciência e a Tecnologia, Portugal, under grant number SFRH/BD/27482/2006.

(S, g), consisting of a set S of states and a function $g{:}S \to GS$ defining the "transitions" of the states. We call the functor G the *type* of the system. For instance, DFA's can be readily seen to correspond to coalgebras of the functor $G(S) = 2 \times S^A$ and image-finite LTS's are obtained by $G(S) = \mathcal{P}_\omega(S)^A$, where $\mathcal{P}_\omega$ is finite powerset.

Under mild conditions, functors G have a *final coalgebra* (unique up to isomorphism) into which every G-coalgebra can be mapped via a unique so-called G-*homomor-phism*. The final coalgebra can be viewed as the universe of all possible G-*behaviours*: the unique homomorphism into the final coalgebra maps every state of a coalgebra to a canonical representative of its behaviour. This provides a general notion of behavioural equivalence: two states are equivalent iff they are mapped to the same element of the final coalgebra. In the case of DFA's, two states are equivalent when they accept the same language; for LTS's, behavioural equivalence coincides with bisimilarity.

For coalgebras of a large but restricted class of functors, we introduced in [7] a language of regular expressions; a corresponding generalisation of Kleene's Theorem; and a sound and complete axiomatization for the associated notion of behavioural equivalence. We derived both the language of expressions and their axiomatization, in a modular fashion, from the functor defining the type of the system.

In recent years, much attention has been devoted to the analysis of probabilistic behaviours, which occur for instance in randomized, fault-tolerant systems. Several different types of systems were proposed: reactive [24, 29], generative [16], stratified [36, 38], alternating [18, 39], (simple) Segala [34, 35], bundle [12] and Pnueli-Zuck [28], among others. For some of these systems, expressions were defined for the specification of their behaviours, as well as axioms to reason about their behavioural equivalence. Examples include [1, 2, 4, 13, 14, 21, 25, 27, 37].

Our previous results [7] apply to the class of so-called Kripke-polynomial functors, which is general enough to include the examples of DFA's and LTS's, as well as many other systems such as Mealy and Moore machines. However, probabilistic systems, weighted automata [15, 33], etc. *cannot* be described by Kripke-polynomial functors. It is the aim of the present paper to identify a class of functors (a) that is general enough to include these and more generally a large class of *quantitative systems*; and (b) to which the methodology developed in [7] can be extended.

To this end, we give a non-trivial extension of the class of Kripke-polynomial functors by adding a functor type that allows the transitions of our systems to take values in a *monoid* structure of quantitative values. This new class, which we shall call quantitative functors, now includes all the types of probabilistic systems mentioned above. We show how to extend our earlier approach to the new setting. As it turns out, the main technical challenge is due to the fact that the behaviour of quantitative systems is inherently *non-idempotent*. As an example consider the expression $1/2 \cdot \varepsilon \oplus 1/2 \cdot \varepsilon'$ representing a probabilistic system that either behaves as ε with probability $1/2$ or behaves as ε' with the same probability. When ε is equivalent to ε', then the system is equivalent to $1 \cdot \varepsilon$ rather than $1/2 \cdot \varepsilon$. This is problematic because idempotency played a crucial role in our previous results to ensure that expressions denote finite-state behaviours. We will show how the lack of idempotency in the extended class of functors can be circumvented by a clever use of the monoid structure. This will allow us to derive for each functor in our new extended class everything we were after: a language of regular expressions;

Table 1. All the expressions are closed and guarded. The congruence and the α-equivalence axioms are implicitly assumed for all the systems. The symbols 0 and $+$ denote, in the case of weighted automata, the empty element and the binary operator of the commutative monoid $\mathbb{S}$ while, for the other systems, denote the ordinary 0 and sum of real numbers. With a slight abuse of notation, we write $\bigoplus_{i \in 1 \cdots n} p_i \cdot \varepsilon_i$ for $p_1 \cdot \varepsilon_1 \oplus \cdots \oplus p_n \cdot \varepsilon_n$.

Weighted automata – $\mathbb{S} \times (\mathbb{S}^{Id})^A$

$\varepsilon ::= \emptyset \mid \varepsilon \oplus \varepsilon \mid \mu x.\varepsilon \mid x \mid s \mid a(s \cdot \varepsilon)$ where $s \in \mathbb{S}$ and $a \in A$

$$\begin{array}{lll}
(\varepsilon_1 \oplus \varepsilon_2) \oplus \varepsilon_3 \equiv \varepsilon_1 \oplus (\varepsilon_2 \oplus \varepsilon_3) & \varepsilon_1 \oplus \varepsilon_2 \equiv \varepsilon_2 \oplus \varepsilon_1 & \varepsilon \oplus \emptyset \equiv \varepsilon \\
a(s \cdot \varepsilon) \oplus a(s' \cdot \varepsilon) \equiv a((s+s') \cdot \varepsilon) & s \oplus s' \equiv s + s' & a(0 \cdot \varepsilon) \equiv \emptyset \\
\varepsilon[\mu x.\varepsilon/x] \equiv \mu x.\varepsilon & \gamma[\varepsilon/x] \equiv \varepsilon \Rightarrow \mu x.\gamma \equiv \varepsilon & 0 \equiv \emptyset
\end{array}$$

Stratified systems – $D_\omega(Id) + (B \times Id) + 1$

$\varepsilon ::= \mu x.\varepsilon \mid x \mid \langle b, \varepsilon \rangle \mid \bigoplus_{i \in 1 \cdots n} p_i \cdot \varepsilon_i \mid \downarrow$ where $b \in B$, $p_i \in (0,1]$ and $\sum_{i \in 1 \ldots n} p_i = 1$

$$\begin{array}{lll}
(\varepsilon_1 \oplus \varepsilon_2) \oplus \varepsilon_3 \equiv \varepsilon_1 \oplus (\varepsilon_2 \oplus \varepsilon_3) & \varepsilon_1 \oplus \varepsilon_2 \equiv \varepsilon_2 \oplus \varepsilon_1 & (p_1 \cdot \varepsilon) \oplus (p_2 \cdot \varepsilon) \equiv (p_1 + p_2) \cdot \varepsilon \\
\varepsilon[\mu x.\varepsilon/x] \equiv \mu x.\varepsilon & \gamma[\varepsilon/x] \equiv \varepsilon \Rightarrow \mu x.\gamma \equiv \varepsilon &
\end{array}$$

Segala systems – $\mathcal{P}_\omega(D_\omega(Id))^A$

$\varepsilon ::= \emptyset \mid \varepsilon \boxplus \varepsilon \mid \mu x.\varepsilon \mid x \mid a(\{\varepsilon'\})$ where $a \in A$, $p_i \in (0,1]$ and $\sum_{i \in 1 \ldots n} p_i = 1$

$\varepsilon' ::= \bigoplus_{i \in 1 \cdots n} p_i \cdot \varepsilon_i$

$$\begin{array}{llll}
(\varepsilon_1 \boxplus \varepsilon_2) \boxplus \varepsilon_3 \equiv \varepsilon_1 \boxplus (\varepsilon_2 \boxplus \varepsilon_3) & \varepsilon_1 \boxplus \varepsilon_2 \equiv \varepsilon_2 \boxplus \varepsilon_1 & \varepsilon \boxplus \emptyset \equiv \varepsilon & \varepsilon \boxplus \varepsilon \equiv \varepsilon \\
(\varepsilon_1' \oplus \varepsilon_2') \oplus \varepsilon_3' \equiv \varepsilon_1' \oplus (\varepsilon_2' \oplus \varepsilon_3') & \varepsilon_1' \oplus \varepsilon_2' \equiv \varepsilon_2' \oplus \varepsilon_1' & \multicolumn{2}{l}{(p_1 \cdot \varepsilon) \oplus (p_2 \cdot \varepsilon) \equiv (p_1 + p_2) \cdot \varepsilon} \\
\varepsilon[\mu x.\varepsilon/x] \equiv \mu x.\varepsilon & \gamma[\varepsilon/x] \equiv \varepsilon \Rightarrow \mu x.\gamma \equiv \varepsilon & &
\end{array}$$

Pnueli-Zuck systems – $\mathcal{P}_\omega D_\omega \mathcal{P}_\omega(Id)^A$

$\varepsilon ::= \emptyset \mid \varepsilon \boxplus \varepsilon \mid \mu x.\varepsilon \mid x \mid \{\varepsilon'\}$ where $a \in A$, $p_i \in (0,1]$ and $\sum_{i \in 1 \ldots n} p_i = 1$

$\varepsilon' ::= \bigoplus_{i \in 1 \cdots n} p_i \cdot \varepsilon_i''$

$\varepsilon'' ::= \emptyset \mid \varepsilon'' \boxplus \varepsilon'' \mid a(\{\varepsilon\})$

$$\begin{array}{llll}
(\varepsilon_1 \boxplus \varepsilon_2) \boxplus \varepsilon_3 \equiv \varepsilon_1 \boxplus (\varepsilon_2 \boxplus \varepsilon_3) & \varepsilon_1 \boxplus \varepsilon_2 \equiv \varepsilon_2 \boxplus \varepsilon_1 & \varepsilon \boxplus \emptyset \equiv \varepsilon & \varepsilon \boxplus \varepsilon \equiv \varepsilon \\
(\varepsilon_1' \oplus \varepsilon_2') \oplus \varepsilon_3' \equiv \varepsilon_1' \oplus (\varepsilon_2' \oplus \varepsilon_3') & \varepsilon_1' \oplus \varepsilon_2' \equiv \varepsilon_2' \oplus \varepsilon_1' & \multicolumn{2}{l}{(p_1 \cdot \varepsilon'') \oplus (p_2 \cdot \varepsilon'') \equiv (p_1 + p_2) \cdot \varepsilon''} \\
(\varepsilon_1'' \boxplus \varepsilon_2'') \boxplus \varepsilon_3'' \equiv \varepsilon_1'' \boxplus (\varepsilon_2'' \boxplus \varepsilon_3'') & \varepsilon_1'' \boxplus \varepsilon_2'' \equiv \varepsilon_2'' \boxplus \varepsilon_1'' & \varepsilon'' \boxplus \emptyset \equiv \varepsilon'' & \varepsilon'' \boxplus \varepsilon'' \equiv \varepsilon'' \\
\varepsilon[\mu x.\varepsilon/x] \equiv \mu x.\varepsilon & \gamma[\varepsilon/x] \equiv \varepsilon \Rightarrow \mu x.\gamma \equiv \varepsilon & &
\end{array}$$

a corresponding Kleene Theorem; and a sound and complete axiomatization for the corresponding notion of behavioural equivalence.

In order to show the effectiveness and the generality of our approach, we apply it to four types of systems: weighted automata; and simple Segala, stratified and Pnueli-Zuck systems. For simple Segala systems, we recover the language and axiomatization presented in [14]. For weighted automata and stratified systems, languages have been defined in [9] and [38] but, to the best of our knowledge, no axiomatization was ever given. Applying our method, we obtain the same languages and, more interestingly, we obtain novel axiomatizations. We also present a completely new framework to reason about Pnueli-Zuck systems. Table 1 summarizes our results.

2 Background

In this section, we present the basic definitions for polynomial functors and coalgebras. We recall, from [7], the language of expressions Exp_G associated with a functor G, the analogue of Kleene's theorem and a sound and complete axiomatization of Exp_G.

Let $\mathbf{Set}$ be the category of sets and functions. Sets are denoted by capital letters $X, Y, \ldots$ and functions by lower case $f, g, \ldots$ The collection of functions from a set X to a set Y is denoted by Y^X. We write $g \circ f$ for function composition, when defined. The product of two sets X, Y is written as $X \times Y$, with projection functions $X \xleftarrow{\pi_1} X \times Y \xrightarrow{\pi_2} Y$. The set 1 is a singleton set written as $1 = \{*\}$. We define $X + Y$ as the set $X \uplus Y \uplus \{\bot, \top\}$, where $\uplus$ is the disjoint union of sets, with injections $X \xrightarrow{\kappa_1} X \uplus Y \xleftarrow{\kappa_2} Y$. Note that the set $X + Y$ is different from the classical coproduct of X and Y, because of the two extra elements $\bot$ and $\top$. These extra elements are used to represent, respectively, underspecification and inconsistency in the specification of systems.

Polynomial functors. In our definition of polynomial functors we will use constant sets equipped with an information order. In particular, we will use join-semilattices. A (bounded) join-semilattice is a set B endowed with a binary operation $\vee_B$ and a constant $\bot_B \in B$. The operation $\vee_B$ is commutative, associative and idempotent. The element $\bot_B$ is neutral w.r.t. $\vee_B$. Every set S can be transformed into a join-semilattice by taking B to be the set of all finite subsets of S with union as join.

We are now ready to define the class of polynomial functors. They are functors $G : \mathbf{Set} \to \mathbf{Set}$, built inductively from the identity and constants, using $\times$, $+$ and $(-)^A$. Formally, the class PF of *polynomial functors* on $\mathbf{Set}$ is inductively defined by putting:

$$PF \ni G ::= Id \mid B \mid G_1 + G_2 \mid G_1 \times G_2 \mid G^A$$

with B a finite join-semilattice and A a finite set. For a set S, $Id(S) = S$, $B(S) = B$, $(G_1 \times G_2)(S) = G_1(S) \times G_2(S)$, $(G_1 + G_2)(S) = G_1(S) + G_2(S)$ and $G^A(S) = \{f \mid f : A \to G(S)\}$ and, for a function $f : S \to T$, $Gf : GS \to GT$ is defined as usual [31].

Typical examples of polynomial functors are $D = 2 \times Id^A$, $M = (B \times Id)^A$ and $St = A \times Id$. These functors represent, respectively, the type of *deterministic automata*, *Mealy machines*, and *infinite streams*.

Our definition of polynomial functors slightly differs from the one of [19, 30] in the use of a join-semilattice as constant functor and in the definition of $+$. This small variation plays an important technical role in giving a full coalgebraic treatment of the language of expressions which we shall introduce later. The intuition behind these extensions becomes clear if one recalls that the set of classical regular expressions carries a join-semilattice structure. Since ordinary polynomial functors can be naturally embedded into our polynomial functors above (because every set can be naturally embedded in the generated free join semilattice), one can use the results of Section 5 to obtain regular expressions (and axiomatization) for ordinary polynomial functors.

Next, we give the definition of the ingredient relation, which relates a polynomial functor G with its *ingredients*, *i.e.* the functors used in its inductive construction. We shall use this relation later for typing our expressions. Let $\lhd \subseteq PF \times PF$ be the least reflexive and transitive relation, written infix, such that

$$G_1 \lhd G_1 \times G_2, \quad G_2 \lhd G_1 \times G_2, \quad G_1 \lhd G_1 + G_2, \quad G_2 \lhd G_1 + G_2, \quad G \lhd G^A.$$

If $F \lhd G$, then F is said to be an *ingredient* of G. For example, 2, Id, $2 \times Id$, and $2 \times Id^A$ are the ingredients of the deterministic automata functor D.

Coalgebras. For an endofunctor G on $\mathbf{Set}$, a G-coalgebra is a pair (S, f) consisting of a set of *states* S together with a function $f : S \to GS$. The functor G, together with the function f, determines the *transition structure* of the G-coalgebra [31]. Examples of coalgebras include deterministic automata, Mealy machines and infinite streams, which are, respectively, coalgebras for the functors D, M and St given above.

A *G-homomorphism* from a G-coalgebra (S, f) to a G-coalgebra (T, g) is a function $h : S \to T$ preserving the transition structure, *i.e.*, such that $g \circ h = Gh \circ f$.

A G-coalgebra (Ω, ω) is said to be *final* if for any G-coalgebra (S, f) there exists a unique G-homomorphism $\mathbf{beh}_S : S \to \Omega$. For every polynomial functor G there exists a final G-coalgebra (Ω_G, ω_G) [31]. The notion of finality plays a key role in defining bisimilarity. For G-coalgebras (S, f) and (T, g) and $s \in S$, $t \in T$, we say that s and t are (G-)bisimilar, written $s \sim t$, if and only if $\mathbf{beh}_S(s) = \mathbf{beh}_T(t)$.

Given a G-coalgebra (S, f) and a subset V of S with inclusion map $i : V \to S$ we say that V is a subcoalgebra of S if there exists $g : V \to GV$ such that i is a homomorphism. Given $s \in S$, $\langle s \rangle \subseteq S$ denotes the subcoalgebra generated by s [31], *i.e.* the set consisting of states that are reachable from s. We will write $\mathit{Coalg}_{\mathit{lf}}(G)$ for the category of G-coalgebras that are *locally finite*: objects are G-coalgebras (S, f) such that for each state $s \in S$ the generated subcoalgebra $\langle s \rangle$ is finite; maps are the usual homomorphisms of coalgebras.

2.1 A Language of Expressions for Polynomial Coalgebras

In order to be able to formulate the generalization of our previous work [7], we first have to recall the main definitions and results concerning the language of expressions associated to a polynomial functor G. Note that in [7] we actually treated Kripke polynomial functors, as mentioned also in the present introduction. In order to give a more uniform and concise presentation, we omit in this section the case of the finite powerset $\mathcal{P}_\omega$ (thus, we only present polynomial functors), which can be recovered as a special instance of the monoidal valuation functor (Section 3). We start by introducing an untyped language of expressions and then we single out the well-typed ones via an appropriate typing system, thereby associating expressions to polynomial functors. Then, we present the analogue of Kleene's theorem.

Let A be a finite set, B a finite join-semilattice and X a set of fixpoint variables. The set of all *expressions* is given by the following grammar (where $a \in A$, $b \in B$):

$$\varepsilon ::= \emptyset \mid \varepsilon \oplus \varepsilon \mid x \mid \mu x.\varepsilon \mid b \mid l(\varepsilon) \mid r(\varepsilon) \mid l[\varepsilon] \mid r[\varepsilon] \mid a(\varepsilon)$$

An expression is *closed* if it has no free occurrences of fixpoint variables x. We denote the set of closed expressions by Exp.

Intuitively, expressions denote elements of final coalgebras. The expressions $\emptyset$, $\varepsilon \oplus \varepsilon$ and $\mu x.\,\varepsilon$ will play a role similar to, respectively, the empty language, the union of languages and the Kleene star in classical regular expressions for deterministic automata. The expressions $l(\varepsilon)$, $r(\varepsilon)$, $l[\varepsilon]$, $r[\varepsilon]$ and $a(\varepsilon)$ denote the left and right hand-side of products and sums and function application, respectively.

Next, we present a typing assignment system that will allow us to associate with each functor G the expressions ε that are valid specifications of G-coalgebras. The typing proceeds following the structure of the expressions and the ingredients of the functors. We type expressions ε using the ingredient relation, for $a \in A$, $b \in B$ and $x \in X$, as follows:

$$\frac{}{\vdash \emptyset : F \lhd G} \qquad \frac{}{\vdash b : B \lhd G} \qquad \frac{}{\vdash x : Id \lhd G} \qquad \frac{\vdash \varepsilon : G \lhd G}{\vdash \mu x.\varepsilon : G \lhd G}$$

$$\frac{\vdash \varepsilon_1 : F \lhd G \quad \vdash \varepsilon_2 : F \lhd G}{\vdash \varepsilon_1 \oplus \varepsilon_2 : F \lhd G} \qquad \frac{\vdash \varepsilon : G \lhd G}{\vdash \varepsilon : Id \lhd G} \qquad \frac{\vdash \varepsilon : F \lhd G}{\vdash a(\varepsilon) : F^A \lhd G}$$

$$\frac{\vdash \varepsilon : F_1 \lhd G}{\vdash l(\varepsilon) : F_1 \times F_2 \lhd G} \quad \frac{\vdash \varepsilon : F_2 \lhd G}{\vdash r(\varepsilon) : F_1 \times F_2 \lhd G} \quad \frac{\vdash \varepsilon : F_1 \lhd G}{\vdash l[\varepsilon] : F_1 + F_2 \lhd G} \quad \frac{\vdash \varepsilon : F_2 \lhd G}{\vdash r[\varepsilon] : F_1 + F_2 \lhd G}$$

Most of the rules are self-explanatory. The rule involving $Id \lhd G$ reflects the isomorphism of the final coalgebra: $\Omega_G \cong G(\Omega_G)$. It is interesting to note that the rule for the variable x guarantees that occurrences of variables in a fixpoint expression are *guarded*: they occur under the scope of expressions $l(\varepsilon)$, $r(\varepsilon)$, $l[\varepsilon]$, $r[\varepsilon]$ and $a(\varepsilon)$. For further details we refer to [7].

The set of G-expressions of well-typed expressions associated with a polynomial functor G is defined by $Exp_G = Exp_{G \lhd G}$, where, for F an ingredient of G:

$$Exp_{F \lhd G} = \{\varepsilon \in Exp \mid \vdash \varepsilon : F \lhd G\} .$$

To illustrate this definition we instantiate it for the functor $D = 2 \times Id^A$.

Example 1 (Deterministic expressions). Let A be a finite set and let X be a set of fixpoint variables. The set Exp_D of well-typed D-expressions is given by the BNF:

$$\varepsilon ::= \emptyset \mid x \mid l(0) \mid l(1) \mid r(a(\varepsilon)) \mid \varepsilon \oplus \varepsilon \mid \mu x.\varepsilon$$

where $a \in A$, $x \in X$, ε is closed and occurrences of fixpoint variables are within the scope of an input action, as can be easily checked by structural induction on the length of the type derivations.

Our derived syntax for this functor differs from classical regular expressions in the use of action prefixing and fixpoint instead of sequential composition and star, respectively. However, as we will soon see (Theorem 1), the expressions in our syntax correspond to deterministic automata and, in that sense, they are equivalent to classical regular expressions.

The language of expressions induces an algebraic description of systems. In [7], we showed that such language is a coalgebra. More precisely, we defined a function $\lambda_{F \lhd G} : Exp_{F \lhd G} \to F(Exp_G)$ and then set $\lambda_G = \lambda_{G \lhd G}$, providing Exp_G with a coalgebraic structure. The function $\lambda_{F \lhd G}$ is defined by double induction on the maximum number of nested unguarded occurrences of μ-expressions in ε and on the length of the proofs for typing expressions. For every ingredient F of a polynomial functor G and $\varepsilon \in Exp_{F \lhd G}$,

Table 2. The function $\mathit{Plus}_{F \lhd G} : F(\mathit{Exp}_G) \times F(\mathit{Exp}_G) \to F(\mathit{Exp}_G)$ and the constant $\mathit{Empty}_{F \lhd G} \in F(\mathit{Exp}_G)$

$$\begin{array}{ll}
\mathit{Empty}_{Id \lhd G} & = \emptyset \\
\mathit{Empty}_{B \lhd G} & = \bot_B \\
\mathit{Empty}_{F_1+F_2 \lhd G} & = \bot \\
\mathit{Empty}_{F_1\times F_2 \lhd G} & = \langle \mathit{Empty}_{F_1 \lhd G}, \mathit{Empty}_{F_2 \lhd G} \rangle \\
\mathit{Empty}_{F^A \lhd G} & = \lambda a.\mathit{Empty}_{F \lhd G} \\
 & \\
\mathit{Plus}_{Id \lhd G}(\varepsilon_1, \varepsilon_2) & = \varepsilon_1 \oplus \varepsilon_2 \\
\mathit{Plus}_{B \lhd G}(b_1, b_2) & = b_1 \vee_B b_2 \\
\mathit{Plus}_{F_1+F_2 \lhd G}(x, \top) & = \mathit{Plus}_{F_1+F_2 \lhd G}(\top, x) = \top \\
\mathit{Plus}_{F_1+F_2 \lhd G}(x, \bot) & = \mathit{Plus}_{F_1+F_2 \lhd G}(\bot, x) = x \\
\mathit{Plus}_{F_1+F_2 \lhd G}(\kappa_i(\varepsilon_1), \kappa_i(\varepsilon_2)) & = \kappa_i(\mathit{Plus}_{F_i \lhd G}(\varepsilon_1, \varepsilon_2)), \quad i \in \{1,2\} \\
\mathit{Plus}_{F_1+F_2 \lhd G}(\kappa_i(\varepsilon_1), \kappa_j(\varepsilon_2)) & = \top \text{ for } i,j \in \{1,2\} \ \mathit{and} \ i \neq j \\
\mathit{Plus}_{F_1\times F_2 \lhd G}(\langle \varepsilon_1, \varepsilon_2\rangle, \langle \varepsilon_3, \varepsilon_4\rangle) & = \langle \mathit{Plus}_{F_1 \lhd G}(\varepsilon_1, \varepsilon_3), \mathit{Plus}_{F_2 \lhd G}(\varepsilon_2, \varepsilon_4)\rangle \\
\mathit{Plus}_{F^A \lhd G}(f, g) & = \lambda a.\ \mathit{Plus}_{F \lhd G}(f(a), g(a))
\end{array}$$

the mapping $\lambda_{F \lhd G}(\varepsilon)$ is given by :

$$\begin{array}{llll}
\lambda_{F \lhd G}(\emptyset) & = \mathit{Empty}_{F \lhd G} & \lambda_{F_1\times F_2 \lhd G}(l(\varepsilon)) & = \langle \lambda_{F_1 \lhd G}(\varepsilon), \mathit{Empty}_{F_2 \lhd G}\rangle \\
\lambda_{F \lhd G}(\varepsilon_1 \oplus \varepsilon_2) & & \lambda_{F_1\times F_2 \lhd G}(r(\varepsilon)) & = \langle \mathit{Empty}_{F_1 \lhd G}, \lambda_{F_2 \lhd G}(\varepsilon)\rangle \\
\quad = \mathit{Plus}_{F \lhd G}(\lambda_{F \lhd G}(\varepsilon_1), \lambda_{F \lhd G}(\varepsilon_2)) & & \lambda_{F_1+F_2 \lhd G}(l[\varepsilon]) & = \kappa_1(\lambda_{F_1 \lhd G}(\varepsilon)) \\
\lambda_{G \lhd G}(\mu x.\varepsilon) & = \lambda_{G \lhd G}(\varepsilon[\mu x.\varepsilon/x]) & \lambda_{F_1+F_2 \lhd G}(r[\varepsilon]) & = \kappa_2(\lambda_{F_2 \lhd G}(\varepsilon)) \\
\lambda_{Id \lhd G}(\varepsilon) & = \varepsilon \quad \text{for } G \neq Id & \lambda_{F^A \lhd G}(a(\varepsilon)) & = \lambda a'.\begin{cases} \lambda_{F \lhd G}(\varepsilon) & a = a' \\ \mathit{Empty}_{F \lhd G} & \mathit{otherwise} \end{cases} \\
\lambda_{B \lhd G}(b) & = b & &
\end{array}$$

Here, $\varepsilon[\mu x.\varepsilon/x]$ denotes syntactic substitution, replacing every free occurrence of x in ε by $\mu x.\varepsilon$. The auxiliary constructs *Empty* and *Plus* are defined in Table 2. Note that we use λ in the right hand side of the equation for $\lambda_{F^A \lhd G}(a(\varepsilon))$ to denote lambda abstraction. This overlap of symbols is safe since when we use it in $\lambda_{F \lhd G}$ it is always accompanied by the type subscript. It is interesting to remark that λ_G is the generalization of the well-known notion of Brzozowski derivative [8] for regular expressions and, moreover, it provides an operational semantics for expressions.

We now present the generalization of Kleene's theorem.

Theorem 1 ([7, Theorem 4]). *Let G be a polynomial functor.*

1. *For every locally finite G-coalgebra (S, g) and for any $s \in S$ there exists an expression $\varepsilon_s \in \mathit{Exp}_G$ such that $\varepsilon_s \sim s$.*
2. *For every $\varepsilon \in \mathit{Exp}_G$, we can construct a coalgebra (S, g) such that S is finite and there exists $s \in S$ with $\varepsilon \sim s$.*

Note that $\varepsilon_s \sim s$ means that the expression ε_s and the (system with initial) state s have the same behaviour. For instance, for DFA's, this would mean that they denote and accept the same regular language. Similarly for ε and s in item 2..

In [7], we presented a sound and complete axiomatization wrt bisimilarity for Exp_G. We will not recall it here because this axiomatization can be recovered as an instance of the one presented in Section 4.

3 Monoidal Valuation Functor

In the previous section we introduced polynomial functors and a language of expressions for specifying coalgebras. Coalgebras for polynomial functors cover many interesting types of systems, such as deterministic and Mealy automata, but not quantitative systems. For this reason, we recall the definition of the *monoidal valuation functor* [17], which will allow us to define coalgebras representing quantitative systems. In the next section, we will provide expressions and an axiomatization for these.

A *monoid* $\mathbb{M}$ is an algebraic structure consisting of a set with an associative binary operation $+$ and a neutral element 0 for that operation. A *commutative monoid* is a monoid where $+$ is also commutative. Examples of commutative monoids include $\mathbf{2}$, the two-element $\{0, 1\}$ boolean algebra with logical "or", and the set $\mathbb{R}$ of real numbers with addition.

A property that will play a crucial role in the rest of the paper is *idempotency*: a monoid is idempotent, if the associated binary operation $+$ is idempotent. For example, the monoid $\mathbf{2}$ is idempotent, while $\mathbb{R}$ is not. Notice that an idempotent commutative monoid is a join-semilattice.

Given a function φ from a set S to a monoid $\mathbb{M}$, we define *support of* φ as the set $\{s \in S \mid \varphi(s) \neq 0\}$.

Definition 1 (Monoidal valuation Functor). *Let* $\mathbb{M}$ *be a commutative monoid. The monoidal valuation functor* $\mathbb{M}^{-}_{\omega}$:$\mathbf{Set} \to \mathbf{Set}$ *is defined as follows. For each set* S, $\mathbb{M}^{S}_{\omega}$ *is the set of functions from* S *to* $\mathbb{M}$ *with finite support. For each function* $h : S \to T$, $\mathbb{M}^{h}_{\omega}$:$\mathbb{M}^{S}_{\omega} \to \mathbb{M}^{T}_{\omega}$ *is the function mapping each* $\varphi \in \mathbb{M}^{S}_{\omega}$ *into* $\varphi^{h} \in \mathbb{M}^{T}_{\omega}$ *defined, for every* $t \in T$, *as*

$$\varphi^{h}(t) = \sum_{s' \in h^{-1}(t)} \varphi(s')$$

Proposition 1. *The functor* $\mathbb{M}^{-}_{\omega}$ *has a final coalgebra.*

Note that the (finite) powerset functor $\mathcal{P}_{\omega}(-)$ coincides with $\mathbf{2}^{-}_{\omega}$. This is often used to represent non-deterministic systems. For example, (image-finite) LTS's (with labels over A) are coalgebras for the functor $\mathcal{P}_{\omega}(-)^{A}$. In the following, to simplify the notation we will always write $\mathbb{M}^{-}$ instead of $\mathbb{M}^{-}_{\omega}$.

By combining the monoidal valuation functor with the polynomial functors, we can model quantitative systems as coalgebras. As an example, we mention weighted automata.

Weighted Automata. A *semiring* $\mathbb{S}$ is a tuple $\langle \mathbb{S}, +, \times, 0, 1\rangle$ where $\langle \mathbb{S}, +, 0\rangle$ is a commutative monoid and $\langle \mathbb{S}, \times, 1\rangle$ is a monoid satisfying certain distributive laws.

Weighted automata [15, 33] are transition systems labelled over a set A and with weights in a semiring $\mathbb{S}$. Moreover, each state is equipped with an output value[1] in $\mathbb{S}$. From a coalgebraic perspective weighted automata are coalgebras for the functor $\mathbb{S} \times (\mathbb{S}^{Id})^{A}$, where we use $\mathbb{S}$ to denote, the commutative monoid of the semiring $\mathbb{S}$. More concretely, a coalgebra for $\mathbb{S} \times (\mathbb{S}^{Id})^{A}$ is a pair $(Q, \langle o, t\rangle)$, where Q is a set of states,

[1] In the original formulation also an input value is considered. To simplify the presentation and following [10] we omit it.

$o : Q \to \mathbb{S}$ is the function that associates an output weight to each state $q \in Q$ and $t : Q \to (\mathbb{S}^Q)^A$ is the transition relation that associates a weight to each transition: $q \xrightarrow{a,s} q' \iff t(q)(a)(q') = s$.

Bisimilarity for weighted automata has been studied in [9] and it coincides with the coalgebraic notion of bisimilarity. (For a proof, see [6].)

Proposition 2. *Bisimilarity for* $\mathbb{S} \times (\mathbb{S}^{Id})^A$ *coincides with the weighted automata bisimilarity defined in [9].*

4 A Non-idempotent Algebra for Quantitative Regular Behaviours

In this section, we will extend the framework presented in Section 2 in order to deal with quantitative systems, as described in the previous section. We will start by defining an appropriate class of functors H, proceed with presenting the language Exp_H of expressions associated with H together with a Kleene like theorem and finally we introduce a sound and complete axiomatization of Exp_H.

Formally, the set QF of *quantitative functors* on **Set** is defined inductively by putting:

$$QF \ni H ::= G \mid \mathbb{M}^H \mid (\mathbb{M}^H)^A \mid \mathbb{M}_1^{H_1} \times \mathbb{M}_2^{H_2} \mid \mathbb{M}_1^{H_1} + \mathbb{M}_2^{H_2}$$

where G is a polynomial functor, $\mathbb{M}$ is a commutative monoid and A is a finite set. Note that we do not allow *mixed* functors, such as $G + \mathbb{M}^H$ or $G \times \mathbb{M}^H$. The reason for this restriction will become clear later in this section when we discuss the proof of Kleene's theorem. In Section 5, we will show how to deal with such mixed functors.

Every definition we presented in Section 2 needs now to be extended to quantitative functors. We start by observing that taking the current definitions and replacing the subscript $F \lhd G$ with $F \lhd H$ does most of the work. In fact, having that, we just need to extend all the definitions for the case $\mathbb{M}^F \lhd H$.

We start by introducing a new expression $m \cdot \varepsilon$, with $m \in \mathbb{M}$, extending the set of untyped expressions, which is now given by:

$$\varepsilon ::= \emptyset \mid \varepsilon \oplus \varepsilon \mid x \mid \mu x.\varepsilon \mid b \mid l(\varepsilon) \mid r(\varepsilon) \mid l[\varepsilon] \mid r[\varepsilon] \mid a(\varepsilon) \mid m \cdot \varepsilon$$

The intuition behind the new expression is that there is a transition between the current state and the state specified by ε with weight m.

The ingredient relation is extended with the rule $H \lhd \mathbb{M}^H$, the type system and $\lambda_{\mathbb{M}^F \lhd H}$ with the following rules:

$$\frac{\varepsilon : F \lhd H}{m \cdot \varepsilon : \mathbb{M}^F \lhd H} \qquad \begin{array}{l} Empty_{\mathbb{M}^F \lhd H} = \lambda\varepsilon'.0 \\ Plus_{\mathbb{M}^F \lhd H}(f, g) = \lambda\varepsilon'.f(\varepsilon') + g(\varepsilon') \\ \lambda_{\mathbb{M}^F \lhd H}(m \cdot \varepsilon) = \lambda\varepsilon'.\begin{cases} m \text{ if } \lambda_{F \lhd H}(\varepsilon) = \varepsilon' \\ 0 \quad otherwise \end{cases} \end{array}$$

where 0 and $+$ are the neutral element and the binary operation of $\mathbb{M}$. Recall that the function $\lambda_H = \lambda_{H \lhd H}$ provides an operational semantics for the expressions. We will soon illustrate this for the case of expressions for weighted automata (Example 2).

Finally, we introduce an equational system for expressions of type $F \lhd H$. We will use the symbol $\equiv \subseteq Exp_{F \lhd H} \times Exp_{F \lhd H}$, omitting the subscript $F \lhd H$, for the least relation satisfying the following:

$(Idempotency)$ $\varepsilon \oplus \varepsilon \equiv \varepsilon, \quad \varepsilon \in Exp_{F \lhd G}$
$(Commutativity)$ $\varepsilon_1 \oplus \varepsilon_2 \equiv \varepsilon_2 \oplus \varepsilon_1$ (FP) $\gamma[\mu x.\gamma/x] \equiv \mu x.\gamma$
$(Associativity)$ $\varepsilon_1 \oplus (\varepsilon_2 \oplus \varepsilon_3) \equiv (\varepsilon_1 \oplus \varepsilon_2) \oplus \varepsilon_3$ $(Unique)$ $\gamma[\varepsilon/x] \equiv \varepsilon \Rightarrow \mu x.\gamma \equiv \varepsilon$
$(Empty)$ $\emptyset \oplus \varepsilon \equiv \varepsilon$

$(B - \emptyset)$	$\emptyset \equiv \bot_B$	$(B - \oplus)$	$b_1 \oplus b_2 \equiv b_1 \vee_B b_2$
$(\times - \emptyset - L)$	$l(\emptyset) \equiv \emptyset$	$(\times - \oplus - L)$	$l(\varepsilon_1 \oplus \varepsilon_2) \equiv l(\varepsilon_1) \oplus l(\varepsilon_2)$
$(\times - \emptyset - R)$	$r(\emptyset) \equiv \emptyset$	$(\times - \oplus - R)$	$r(\varepsilon_1 \oplus \varepsilon_2) \equiv r(\varepsilon_1) \oplus r(\varepsilon_2)$
$(-^A - \emptyset)$	$a(\emptyset) \equiv \emptyset$	$(-^A - \oplus)$	$a(\varepsilon_1 \oplus \varepsilon_2) \equiv a(\varepsilon_1) \oplus a(\varepsilon_2)$
$(\mathbb{M}^- - \emptyset)$	$(0 \cdot \varepsilon) \equiv \emptyset$	$(\mathbb{M}^- - \oplus)$	$(m \cdot \varepsilon) \oplus (m' \cdot \varepsilon) \equiv (m + m') \cdot \varepsilon$
$(+ - \oplus - L)$	$l[\varepsilon_1 \oplus \varepsilon_2] \equiv l[\varepsilon_1] \oplus l[\varepsilon_2]$	$(+ - \oplus - R)$	$r[\varepsilon_1 \oplus \varepsilon_2] \equiv r[\varepsilon_1] \oplus r[\varepsilon_2]$
$(\alpha - equiv)$	$\mu x.\gamma \equiv \mu y.\gamma[y/x]$ if y not free in γ	$(+ - \oplus - \top)$	$l[\varepsilon_1] \oplus r[\varepsilon_2] \equiv l[\emptyset] \oplus r[\emptyset]$

$(Cong)$ If $\varepsilon \equiv \varepsilon'$ then $\varepsilon \oplus \varepsilon_1 \equiv \varepsilon' \oplus \varepsilon_1$, $\mu x.\varepsilon \equiv \mu x.\varepsilon'$, $l(\varepsilon) \equiv l(\varepsilon')$, $r(\varepsilon) \equiv r(\varepsilon')$, $l[\varepsilon] \equiv l[\varepsilon']$, $r[\varepsilon] \equiv r[\varepsilon']$, $a(\varepsilon) \equiv a(\varepsilon')$, and $m \cdot \varepsilon \equiv m \cdot \varepsilon'$.

We shall write $Exp/_{\equiv}$ for the set of expressions modulo $\equiv$.

Note that $(Idempotency)$ only holds for $\varepsilon \in Exp_{F \lhd G}$. The reason why it cannot hold for the remaining functors comes from the fact that a monoid is, in general, not idempotent. Thus, $(Idempotency)$ would conflict with the axiom $(\mathbb{M}^- - \oplus)$, which allows us to derive, for instance, $(2 \cdot \emptyset) \oplus (2 \cdot \emptyset) \equiv 4 \cdot \emptyset$. In the case of an idempotent commutative monoid $\mathbb{M}$, $(Idempotency)$ follows from the axiom $(\mathbb{M}^- - \oplus)$.

Lemma 1. *Let $\mathbb{M}$ be an idempotent commutative monoid. For every expression $\varepsilon \in Exp_{\mathbb{M}^F \lhd H}$, one has $\varepsilon \oplus \varepsilon \equiv \varepsilon$.*

Example 2 (Expressions for weighted automata). The syntax automatically derived from our typing system for the functor $W = \mathbb{S} \times (\mathbb{S}^{Id})^A$ is the following.

$$\begin{aligned} \varepsilon &::= \ \emptyset \mid \varepsilon \oplus \varepsilon \mid x \mid \mu x.\varepsilon \mid l(s) \mid r(\varepsilon') \\ \varepsilon' &::= \ \emptyset \mid \varepsilon' \oplus \varepsilon' \mid a(\varepsilon'') \\ \varepsilon'' &::= \ \emptyset \mid \varepsilon'' \oplus \varepsilon'' \mid s \cdot \varepsilon \end{aligned}$$

where $s \in \mathbb{S}$, $a \in A$ and all the occurrences of x are guarded. The semantics of these expressions is given by the function $\lambda_{W \lhd W}$ (hereafter denoted by λ_W) which is an instance of the general $\lambda_{F \lhd H}$ defined above. It is given by:

$$\begin{aligned} \lambda_W(\emptyset) &= \langle 0, \lambda a.\lambda \varepsilon.0 \rangle \\ \lambda_W(\varepsilon_1 \oplus \varepsilon_2) &= \langle s_1 + s_2, \lambda a.\lambda \varepsilon.(f(a)(\varepsilon) + g(a)(\varepsilon)) \rangle \\ &\quad \text{where } \langle s_1, f \rangle = \lambda_W(\varepsilon_1) \text{ and } \langle s_2, g \rangle = \lambda_W(\varepsilon_2) \\ \lambda_W(\mu x.\varepsilon) &= \lambda_W(\varepsilon[\mu x.\varepsilon/x]) \\ \lambda_W(l(s)) &= \langle s, \lambda a.\lambda \varepsilon.0 \rangle \\ \lambda_W(r(\varepsilon')) &= \langle 0, \lambda_{(\mathbb{S}^{Id})^A \lhd W}(\varepsilon') \rangle \end{aligned}$$

$$\begin{aligned} \lambda_{(\mathbb{S}^{Id})^A \lhd W}(\emptyset) &= \lambda a.\lambda \varepsilon.0 \\ \lambda_{(\mathbb{S}^{Id})^A \lhd W}(\varepsilon_1 \oplus \varepsilon_2) &= \lambda a.\lambda \varepsilon.(f_1(a)(\varepsilon) + f_2(s)(\varepsilon)) \\ &\quad \text{where } f_i = \lambda_{(\mathbb{S}^{Id})^A \lhd W}(\varepsilon_i), i \in \{1,2\} \\ \lambda_{(\mathbb{S}^{Id})^A \lhd W}(a(\varepsilon'')) &= \lambda a'. \begin{cases} \lambda_{\mathbb{S}^{Id} \lhd W}(\varepsilon'') & a = a' \\ \lambda \varepsilon.0 & oth. \end{cases} \end{aligned}$$

$$\begin{aligned} \lambda_{(\mathbb{S}^{Id}) \lhd W}(\emptyset) &= \lambda \varepsilon.0 \\ \lambda_{(\mathbb{S}^{Id}) \lhd W}(\varepsilon_1 \oplus \varepsilon_2) &= \lambda \varepsilon.(f_1(\varepsilon) + f_2(\varepsilon)) \\ &\quad \text{where } f_i = \lambda_{(\mathbb{S}^{Id}) \lhd W}(\varepsilon_i), i \in \{1,2\} \\ \lambda_{(\mathbb{S}^{Id}) \lhd W}(s \cdot \varepsilon) &= \lambda \varepsilon'. \begin{cases} s & \varepsilon = \varepsilon' \\ 0 & oth. \end{cases} \end{aligned}$$

The function λ_W assigns to each expression ε a pair $\langle s, t\rangle$, consisting of an output weight $s \in \mathbb{S}$ and a function $t : A \to \mathbb{S}^{Exp_W}$. For a concrete example, let $\mathbb{S} = \mathbb{R}$ and consider $\varepsilon = \mu x.r(a(2 \cdot x \oplus 3 \cdot \emptyset)) \oplus l(1) \oplus l(2)$. The semantics of this expression, obtained by λ_W is described by the weighted automaton below.

$$\begin{tikzcd}
\varepsilon \arrow[loop above, "{a,2}"] \arrow[r, "{a,3}"] \arrow[d, Rightarrow] & \emptyset \arrow[d, Rightarrow] \\
3 & 0
\end{tikzcd}$$

In Table 1 a more concise syntax for expressions for weighted automata is presented. To derive that syntax from the one automatically generated, we first write ε' as

$$\varepsilon' ::= \ \emptyset \mid \varepsilon' \oplus \varepsilon' \mid a(s \cdot \varepsilon)$$

using the axioms $a(\emptyset) \equiv \emptyset$ and $a(\varepsilon_1'' \oplus \varepsilon_2'') \equiv a(\varepsilon_1'') \oplus a(\varepsilon_2'')$. Similarly, using $r(\emptyset) = \emptyset$ and $r(\varepsilon_1' \oplus \varepsilon_2') \equiv r(\varepsilon_1') \oplus r(\varepsilon_2')$, we can write ε as follows.

$$\varepsilon ::= \ \emptyset \mid \varepsilon \oplus \varepsilon \mid x \mid \mu x.\varepsilon \mid l(s) \mid r(a(s \cdot \varepsilon))$$

In Table 1, we abbreviate $l(s)$ to s and $r(a(s{\cdot}\varepsilon))$ to $a(s{\cdot}\varepsilon)$, without any risk of confusion. Note that the axioms presented in Table 1 also reflect the changes in the syntax of the expressions.

We are now ready to formulate the analogue of Kleene's theorem for quantitative systems.

Theorem 2 (Kleene's theorem for quantitative functors). *Let H be a quantitative functor.*

1. *For every locally finite H-coalgebra (S, h) and for every $s \in S$ there exists an expression $\varepsilon_s \in Exp_H$ such that $s \sim \varepsilon_s$.*
2. *For every $\varepsilon \in Exp_H$, there exists a finite H-coalgebra (S, h) with $s \in S$ such that $s \sim \varepsilon$.*

The proof of the theorem can be found in [6], but let us explain what are the technical difficulties that arise when compared with Theorem 1, where only polynomial functors are considered.

In the proof of item 2. in Theorem 1, we start by constructing the subcoalgebra generated by ε, using the fact that the set Exp_G has a coalgebra structure given by λ_G. Then, we observe that such subcoalgebra might not be finite and, following a similar result for classical regular expressions, we show that finiteness can be obtained by taking the subcoalgebra generated modulo $(Associativity)$, $(Commutativity)$ and $(Idempotency)$ (ACI).

Consider for instance the expression $\varepsilon = \mu x.r(a(x \oplus x))$ of type $D = 2 \times Id^A$. The subcoalgebras generated with and without applying ACI are the following:

$$\begin{tikzcd}
\varepsilon \arrow[loop above, "a"] & & \varepsilon \arrow[r, "a"] & \varepsilon \oplus \varepsilon \arrow[r, "a"] & (\varepsilon \oplus \varepsilon) \oplus (\varepsilon \oplus \varepsilon) \arrow[r, "a"] & \cdots
\end{tikzcd}$$

We cannot apply ACI in the quantitative setting, since the idempotency axiom does not hold anymore. However, surprisingly enough, in the case of the functor $\mathbb{M}^H$, we are able to prove finiteness of the subcoalgebra $\langle \varepsilon \rangle$ just by using $(Commutativity)$ and $(Associativity)$. The key observation is that the monoid structure will be able to avoid the infinite scenario described above. In fact, for the functor M^H one can prove that, if $\varepsilon \oplus \varepsilon$ is one of the successors of ε then the successors of $\varepsilon \oplus \varepsilon$ will all be contained in the set of direct successors of ε, which we know is finite . What happens is concisely captured by the following example. Take the expression $\varepsilon = \mu x.2 \cdot (x \oplus x)$ for the functor $\mathbb{R}^{Id}$. Then, the subcoalgebra generated by ε is depicted in the following picture:

$$\begin{tikzcd}
\varepsilon \arrow[r, "2"] & \varepsilon \oplus \varepsilon \arrow[loop above, "4"]
\end{tikzcd}$$

In this manner, we are able to deal with the base cases G (polynomial functor) and $\mathbb{M}^H$ of the inductive definition of the set of quantitative functors. Moreover, the functors $\mathbb{M}^H \times \mathbb{M}^H$ and $\mathbb{M}^H + \mathbb{M}^H$ inherit the above property from $\mathbb{M}^H$ and do not pose any problem in the proof of Kleene's theorem. The syntactic restriction that excludes mixed functors is needed because of the following problem. Take as an example the functor $\mathbb{M}^{Id} \times Id^A$. A well-typed expression for this functor would be $\varepsilon = \mu x.r(a(x \oplus x \oplus l(2 \cdot x) \oplus l(2 \cdot x)))$. It is clear that we cannot apply idempotency in the subexpression $x \oplus x \oplus l(2 \cdot x) \oplus l(2 \cdot x)$ and hence the subcoalgebra generated by ε will be infinite:

$$\begin{tikzcd}
\varepsilon \arrow[r, "a"] & \varepsilon' \arrow[r, "a"] \arrow[l, bend left, "4"] & \varepsilon' \oplus \varepsilon' \arrow[r, "a"] \arrow[ll, bend left, "8"] & (\varepsilon' \oplus \varepsilon') \oplus (\varepsilon' \oplus \varepsilon') \arrow[r, "a"] \arrow[lll, bend left, "16"] & \cdots
\end{tikzcd}$$

with $\varepsilon' = \varepsilon \oplus \varepsilon \oplus l(2 \cdot \varepsilon) \oplus l(2 \cdot \varepsilon)$. We will show in the next section how to overcome this problem.

Let us summarize what we have achieved so far: we have presented a framework that allows, for each quantitative functor $H \in QF$, the derivation of a language Exp_H. Moreover, Theorem 2 guarantees that for each expression $\varepsilon \in Exp_H$, there exists a finite H-coalgebra (S, h) that contains a state $s \in S$ bisimilar to ε and, conversely, for each locally finite H-coalgebra (S, h) and for every state in s there is an expression $\varepsilon_s \in Exp_H$ bisimilar to s. The proof of Theorem 2, which can be found in [6], shows how to compute the H-coalgebra (S, h) corresponding to an expression ε and vice-versa.

The axiomatization presented above is sound and complete:

Theorem 3 (Soundness and Completeness). *Let H be a quantitative functor and let $\varepsilon_1, \varepsilon_2 \in Exp_H$. Then, $\varepsilon_1 \sim \varepsilon_2 \iff \varepsilon_1 \equiv \varepsilon_2$.*

The proof of this theorem follows a similar strategy as in [7, 20] and can be found in [6].

5 Extending the Class of Functors

In the previous section, we introduced regular expressions for the class of quantitative functors. In this section, by employing standard results from the theory of coalgebras, we show how to use such regular expressions to describe the coalgebras of many more functors, including the mixed functors we mentioned in Section 4.

Given F and G two endofunctors on $\mathbf{Set}$, a *natural transformation* $\alpha{:}F \Rightarrow G$ is a family of functions $\alpha_S{:}F(S) \to G(S)$ (for all sets S), such that for all functions $h{:}T \to U$, $\alpha_U \circ F(h) = G(h) \circ \alpha_T$. If all the α_S are injective, then we say that α is injective.

Proposition 3. *An injective natural transformation $\alpha{:}F \Rightarrow G$ induces a functor $\alpha \circ (-) : \mathit{Coalg}_{\mathit{lf}}(F) \to \mathit{Coalg}_{\mathit{lf}}(G)$ that preserves and reflects bisimilarity.*

This result (proof can be found in [6]) allows us to extend both regular expressions and axiomatization to many functors. Indeed, consider a functor F that is not quantitative, but that has an injective natural transformation α into some quantitative functor H. A (locally finite) F-coalgebra can be translated into a (locally finite) H-coalgebra via the functor $\alpha \circ (-)$ and then it can be characterized by using expressions in Exp_H (as we will show soon, for the converse some care is needed). The axiomatization for Exp_H is still sound and complete for F-coalgebras, since the functor $\alpha \circ (-)$ preserves and reflects bisimilarity.

However, notice that Kleene's theorem does not hold anymore, because not all the expressions in Exp_H denote F-regular behaviours or, more precisely, not all expressions of Exp_H are equivalent to H-coalgebras that are in the image of $\alpha \circ (-)$. Thus, in order to retrieve Kleene's theorem, one has just to exclude such expressions. In many situations, this is feasible by simply imposing some syntactic constraints on Exp_H.

As an example, we recall the definition of the probability functor that, in the next section, will allow us to derive regular expressions for probabilistic systems.

Definition 2 (Probability functor). *A probability distribution over a set S is a function $d : S \to [0,1]$ such that $\sum_{s \in S} d(s) = 1$. The probability functor $\mathcal{D}_\omega{:}\mathbf{Set} \to \mathbf{Set}$ is defined as follows. For all sets S, $\mathcal{D}_\omega(S)$ is the set of probability distributions over S with finite support. For all functions $h : S \to T$, $\mathcal{D}_\omega(h)$ maps each $d \in \mathcal{D}_\omega(S)$ into d^h as defined in Definition 1.*

Now recall the functor $\mathbb{R}^{Id}$ from Section 3. Note that for any set S, $D_\omega(S) \subseteq \mathbb{R}^S$ since probability distributions are also functions from S to $\mathbb{R}$. Let ι be the family of inclusions $\iota_S{:}D_\omega(S) \to \mathbb{R}^S$. It is easy to see that ι is a natural transformation between D_ω and $\mathbb{R}^{Id}$ (the two functors are defined in the same way on arrows). Thus, in order to specify D_ω-coalgebras, we can use $\varepsilon \in \mathit{Exp}_{\mathbb{R}^{Id}}$ which are the closed and guarded expressions given by $\varepsilon ::= \emptyset \mid \varepsilon \oplus \varepsilon \mid x \mid \mu x.\varepsilon \mid r \cdot \varepsilon$, for $r \in \mathbb{R}$. However, this language allows us to specify $\mathbb{R}^{Id}$-behaviours that are not D_ω-behaviours, such as for example, $\mu x.2 \cdot x$ and $\mu x.0 \cdot x$. In order to obtain a language that specifies all and only the regular D_ω-behaviours, it is enough to restrict the syntax of $\mathit{Exp}_{\mathbb{R}^{Id}}$, as follows:

$$\varepsilon ::= x \mid \mu x.\varepsilon \mid \bigoplus_{i \in 1\ldots n} p_i \cdot \varepsilon_i \quad \text{for } p_i \in (0,1] \text{ such that } \sum_{i \in 1\ldots n} p_i = 1$$

where, with a slight abuse of notation, $\bigoplus_{i \in 1\ldots n} p_i \cdot \varepsilon_i$ denotes $p_1 \cdot \varepsilon_1 \oplus \cdots \oplus p_n \cdot \varepsilon_n$.

In the next section, we will use this kind of syntactic restrictions for defining regular expressions of probabilistic systems.

For another example, consider the functors Id and $\mathcal{P}_\omega(Id)$. Let τ be the family of functions $\tau_S : S \to \mathcal{P}_\omega(S)$ mapping each $s \in S$ in the singleton set $\{s\}$. It is easy to see that τ is an injective natural transformation. With the above observation, we can also get regular expressions for the functor $\mathbb{M}^{Id} \times Id^A$ that, as discussed in Section 4, does not belong to our class of quantitative functors. Indeed, by extending τ, we can construct an injective natural transformation $\mathbb{M}^{Id} \times Id^A \Rightarrow \mathbb{M}^{Id} \times \mathcal{P}_\omega(Id)^A$.

In the same way, we can construct an injective natural transformation from the functor $D_\omega(Id)+(A\times Id)+1$ (that is the type of stratified systems) into $\mathbb{R}^{Id}+(A\times\mathcal{P}_\omega(Id))+1$. Since the latter is a quantitative functor, we can use its expressions and axiomatization for stratified systems. But since not all its expressions define stratified behaviours, we again have to restrict the syntax.

The procedure of appropriately restricting the syntax usually requires some ingenuity. We shall see that in many concrete cases, as for instance D_ω above, it is fairly intuitive which restriction to choose.

6 Probabilistic Systems

Many different types of probabilistic systems have been defined in literature: reactive, generative, stratified, alternating, (simple) Segala, bundle and Pnueli-Zuck. Each type corresponds to a functor, and the systems of a certain type are coalgebras for the corresponding functor. A systematic study of all these systems as coalgebras was made in [5]. In particular, Fig.1 of [5] provides a full correspondence between types of systems and functors. By employing this correspondence, we can use our framework in order to derive regular expressions and axiomatizations for all these types of probabilistic systems.

In order to show the effectiveness of our approach, we have derived expressions and axioms for three different types of probabilistic systems: simple Segala, stratified and Pnueli-Zuck. Table 1 shows the expressions and the axiomatizations that we have obtained, after some simplification of the canonically derived syntax (which is often verbose and redundant).

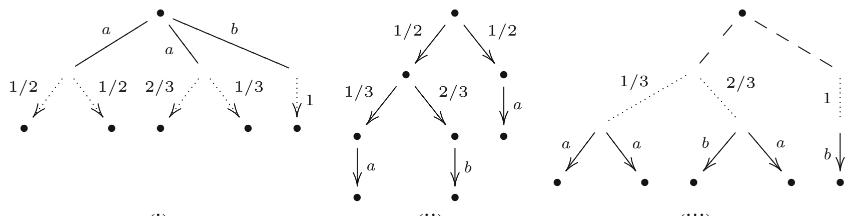


Fig. 1. (i) A simple Segala system, (ii) a stratified system and (iii) a Pnueli-Zuck system

Simple Segala systems. Simple Segala systems are coalgebras of type $\mathcal{P}_\omega(D_\omega(Id))^A$ (recall that $\mathcal{P}_\omega$ is the functor $\mathbf{2}^-$). These are like labelled transition systems, but each labelled transition leads to a probability distribution of states instead of a single state. An example is shown in Fig.1(i).

Table 1 shows expressions and axioms for simple Segala systems. In the following we show how to derive these. As described in Section 5, we can derive the expressions for $\mathcal{P}_\omega(\mathbb{R}^{Id})^A$ instead of $\mathcal{P}_\omega(D_\omega(Id))^A$, and then impose some syntactic constraints on $Exp_{\mathcal{P}_\omega(\mathbb{R}^{Id})^A}$ in order to characterize all and only the $\mathcal{P}_\omega(D_\omega(Id))^A$ behaviours. By simply applying our typing systems to $\mathcal{P}_\omega(\mathbb{R}^{Id})^A$, we derive the expressions:

$$\begin{aligned} \varepsilon &::= \emptyset \mid \varepsilon \oplus \varepsilon \mid x \mid \mu x.\varepsilon \mid a(\varepsilon') \\ \varepsilon' &::= \emptyset \mid \varepsilon' \oplus \varepsilon' \mid 1 \cdot \varepsilon'' \mid 0 \cdot \varepsilon'' \\ \varepsilon'' &::= \emptyset \mid \varepsilon'' \oplus \varepsilon'' \mid p \cdot \varepsilon \end{aligned}$$

where $a \in A$, $p \in \mathbb{R}$ and 0 and 1 are the elements of the boolean monoid $\mathbf{2}$.

Now, observe that the syntax for ε', due to the axiom $0 \cdot \varepsilon'' \equiv \emptyset$ can be reduced to

$$\varepsilon' ::= \emptyset \mid \varepsilon' \oplus \varepsilon' \mid 1 \cdot \varepsilon''$$

which, because of $a(\varepsilon'_1) \oplus a(\varepsilon'_2) \equiv a(\varepsilon'_1 \oplus \varepsilon'_2)$ and $a(\emptyset) \equiv \emptyset$ is equivalent to the simplified syntax:

$$\varepsilon ::= \emptyset \mid \varepsilon \oplus \varepsilon \mid x \mid \mu x.\varepsilon \mid a(\{\varepsilon''\})$$

Here, and in what follows, $\{\varepsilon''\}$ abbreviates $1 \cdot \varepsilon''$. Note that the axiomatization would have to include the axiom $a(\{\varepsilon''\}) \oplus a(\{\varepsilon''\}) \equiv a(\{\varepsilon''\})$, as a consequence of $(\mathbb{M}^- - \oplus)$ and $(-^A - \oplus)$. However, this axiom is subsumed by the $(Idempotency)$ axiom, which we add to the axiomatization, since it holds for expressions $\varepsilon \in Exp_{\mathcal{P}_\omega(\mathbb{R}^{Id})^A}$. This, combined with the restrictions to obtain $\mathbb{D}_\omega$ out of $\mathbb{R}^{Id}$, leads to the expressions and the axiomatization in Table 1 where, in order to avoid confusion, we use $\boxplus$ instead of $\oplus$, making a clear distinction between the idempotent and non-idempotent sums.

As an example, the expression $a(\{1/2 \cdot \emptyset \oplus 1/2 \cdot \emptyset\}) \boxplus a(\{1/3 \cdot \emptyset \oplus 2/3 \cdot \emptyset\}) \boxplus b(\{1 \cdot \emptyset\})$ describes the simple Segala system in Fig. 1(i).

Stratified systems. Stratified systems are coalgebras of the functor $D_\omega(Id) + (B \times Id) + 1$. Each state of these systems either performs unlabelled probabilistic transitions or one B-labelled transition or it terminates. To get the intuition for the syntax presented in Table 1, note that the stratified system in Fig. 1(ii) would be specified by the expression $1/2 \cdot (1/3 \cdot \langle a, \downarrow \rangle \oplus 2/3 \cdot \langle b, \downarrow \rangle) \oplus 1/2 \cdot \langle a, \downarrow \rangle$. Again, we added some syntactic sugar to our original regular expressions: $\downarrow$, denoting termination, corresponds to our expression $r[r[1]]$, while $\langle b, \varepsilon \rangle$ corresponds to $r[l[l(b) \oplus r(\{\varepsilon\})]]$. The derivation of the simplified syntax and axioms follows a similar strategy as in the previous example and thus is omitted here. As described in Section 5, we first derive expressions and axioms for $\mathbb{R}^{Id} + (B \times \mathcal{P}_\omega(Id)) + 1$ and then we restrict the syntax to characterize only $D_\omega(Id) + (B \times Id) + 1$-behaviours.

Pnueli-Zuck systems. These systems are coalgebras of the functor $\mathcal{P}_\omega D_\omega(\mathcal{P}_\omega(Id))^A$. Intuitively, the ingredient $\mathcal{P}_\omega(Id)^A$ denotes A-labelled transitions to other states. Then, $D_\omega(\mathcal{P}_\omega(Id))^A$ corresponds to a probability distribution of labelled transitions and then, each state of a $\mathcal{P}_\omega D_\omega(\mathcal{P}_\omega(Id))^A$-coalgebra performs a non deterministic choice amongst probability distributions of labelled transitions. The expression $\{1/3 \cdot a(\{\emptyset\}) \boxplus a(\{\emptyset\}) \oplus 2/3 \cdot (b(\{\emptyset\}) \boxplus a(\{\emptyset\}))\} \boxplus \{1 \cdot b(\{\emptyset\})\}$ specifies the Pnueli-Zuck system in Fig. 1 (iii). Notice that we use the same symbol $\boxplus$ for denoting two different kinds of non-deterministic choice. This is safe, since they satisfy the same axioms. Again, the derivation of the simplified syntax and axioms is omitted here.

7 Conclusions

We presented a general framework to canonically derive expressions and axioms for quantitative regular behaviours. To illustrate the effectiveness and generality of our approach we derived expressions and equations for weighted automata, simple Segala, stratified and Pnueli-Zuck systems.

We recovered the syntaxes proposed in [10, 14, 38] for the first three models and the axiomatization of [14]. For weighted automata and stratified systems we derived new axiomatizations and for Pnueli-Zuck systems both a novel language of expressions and axioms. It should be remarked that [10, 14, 38] considered process calculi that are also equipped with the parallel composition operator and thus they slightly differ from our languages, which are more in the spirit of Kleene and Milner's expressions. Also [4, 13, 37] study expressions without parallel composition for probabilistic systems. These provide syntax and axioms for generative systems, Segala systems and alternating systems, respectively. For Segala systems our approach will derive the same language of [13], while the expressions in [37] differ from the ones resulting from our approach, since they use a probabilistic choice operator $+_p$. For alternating systems, our approach would bring some new insights, since [4] considers only expressions without recursion.

Acknowledgments. The authors are grateful to Catuscia Palamidessi for interesting discussions and pointers to the literature.

References

1. Aceto, L., Ésik, Z., Ingólfsdóttir, A.: Equational axioms for probabilistic bisimilarity. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 239–253. Springer, Heidelberg (2002)
2. Baeten, J., Bergstra, J., Smolka, S.: Axiomization probabilistic processes: Acp with generative probabililties (extended abstract). In: Cleaveland [11], pp. 472–485
3. Baeten, J., Klop, J. (eds.): CONCUR 1990. LNCS, vol. 458. Springer, Heidelberg (1990)
4. Bandini, E., Segala, R.: Axiomatizations for probabilistic bisimulation. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 370–381. Springer, Heidelberg (2001)
5. Bartels, F., Sokolova, A., de Vink, E.: A hierarchy of probabilistic system types. TCS 327(1-2), 3–22 (2004)
6. Bonchi, F., Bonsangue, M., Rutten, J., Silva, A.: Deriving syntax and axioms for quantitative regular behaviours. CWI technical report (2009)
7. Bonsangue, M., Rutten, J., Silva, A.: An algebra for Kripke polynomial coalgebras. In: LICS (to appear, 2009)
8. Brzozowski, J.: Derivatives of regular expressions. Journal of the ACM 11(4), 481–494 (1964)
9. Buchholz, P.: Bisimulation relations for weighted automata. TCS 393(1-3), 109–123 (2008)
10. Buchholz, P., Kemper, P.: Quantifying the dynamic behavior of process algebras. In: de Luca, L., Gilmore, S. (eds.) PROBMIV 2001, PAPM-PROBMIV 2001, and PAPM 2001. LNCS, vol. 2165, pp. 184–199. Springer, Heidelberg (2001)
11. Cleaveland, R. (ed.): CONCUR 1992. LNCS, vol. 630. Springer, Heidelberg (1992)
12. D'Argenio, P., Hermanns, H., Katoen, J.-P.: On generative parallel composition. ENTCS 22 (1999)
13. Deng, Y., Palamidessi, C.: Axiomatizations for Probabilistic Finite-State Behaviors. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 110–124. Springer, Heidelberg (2005)

14. Deng, Y., Palamidessi, C., Pang, J.: Compositional reasoning for probabilistic finite-state behaviors. In: Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R. (eds.) Processes, Terms and Cycles: Steps on the Road to Infinity. LNCS, vol. 3838, pp. 309–337. Springer, Heidelberg (2005)
15. Droste, M., Gastin, P.: Weighted Automata and Weighted Logics. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 513–525. Springer, Heidelberg (2005)
16. Giacalone, A., Jou, C., Smolka, S.: Algebraic reasoning for probabilistic concurrent systems. In: Broy, Jones (eds.) Proc. of IFIP TC 2 (1990)
17. Gumm, H., Schröder, T.: Monoid-labeled transition systems. ENTCS 44(1) (2001)
18. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Form. Asp. Comp. 6(5), 512–535 (1994)
19. Jacobs, B.: Many-sorted coalgebraic modal logic: a model-theoretic study. ITA 35(1), 31–59 (2001)
20. Jacobs, B.: A Bialgebraic Review of Deterministic Automata, Regular Expressions and Languages. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 375–404. Springer, Heidelberg (2006)
21. Jou, C., Smolka, S.: Equivalences, congruences, and complete axiomatizations for probabilistic processes. In: Baeten, Klop [3], pp. 367–383
22. Kleene, S.: Representation of events in nerve nets and finite automata. Automata Studies, 3–42 (1956)
23. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. In: Logic in Computer Science, pp. 214–225 (1991)
24. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Inf. Comp. 94(1), 1–28 (1991)
25. Larsen, K., Skou, A.: Compositional verification of probabilistic processes. In: Cleaveland [11], pp. 456–471
26. Milner, R.: A complete inference system for a class of regular behaviours. J. Comp. Syst. Sci. 28(3), 439–466 (1984)
27. Mislove, M., Ouaknine, J., Worrell, J.: Axioms for probability and nondeterminism. ENTCS 96, 7–28 (2004)
28. Pnueli, A., Zuck, L.: Probabilistic verification by tableaux. In: LICS, pp. 322–331. IEEE, Los Alamitos (1986)
29. Rabin, M.: Probabilistic automata. Information and Control 6(3), 230–245 (1963)
30. Rößiger, M.: Coalgebras and modal logic. ENTCS 33 (2000)
31. Rutten, J.: Universal coalgebra: a theory of systems. TCS 249(1), 3–80 (2000)
32. Salomaa, A.: Two complete axiom systems for the algebra of regular events. J. ACM 13(1), 158–169 (1966)
33. Schützenberger, M.: On the definition of a family of automata. Information and Control 4(2-3), 245–270 (1961)
34. Segala, R.: Modeling and verification of randomized distributed real-time systems. PhD thesis, MIT (1995)
35. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 481–496. Springer, Heidelberg (1994)
36. Smolka, S., Steffen, B.: Priority as extremal probability. In: Baeten, Klop [3], pp. 456–466
37. Stark, E., Smolka, S.: A complete axiom system for finite-state probabilistic processes. In: Plotkin, et al. (eds.) Proof, Language, and Interaction, pp. 571–596. MIT Press, Cambridge (2000)
38. van Glabbeek, R., Smolka, S., Steffen, B.: Reactive, generative and stratified models of probabilistic processes. Inf. Comput. 121(1), 59–80 (1995)
39. Vardi, M.: Automatic verification of probabilistic concurrent finite-state programs. In: FOCS, pp. 327–338. IEEE, Los Alamitos (1985)

Weighted Bisimulation in Linear Algebraic Form[⋆]

Michele Boreale

Università di Firenze
Dipartimento di Sistemi e Informatica, Viale Morgagni 65, I-50134, Firenze
boreale@dsi.unifi.it

Abstract. We study bisimulation and minimization for weighted automata, relying on a geometrical representation of the model, linear weighted automata (LWA). In a LWA, the state-space of the automaton is represented by a vector space, and the transitions and weighting maps by linear morphisms over this vector space. Weighted bisimulations are represented by sub-spaces that are invariant under the transition morphisms. We show that the largest bisimulation coincides with weighted language equivalence, can be computed by a geometrical version of partition refinement and that the corresponding quotient gives rise to the minimal weighted-language equivalence automaton. Relations to Larsen and Skou's probabilistic bisimulation and to classical results in Automata Theory are also discussed.

1 Introduction

We study bisimulation [17, 18], quotients and minimization in finite-state weighted automata over the field of real numbers. The motivation of this study is twofold. In the past two decades, bisimulation has been adapted to several flavours of probabilistic and stochastic systems. There is a large body of literature on the subject, see e.g. [1, 4, 5, 11, 15] and references therein. These definitions are presented in different formats and notations and give rise, in general, to different equivalences, whose mutual relationships are difficult to asses. At the same time, one wonders if any alternative, linear-time equivalence exists that might replace bisimulation-based equivalences in certain situations. On ordinary LTS's, language equivalence, aka may testing [9], is known to be more generous than bisimulation and appropriate to reason on certain classes of properties, like safety ones. Unfortunately, it is also much more complex to decide than bisimilarity [16]. In practice, one often uses bisimulation as an incomplete proof technique for language equivalence. One wonders to what extent this state of things carries over to the weighted setting.

In an ordinary automaton, each state is associated with a recognized language. Likewise, in a weighted automaton, each state q is associated with a recognized *weighted* language $\sigma(q)$, that is, a set of words each coming with a weight

⋆ Work partially supported by EU within the FET-GC2 initiative, project SENSORIA.

(probability/multiplicity/cost...). Classically, weighted languages are known as *formal power series* [2]. Two states q and q' are weighted language equivalent if $\sigma(q) = \sigma(q')$. It is worth to notice that, in the case of probabilistic automata, $\sigma(q)$ has a very natural interpretation: the weight of a word $x = a_1 \cdots a_n$ in $\sigma(q)$ is just the probability of observing the word x if the execution of the system starts from q. Or, to phrase it in the testing equivalence jargon [9], it is the probability that the system passes the test $\overline{a}_1. \cdots \overline{a}_n.\omega$ starting its execution from q. In other weighted settings, like the counting automata of [19], language equivalence enjoys an equally natural interpretation. Now, on ordinary automata, one way of computing the minimal language equivalent automaton is to first make the original automaton deterministic and then quotient the result of this operation by the largest bisimulation. It is also known that the first operation, Kleene powerset construction, takes exponential time. One wonders what is an equivalent construction in the weighted setting. Or, in other words, what is, if any, the form of bisimulation underpinned by language-preserving minimization in weighted automata. Note that a polynomial weighted-language-preserving minimization procedure for weighted automata has been known for more than forty years [22]. This leads us to the second motivation for our study, that is, to clarify the connections of bisimulation for weighted transition systems and similar structures to classical results in Automata and Language Theory.

We undertake our study by first introducing a linear algebraic representation of weighted automata, *linear weighted automata* (LWA, Section 2). In the familiar representation, transitions of a weighted automaton can be viewed as maps taking each individual state into a set of states, each having its own weight. It is useful to view this set as a formal linear combination of states. It is then natural to extend the transition maps so that they take linear combinations to linear combinations (of states). This leads to the notion of LWA, where the state-space of an automaton is a vector space – the set of linear combinations of states – and the transitions are linear maps. In this formulation, it is natural to define a *linear weighted bisimulation* (Section 3) as a sub-space that is invariant under the transition maps and is included in the kernel of the weight function. This definition retains the nice coinductive proof technique found in ordinary bisimulation: to prove two stated related, it is sufficient to exhibit a "small" bisimulation relation containing them as a pair. We show that the largest linear weighted bisimulation equivalence exists and coincides with coincides with weighted language equivalence. Moreover, it can be effectively computed by a geometrical version of the partition refinement algorithm (Section 4). Or, more accurately, a basis of the corresponding sub-space can be effectively computed. The resulting algorithm is polynomial in the dimension, i.e. the number of states, of the underlying weighted automaton. We next show that taking the quotient of a LWA by a bisimulation corresponds, geometrically, to projecting the original state-space onto the *(orthogonal) complement* of the sub-space representing the bisimulation (Section 5). When the chosen bisimulation is the largest one, this operation results into a practical method for constructing the minimal language-equivalent LWA out of a given one.

The overall construction resembles that for ordinary automata, with determinization corresponding to building the LWA. The important difference is that here there is no exponential blow-up involved. When we specialize this construction to automata with an initial state (Section 7), we re-discover essentially the original minimization algorithm proposed by Schützenberger [22]. The minimal form is canonical, in the sense that minimal LWA's representing the same weighted language are isomorphic. We also compare linear weighted bisimilarity to the probabilistic bisimulation of Larsen and Skou [15] and find the latter to be to be strictly finer than the former (Section 6).

Our work is related to recent and less recent work by Buchholz and Kemper [5, 6], by Rutten [19–21] and by Stark [24], who have studied some of the issues we consider here (see Section 8).

In the paper, we will make use of a few concepts from elementary Linear Algebra, whose description can be found in any introductory textbook, such as [14]. Due to lack of space, most proofs have been omitted in this short version: they can be found in the full version available online [3].

2 Linear Weighted Automata

In the sequel, we fix a finite, non-empty alphabet A of *actions*. V will denote a finite-dimensional inner product space, that is a vector space over $\mathbb{R}$ equipped with a inner product $\langle\cdot,\cdot\rangle : V\times V \to \mathbb{R}$ (in fact, inner product will not come into use until Section 4). We will often refer to elements of a vector space as "vectors"; the null vector will be always denoted by 0, as the context will be always sufficient to avoid confusion with the zero scalar. We shall often omit brackets surrounding function arguments, writing e.g. Tv rather than $T(v)$, unless this jeopardizes readability. "Homomorphism" is shortened as "morphism".

Definition 1 (weighted automaton in linear form). *A* linear weighted automaton *(*LWA*, for short) is a triple $L = (V, \{T_a\}_{a\in A}, \phi)$, where V is a inner product space over $\mathbb{R}$, and $T_a : V \to V$, for each $a \in A$, and $\phi : V \to \mathbb{R}$ are morphisms. The* dimension *of L is $\dim(L) \triangleq \dim(V)$.*

The three elements of a LWA L are referred to as: the *state-space* V, the *transition functions* T_a, $a \in A$, and the *(final) weight function* ϕ, respectively. We do not consider yet initial distributions on states, that will be the subject of Section 7. In the rest of the section, L will denote a generic LWA $(V, \{T_a\}_{a\in A}, \phi)$. A family of morphisms indexed over A, $\{T_a\}_{a\in A}$, induces a family of morphisms indexed over A^*, $\{T_x\}_{x\in A^*}$, defined as follows: for each $v \in V$, $T_\epsilon v \triangleq v$, $T_{ax}v \triangleq T_x T_a v$. Recall that a *formal power series* (FPS for short) over A and $\mathbb{R}$ is a function $\sigma : A^* \to \mathbb{R}$. We view a FPS as the same thing as a *weighted language*, a generalization of the usual notion of language where which word comes with a real multiplicity, possibly 0 to indicate absence.

Definition 2 (weighted language semantics). *The* weighted language *associated by L to any $v \in V$ is the* FPS *$\sigma_L(v) : A^* \to \mathbb{R}$ defined by: $\sigma_L(v)(x) \triangleq \phi(T_x v)$ for each $x \in A^*$. We say u and v are* weighted-language equivalent *if $\sigma_L(u) = \sigma_L(v)$.*

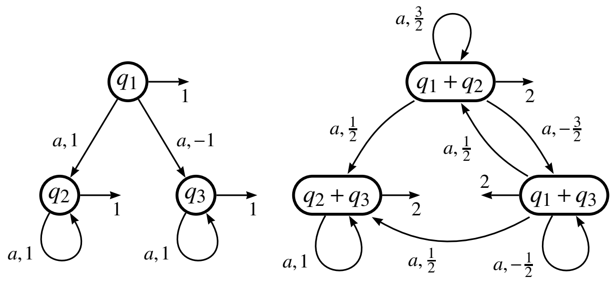


Fig. 1. Two weighted automata representing the same linear weighted automaton

Weighted automata are often represented in matrix form. Ignoring for the moment initial states, a weighted automaton (WA, for short) W is often defined as a triple $(Q, \{M_a\}_{a\in A}, f)$, where: $Q = (q_1, ..., q_n)$ is an ordered finite set of states; each $M_a \in \mathbb{R}^{n\times n}$ is a real-valued square matrix, with $M_a(i, j)$ specifying the weight of the a-transition from q_j to q_i; and $f \in \mathbb{R}^{1\times n}$ is a real-valued row vector, describing the final weights assigned to the q_i's (see e.g. [5]). The weighted language semantics of W can be described as follows. Given an initial distribution on states specified by a column vector $s \in \mathbb{R}^{n\times 1}$, the FPS associated to s by W is given by: $\sigma_W(s)(x) \triangleq fM_x s$, where for $x = a_1 \cdots a_k$, M_x is the product matrix $M_{a_n} \cdots M_{a_1}$ (with $M_\epsilon = I$). The matrix representation corresponds, up to the ordering of states, to the familiar graphical representation (see the next example).

It should be evident that the matrix formulation is equivalent to the one given in Definition 1. More precisely, given a LWA L, by fixing an ordered basis $Q = (e_1, ..., e_n)$ of V one determines a WA $W_{L,Q} = (Q, \{M_a\}_{a\in A}, f)$, where M_a (resp. f) is the matrix (resp. row vector) representing T_a (resp. ϕ) in the basis Q. This correspondence preserves weighted-language semantics, that is, for each $v \in V$, $\sigma_L(v) = \sigma_{W_{L,Q}}(s)$, where s is the column vector of coordinates of v in Q. Conversely, a WA $W = (Q, \{M_a\}_{a\in A}, f)$ determines a LWA $L_W = (\mathbb{R}^Q, \{T_a\}_{a\in A}, \phi)$, by considering $\mathbb{R}^Q$ as the (free) vector space with the expected inner product ($g \cdot h \triangleq \sum_{q\in Q} g(q) \cdot h(q)$), and taking T_a (resp. ϕ) to be the linear morphism on $\mathbb{R}^Q$ represented by the matrix M_a (resp. row-vector f) in the basis[1] Q. Again, this correspondence preserves the weighted-language semantics. The two constructions are inverse of one another, e.g. one has $W_{L_W,Q} = W$.

Example 1. Let $A = \{a\}$ be a singleton alphabet and $Q = (q_1, q_2, q_3)$. Consider the WA $W = (Q, \{M_a\}, f)$ represented by the graph in Fig. 1, on the left. Transitions having weight 0 are not displayed. The standard convention is adopted to specify final weights: $\textcircled{$q_i$}\xrightarrow[r]{}$ means that $f_i = r$ (this graphical representation alone actually determines the matrix representation up to the ordering of the states q_1, q_2, q_3). The WA W gives rise to the LWA $L = L_W = (\mathbb{R}^Q, \{T_a\}, \phi)$. Another representation of the same LWA L, this time w.r.t. the basis $Q' = (q_1+q_2, q_2+q_3, q_1+q_3)$,

[1] As customary, we identify each $q \in Q$ with $\delta_q \in \mathbb{R}^Q$ defined as: $\delta_q(q') = 1$ if $q' = q$, $\delta_q(q') = 0$ otherwise. Under this identification, we have $Q \subseteq \mathbb{R}^Q$.

is given by the automaton W' in Fig. 2 on the right. That is, we have that $L = L_W = L_{W'}$. Note that, in general, WA's derived from of the same LWA by fixing different bases have *similar* transition matrices [14]. The difference between them may only be relevant for computational purposes. As an example one will generally prefer to work with the representation on the left rather than with the one on the right above. More discussion on similarity can be found in [3].

3 Linear Weighted Bisimulation

We first show how to represent binary relations over V as sub-spaces of V, following [24].

Definition 3 (linear relations). *Let U be a sub-space of V. The binary relation R_U over V is defined by: $u\, R_U\, v$ if and only if $u - v \in U$. A relation R is* linear *if there is a subspace U such that $R = R_U$.*

Note that a linear relation is a total equivalence relation on V. Let now R be *any* binary relation over V. There is a canonical way of turning R into a linear relation, which we describe in the following. The *kernel* of R is defined by: $\mathrm{ker}(R) \stackrel{\triangle}{=} \{u - v \,|\, uRv\}$. The *linear extension* of R, denoted R^ℓ, is defined by: $uR^\ell v$ if and only if $(u - v) \in \mathrm{span}(\mathrm{ker}(R))$. The following lemma summarizes two useful facts about linear relations.

Lemma 1. *(1) Let U be a sub-space of V, then $\mathrm{ker}(R_U) = U$. (2) Given any binary relation R, R^ℓ is the smallest linear relation containing R.*

According to the first part of the above lemma, a linear relation R is completely described by its kernel, which is a sub-space, that is

$$uRv \quad \text{if and only if} \quad (u - v) \in \mathrm{ker}(R). \tag{1}$$

Conversely, to any sub-space $U \subseteq V$ there corresponds, by definition, a linear relation R_U whose kernel is U. Hence, without loss of generality, *we can identify linear relations on V with sub-spaces of V*. For example, by slight abuse of notation, we can write $u\,U\,v$ instead of $u R_U v$; and conversely, we will sometime denote by R the sub-space $\mathrm{ker}(R)$, for a linear relation R. The context will be sufficient to tell whether we are actually referring to a linear relation or to the corresponding sub-space (kernel). Note that the sub-space $\{0\}$ corresponds to the identity relation on V, that is $R_{\{0\}} = Id_V$. In fact: $u\, Id_V\, u$ iff $u = v$ iff $u - v = 0$. Similarly, the space V itself corresponds the universal relation on V. Another consequence of the preceding lemma, part (2), is that it is not restrictive to confine ourselves, as we do below, to relations over V that are linear. Note that, again by virtue of (2), $R^\ell = R$ if R is linear, hence $(\cdot)^\ell$ is idempotent: $(R^\ell)^\ell = R^\ell$.

We are now set to define linear weighted bisimulation. The definition relies on the familiar step-by-step game on transitions, plus an initial condition requiring that two related states have the same weight. We christen this form of bisimulation *linear* to stress the difference with other forms of bisimulation proposed for WA's [5]. In the rest of the section, we let L denote a generic LWA $(V, \{T_a\}_{a \in A}, \phi)$.

Definition 4 (linear weighted bisimulation). *Let L be a* LWA. *A linear relation R over V is a* linear weighted L-bisimulation *(L-bisimulation, for short) if whenever uRv then: (a) $\phi(u) = \phi(v)$ and (b) $T_a u \; R \; T_a v$ for each $a \in A$.*

That a largest L-bisimulation, denoted $\sim_L$, exists, is quite obvious: in analogy with the ordinary case, one takes the *span* of the union of all L-bisimulations, and checks that it is in turn a L-bisimulation, the largest one. Note that the mentioned union is non-empty, as e.g. the identity relation is a L-bisimulation. We shall give two characterizations of $\sim_L$, one in terms of language equivalence (Theorem 1) and one in algorithmic terms (Theorem 2). A useful property of bisimulation on ordinary transition systems is that, to prove two states related, exhibiting a "small" relation containing the given pair is sufficient. This property is preserved in the present setting, despite the fact that Definition 4 mentions linear, hence total, relations on V.

Lemma 2. *Let L be a* LWA *and R be a binary relation over V satisfying clauses (a) and (b) of Definition 4. Then R^ℓ is the smallest weighted L-bisimulation containing R.*

The following lemma provides a somewhat handier characterization of linear weighted bisimulation. Let us say that a sub-space U is *T-invariant* if $T(U) \subseteq U$. Bisimulations are transition-invariant relations that refine the kernel of ϕ.

Lemma 3. *Let L be a* LWA *and R be linear relation over V. R is a L-bisimulation if and only if (a)* $\ker(\phi) \supseteq R$, *and (b) R is T_a-invariant for each $a \in A$.*

The largest L-bisimulation $\sim_L$ coincides with the weighted-language equivalence.

Theorem 1. *For any $u, v \in V$, we have that $u \sim_L v$ if and only if $\sigma_L(u) = \sigma_L(v)$.*

4 Partition Refinement

"Two well-known concepts from Linear Algebra, orthogonal complements and transpose morphisms, are used to describe geometrically two basic operations of the algorithm, relation complementing and "arrows reversing", respectively."

Let U, W be sub-spaces of V. We recall that the orthogonal complement $U^\perp$ enjoys the following properties: (i) $U^\perp$ is a sub-space of V; (ii) $(\cdot)^\perp$ reverses inclusions, i.e. if $U \subseteq W$ then $W^\perp \subseteq U^\perp$; (iii) $(\cdot)^\perp$ is an involution, that is $(U^\perp)^\perp = U$. These three properties suggest that $U^\perp$ can be regarded as a *complement*, or negation, of U seen as a relation. Another useful property is: (iv) $\dim(U^\perp) + \dim(U) = \dim(V)$. Concerning transpose morphisms, we have the following definition. The need for ortho-normal bases is explained in the remark below.

Definition 5 (transpose morphism). *Let $T : V \to V$ be any endomorphism on V. Fix any ortho-normal basis of V and let M be the square matrix representing T in this basis. We let the* transpose of T, *written ${}^{\mathrm{t}}T$, be the endomorphism $V \to V$ represented by ${}^{\mathrm{t}}M$ in the given basis.*

Remark 1. It is easy to check that the definition of ${}^{\mathrm{t}}T$ does *not* depend on the choice of the ortho-normal basis: this is a consequence of fact that the change of basis matrix N between two ortho-normal bases is unitary ($N^{-1} = {}^{\mathrm{t}}N$). The transpose operator is of course an involution, in the sense that ${}^{\mathrm{t}}({}^{\mathrm{t}}T) = T$.

Transpose morphisms and orthogonal spaces are connected via the following property, which is crucial to the development of the partition refinement algorithm. It basically asserts that T-invariance of R corresponds to ${}^{\mathrm{t}}T$-invariance of the complementary relation $R^{\perp}$.

Lemma 4. *Let U be a sub-space of V and T be an endomorphism on V. If U is T-invariant then $U^{\perp}$ is ${}^{\mathrm{t}}T$-invariant.*

An informal preview of the algorithm is as follows. Rather than computing directly the sub-space representing $\sim_L$, the algorithm computes the sub-space representing the complementary relation. To this end, the algorithm starts from a relation R_0 that is the complement of the relation identifying vectors with equal weights, then incrementally computes the space of all states that are *backward* reachable from R_0. The largest bisimulation is obtained by taking the complement of this space. Geometrically, "going backward" means working with the transpose transition functions ${}^{\mathrm{t}}T_a$ rather than with T_a. Taking the complement of a relation actually means taking its orthogonal complement. Recall that $U + W \stackrel{\triangle}{=} \mathrm{span}(U \cup W)$.

Theorem 2 (partition refinement). *Let L be a* LWA. *Consider the sequence $(R_i)_{i\geq 0}$ of sub-spaces of V inductively defined by: $R_0 = \ker(\phi)^{\perp}$ and $R_{i+1} = R_i + \sum_{a\in A} {}^{\mathrm{t}}T_a(R_i)$. Then there is $j \leq \dim(L)$ s.t. $R_{j+1} = R_j$. The largest L-bisimulation is $\sim_L = R_j^{\perp}$.*

Proof. Since $R_0 \subseteq R_1 \subseteq R_2 \subseteq \cdots \subseteq V$, the sequence of the dimensions of these spaces is non-decreasing. As a consequence, for some $j \leq \dim(V)$, we get $\dim(R_j) = \dim(R_{j+1})$. Since $R_j \subseteq R_{j+1}$, this implies $R_j = R_{j+1}$.

We next show that $R_j^{\perp}$ is a L-bisimulation. Indeed, by the properties of the orthogonal complement: (a) $\ker(\phi)^{\perp} \subseteq R_j$ implies $(\ker(\phi)^{\perp})^{\perp} = \ker(\phi) \supseteq R_j^{\perp}$. Moreover: (b) for any action a, ${}^{\mathrm{t}}T_a(R_j) \subseteq {}^{\mathrm{t}}T_a(R_j) + R_j \subseteq R_{j+1} = R_j$ implies, by Lemma 4, that ${}^{\mathrm{t}}({}^{\mathrm{t}}T_a(R_j^{\perp})) = T_a(R_j^{\perp}) \subseteq R_j^{\perp}$; by (a), (b) and Lemma 3, we conclude that $R_j^{\perp}$ is an L-bisimulation.

We finally show that any L-bisimulation S is included in $R_j^{\perp}$. We do so by proving that for each i, $S \subseteq R_i^{\perp}$, thus, in particular $S \subseteq R_j^{\perp}$. We proceed by induction on i. Again by Lemma 3, we know that $R_0^{\perp} = \ker(\phi) \supseteq S$. Assume now $S \subseteq R_i^{\perp}$, that is, $S^{\perp} \supseteq R_i$. For each action a, by Lemma 3 we have that $T_a(S) \subseteq S$, which implies ${}^{\mathrm{t}}T_a(S^{\perp}) \subseteq S^{\perp}$ by Lemma 4. Hence $S^{\perp} \supseteq {}^{\mathrm{t}}T_a(S^{\perp}) \supseteq {}^{\mathrm{t}}T_a(R_i)$, where the last inclusion stems from $S^{\perp} \supseteq R_i$. Since this holds for each a, we have that $S^{\perp} \supseteq \sum_a {}^{\mathrm{t}}T_a(R_i) + R_i = R_{i+1}$. Taking the orthogonal complement on both sides reverses the inclusion and yields the wanted result.

Remark 2. What is being "refined" in the algorithm above are not, of course, the sub-spaces R_i, but their orthogonal complements: $R_0^{\perp} \supseteq R_1^{\perp} \supseteq \cdots \supseteq R_j^{\perp} = \sim_L$. One could also devise a version of the above algorithm that starts from $\ker(\phi)$ and refines it working forward, operating with intersections of sub-spaces rather than with sums. This "forward" version appears to be less convenient computationally as $\ker(\phi)$ is a large sub-space: since $\phi : V \to \mathbb{R}$ with $\dim(\mathbb{R}) = 1$, by virtue of the fundamental identity relating the dimensions of the kernel and of the image of a morphism, we have that $\dim(\ker(\phi)) \geq \dim(V) - 1$.

By virtue of (1), checking $u \sim_L v$, for any pair of vectors u and v, is equivalent to checking $u - v \in \ker(\sim_L)$. This can be done by first computing a basis of $\sim_L$ and then checking for linear (in)dependence of $u - v$ from this basis. Alternatively, and more efficiently, one can check whether $u - v$ is in the orthogonal complement of R_j, by showing that $u - v$ is orthogonal to each element of a basis of R_j. Thus, our task reduces to computing one such basis. To do so, we fix any orthonormal basis B of V: all the computations are carried out representing coordinates in this basis. Let f and M_a $(a \in A)$ be the row-vector and matrices, respectively, representing the weight and transition functions of the LWA in this basis. Then a sequence of basis $B_0, ..., B_j$ for the sub-spaces $R_0, ..., R_j$ can be iteratively computed starting with $B_0 = \{v_0\}$, where v_0 is the vector represented by f, which is a basis for $\ker(\phi)^\perp$. The resulting algorithm requires a polynomial (cubic) number of floating point operations in the dimension of the automaton. The algorithm is illustrated below.

Example 2. Consider the LWA $L = (V, \{T_a\}, \phi)$, with $V = \mathbb{R}^Q$ and $Q = (q_1, q_2, q_3)$, given in Example 1. The WA describing L w.r.t. Q is the one depicted in Fig. 1, on the left. Q is an ortho-normal basis, so that it is easy to represent the transpose transitions ${}^tT_a$. According to the above outline of the algorithm, since $f = (1, 1, 1)$ represents ϕ in Q, we have that $R_0 = \ker(\phi)^\perp$ is spanned by $v_0 = q_1 + q_2 + q_3$. Next, we apply the algorithm to build the B_i's as described above. Manually, the computation of the vectors ${}^tT_a v$ can be carried out by looking at the transitions of the WA with arrows reversed. Since ${}^tT_a(q_1 + q_2 + q_3) = q_1 + q_2 - q_1 + q_3 = q_2 + q_3$ and ${}^tT_a(q_1 + q_2 + q_3) = q_2 + q_3$, we obtain $B_0 = \{q_1 + q_2 + q_3\}$, then $B_1 = \{q_1 + q_2 + q_3,\ q_2 + q_3\}$ and finally $B_2 = B_1$. Hence B_1 is a basis of $(\sim_L)^\perp$. As an example, let us check that $q_1 \sim_L q_1 + q_2 - q_3$. To this purpose, note that the difference vector $(q_1 + q_2 - q_3) - q_1 = q_2 - q_3$ is orthogonal to each elements of B_1, which is equivalent to $q_1 \sim_L q_1 + q_2 + q_3$.

5 Quotients

The purpose of the quotient operation is to obtain a reduced automaton that has the same semantics as the original one. Let us make the notions of reduction and minimality precise first.

Definition 6 (reduction, minimality). *Let L and L' be two* LWA*'s having V and V', respectively, as underlying state-spaces. Let $h : V \to V'$ be a morphism. We say (h, L') is a* reduction *of L if $\dim(L') \leq \dim(L)$ and for each $v \in V$, $\sigma_L(v) = \sigma_{L'}(hv)$. We say L is* minimal *if for every reduction (h, L') of L we have $\dim(L) = \dim(L')$.*

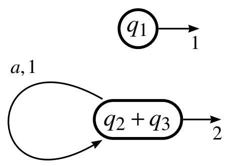


Fig. 2. A minimal weighted automaton

We now come to the actual construction of the minimal automaton. This is basically obtained by quotienting the original space by the largest bisimulation. In inner product spaces, there is a canonical way of representing quotients as orthogonal complements. Let U be any sub-space of V. Then V can be decomposed as the direct sum of two sub-spaces: $V = U \oplus U^\perp$ and $\dim(V) = \dim(U) + \dim(U^\perp)$. This means that any element $v \in V$ can be written in a *unique way* as a sum $v = u + w$ with $u \in U$ and $w \in U^\perp$. The orthogonal *projection* of V onto $U^\perp$ is the morphism $\pi : V \to U^\perp$ defined as $\pi(u + w) \stackrel{\triangle}{=} w$. The following lemma says that taking the quotient of V by a linear relation (whose kernel is) U amounts to "collapsing" vectors of V along the U-direction. Or, in other words, to projecting vectors of V onto $U^\perp$, the sub-space orthogonal to U (this is a well known result in Linear Algebra).

Lemma 5. *Let U be a sub-space of V and $\pi : V \to U^\perp$ be the projection onto $U^\perp$. Then for each $u, v \in V$: (a) $u\, U\, v$ if and only if $\pi u = \pi v$; (b) $u\, U\, \pi u$.*

In view of the above lemma, we will sometimes denote the orthogonal complement of U w.r.t. V as "V/U". In what follows, L denotes a LWA $(V, \{T_a\}_{a \in A}, \phi)$. We shall make use of the morphisms $(\pi T)_a \stackrel{\triangle}{=} \pi \circ T_a$, for $a \in A$.

Definition 7 (quotient automaton). *Let R be a L-bisimulation and let π be the projection function onto V/R. We let the* quotient automaton *L/R be $(V/R, \{T^{\mathrm{q}}_a\}_{a \in A}, \phi^{\mathrm{q}})$ where $T^{\mathrm{q}}_a = (\pi T_a)_{|V/R}$ and $\phi^{\mathrm{q}} = \phi_{|V/R}$.*

Theorem 3 (minimal automaton). *Let R be a L-bisimulation and π be the projection function from V onto V/R. Then $(\pi, L/R)$ is a reduction of L such that: (a) $\dim(L/R) = \dim(L) - \dim(R)$, and (b) for each $u, v \in V$, $u \sim_L v$ if and only if $\pi u \sim_{L/R} \pi v$. Moreover, if R is $\sim_L$, then L/R is minimal and the following* coinduction *principle holds: for each $u, v \in V$, $u \sim_L v$ if and only if $\pi u = \pi v$.*

It is well-known that, when B is an orthogonal basis, for each $v \in V$, the projection of v onto the space spanned by B, πv, can be written as

$$\pi v = \sum_{e \in B} \frac{\langle v, e \rangle}{\langle e, e \rangle} e. \tag{2}$$

One can give a (concrete) representation of the minimal LWA in terms of a WA, by first computing an orthogonal basis of the quotient space $V/\sim_L$ and then representing the transition T^{q}_a and and final weight ϕ^{q} functions in this basis using the above formula. This is illustrated in the example below.

Example 3. Let us consider again the LWA in Example 2. We give a representation of $L/\sim_L$ as a WA. From Example 2, we know that a basis of $V/\sim_L$ is $B = \{q_1+q_2+q_3, q_2+q_3\}$. It is convenient to turn B into an orthogonal basis, applying an the Gram-Schmidt's [14] orthogonalizing method. We find that $B' = \{q_1, q_2+q_3\}$ is an orthogonal basis of $V/\sim_L$. We now represent the transition function in B'. That is, for any $e \in B'$, we express each $T_a^{\mathrm{q}} e$ as a linear combination of elements of B'. Applying the identity (2), we find that

$$T_a^{\mathrm{q}} q_1 = \pi(q_2 - q_3) = 0$$
$$T_a^{\mathrm{q}}(q_2+q_3) = \pi(q_2+q_3) = q_2+q_3\,.$$

Concerning the weight function, we have: $\phi^{\mathrm{q}}(q_1) = 1$ and $\phi^{\mathrm{q}}(q_2+q_3) = 2$. The resulting WA, which represents the LWA $L/\sim_L$ w.r.t. the basis B', is graphically represented in in Fig. 2. According to Theorem 3, the projection function π turns pairs of bisimilar elements of L into identical ones of $L/\sim_L$. As an example, the relation $q_1 \sim_L q_1+q_2-q_3$ becomes an identity once projected onto $V/\sim_L$: indeed, $\pi q_1 = q_1$ and $\pi(q_1+q_2-q_3) = \pi(q_1)+\pi(q_2-q_3) = q_1+0 = q_1$.

6 Probabilistic Bisimulation

The notion of probabilistic bisimulation was introduced by Larsen and Skou [15], as a generalization to probabilistic transition systems of the older notion of *lumpability* for Markov chains, due to Kemeny and Snell [13]. The notion of probabilistic transition system itself can be found in a number of variations in the literature; see e.g. [1, 11] and references therein. Below, we deal with the the one called *reactive* probabilistic transition system. We comment on another version, the *generative* one, at the end of this section.

In this section, for any finite set Q, $f \in \mathbb{R}^Q$ and $X \subseteq Q$, we let $|f|_X \triangleq \sum_{q\in X} f(q)$. We abbreviate $|f|_Q$ just as $|f|$. A probabilistic transition system is just a weighted automaton with all final weights implicitly set to 1, and where the weights of arcs outgoing a node satisfy certain restrictions.

Definition 8 (probabilistic bisimulation). *A* (finite, reactive) probabilistic transition system *(*PTT*, for short) is a pair $P = (Q, \{t_a\}_{a\in A})$, where Q is finite set of states and $\{t_a\}_{a\in A}$ is a family of functions $Q \to (\mathbb{R}^+)^Q$, such that for each $a \in A$ and $q \in Q$, $|t_a(q)|$ equals* 1 *or* 0*. An equivalence relation $\mathcal{S}$ over Q is a* probabilistic bisimulation *on P if whenever $q\,\mathcal{S}\,q'$ then, for each equivalence class $C \in Q/\mathcal{S}$ and each $a \in A$, $|t_a(q)|_C = |t_a(q')|_C$.*

It is not difficult to see that a largest probabilistic bisimulation on P, denoted $\sim_P$, exists. Let $P = (Q, \{t_a\}_{a\in A})$ be a probabilistic transition system. Any transition function t_a, being defined over Q, which is a basis of $\mathbb{R}^Q$ seen as a free vector space, is extended linearly to an endomorphism on the whole $\mathbb{R}^Q$: we denote this extension by the same name, t_a. With this notational convention, every PTT P determines a LWA $\hat{P} = (\mathbb{R}^Q, \{t_a\}_{a\in A}, \phi)$, where ϕ takes on the value 1 on each element of Q and is extended linearly to the whole space. Note that the semantics of a

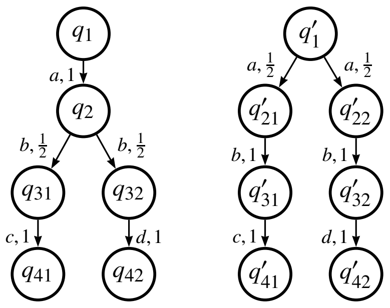


Fig. 3. A probabilistic transition system

PTT is independent of final weights on states; we achieve the same effect here by setting $\phi(q) = 1$, the neutral element of product.

We establish below that, over Q, the largest linear weighted bisimulation, $\sim_{\hat{P}}$, is coarser than the largest probabilistic bisimulation, $\sim_P$. A similar result was already proven by Stark in [24], building on an alternative characterization of probabilistic bisimulation due to Jonsson and Larsen [12]. Our proof is more direct and relies on the following lemma.

Lemma 6. *Let $\mathcal{S}$ be a probabilistic bisimulation on the* PTT $(Q, \{t_a\}_{a\in A})$. *Let $f, g \in \mathbb{R}^Q$ s.t. for each $C \in Q/\mathcal{S}$, $|f|_C = |g|_C$. Then for each $C \in Q/\mathcal{S}$ and $a \in A$, $|t_a f|_C = |t_a g|_C$.*

Theorem 4. *Let $P = (Q, \{t_a\}_{a\in A}, \phi)$ be a* PTT. *If $q \sim_P q'$ in P then $q \sim_{\hat{P}} q'$ in $\hat{P}$.*

Proof. (Sketch) It is shown that that the linear relation $\sim_P^\ell$ over $\mathbb{R}^Q$ defined by: $f \sim_P^\ell g$ if and only if (i) $|f| = |g|$, and (ii) for all $a \in A$ and all equivalence classes $C \in Q/\sim_P$, $|t_a f|_C = |t_a g|_C$, is a linear weighted bisimulation. Lemma 6 is used to check requirement (b) of the definition.

To show that $\sim_P$ makes *less* identifications than $\sim_{\hat{P}}$, we consider the following example.

Example 4. The WA P in Fig. 3, with final weights all implicitly set to 1, is a PTT. Let $L = L_P$ be the corresponding LWA. It is easy to check that $q_1 \sim_L q'_1$. Indeed, consider the "small" relation R, defined thus

$$R = \{(q_1, q'_1), (q_2, \frac{1}{2}(q'_{21} + q'_{22})), (\frac{1}{2}(q_{31} + q_{32}), \frac{1}{2}(q'_{31} + q'_{32})), (\frac{1}{2}(q_{41} + q_{42}), \frac{1}{2}(q'_{41} + q'_{42})), (0,0)\}.$$

One checks that this relation satisfies the clauses of bisimulation. Applying Lemma 2, one thus finds that R^ℓ is a linear weighted bisimulation, hence $q_1 \sim_L q'_1$. In an even more direct fashion, one just checks that $\sigma_L(q_1) = \sigma_L(q'_1)$ and applies Theorem 1. On the other hand, q_1 and q'_1 are not related by any probabilistic bisimulation. In fact, any such relation should group e.g. q_{31} and q_{32} in the same equivalence class: but this is impossible, because q_{31} has a c-transition, whereas q_{32} has not.

The above example highlights the fundamental difference between probabilistic and linear weighted bisimulations. After each step, linear weighted bisimulation may relate "point" states to linear combinations of states: e.g., starting from q_1 and q'_1 and taking an a-step, q_2 and $\frac{1}{2}(q'_{21}+q'_{22})$ are related. This is not possible, by definition, in probabilistic bisimulation. A practical consequence of these results is that quotienting by the largest linear weighted bisimulation yields a minimal automaton that may be smaller than the one obtained when quotienting by probabilistic bisimilarity.

As hinted at the beginning of this section, a different version of probabilistic transition systems exists, the *generative* one. In this version, the requirement "for each $a \in A$ and $q \in Q$, $|t_a(q)|$ equals 1 or 0" is replaced by "for each $q \in Q$, $\sum_{a \in A} |t_a(q)|$ equals 1 or 0"[2]. The results discussed in this section carry over to this class of transition systems.

7 Weighted Automata with an Initial State

WA's are sometimes presented as featuring an initial distribution on states, see e.g. Buchholz's [5]. WA's with an initial distribution are also known as *linear representations* in Automata Theory [2]. In terms of LWA's, assuming an initial distribution on states is equivalent to choosing a distinguished initial vector, or root, so that we can define a *rooted* LWA as a pair (v, L), where $v \in V$. When minimizing, now one has now to take care of preserving only the semantics of the root. In particular, states that are not reachable from the root can be discarded right away, thus allowing for a potentially smaller reduced automaton.

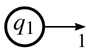
We give here only a brief outline the minimization procedure, which is explained in detail in the full version [3]. The algorithm now consists of two steps: first, the sub-space reachable from the root is computed; second, the sub-automaton obtained by restricting the original one to this sub-space is minimized, according to the method described in Section 5. The second step is clearly a quotient operation involving bisimulation. But in fact, also the first step can be seen as a quotient operation. Indeed, if one looks at the minimization algorithm described in Theorem 2, one sees that computing the sub-space reachable from the root v, and spanned by the vectors $\{T_x v | x \in A^*\}$, is equivalent to computing the largest bisimulation of a LWA with transitions reversed and with a final weight function ψ such that $\mathrm{span}(v) = \ker(\psi)^{\perp}$. Let us denote by $\sim_L^{\mathrm{op}}$ the largest bisimulation in this reverse LWA. More formally, we have the following

Theorem 5 (minimal rooted LWA). *Let (v, L) be a rooted LWA with $L = (V, \{T_a\}_a, \phi)$. Let π be the projection function $V \to V/\sim_L$. Consider the rooted LWA (v_*, L_*), where $v_* = \pi v$ and $L_* = (V_*, \{T_{*a}\}_a, \phi_*)$ is given by $V_* = \pi(V/\sim_L^{\mathrm{op}})$, $T_{*a} = (\pi T_a)_{|V_*}$ and $\phi_* = \phi_{|V_*}$. Then (v_*, L_*) is a minimal reduct of (v, L).*

[2] Modulo the addition of self-loops to sink states, generative and reactive transition systems correspond to Markov Chains and to Markov Decision Processes, respectively.

This construction can be seen essentially as the original two-phase minimization algorithm given by Schützenberger [22], modulo the fact that here the two phases (computing the reachable sub-space and then reducing it) are both described in terms of bisimulation quotients.

When we apply this procedure to the rooted LWA (q_1,L), where L is the LWA of Example 1, we get a minimal rooted LWA that can be represented by the following WA:



Finally, one can show that any two minimal LWA's representing the same FPS are isomorphic, in particular they have the same dimension. This shows that the (minimal) dimension is a feature of FPS's rather than of rooted LWA's. We omit the details here.

8 Related and Further Work

Our formulation of linear weighted bisimulation is primarily related to the definition of *Σ-congruence* put forward by Stark [24]. Σ-congruence is introduced in order to provide a simple formulation of behavioural equivalence in a model of stochastic systems, *Probabilistic Input/Output Automata*, and relate this notion to standard probabilistic bisimulation. In the form studied by Stark, weighted automata do not feature final (nor initial) weights. This form is subsumed by ours once we assign the final weight 1 to all elements of the basis. In this special case, Σ-congruence and linear weighted bisimulation coincide. The representation of linear relations in terms of their kernels is already present in [24]. Partition refinement and quotient/minimization are not tackled, though. A related equivalence for stochastic systems, under the name behaviour equivalence, is studied in [25, 26] (while weighted equivalence indicates there yet another equivalence).

Buchholz and Kemper have put forward a definition of bisimulation for weighted automata over a generic semiring [5, 6]. A largest such bisimulation can be computed by a partition refinement algorithm that works on a matrix representation of the automata [5]; both a forward and a backward version of the equivalence and of the algorithm are investigated. A definition of "aggregated" automaton, corresponding to a quotient, is presented, but a notion of canonical representation is not put forward. Akin to the probabilistic one of Larsen and Skou, and differently from ours and Stark's, Buchholz and Kemper's bisimulations never relate a "point" state to a linear combinations of states. As a consequence, when instantiating the semiring in their framework to $\mathbb{R}$, their notion of equivalence is stricter than ours – and than weighted language equivalence – for the same reasons discussed in Example 4.

Weighted automata and formal power series play a central role in a few recent and less recent papers of Rutten [19–21] on coinduction and (multivariate) streams – another name for FPS's. In [20], weighted automata are used to provide a more compact representation for streams than deterministic (Moore) automata do. Closely related to ours is also [21], where linear representations very similar to our LWA's are considered. Bisimulation is defined over streams – seen as deterministic Moore automata – and two states of a weighted automaton are related iff they generate the same stream. This approach is also taken in [19], where it is shown that infinite weighted automata can be used to enumerate a wide class of combinatorial objects. The stream-based definition can be used to prove an infinite automaton equivalent to a "small" one. The latter can be directly mapped to a closed expressions for the generating function of the enumerated objects.

Weighted automata were first introduced in Schützenberger's classical paper [22], where a minimization algorithm was also discussed. This algorithm has been reformulated in a more algebraic fashion in Berstel and Reutenauer's book [2]. Other descriptions of the algorithm can be found in [7, 10]. Here we explicitly connect this algorithm to the notion of bisimulation. Hopefully, this connection will make the algorithm itself accessible to a larger audience.

Due to lack of space, we have not presented results concerning composition of automata. Indeed, it is quite easy to prove that both direct sum (juxtaposition) and tensor product (synchronization) of LWA's preserve bisimulation equivalence (see also Stark's [24, Section 3]). Also, the results presented here can be extended to the case of vector spaces over a generic field, relying on the concept of dual space (see [3]).

There are several possible directions for future work. One would like to extend the present approach to the case of infinite WA's. This would provide proof techniques, if not effective algorithms, that could be used to reason in a more systematic manner on the counting automata of [19]. Also, it would be interesting to cast the present results in a more explicit co-algebraic setting: this would put them in a deeper perspective and possibly help to explain certain aspects not clear at moment, such as, why the blow up of the ordinary case goes away. It would also be practically relevant to identify classes of properties preserved by linear weighted bisimilarity on probabilistic systems: a preliminary investigation shows that reachability is one such class. The relations of linear weighted bisimilarity to other notions of equivalences/preorders [8, 23] that also relate distributions, rather than individual states, deserves further attention.

References

1. Baier, C., Engelen, B., Majster-Cederbaum, M.E.: Deciding Bisimilarity and Similarity for Probabilistic Processes. Journal of Computer and System Sciences 60(1), 187–231 (2000)
2. Berstel, J., Reutenauer, C.: Rational Series and Their Languages. EATCS Monograph Series. Springer, Heidelberg (1988); New edition, Noncommutative Rational Series With Applications (2008), http://www-igm.univ-mlv.fr/~berstel/LivreSeries/LivreSeries.html
3. Boreale, M.: Weighted bisimulations in linear algebraic form. Full version of the present paper (2009), http://rap.dsi.unifi.it/~boreale/papers/WBG.pdf
4. Buchholz, P.: Exact Performance Equivalence: An Equivalence Relation for Stochastic Automata. Theoretical Computer Science 215(1-2), 263–287 (1999)
5. Buchholz, P.: Bisimulation relations for weighted automata. Theoretical Computer Science 393(1-3), 109–123 (2008)

6. Buchholz, P., Kemper, P.: Quantifying the dynamic behavior of process algebras. In: de Luca, L., Gilmore, S. (eds.) PROBMIV 2001, PAPM-PROBMIV 2001, and PAPM 2001. LNCS, vol. 2165, pp. 184–199. Springer, Heidelberg (2001)
7. Cardon, A., Crochemore, M.: Determination de la representation standard d'une serie reconnaissable. RAIRO Theor. Informatics and Appl. 14, 371–379 (1980)
8. Deng, Y., van Glabbeek, R.J., Hennessy, M., Morgan, C.C.: Characterising testing preorders for finite probabilistic processes. Logical Methods in Computer Science 4(4:4) (2008)
9. De Nicola, R., Hennessy, M.: Testing Equivalences for Processes. Theoretical Compututer Science 34, 83–133 (1984)
10. Flouret, M., Laugerotte, E.: Noncommutative minimization algorithms. Inform. Process. Lett. 64, 123–126 (1997)
11. van Glabbeek, R.J., Smolka, S.A., Steffen, B., Tofts, C.M.N.: Reactive, Generative, and Stratified Models of Probabilistic Processes. In: LICS 1990, pp. 130–141 (1990)
12. Jonsson, B., Larsen, K.G.: Specification and Refinement of Probabilistic Processes. In: LICS 1991, pp. 266–277 (1991)
13. Kemeny, J.G., Snell, J.L.: Finite Markov Chains. Springer, Heidelberg (1976)
14. Lang, S.A.: Introduction to Linear Algebra, 2/e. Springer, Heidelberg (1997)
15. Larsen, K.G., Skou, A.: Bisimulation through Probabilistic Testing. Information and Compututation 94(1), 1–28 (1991)
16. Meyer, A.R., Stockmeyer, L.J.: Word problems requiring exponential time. In: STOC 1973, pp. 1–9 (1973)
17. Milner, R.: A Calculus of Communicating Systems. Prentice-Hall, Englewood Cliffs (1989)
18. Park, D.: Concurrency and Automata on Infinite Sequences. Theoretical Computer Science, 167–183 (1981)
19. Rutten, J.J.M.M.: Coinductive counting with weighted automata. Journal of Automata, Languages and Combinatorics 8(2), 319–352 (2003)
20. Rutten, J.J.M.M.: Behavioural differential equations: a coinductive calculus of streams, automata, and power series. Theoretical Computer Science 308(1-3), 1–53 (2003)
21. Rutten, J.J.M.M.: Rational streams coalgebraically. Logical Methods in Computer Science 4(3) (2008)
22. Schützenberger, M.P.: On the Definition of a Family of Automata. Information and Control 4(2-3), 245–270 (1961)
23. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. PhD thesis, MIT (1995)
24. Stark, E.W.: On Behavior Equivalence for Probabilistic I/O Automata and its Relationship to Probabilistic Bisimulation. Journal of Automata, Languages and Combinatorics 8(2), 361–395 (2003)
25. Stark, E.W., Cleaveland, R., Smolka, S.A.: Probabilistic I/O Automata: Theories of Two Equivalences. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 343–357. Springer, Heidelberg (2006)
26. Wu, S.-H., Smolka, S.A., Stark, E.W.: Composition and behaviors of probabilistic I/O automata. Theoretical Computer Science 176(1-2), 1–38 (1997)

A Logic-Based Framework for Reasoning about Composite Data Structures[⋆]

Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu

LIAFA, University Paris Diderot and CNRS, 75205 Paris 13, France
{abou,cezarad,cenea,sighirea}@liafa.jussieu.fr

Abstract. We define a logic, called CSL, for the specification of complex data structures, and we show its use in program verification. Our framework allows to handle programs with dynamic linked structures and arrays carrying unbounded data, as well as the composition of these structures. The formulas in CSL allow a limited form of alternation between existential and universal quantifiers and they can express (1) constraints on reachability between positions in the heap following some pointer fields, (2) linear constraints on the lengths of the lists and the indexes of the arrays, and (3) constraints on the values of the data attached to these positions. For data constraints, the logic CSL is parameterized by a first-order logic over the considered data domain. We prove that the satisfiability problem of CSL is decidable whenever the underlying data logic is decidable and that CSL is closed under the computation of the strongest post-condition in the considered class of programs.

1 Introduction

Program verification requires reasoning about complex, unbounded size structures that may carry data ranging over infinite domains. Examples of such structures are multi-linked data structures, arrays, as well as compositions of these structures (e.g., lists of doubly linked lists of arrays of ... integers). Programs manipulating such structures can perform operations that may modify the shape of these structures (due to dynamic creation and destructive updates) as well as the data attached to their elements. An important issue is the design of logic-based frameworks allowing to express assertions about program configurations (at given control points), and then to check automatically the validity of these assertions, for all computations. This leads to the challenging problem of finding compromises between expressiveness and decidability.

Concerning expressiveness, it is important in the context we consider, to be able to use (1) some form of reachability predicates (or ordering predicates) between positions in the structures, and (2) constraints on the data attached to these positions. It is also important to have a framework which allows to handle data structures beyond words and trees. For instance, logics on words do not allow to reason in a natural way about transformations (such as destructive updates) of linked lists. Similarly, transformations of linked tree structures may lead in intermediary steps to structures which are not trees. Furthermore, many of the commonly used data structures are composite and combine several kinds of linked structures and arrays, containing data as well as pointers to elements in different substructures.

⋆ Partially supported by the french ANR project AVERISS and the RNTL project AVERILES.

From the point of view of decidability, the problem is that, regardless of the unbounded data issue, beyond tree-like structures, the use of reachability predicates may lead to undecidability when quantification over positions is not restricted. Moreover, when unbounded data domains are considered, even for relatively simple structures such as words, the obtained logics are undecidable in general.

In this paper, we propose a logic called CSL (for Composite Structures Logic) which, we believe, offers such a good compromise we are seeking. The logic CSL is parameterized by a first-order logic over some data domain. Models of CSL are graphs (of arbitrary size and shape) where each vertex has a type. Types correspond to a fixed number of data fields and a fixed number of pointer fields. Pointers define successor relations in the structure (they correspond to edges in the graph). They are also typed in the sense that their origins and their targets are always vertices of some fixed types. Special pointer fields are used to encode arrays. These fields are supposed to define distinct acyclic paths.

Location variables (which can be existentially or universally quantified) are used to refer to positions (vertices) in the models. Formulas in CSL allow to express that two positions are related by a path following (forward and/or backward) some specific pointer fields. For instance, saying that positions u and v are related by a doubly linked path can be written $u \xrightarrow{\{f,\overline{b}\}} v$ where f (resp. b) is the forward (resp. backward) successor field. Also, we can represent the address of an array element by a term of the form $a[i]$ where a is an array variable which is assigned to a position representing the first element of the array and i is an index variable representing the distance to that position on a path that uses the distinguished array field. These terms can be used to specify linked structures containing elements from different arrays. Array and index variables can be existentially or universally quantified.

Putting arithmetical constraints on index variables allows to reason about distances between positions of elements in the arrays. This can be generalized to linked structures using formulas of the form $u \xrightarrow{\{f,\overline{b}\},\ell} v$, where ℓ is an index variable representing the length of the path between u and v. Then, our logic allows to use linear constraints on the set of existential index variables and order constraints on the universal index variables (we can compare two universal variables or we can compare a universal variable to an expression built using existential variables).

Moreover, the data attached to the location variables in the formula can be constrained using formulas in the underlying (first-order) data logic.

The quantification over positions (location, array, and index variables) in CSL formulas is restricted according to an ordered partition on the vertex types. This partition decomposes the graph into classes and assigns to each vertex in the graph a level. (Notice, that we do not require that the induced graph by this partition be a tree.) Then, roughly speaking, by associating a level to positions and quantifiers over them, the quantification part of CSL formulas has the form $\exists_k^* \forall_k^* \exists_{k-1}^* \forall_{k-1}^* \cdots \exists_1^* \forall_1^*$. (A precise definition is given later in the paper.) Here it is important to notice that we allow some form of alternation between existential and universal quantifiers on positions, but this alternation is restricted according to the considered ordered partition of the structure. Allowing such quantifier alternation is important for the definition of (1) initial conditions, and of (2) precise invariants. Indeed, to define a (potentially infinite) family of possible initial structures, we need to require the existence of some links/elements. (For instance, that each element in some array points to a nonempty list with at least some number of elements carrying distinct/equal data.) Similarly, while in many cases invariants are universally quantified formulas, there are cases where the existence of some links/elements is important. For instance, a program which inserts an element in a list assuming that it contains at least some elements is prone to errors although it satisfies a weaker invariant where this assumption (on the minimal length) is not specified.

We prove that the satisfiability problem of CSL can be reduced to the satisfiability problem of its underlying data logic. The proof is based, roughly, on computing a finite number of minimal-size models for the given formula, and on interpreting universal quantifiers on the positions of these models. Defining such minimal models is nontrivial in presence of shape constraints (reachability predicates), of list and array length constraints, as well as of nesting of data structures.

Another important fact we prove is that CSL is effectively closed under the computation of post images (i.e., strongest post condition). We show how this result, together with the decidability of the satisfiability problem can be used in pre-post condition reasoning and invariant checking. In [8], we illustrate our framework on a nontrivial example of a program manipulating composite data structures. For lack of space, detailed proofs are omitted here and can be found in [8].

2 Modeling Programs with Dynamic Heap

2.1 Programs

We consider strongly typed, heap-manipulating programs. The types used in these programs are either basic or user defined types. Basic types (boolean, integers, etc.) are represented by an abstract domain/type $\mathbb{D}$ on which are defined a set $\mathbb{O}$ of operations and a set $\mathbb{P}$ of predicates. User defined types are either references to records or arrays of records, where a *record* is a finite set of typed fields. Programs manipulate the heap using memory allocation/deallocation statements (`new`/`free`), record field updates (`x->f:=...`), variable assignments (`x:=...`), and array cell updates (`a[i]:=...`). The only operations allowed on references are the field access (`x->f`) and the array access (`a[i]`); no arithmetics on references is allowed. The control is changed by sequential composition, conditionals, and "while" loops. A program configuration is given by the state of the heap and by the valuation of the program variables either to values in $\mathbb{D}$ or to addresses in the heap.

Example 1. The program in Figure 1 declares, using a C syntax, two record types defining an array of doubly linked lists. The record `a_ty` (lines 2–6) stores an integer (field `id`) and a reference to a record of type `dll_ty` (field `dll`). The record `dll_ty` (lines 1 and 7–11) defines doubly linked lists storing a boolean (field `flag`) and a reference to an `a_ty` record (field `root`). The variable `v` is declared (line 12) as an array of `a_ty` records. Figure 1 also gives a possible heap configuration for this program.

2.2 Heaps as Graphs

We model a program heap by a labeled oriented graph called *heap graph*. Vertices in this graph represent values of record types. Labeled edges represent the relations between

```
typedef struct _dll_ty dll_ty;
typedef struct _a_ty   a_ty;
struct _a_ty {
  int     id;
  dll_ty* dll;
};
struct _dll_ty {
  a_ty*   root;
  bool    flag;
  dll_ty* next, prev;
};
a_ty v[N];
```

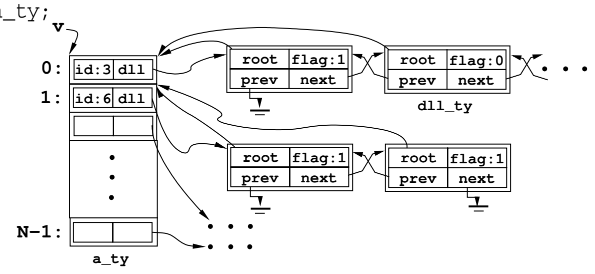


Fig. 1. A program declaring an array of doubly linked lists v and a heap for v

records defined by their fields: an edge $v \xrightarrow{f} v'$ models the field f in the record modeled by v which stores a reference to the record modeled by v'. To represent fields of basic type in records, we label the vertices of the graph by a valuation of these fields. Also, an array of records is modeled using an edge, labeled by a special field called *array field*, to relate two successive cells of the array.

The vertices of the heap graphs are typed using the ***program type system*** Σ which is defined by a set of types $\mathcal{T}$, a set of (data/pointer) field symbols, and a typing function τ mapping each symbol into a (functional) type defined over $\mathcal{T}$. The components of Σ are detailed in Table 1. The set $\mathcal{T}$ contains the basic type $\mathbb{D}$ as well as the user defined records. The field symbols belong to several classes detailed in the column *Components*, depending on their typing given in the column *Typing*. For example, τ associates with a data field g in a record $R \in \mathcal{RT}$ the type $R \to \mathbb{D}$ (i.e., g represents a mapping from R to $\mathbb{D}$). Generic notations for representatives of these sets are given in the column *Elements*. The last column illustrates each notion with elements from Example 1.

To represent "inverse pointers" we introduce a new field symbol $\overline{f}$ for every field $f \in \mathcal{PF} \setminus \mathcal{AF}$, and we assume that $\overline{\overline{f}} = f$. We extend the "inverse" notation to set of fields, i.e., $\overline{F} = \{\overline{f} | f \in F\}$ for any set $F \subseteq \mathcal{PF} \setminus \mathcal{AF}$. We denote by $\mathcal{PF}^* = \mathcal{PF} \cup \overline{\mathcal{PF} \setminus \mathcal{AF}}$.

Table 1. Type system associated with programs

Σ	*Components*	*Elements*	*Typing*	*... in Example 1*
Types	$\mathbb{D}$ basic domain	d		`int`
	$\mathcal{RT}$ record types	$R, R', \ldots$	$\mathbb{D} \notin \mathcal{RT}$	`a_ty`, `dll_ty`
Field Symbols	$\mathcal{DF}$ data fields	$g, g^1, g^2, \ldots$	$\tau(g) = R \to \mathbb{D}$	$\tau($`id`$) =$ `a_ty` $\to$ `int` $\tau($`flag`$) =$ `dll_ty` $\to$ `int`
	$\mathcal{PF}$ pointer fields $\mathcal{DF} \cap \mathcal{PF} = \emptyset$	$a, f, h, \ldots$	$\tau(h) = R \to R'$	$\tau($`dll`$) =$ `a_ty` $\to$ `dll_ty`, $\tau($`root`$) =$ `dll_ty` $\to$ `a_ty`
	$\mathcal{PF}_r$ recursive fields $\mathcal{PF}_r \subseteq \mathcal{PF}$	$f, f^1, f^2, \ldots$	$\tau(f) = R \to R$	$\tau($`next`$) =$ `dll_ty` $\to$ `dll_ty`, $\tau($`prev`$) =$ `dll_ty` $\to$ `dll_ty`
	$\mathcal{AF}$ array fields $\mathcal{AF} \subseteq \mathcal{PF}_r$	$a, \ldots$	$\tau(a) = R \to R$	$\tau(a) =$ `a_ty` $\to$ `a_ty`

Then, a ***heap graph*** over the type system Σ is an oriented labeled graph $G = (V, E, Lab, L, P, D)$, where
– V is a finite set of vertices,
– $E \subseteq V \times V$ is a finite set of directed edges,
– Lab is a finite set of vertex labels,
– $L : V \to 2^{Lab}$ is a labeling function for vertices with sets of vertex labels,
– $P : E \to 2^{\mathcal{PF}^*}$ is a labeling function for edges with sets of pointer fields, and
– $D : V \to [\mathcal{DF} \rightharpoonup \mathbb{D}]$ is a labeling function for vertices with a data field valuation.

We extend the typing function τ of Σ to vertices in G, i.e., we consider that τ associates a type $\tau(v) \in \mathcal{RT}$ with each vertex $v \in V$. A heap graph is ***well typed*** in Σ, if the labeling functions P and D are consistent with the typing of the vertices and the field symbols by τ. A heap graph G is ***well formed*** if it satisfies the following constraints:
– *determinism*: For all $v_1, v_2, v_3 \in V$, we have that (1) if $\{(v_1,v_2),(v_1,v_3)\} \subseteq E$ and $P(v_1,v_2) \cap P(v_1,v_3) \cap \mathcal{PF} \neq \emptyset$ then $v_2 = v_3$, (2) if $\{(v_1,v_2),(v_3,v_1)\} \subseteq E$ and $P(v_1,v_2) \cap \overline{P(v_3,v_1) \setminus \mathcal{AF}} \cap \mathcal{PF} \neq \emptyset$ then $v_2 = v_3$, (3) if $\{(v_2,v_1),(v_3,v_1)\} \subseteq E$ and $P(v_2,v_1) \cap P(v_3,v_1) \cap \overline{\mathcal{PF} \setminus \mathcal{AF}} \neq \emptyset$ then $v_2 = v_3$;
– *array well-formedness*: array fields create acyclic distinct paths in G.

From now on, we consider that heap graphs are well typed and well formed.

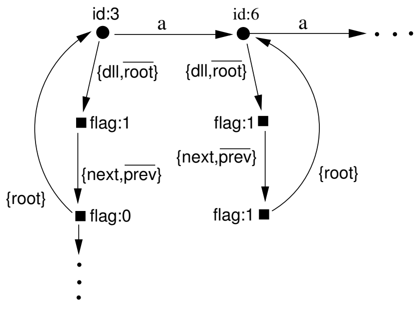

id:3
a
id:6
a
{dll,$\overline{\text{root}}$}
{dll,$\overline{\text{root}}$}
flag:1
flag:1
{next,$\overline{\text{prev}}$}
{next,$\overline{\text{prev}}$}
{root}
{root}
flag:0
flag:1

Fig. 2. A heap graph for Ex. 1

Example 2. The heap graph model corresponding to Example 1 is given in Figure 2. We use different shapes for the vertices in order to point out their different types: circles for vertices representing `a_ty` records and boxes for vertices representing `dll_ty` records. Vertices are labeled only by the values of the data fields since we consider $Lab = \emptyset$. The edges labeled by the array field a represent the relation between successive cells in the array `v`. The labels $\overline{\texttt{prev}}$ and $\overline{\texttt{root}}$ represent the inverse of the relations defined by `prev` and `root` respectively.

2.3 Reasoning about Programs

Our aim is to define a logic-based framework for checking assertions on program configurations at given control locations. For that, a basic operation is the computation of post images of configurations for given program statements. Given a program configuration C (i.e., a heap graph and a valuation of the program variables), and given a program statement S, let $\mathsf{post}(S,C)$ denote the configuration obtained by executing S on C. This definition can be generalized as usual to sets of configurations.

Then, we are interested in *inductive invariant checking* and *pre/post-condition checking*. The first problem consists in, given a set of initial configurations $Init$, and a set of configurations Inv, deciding whether (1) $Init \subseteq Inv$, and (2) for every statement S of the program, $\mathsf{post}(S, Inv) \subseteq Inv$. Pre/post-condition checking consists in, given two sets of configurations $PreCond$ and $PostCond$, and given a statement S, checking whether

$\mathsf{post}(S, PreCond) \subseteq PostCond$. Therefore, the issue we are addressing is to provide a logic which allows to:

– carry out automatically the kind of reasoning mentionned above. For that, it must have closure properties under (some) boolean operations and under post-image computation, and it must also have a decidable satisfiability/validity problem.
– express relevant assertions about heap manipulating programs. These properties may refer to several aspects such as the structure of the heap graph, the data attached to the elements of the heap, and the sizes of some parts of the heap.

Example 3. Consider the heap of Example 1. Assertions one can be interested to express on this heap are:

Structure: "the array `v` contains in each cell a reference to an acyclic doubly linked list" (`doubly-ll`), "each cell in the doubly linked lists stores in the field `root` a reference to the entry of the array referencing the list" (`root-fld`), or "doubly linked lists are cyclic" (`doubly-cll`).

Data: "the array `v` is sorted w.r.t. the values of the field `id`" (`sorted-id`), or "it exists a list with all fields `flag` set to 1" (`flag-1`).

Sizes: "each doubly-linked list has at least two elements" (`dll-len2`), or "the array `v` is sorted in decreasing order of lengths of lists stored" (`dll-len`).

3 Generalized Composite Structures Logic

We introduce hereafter a logic on program heaps called *Generalized Composite Structures Logic* (gCSL). The logic we are seeking for, i.e., which satisfies the expressiveness, closure, and decidability properties mentioned in the end of the last section, will be defined as a fragment of gCSL in the next section.

Given a type system $\Sigma = (\mathbb{D}, \mathcal{RT}, \mathcal{DF}, \mathcal{PF}, \mathcal{PF}_r, \mathcal{AF}, \tau)$, a finite set of (heap location) labels *Lab*, and a first-order logic $\mathsf{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$ on data, the set of formulas of gCSL is defined in Figure 3. As usual, conjunction ($\wedge$), implication ($\Longrightarrow$), and universal quantification ($\forall$) can be defined in terms of $\neg$, $\vee$, and $\exists$. Intuitively, an atomic formula $v \xrightarrow{A,B,ind} v'$ is a *reachability predicate* expressing the fact that there is a nonempty path in the heap graph relating the locations designated by v and v' such that (1) all its vertices are distinct, except maybe for the extremal points, (2) all its edges are labeled by a set which includes A, (3) all its vertices are not in B, and (4) its length (i.e., the number of its edges) is *ind* which is either an index variable, or a constant. (The formula $v \xrightarrow{A,B} v'$ is similar but it does not allow to refer to the length of the path.) Index variables can also appear in terms of the form $a[ind]$ designating locations in arrays (i.e., the location reachable from the initial position of the array after *ind* array edges). Index variables can be used in linear arithmetical constraints. The logic gCSL allows quantification over all kinds of variables. We assume w.l.o.g. that each variable is quantified at most once.

Using formulas of the form $v = t$, gCSL allows terms of the form $a[Exp]$: a formula ϕ containing $a[Exp]$ can be seen as the formula $Exp - i = 0 \wedge \phi(a[i]/a[Exp])$ where i is a fresh index variable. Similarly, gCSL allows data terms of the form $g(a[Exp])$ and $g(f(v))$. Also, formulas of the form $a[i] \xrightarrow{A,B,ind} b[j]$ are abbreviations of $v_1 = a[i] \wedge v_2 =$

$b[j] \wedge v_1 \xrightarrow{A,B,ind} v_2$ for some fresh location variables v_1 and v_2. Finally, we allow writing $Exp < Exp'$ instead of $Exp - Exp' < 0$.

We extend the typing function τ of Σ to the variables of the logic, and then to terms, and we assume that formulas are type consistent. For instance, τ associates records in $\mathcal{RT}$ with location variables, and then (1) for every location term $h(v)$, if $\tau(h) = R \to R'$ for some $R, R' \in \mathcal{RT}$, then we must have $\tau(v) = R$ and $\tau(h(v)) = R'$, and (2) for every formula of the form $v \xrightarrow{A,B,ind} v'$, we require that $\tau(v) = \tau(v') = \tau(z)$ for every $z \in B$, and that for every $f \in A$, $\tau(f) = \tau(v) \to \tau(v)$.

$v, v' \in Loc$	location variable	$a \in Arr$	array variable
$d \in Data$	data variable	$i \in Ind$	index variable
$\ell \in Lab$	label	$ind \in Ind \cup \mathbb{N}$	index
$p \in \mathbb{P}$	predicate over $\mathbb{D}$	$ct \in \mathbb{N}$	integer constant
$o \in \mathbb{O}$	operator over $\mathbb{D}$	$h \in \mathcal{PF} \setminus \mathcal{PF}_r$,	$g \in \mathcal{DF}$

Location terms: $t ::= v \mid a[ind] \mid h(v)$

Data terms: $dt ::= d \mid o(dt, \ldots, dt) \mid g(v)$

Index expressions: $Exp ::= i \mid ct \mid Exp + Exp \mid Exp - Exp$

Formulas:
$$\begin{aligned}\varphi ::= {} & \ell(v) \mid v = t \mid \\ & v \xrightarrow{A,B,ind} v' \mid v \xrightarrow{A,B} v' \mid \\ & p(dt, \ldots, dt) \mid Exp < ct \mid Exp = ct \mid \\ & \exists v.\ \varphi \mid \exists a.\ \varphi \mid \exists d.\ \varphi \mid \exists i.\ \varphi \mid \neg\varphi \mid \varphi \vee \varphi, \\ & \text{where } A \neq \emptyset, A \subseteq (\mathcal{PF}_r \setminus \mathcal{AF}) \cup \overline{\mathcal{PF}_r \setminus \mathcal{AF}}, \text{ and } B \subseteq Loc.\end{aligned}$$

Fig. 3. Syntax of gCSL

A gCSL formula is interpreted over a heap graph $G = (V, E, Lab, L, P, D)$ w.r.t type preserving valuations of free variables. Let $\mu : Loc \rightharpoonup V$, $\nu : Ind \rightharpoonup \mathbb{N}$, and $\delta : Data \rightharpoonup \mathbb{D}$ be a valuation of location, index, and data variables, respectively. Array variables are mapped to *array vertices*. These vertices have no incoming edge labeled by an array field, i.e., they are first elements of (possibly singleton) arrays in G. For any array vertex w, let $Dom(w)$ (the domain of w) be the length of the maximal path starting in w where all the edges are labeled by some $a \in \mathcal{AF}$ (this array field should be unique by the well-formedness of G). Let AV denote the set of array vertices in V and $\theta : Arr \rightharpoonup AV$ be the valuation of array variables.

The interpretation of a location term is either a vertex in G or $\perp$. It is defined as follows: (1) $\langle\langle v \rangle\rangle_{G,\mu,\theta,\nu} = \mu(v)$; (2) for any $f \in \mathcal{PF} \setminus \mathcal{PF}_r$, $\langle\langle f(v) \rangle\rangle_{G,\mu,\theta,\nu} = w$, where $f \in P(\mu(v), w)$ or $\overline{f} \in P(w, \mu(v))$, if such a vertex w exists, or $\perp$ otherwise; (3) for any $a \in Arr$ and $i \in Ind$, if $\nu(i) > Dom(\theta(a))$ then $\langle\langle a[i] \rangle\rangle_{G,\mu,\theta,\nu} = \perp$, otherwise, $\langle\langle a[i] \rangle\rangle_{G,\mu,\theta,\nu}$ is the vertex v reachable from $\theta(a)$ by a path of $\nu(i)$ edges labeled by an array field. Data terms are interpreted in $\mathbb{D}$ as follows: (1) $\langle\langle d \rangle\rangle_{G,\mu,\delta} = \delta(d)$; (2) for any $g \in \mathcal{DF}$, $\langle\langle g(v) \rangle\rangle_{G,\mu,\delta} = D(\mu(v))(g)$; (3) for any $o \in \mathbb{O}$, $\langle\langle o(dt_1, \ldots, dt_n) \rangle\rangle_{G,\mu,\delta} = o(\langle\langle dt_1 \rangle\rangle_{G,\mu,\delta}, \ldots, \langle\langle dt_n \rangle\rangle_{G,\mu,\delta})$. The interpretation $\langle\langle Exp \rangle\rangle_\nu$ of an index expression as an integer is defined as usual.

Then, the interpretation $[\![\varphi]\!]_{G,\mu,\theta,\nu,\delta}$ of a gCSL formula φ is a value in $\{0,1,\bot\}$. Boolean operators are interpreted as usual on $\{0,1\}$, and the boolean composition of $\bot$ with any value leads to $\bot$. Then, for every formula φ, if there exists a sub-term t of φ such that $\langle\langle t\rangle\rangle_{G,\mu,\theta,\nu} = \bot$ then $[\![\varphi]\!]_{G,\mu,\theta,\nu,\delta} = \bot$. Otherwise, $[\![\varphi]\!]_{G,\mu,\theta,\nu,\delta} = 1$ iff $G \models_{\mu,\theta,\nu,\delta} \varphi$ and $[\![\varphi]\!]_{G,\mu,\theta,\nu,\delta} = 0$ iff $G \not\models_{\mu,\theta,\nu,\delta} \varphi$, where the relation $\models$ is defined as follows (only interesting cases are considered):

$G \models_{\mu,\theta,\nu,\delta} \ell(v)$ iff $\ell \in L(\mu(v))$

$G \models_{\mu,\theta,\nu,\delta} v = t$ iff $\mu(v) = \langle\langle t\rangle\rangle_{G,\mu,\theta,\nu}$

$G \models_{\mu,\theta,\nu,\delta} v \xrightarrow{A,B,ind} v'$ iff there exists a path $\mu(v) = w_0, w_1, \ldots, w_m = \mu(v')$ s.t. $m = \nu(ind) \geq 1$, $w_j \neq w_{j'}$ if $j \neq j'$ and $(j,j') \notin \{(0,m),(m,0)\}$, and for any $j \geq 1$, $A \subseteq P(w_j, w_{j+1})$ or $\overline{A} \subseteq P(w_{j+1}, w_j)$ and for any $x \in B$, $w_j \neq \mu(x)$,

$G \models_{\mu,\theta,\nu,\delta} Exp < ct$ iff $\langle\langle Exp\rangle\rangle_\nu < ct$,

$G \models_{\mu,\theta,\nu,\delta} p(dt_1,..,dt_n)$ iff $p(\langle\langle dt_1\rangle\rangle_{G,\mu,\delta}, \ldots, \langle\langle dt_n\rangle\rangle_{G,\mu,\delta})$

$G \models_{\mu,\theta,\nu,\delta} \exists v.\ \varphi$ iff there exists $w \in V$ with $\tau(w) = \tau(v)$ such that $G \models_{\mu[v\leftarrow w],\theta,\nu,\delta} \varphi$

$G \models_{\mu,\theta,\nu,\delta} \exists i.\ \varphi$ iff there exists $m \in \mathbb{N}$ such that $G \models_{\mu,\theta,\nu[i\leftarrow m],\delta} \varphi$.

We omit the subscripts of $[\![\cdot]\!]$ and $\models$ when they are clear from the context.

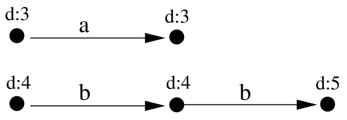


Fig. 4. A heap graph with arrays

Remark: Recall that given a heap graph G and some valuations of the variables, the value of a term $a[i]$ is well-defined ($\neq \bot$) if the value of i is in the domain of the array associated with a. Then, the interpretation of a formula $\exists i.\ \varphi$ (resp. $\forall i.\ \varphi$) is 1 if the interpretation of φ is 1 for some value (resp. for all values) of i in the domains of all arrays a such that $a[i]$ occurs in φ. For example, on the heap graph of Figure 4, $[\![\exists i.\ (d(a[i]) = 3 \wedge d(b[i]) = 4)]\!] = 1$, $[\![\exists i.\ (d(a[i]) = 3 \wedge d(b[i]) = 5)]\!] = 0$, and $[\![\exists i.\ (d(a[2]) = 3 \vee d(b[i]) = 5)]\!] = \bot$. Notice also that the formula $[\![\exists i.\ (d(a[i]) = 4 \vee d(b[i]) = 5)]\!] = 0$ whereas $[\![(\exists i.\ d(a[i]) = 4) \vee (\exists i.\ d(b[i]) = 5)]\!] = 1$. This shows that $\vee$ (resp. $\wedge$) does not distribute, in general, w.r.t. $\exists$ (resp. $\forall$). However, these distributivity properties hold in the fragment of gCSL without arrays.

Example 4. Consider the heap graph of Figure 2 and its properties given in Example 3. These properties can be expressed in gCSL (assuming as data logic the first order linear arithmetics over integers, for instance) as follows:

Structure

$$\texttt{doubly-ll}(v) \equiv \forall i.\ \exists dl_i.\ \Big(dl_i = \texttt{dll}(v[i]) \wedge$$
$$\big(dl_i = \texttt{null} \vee (dl_i \xrightarrow{\{\texttt{next}\},\emptyset,1} \texttt{null} \wedge dl_i \xrightarrow{\{\texttt{prev}\},\emptyset,1} \texttt{null}) \vee$$
$$\exists dl.\ (dl_i \xrightarrow{\{\texttt{next},\overline{\texttt{prev}}\},\emptyset} dl \wedge dl \xrightarrow{\{\texttt{next}\},\emptyset,1} \texttt{null} \wedge dl_i \xrightarrow{\{\texttt{prev}\},\emptyset,1} \texttt{null})\Big)$$

$$\texttt{root-fld}(v) \equiv \forall i.\ \forall v_i, dl.\ (v[i] = v_i \wedge \texttt{dll}(v_i) \xrightarrow{\{\texttt{next}\},\emptyset} dl) \Longrightarrow \texttt{root}(dl) = v_i$$

$$\texttt{doubly-cll}(v) \equiv \forall i.\ \forall dl_i.\ (dl_i = \texttt{dll}(v[i]) \Longrightarrow dl_i \xrightarrow{\{\texttt{next},\overline{\texttt{prev}}\},\emptyset} dl_i)$$

where we assume that null is a free location variable of type dll_ty, and that $\texttt{null} \xrightarrow{\{\texttt{next},\overline{\texttt{prev}}\},\emptyset,1} \texttt{null}$ holds.

Data

$$\texttt{sorted-id}(v) \equiv \forall i,j.\ \big(i < j \Longrightarrow \texttt{id}(v[i]) < \texttt{id}(v[j])\big)$$

$$\texttt{flag-1}(v) \equiv \exists i.\ \exists dl_i.\ \big(dl_i = \texttt{dll}(v[i]) \wedge \forall y.\ (dl_i \xrightarrow{\{\texttt{next}\},\emptyset} y \Longrightarrow \texttt{flag}(y) = 1)\big)$$

Sizes

$$\texttt{dll-len2}(v) \equiv \forall i.\ \exists dl_i, dl'_i.\ (dl_i = \texttt{dll}(v[i]) \wedge dl_i \neq dl'_i \wedge dl'_i \neq \texttt{null} \wedge dl_i \xrightarrow{\{\texttt{next},\overline{\texttt{prev}}\},\{\texttt{null}\}} dl'_i)$$

$$\texttt{dll-len}(v) \equiv \forall j,j'.\ \Big(j < j' \Longrightarrow \exists dl_j, dl'_j, l, l'.\ \big(dl_j = \texttt{dll}(v[j]) \wedge dl'_j = \texttt{dll}(v[j']) \wedge dl_j \neq \texttt{null} \wedge dl_j \xrightarrow{\{\texttt{next}\},\emptyset,l} \texttt{null} \wedge dl'_j \xrightarrow{\{\texttt{next}\},\emptyset,l'} \texttt{null} \wedge l' \leq l\big)\Big)$$

By existing results shown, e.g., in [11,9], it can be easily deduced that the satisfiability problem of gCSL is undecidable already for the fragment $\forall^*\exists^*$ even if the considered heap structures are linear (i.e., one dimensional arrays or singly-linked lists) and the data logic is simply $(\mathbb{N}, =)$ (i.e., an enumerable data domain with only equality). When the data domain is finite, the satisfiability problem of gCSL is undecidable in general since it subsumes the first-order logic on graphs with reachability [6].

4 The Logic CSL

We define hereafter the *Composite Structures Logic* (CSL) as a fragment of gCSL. We show in the end of the section that CSL is closed under post-image computation for all statements in the class of programs considered in Section 2.1. We prove in the next section that CSL has a decidable satisfiability problem.

To get the decidability result, we must restrict the use of the formulas involving quantifier alternation of the form $\forall^*\exists^*$ (see the end of the last section). On the other hand, it is important to allow some forms of these formulas since they may correspond to natural assertions (such as doubly-ll, dll-len, and dll-len2 in Example 4). To introduce the restriction on these formulas, let us consider a gCSL formula of the form $\exists^*\forall^*\exists^*\forall^* \ldots \exists^*\forall^*.\ \phi$ where ϕ is quantifier free. Then, we impose basically that if a variable v is existentially quantified within the scope of a universal quantification of some variable v', then the type of v' must be different from the one of v and from the types of all the variables which are (universally or existentially) quantified after (under the scope of) v. This restriction is defined through the introduction of a notion of ordered partition of the set of types.

Ordered partitions of types: Let N be a natural number such that $1 \leq N \leq |\mathcal{RT}|$. Then, an *ordered partition over the set of types* $\mathcal{RT}$ is a mapping $\sigma : \mathcal{RT} \to \{1,\ldots,N\}$. A type $R \in \mathcal{RT}$ is of *level* k, for some $k \in \{1,\ldots,N\}$, iff $\sigma(R) = k$.

We extend an ordered partition σ to $\mathbb{D}$ and we assume that it associates level 0 to all elements of $\mathbb{D}$. Ordered partitions of the set of types induce ordered partitions on heap graphs: the notion of level is transfered from types to vertices in heap graphs. These partitions correspond to natural decompositions of heap structures into sub-structures according to the type definitions in the program. For example, in the data structure of Example 1, we may consider two levels: the first level contains the doubly-linked lists (i.e., $\sigma(\texttt{dll\_ty}) = 1$) and the second level contains the array (i.e., $\sigma(\texttt{a\_ty}) = 2$). Notice that the quotient graph induced by this partition is cyclic: the edges labeled by `dll` go from level 2 to level 1, and edges labeled by `root` go from level 1 to level 2.

k-stratified formulas: Given an ordered partition σ over $\mathcal{RT}$, we consider that $\sigma(v) = \sigma(\tau(v))$ for any location or array variable v. For an index variable i, we let the level $\sigma(i)$ to be fixed arbitrarily. Then, for each level k, with $1 \leq k \leq N$, let Q_k be a sequence of quantifiers over variables of level k defined as follows:

$$Q_k = \exists \mathbf{a}^k \; \exists \mathbf{x}^{\leq k} \; \exists \mathbf{i}^k \; \exists \mathbf{d}^k \; \forall \mathbf{b}^k \; \forall \mathbf{y}^k \; \forall \mathbf{j}^k$$

where $\mathbf{a}^k$ and $\mathbf{b}^k$ are sets of array variables of level k, $\mathbf{x}^{\leq k}$ is a set of location variables of level less than or equal to k, $\mathbf{y}^k$ is a set of location variables of level k, $\mathbf{i}^k$ and $\mathbf{j}^k$ are sets of index variables of level k, and $\mathbf{d}^k$ is a set of data variables. (There is no level restriction on data variables.)

A gCSL formula is k-stratified if it is of the form $Q_k \; Q_{k-1} \; \ldots Q_1 \; Q. \; \phi$, where Q is a set of quantifiers over data variables and ϕ is a quantifier-free formula.

The fragment CSL$_k$: To define CSL formulas, we consider in addition to stratification some restrictions on the occurrences of universally quantified location, array, and index variables. These restrictions are necessary for our proof technique for showing the decidability of the satisfiability problem, which is based on establishing a small model property. We define the fragment CSL$_k$ to be the smallest set of formulas which is closed under disjunction and conjunction, and which contains the set of all k-stratified formulas satisfying the following constraints:

UNIVIDX: two universally quantified index variables j and j' can not be used in any linear expression but only in atomic formulas $j - j' < 0$, and $j - j' = 0$;

REACH1: in the reachability sub-formula $v \xrightarrow{A,B,ind} v'$ or $v \xrightarrow{A,B} v'$, the set of forbidden locations B contains only free or existentially quantified location variables;

REACH2: in the reachability sub-formula $v \xrightarrow{A,B,ind} v'$, the location variables v and v', and the index variable ind, are free or existentially quantified;

LEV: the constraints on lengths of lists and array indexes must involve only one level, that is: (1) for any atomic formula $v \xrightarrow{A,B,i} v'$, $\sigma(i) = \sigma(v)$, (2) for any term $a[i]$, $\sigma(i) = \sigma(a)$, and (3) for any atomic formulas $Exp < ct$ or $Exp = ct$, all index variables in Exp have the same level.

The set of formulas of CSL is the union of the fragments CSL$_k$ for all $k \in \{1, \ldots, N\}$. Despite the syntactical restrictions in CSL, the logic is still quite powerful. It extends several existing decidable logics (see Section 6). Notice for instance that all formulas given in Example 4 are in CSL (for the ordered partition defined above).

Closure under post image computation: We call *basic statement* any program statement of the class of programs defined in Section 2.1 that is different from a "while" loop. Then, we can prove the following fact. (The proof is technical and it is omitted here)

Theorem 1. *For any basic statement S and any CSL formula φ,* $\mathsf{post}(S, \varphi)$ *is CSL-definable and it can be computed in linear time.*

The theorem above is important for carrying out pre/post-condition reasoning (assuming we have an annotated program with assertions and loop invariants) and for inductive invariance checking.

A fragment for expressing invariants: The logic CSL is closed under disjunction and conjunction, but not under negation. For checking inductive invariance (in particular that $\mathsf{post}(S, Inv) \subseteq Inv$ which is equivalent to $\mathsf{post}(S, Inv) \cap \overline{Inv} = \emptyset$), we need a fragment which is closed under negation. We introduce hereafter such a fragment.

For each level k, let ICSL_k be the smallest fragment of CSL which is closed under all boolean operations and which contains formulas of the form:

$$S_k\ S_{k-1}\ \ldots S_1\ S.\ \phi$$

where $S_k \in \{\exists \mathbf{a}^k \exists \mathbf{i}^k \exists \mathbf{x}^k, \forall \mathbf{b}^k \forall \mathbf{y}^k \forall \mathbf{j}^k\}$, S is a set of quantifiers over data variables and ϕ is a quantifier-free formula in CSL such that:

BoundIdx: two quantified index variables i and i' can not be used in any linear expression but only in atomic formulas $i - i' < 0$, and $i - i' = 0$;

Reach3: in any reachability sub-formula $v \xrightarrow{A,B,ind} v'$, v, v' and ind are free variables.

Then, let $\mathsf{ICSL} = \bigcup_{1 \leq k \leq N} \mathsf{ICSL}_k$. Notice that ICSL allows alternation between quantifiers provided that they concern different levels.

Examples of ICSL formulas are those given in Example 4, except `doubly-ll` and `dll-len`. Actually, it is possible to give an equivalent formula to `doubly-ll` which is in ICSL (see [8]). The CSL formula `dll-len` can be used to constrain the initial condition of the program.

5 Deciding the Satisfiability Problem for CSL

We prove that CSL has a decidable satisfiability problem. For the sake of readability, we present the proof of this result in two steps. First, we prove it for the fragment CSL_1 (i.e., all location/array variables are in the same class of the considered ordered partition of the types), and then we show the extension to the full CSL.

5.1 The Case of CSL_1

This subsection presents a sketch of the proof of the following fact:

Theorem 2. *The satisfiability problem for closed CSL_1 formulas is decidable provided that the satisfiability problem of $\mathsf{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$ is decidable.*

Let φ be a closed CSL_1 formula. We reduce its satisfiability to the satisfiability of a $\mathsf{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$ formula. The reduction is in three steps: (1) we prove that if φ has a model (i.e., a heap graph), then it must have a model of bounded size where the bound can be effectively computed from the syntax of φ, (2) for any given finite model G of φ, we construct an equi-satisfiable formula without universal quantification over location, array, and index variables, and finally (3) we construct an equi-satisfiable formula without existential quantification over location, array, and index variables. The so obtained formula is in $\mathsf{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$.

The last two steps of the reduction do not present any difficulties. By fixing a finite model, the interpretation domain of the universally quantified variables is also fixed, and therefore universal quantification can be eliminated. Then, existential quantifiers over location, index, and array variables can be eliminated by replacing in the formula the terms of the form $g(v)$ and $g(a[i])$ by fresh data variables $d_{g,v}$ and $d_{g,a,i}$ respectively, and by quantifying existentially these new data variables.

We now describe the first (and main) step of the reduction. Let us introduce the problem on an example. Consider the formula (defined over natural numbers):

$$\psi_1 = \exists x,q,z.\ \Big(q \neq z \wedge x \xrightarrow{\{f\},\emptyset} q \wedge x \xrightarrow{\{f\},\emptyset} z\ \wedge\ g(x) = 0 \wedge g(q) = 2\ \wedge \\ \forall y,y'.\ \big(x \xrightarrow{\{f\},\emptyset} y \xrightarrow{\{f\},\emptyset} y' \wedge y \neq y' \Rightarrow g(y) < g(y')\big)\Big) \tag{1}$$

which says that there is an f-path from x to q and an f-path from x to z, the data attached with x (resp. q) is 0 (resp. 2), and the data are ordered along f-paths. It can be checked that ψ_1 is satisfiable and has two minimal models of size three, either the graph $x \xrightarrow{f} q \xrightarrow{f} z$, or the graph $x \xrightarrow{f} z \xrightarrow{f} q$ (here we identify vertices with the variables they represent). Notice that the three vertices corresponding to x, q, and z must be part of a same f-path since heap graphs are deterministic by definition. So, in this case the size of a minimal model is the number of existentially quantified variables. Consider now a new formula obtained from ψ_1 by constraining the lengths of the paths relating x to q and to z.

$$\psi_2 = \exists x,q,z.\ \exists l_1,l_2.\ \Big(q \neq z \wedge x \xrightarrow{\{f\},\emptyset,l_1} q \wedge x \xrightarrow{\{f\},\emptyset,l_2} z\ \wedge\ g(x) = 0 \wedge g(q) = 2\ \wedge \\ l_1 + l_2 \geq 8\ \wedge\ \forall y,y'.\ \big(x \xrightarrow{\{f\},\emptyset} y \xrightarrow{\{f\},\emptyset} y' \wedge y \neq y' \Rightarrow g(y) < g(y')\big)\Big) \tag{2}$$

Again, since x, q, and z must be part of a same f-path, there are two different possible "heap graph templates" to consider:

$$x \xrightarrow{f,l_1} q \xrightarrow{f,l} z \tag{3}$$
$$x \xrightarrow{f,l_2} z \xrightarrow{f,l} q \tag{4}$$

where each edge represents an f-path (of some length denoted l_1, l_2, or l), and we need to determine the minimal lengths of each of these paths, taking into account the constraints imposed by the formula ψ_2. First, it can be seen that with the first (resp. second) template above there is an associated constraint on the lengths which is $l_1 + l = l_2 \wedge l_1 + l_2 \geq 8$ (resp. $l_2 + l = l_1 \wedge l_1 + l_2 \geq 8$). For the first (resp. second) template, the set of minimal solutions (w.r.t. the usual ordering on vectors of natural numbers) for the constraints on l_1, l_2, and l is $\{(3,5,2),(2,6,4),(1,7,6)\}$ (resp. $\{(5,3,2),(6,2,4),(7,1,6)\}$). However, not all of these minimal solutions lead to minimal models of the formula. This is due to the fact that data constraints can also impose constraints on the path lengths. Here for instance, it is clear that l_1 must be at most 2. This means that if the formula ψ_2 has a model, a minimal model of it should correspond to the first template with either $l_1 = 2$, $l_2 = 6$, and $l = 4$, or $l_1 = 1$, $l_2 = 7$, and $l = 6$. In the first case, the minimal model is obtained from the template by inserting new vertices, one vertex between x and q, and five vertices between q and z.

In fact, our proof shows that we can build from ψ_2 an equivalent formula ψ_2' such that, if ψ_2 has a model, then it must have a model of size less or equal than the number of existentially quantified variables in ψ_2'. The construction of ψ_2' is based on the consideration of all possible minimal models built out of the templates as shown above. We give now the construction in more details. We proceed again in several steps.

Heap graph templates: A template defines a set of heap graphs. It is given by (1) a graph where vertices correspond to some distinguished vertices in the defined heap graphs, and edges correspond to paths between these vertices, and (2) a constraint relating the lengths of these paths. Given a CSL_1 formula, templates are built by considering as distinguished vertices positions in heap graphs corresponding to existentially quantified location, array, or index variables (i.e., positions which are imposed by the formula), and by considering the different ways these positions may or may not be equal, as well as the different ways these positions should or should not share the same paths (due to the determinism assumption on heap graphs).

Formally, let $\varphi = \exists\mathbf{a}\ \exists\mathbf{i}\ \exists\mathbf{x}\ \exists\mathbf{d}\ \forall\mathbf{b}\ \forall\mathbf{j}\ \forall\mathbf{y}\ \{\exists d, \forall d\}^*.\ \phi$ be a closed CSL_1 formula (ϕ is quantifier-free). We denote by $Pos(\varphi)$ the set of existential positions used in ϕ, i.e., $Pos(\varphi) = \mathbf{x} \cup \{a[i] \mid a \in \mathbf{a}, i \in \mathbb{N} \cup \mathbf{i},\ a[i] \text{ is a term in } \phi\} \cup \{a[i_{Exp}] | Exp \not\in \mathbf{i},\ Exp \text{ is a maximal index expression over } \mathbf{i} \text{ in } \phi\}$. Moreover, we suppose that $Pos(\varphi)$ contains for each array a in $\mathbf{a}$ the position $a[0]$. We denote by $\mathbf{i}_{Pos(\varphi)}$ the set of fresh index variables i_{Exp} associated with index expressions Exp of ϕ in the definition of $Pos(\varphi)$. Then, a *template* for φ is a labeled directed graph $T = (V, E, L_V, L_E, C)$ where

– V is a set of vertices, E is a multi-set of edges over V,
– $L_V : V \longrightarrow 2^{Pos(\varphi)} \times 2^{Lab}$ is a labeling of vertices,
– $L_E : E \longrightarrow 2^{\mathcal{PF}^*} \times \mathbf{i}_T$, where $\mathbf{i}_T \subseteq Ind$, is a labeling of edges such that (1) T is deterministic, i.e., $L_E(e_1)|_1 \cap L_E(e_2)|_1 = \emptyset$, for any edges e_1 and e_2 starting in the same vertex, and (2) $L_E(e_1)|_2 \neq L_E(e_2)|_2$ for any two edges $e_1 \neq e_2$,
– C is a conjunction of quantifier free constraints over variables in $\mathbf{i} \cup \mathbf{i}_T$ corresponding to the constraints on $\mathbf{i}$ in φ.

The set of templates built as above is denoted by $\mathcal{T}_{\varphi,\exists}$. We associate with each template a CSL_1 formula which characterizes the set of heap graphs defined by the template. Given $T = (V, E, L_V, L_E, C) \in \mathcal{T}_{\varphi,\exists}$, the *characteristic formula* of T is the quantifier free formula in CSL_1 with free variables in $\mathbf{a} \cup \mathbf{x} \cup \mathbf{i} \cup \mathbf{i}_T \cup \mathbf{i}_{Pos(\varphi)}$ defined by:

$$\rho_T = \bigwedge_{\substack{e=(v_1,v_2)\in E, L_E(e)=(A,i'),\\ x\in L_V(v_1)|_1,\, y\in L_V(v_2)|_1}} x \xrightarrow{A,\emptyset,i'} y \ \wedge \bigwedge_{\substack{v\in V\\ a,b\in L_V(v)|_1}} a = b \ \wedge \bigwedge_{i_{Exp}\in Pos(\varphi)} i_{Exp} = Exp \wedge C$$

Proposition 1. *For every model G of φ there exists a template $T \in \mathcal{T}_{\varphi,\exists}$ and valuations μ, θ, ν, δ for the free variables in ρ_T s.t. $G \models_{\mu,\theta,\nu,\delta} \rho_T$.*

This proposition implies obviously that if $\mathcal{T}_{\varphi,\exists} = \emptyset$ then φ is unsatisfiable. Notice that the converse is false. Then, we can prove the following fact:

Lemma 1. *The formula φ is equivalent to*

$$\varphi_1 = \bigvee_{T\in\mathcal{T}_{\varphi,\exists}} \exists\mathbf{a}\ \exists\mathbf{i}'\ \exists\mathbf{x}\ \exists\mathbf{d}\ \forall\mathbf{b}\ \forall\mathbf{j}\ \forall\mathbf{y}\ \{\exists d, \forall d\}^*.\ \phi \wedge \rho_T$$

where $\mathbf{i}' = \mathbf{i} \cup \mathbf{i}_T \cup \mathbf{i}_{Pos(\varphi)}$.

Computing a set of bounds: Given a template $T \in \mathcal{T}_{\varphi,\exists}$ we compute a finite set of solutions, denoted $\mathcal{M}_T$, for the index constraints in T. These solutions are all Pareto optimal w.r.t. the values of the index variables $\mathbf{i}_T$ that label edges in T. Pareto optimality means that for any $\mathsf{m} \in \mathcal{M}_T$ there is no solution of the constraints which is smaller or equal than m with respect to every variable, and strictly smaller than m with respect to at least one variable. We need to consider all the Pareto optimal solutions because minimizing only some of the variables in $\mathbf{i}_T$ does not lead necessarily to a model of φ since data constraints may impose upper bounds on the path lengths. For instance, for the formula ψ_2 given in (2), if we consider the first template whose graph is given in (3), the minimization according to l_2 (alone) leads to the solution $(3,5,2)$ which does not correspond to a model of ψ_2. (In that example, the right solutions to consider are $(2,6,4)$ and $(1,7,6)$ as we said earlier.)

The set $\mathcal{M}_T$ is computed using a multi-objective integer linear program (MOILP, for short) of the following form: `minimize` $\{l \mid l \in L_E|_2\}$ `subject to` C.

Template expanding: Given a $T \in \mathcal{T}_{\varphi,\exists}$ and an $\mathsf{m} \in \mathcal{M}_T$, we define a set of expanded templates $\mathcal{T}_{\varphi,T,\mathsf{m}}$ where each element is obtained by:
– replacing each edge in T by a path of the length specified in m. For this, new vertices are introduced, each of them being labeled with a fresh location variable, and moreover, each edge in these paths is labeled with a fresh index variable. Let $\mathbf{x}_\mathsf{m}$ and $\mathbf{i}_\mathsf{m}$ be the sets of these new location and index variables respectively.
– guessing labels in *Lab* for each new vertex, and guessing some additional edges involving the new vertices (while preserving determinism).

Proposition 2. *For every model G of the formula φ, there is a template $T \in \mathcal{T}_{\varphi,\exists}$, a solution $\mathsf{m} \in \mathcal{M}_T$, a template $T' \in \mathcal{T}_{\varphi,T,\mathsf{m}}$, and some valuations μ,θ,ν,δ for the free variables in $\rho_{T'}$ such that $G \models_{\mu,\theta,\nu,\delta} \rho_{T'}$.*

Then, we can prove the following fact:

Lemma 2. *The formula φ_1 is equivalent to*

$$\varphi_2 = \bigvee_{T \in \mathcal{T}_{\varphi,\exists}} \bigvee_{\mathsf{m} \in \mathcal{M}_T} \bigvee_{T' \in \mathcal{T}_{\varphi,T,\mathsf{m}}} \exists \mathbf{a}\, \exists \mathbf{i}'\, \exists \mathbf{x}\, \exists \mathbf{d}\, \exists \mathbf{i}_\mathsf{m}\, \exists \mathbf{x}_\mathsf{m}\, \forall \mathbf{b}\, \forall \mathbf{y}\, \forall \mathbf{j}\, \{\exists d, \forall d\}^*.\ \phi \wedge \rho_{T'}$$

where $\mathbf{x}_\mathsf{m}$ and $\mathbf{i}_\mathsf{m}$ denote the newly added location and index variables in T'.

A small model property: The formula φ_2 above is an expanded form of the original formula φ for which we can prove the following fact (the proof is omitted here):

Lemma 3. *Let $\varphi_{T,\mathsf{m},T'} = \exists \mathbf{a}\, \exists \mathbf{i}'\, \exists \mathbf{x}\, \exists \mathbf{d}\, \exists \mathbf{i}_\mathsf{m}\, \exists \mathbf{x}_\mathsf{m}\, \forall \mathbf{b}\, \forall \mathbf{y}\, \forall \mathbf{j}\, \{\exists d, \forall d\}^*.\ \phi \wedge \rho_{T'}$ be a disjunct of φ_2. Then, if $\varphi_{T,\mathsf{m},T'}$ is satisfiable, it must have a model of size less than or equal to the number of its existentially quantified index, array, and location variables.*

Then, having determined a bound on the models, for checking the satisfiability of φ, the two last steps can be done as described in the beginning of the section (i.e., elimination of the universal and then of the existential quantification over location, array, and index variables), leading to a formula in $\mathsf{FO}(\mathbb{D},\mathbb{O},\mathbb{P})$.

Complexity: Let us define the *size of a* CSL *formula* to be the number of variables, operators, and predicates, plus the integer constants. Then, the reduction presented above is a nondeterministic procedure where all the choices can be done in polynomial time with respect to the size of the formula. Also, it has access to an oracle with the same complexity as the complexity of solving MOILPs. All we know about the complexity of the MOILP problem is that it is NP-hard [12] and that it is polynomial when the number of variables is fixed [5]. Therefore, if we fix the number of universally quantified variables, the overall complexity of the procedure is NP^{MOILP}.

5.2 Extension to the Full CSL

We show briefly in this section the generalization of Theorem 2 to CSL.

Theorem 3. *The satisfiability of a* CSL *formula can be reduced to the satisfiability of a formula in the underlying data logic* $\mathsf{FO}(\mathbb{D},\mathbb{O},\mathbb{P})$.

The reduction is defined inductively on the quantifier level. It uses the fact that CSL_0 is $\mathsf{FO}(\mathbb{D},\mathbb{O},\mathbb{P})$ and the following lemma:

Lemma 4. *For every $k \geq 1$, the satisfiability of a CSL_k formula can be reduced to the satisfiability of a CSL_{k-p} formula, for some $0 < p \leq k$.*

We sketch hereafter the proof of the lemma above. Let $\varphi = \exists \mathbf{a}^k\ \exists \mathbf{i}^k\ \exists \mathbf{x}^{\leq k}\ \exists \mathbf{d}^k\ \forall \mathbf{b}^k\ \forall \mathbf{j}^k\ \forall \mathbf{y}^k\ Q_{k-1}\ \ldots Q_1 Q.\ \phi$ be a CSL_k formula. It can be observed that ϕ can relate vertices of different levels only by atomic formulas of the form $f(x) = y$ or by data constraints. Using this fact, we can build heap graph templates by considering each level independently from the others. We define, for the level k, a set of templates $\mathcal{T}_{k,\varphi,\exists}$, and then, a set of bounds is computed leading to a set of expanded templates, following the same lines as the ones followed in the proof of Theorem 2. Then, an expanded formula w.r.t. these templates can be constructed, allowing to establish a small property w.r.t. level k: Given a fixed model of the formula, it is possible to define a model whose size at level k is bounded by the number of vertices in the considered expanded templates. Based on this small model property, it is possible to build an equi-satisfiable formula without universally quantified location, array, and index variables of level k. It remains to eliminate the existentially quantified variables of level k. Let $k - p$ be the second greatest level (after k) of variables in φ. Then, the elimination of existential variables can be done as in the case of CSL_1 formulas, but here, we need also to deal with the variables appearing in atomic formulas of the form (1) $f(y) = x$, where y is of level k and x is of level less than k, or of the form (2) $f(x) = y$, where y is of level k and x is of level less than k. A formula of the first kind is replaced by an equality $z_{y,f} = x$ where $z_{y,f}$ is a fresh variable of the same type as x, and a formula of the second kind is replaced by a formula $l_{f,y}(x)$ where $l_{f,y}$ is a fresh label in *Lab* (all the vertices that had up-going edges labeled by f towards the vertex denoted by y are labeled by $l_{f,y}$).

The complexity of the reduction is similar to the one for CSL_1.

6 Related Work

Various approaches have been developed for the analysis of programs with dynamic data structures including works based on abstraction techniques, e.g., [1,3,17], on logics for reasoning about graph structures, e.g., [11,13,18,14,16], and on automata-theoretic techniques, e.g. [7,10,15]. Only a few of them are able to handle composite data structures. The approach in [3] introduces an abstraction domain composed of separation logic formulas. These formulas use a class of higher-order predicates which define recursively the composite data structure. The focus in that work is on the heap shape properties, assuming that data have been abstracted away. In contrast, our approach allows precise invariant checking to reason about the same type of structures, taking into account constraints on data and sizes: CSL allows (1) to specify (a finite number of) shared locations, (2) to reason about the lengths of lists and arrays, and (3) to model explicitly data over infinite domains. In [14] the logic LISBQ for reasoning about composite data structures is introduced. The definition of LISBQ is based on a partial order on the types of the program which forbids having links going from a smaller type to a greater one, whereas in CSL, although we use an ordered partition of the types, we do not impose constraints on the fields of the linked structures (we allow having edges going from one class to another one, and other edges going back). Formulas in LISBQ may contain only universal quantifiers, and the restrictions on LISBQ formulas do not allow to reason completely about doubly-linked lists as it is possible in CSL (or even in CSL_1). For example, `doubly-ll(v)` or `dll-len2(v)` are not expressible in LISBQ. Our logic allows to specify multi-linked lists and arrays, to express constraints on the lengths of the lists, and formulas in CSL may contain alternations of universal and existential quantifiers. (In fact, LISBQ can be seen as a strict fragment of CSL_1.) In LISBQ, the underlying data logic must be Presburger arithmetics or equality with uninterpreted functions, whereas it is possible to use in CSL any decidable logic.

Decidable logics for reasoning about memory heaps have been proposed in, e.g., [1,2,4,18]. These logics focus mainly on shape constraints and assume that the data domain is finite. Our logic allows an infinite data domain and it is incomparable w.r.t. the class of handled structures. For example, LRP [18] can specify unbounded trees but it cannot specify arrays of doubly linked lists (property `doubly-ll(v)`) which is possible in CSL.

Recently, several works have addressed the issue of reasoning about programs manipulating structures with unbounded data, e.g. [9,11,13,14]. The logic in [9] allows to reason about words but it is not closed under strongest postcondition computation for pointer manipulation operations. Then, the approaches from [11,13] allow to reason about arrays. If we restrict CSL_1 to arrays, we obtain a fragment which is incomparable to the Array Property Fragment in [11]. CSL_1 does not allow to compare data in the arrays with indexes. On the other hand, CSL_1 allows strict inequalities between universal index variables (see `sorted-id(v)`). When data are not integers (it is not possible to compare data with indexes) dealing in CSL_1 with strict inequalities between universal index variables solves an open problem stated in Section 5 of [11]. The approach in [13] allows for more expressive constraints on index variables due to the fact that it considers only arrays with integer data.

7 Conclusions and Future Work

We have introduced an expressive logic for reasoning about programs manipulating composite data structures storing data values. Our main result is the decidability of its satisfiability problem which is established using a reduction to the satisfiability problem of the underlying data logic. This allows in practice to use SMT solvers to implement the decision procedure.

The definition of CSL is based on syntactical restrictions allowing to prove a small model property for the logic. This property is lost if any of these restrictions is relaxed (in the sense that the size of the minimal models cannot be computed from the syntax of the formula, regardless of the considered data domain; it can rather be arbitrarily large depending on this domain). Nevertheless, it could be possible that some of these restrictions be relaxed without loss of decidability. For instance, on could adopt an alternative approach such as the one used in [13] which consists in reducing the satisfiability problem to the reachability problem in a class of extended automata.

Future work includes also (1) efficient techniques and heuristics for universal quantifier elimination (2) defining abstraction techniques based on expressing predicates in CSL, (3) developing techniques for automatic generation of invariants in CSL, (4) developing techniques for checking termination using CSL formulas for reasoning about ranking functions, etc.

References

1. Balaban, I., Pnueli, A., Zuck, L.D.: Shape analysis of single-parent heaps. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 91–105. Springer, Heidelberg (2007)
2. Benedikt, M., Reps, T.W., Sagiv, S.: A decidable logic for describing linked data structures. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 2–19. Springer, Heidelberg (1999)
3. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
5. Blanco, V., Puerto, J.: Short rational generating functions for multiobjective linear integer programming. arXiv:0712.4295v3 (2008)
6. Borger, E., Gradel, E., Gurevich, Y.: The Classical Decision Problem. Perspectives of Mathematical Logic. Springer, Heidelberg (1997)
7. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
8. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. Technical report. LIAFA, University Paris 7 & CNRS
9. Bouajjani, A., Habermehl, P., Jurski, Y., Sighireanu, M.: Rewriting systems with data. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) FCT 2007. LNCS, vol. 4639, pp. 1–22. Springer, Heidelberg (2007)
10. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)

11. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005)
12. Ehrgott, M.: A survey and annotated bibliography of multiobjective combinatorial optimization. OR Spectrum 22(4), 425–460 (2000)
13. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 474–489. Springer, Heidelberg (2008)
14. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: POPL, pp. 171–182. ACM, New York (2008)
15. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI, pp. 221–231. ACM, New York (2001)
16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
17. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
18. Yorsh, G., Rabinovich, A.M., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. J. Log. Algebr. Program. 73(1-2), 111–142 (2007)

Measuring Permissivity in Finite Games

Patricia Bouyer[1,⋆], Marie Duflot[2], Nicolas Markey[1,⋆], and Gabriel Renault[3]

[1] LSV, CNRS & ENS Cachan, France
{bouyer,markey}@lsv.ens-cachan.fr
[2] LACL, Université Paris 12, France
duflot@univ-paris12.fr
[3] Département Informatique, ENS Lyon, France
gabriel.renault@ens-lyon.fr

Abstract. In this paper, we extend the classical notion of strategies in turn-based finite games by allowing several moves to be selected. We define and study a quantitative measure for permissivity of such strategies by assigning penalties when blocking transitions. We prove that for reachability objectives, most permissive strategies exist, can be chosen memoryless, and can be computed in polynomial time, while it is in $\mathsf{NP} \cap \mathsf{coNP}$ for discounted and mean penalties.

1 Introduction

Finite games. Finite games have found numerous applications in computer science [Tho02]. They extend finite automata with several players interacting on the sequence of transitions being fired. This provides a convenient way for reasoning about open systems (subject to *uncontrollable actions* of their environment), and for verifying their correctness. In that setting, *correctness* generally means the existence of a controller under which the system always behaves according to a given specification. A controller, in that terminology, is nothing but a strategy in the corresponding game, played on the automaton of the system, against the environment.

Our framework. In this paper, we propose a new framework for computing *permissive* controllers in finite-state systems. We assume the framework of two-player turn-based games (where the players are Player $\Diamond$ and Player $\Box$, with the controller corresponding to Player $\Diamond$). The classical notion of (deterministic) strategy in finite (turn-based) games is extended into the notion of *multi-strategy*, which allows several edges to be enabled. The permissivity of such a multi-strategy is then measured by associating penalties to blocking edges (each edge may have a different penalty). A strategy is more permissive than an other one if its penalty is weaker, *i.e.*, if it blocks fewer (or less expensive) edges.

We focus on reachability objectives for the controller, that is, the first aim of Player $\Diamond$ will be to reach a designated set of winning states (whatever Player $\Box$

⋆ These authors were partly supported by the French project DOTS (ANR-06-SETI-003), by the European project QUASIMODO (FP7-ICT-STREP- 214755), and by the ESF project GASICS.

does). The second aim of Player $\Diamond$ will be to minimize the penalty assigned to the set of outcomes generated by the multi-strategy.

Formally we consider *weighted (finite) games*, which are turn-based finite games with non-negative weights on edges. In each state, the penalty assigned to a multi-strategy is the sum of the weights of the edges *blocked* by the multi-strategy. Several ways of measuring the penalty of a strategy can then be considered: in this paper, we consider three ways of counting penalties along outcomes (sum, discounted sum, and mean value) and then set the penalty of a multi-strategy as the maximal penalty of its outcomes.

We will be interested in several problems: (*i*) does Player $\Diamond$ have a winning multi-strategy for which the penalty is no more than a given threshold? (*ii*) compute the infimal penalty that Player $\Diamond$ can ensure while reaching her goal; (*iii*) synthesize (almost-)optimal winning multi-strategies, and characterize them (in terms of memory and regularity).

Our results. We first prove that our games with penalties can be transformed into classical weighted games [ZP96, LMO06] with an exponential blowup, and that the converse reduction is polynomial.

Then, we prove that we can compute optimal and memoryless multi-strategies for optimal reachability in PTIME. The proof is in three steps: first, using our transformation to weighted games and results of [LMO06], we obtain the existence of optimal memoryless multi-strategies; we then propose a polynomial-time algorithm for computing an optimal winning multi-strategy *with* memory; finally, we show how we can get rid of the memory in such a multi-strategy, which yields the expected result.

We then focus on two other ways of computing penalties, namely the discounted sum and the mean value, and we prove that optimal multi-strategies may not exist, or may require memory. We further prove that we can compute the optimal discounted penalty in NP∩coNP, and that we can search for almost-optimal winning multi-strategies as a pair (σ_1, σ_2) of memoryless multi-strategies and that we need to play σ_1 for some time before following σ_2 in order to reach the goal. The longer we play σ_1, the closer we end up to the optimal discounted penalty. The same holds for the mean penalty before reaching the goal.

As side-results, we obtain the complexity of computing strategies in weighted games with a combined objective of reaching a goal state and optimizing the accumulated cost. This can be seen as the *game version* of the shortest path problem in weighted automata. Regarding accumulated costs, this was already a by-product of [LMO06]; we show here that for discounted and mean costs, optimal or memoryless optimal strategies do not necessarily exist, but almost-optimal strategies can be obtained as a "pair" of memoryless strategies.

Related work. This quantitative approach to permissivity is rather original, and does not compare to either of the approaches found in the literature [BJW02, PR05]. Indeed classical notions of permissivity imply the largest sets of generated plays. This is not the case here, where an early cut of a branch/edge of the game may avoid a large penalty later on for blocking many edges. However our notion of multi-strategy coincides with the non-deterministic strategies of [BJW02] and [Lut08].

Our work also meets the problem proposed in [CHJ05] of considering mixed winning objectives, one which is qualitative (parity in their special case), and one which is quantitative (mean-payoff in their special case). The same kind of mixed objectives is considered when extending ATL with quantitative constraints [LMO06, HP06].

The rest of this paper is organized as follows. In the next section, we introduce our formalism of multi-strategies and penalties. We also explain the link with the classical framework of games with costs. Section 3 is devoted to our polynomial-time algorithm for computing most permissive strategies. Section 4 deals with the case of discounted and mean penalty. By lack of space, several proofs are omitted.

2 Weighted Games with Reachability Objectives

2.1 Basic Definitions

Weighted games. A *(finite) weighted game* is a tuple $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ where $V_\Box$ and $V_\Diamond$ are finite sets of states (said to belong to Player $\Box$ and Player $\Diamond$, resp.); writing $V = V_\Box \cup V_\Diamond \cup \{☺, ☹\}$ where ☺ and ☹ are two distinguished states not belonging to $V_\Box \cup V_\Diamond$, $E \subseteq V \times V$ is a finite set of edges; and $\mathsf{weight}\colon E \to \mathbb{N}$ is a function assigning a weight to every edge. We assume (w.l.o.g.) that the states ☺ and ☹ have no outgoing edges (they are respectively the winning and losing states). If $v \in V$, we write vE (resp. Ev) for $E \cap (\{v\} \times V)$ (resp. $E \cap (V \times \{v\})$) for the set of edges originating from (resp. targetting to) v.

A *run* ϱ in G is a finite or infinite sequence of states $(v_i)_{0 \le i \le p}$ (for some $p \in \mathbb{N} \cup \{\infty\}$) such that $e_i = (v_{i-1}, v_i) \in E$ when $0 < i \le p$. We may also write for such a run $\varrho = (v_0 \to v_1 \to v_2 \cdots)$, or $\varrho = (e_i)_{i \ge 1}$[1], or the word $\varrho = v_0\, v_1\, v_2 \ldots$ The length of ϱ, denoted by $|\varrho|$, is $p + 1$. For finite-length runs, we write $\mathsf{last}(\varrho)$ for the last state v_p. Given $r < |\varrho|$, the r-th prefix of $\varrho = (v_i)_{0 \le i \le p}$ is the run $\varrho_{\le r} = (v_i)_{0 \le i \le r}$. Given a finite run $\varrho = (v_i)_{0 \le i \le p}$ and a transition $e = (v, v')$ with $v = v_p$, we write $\varrho \xrightarrow{e}$, or $\varrho \to v'$, for the run $\varrho = (v_i)_{0 \le i \le p+1}$ with $v_{p+1} = v'$.

We write $\mathsf{Runs}^{<\omega}_{\mathsf{G}}$ (resp. $\mathsf{Runs}^{\omega}_{\mathsf{G}}$) for the set of finite (resp. infinite) runs in G, and $\mathsf{Runs}_{\mathsf{G}} = \mathsf{Runs}^{<\omega}_{\mathsf{G}} \cup \mathsf{Runs}^{\omega}_{\mathsf{G}}$. In the sequel, we omit the subscript G when no ambiguity may arise.

Multi-strategies. A *multi-strategy* for Player $\Diamond$ is a function

$$\sigma\colon \left\{\varrho \in \mathsf{Runs}^{<\omega} \mid \mathsf{last}(\varrho) \in V_\Diamond\right\} \to 2^E$$

such that, for all $\varrho \in \mathsf{Runs}^{<\omega}$, we have $\sigma(\varrho) \subseteq vE$ with $v = \mathsf{last}(\varrho)$. A multi-strategy is *memoryless* if $\sigma(\varrho) = \sigma(\varrho')$ as soon as $\mathsf{last}(\varrho) = \mathsf{last}(\varrho')$. A memoryless

[1] These notations are equivalent since we assume that there can only be one edge between two states.

multi-strategy σ can be equivalently represented as a mapping $\sigma' \colon V_\Diamond \to 2^E$, with $\sigma(\varrho) = \sigma'(\mathsf{last}(\varrho))$.

Multi-strategies extend the usual notion of *strategies* by selecting several possible moves (classically, a strategy is a multi-strategy σ such that for every $\varrho \in \mathsf{Runs}^{<\omega}$ with $\mathsf{last}(\varrho) \in V_\Diamond$, the set $\sigma(\varrho)$ is a singleton). The aim of this paper is to compare multi-strategies and to define and study a quantitative notion of *permissivity* of a multi-strategy.

Given a multi-strategy σ for Player $\Diamond$, the set of outcomes of σ, denoted $\mathsf{Out}(\sigma) \subseteq \mathsf{Runs}$, is defined as follows:

- for every state $v \in V$, the run v is in $\mathsf{Out}^{<\omega}(\sigma)$;
- if $\varrho \in \mathsf{Out}^{<\omega}(\sigma)$ and $\sigma(\varrho)$ is defined and non-empty, then for every $e \in \sigma(\varrho)$, the run $\varrho \xrightarrow{e}$ is in $\mathsf{Out}^{<\omega}(\sigma)$;
- if $\varrho \in \mathsf{Out}^{<\omega}(\sigma)$ and $\mathsf{last}(\varrho) = v \in V_\Box$, then for every edge $e \in vE$, the run $\varrho \xrightarrow{e}$ is in $\mathsf{Out}^{<\omega}(\sigma)$;
- if $\varrho \in \mathsf{Runs}^\omega$ and if all finite prefixes ϱ' of ϱ are in $\mathsf{Out}^{<\omega}(\sigma)$, then $\varrho \in \mathsf{Out}^\omega(\sigma)$.

We write $\mathsf{Out}(\sigma) = \mathsf{Out}^{<\omega}(\sigma) \cup \mathsf{Out}^\omega(\sigma)$. A run ϱ in $\mathsf{Out}(\sigma)$ is *maximal* whenever it is infinite, or it is finite and either $\sigma(\varrho) = \varnothing$, or $\mathsf{last}(\varrho)$ has no outgoing edge (*i.e.*, the set vE, with $v = \mathsf{last}(\varrho)$, is empty). If ϱ_0 is a finite outcome of σ, we write $\mathsf{Out}(\sigma, \varrho_0)$ (resp. $\mathsf{Out}^{\max}(\sigma, \varrho_0)$) for the set of outcomes (resp. maximal outcomes) of σ having ϱ_0 as a prefix. A multi-strategy σ is *winning* after ϱ_0 if every run $\varrho \in \mathsf{Out}^{\max}(\sigma, \varrho_0)$ is finite and has $\mathsf{last}(\varrho) = ☺$. A finite run ϱ_0 is *winning* if it admits a winning multi-strategy after ϱ_0. Last, a strategy is winning if it is winning from any winning state (seen as a finite run).

Penalties for multi-strategies. We define a notion of permissivity of a multi-strategy by counting the weight of transitions that the multi-strategy blocks along its outcomes. If σ is a multi-strategy and ϱ_0 is a finite run, the *penalty* of σ after ϱ_0, denoted $\mathsf{penalty}(\sigma, \varrho_0)$, is defined as $\sup\{\mathsf{penalty}_{\sigma,\varrho_0}(\varrho) \mid \varrho \in \mathsf{Out}^{\max}(\sigma, \varrho_0)\}$ where $\mathsf{penalty}_{\sigma,\varrho_0}(\varrho)$ is defined inductively, for every finite run $\varrho \in \mathsf{Out}(\sigma, \varrho_0)$, by:

- $\mathsf{penalty}_{\sigma,\varrho_0}(\varrho_0) = 0$;
- if $\mathsf{last}(\varrho) \notin V_\Diamond$ and $(\mathsf{last}(\varrho), v) \in E$, then $\mathsf{penalty}_{\sigma,\varrho_0}(\varrho \to v) = \mathsf{penalty}_{\sigma,\varrho_0}(\varrho)$;
- if $\mathsf{last}(\varrho) \in V_\Diamond$ and $(\mathsf{last}(\varrho), v) \in \sigma(\varrho)$, then

$$\mathsf{penalty}_{\sigma,\varrho_0}(\varrho \to v) \;=\; \mathsf{penalty}_{\sigma,\varrho_0}(\varrho) \;+ \sum_{(\mathsf{last}(\varrho),v') \in (E \setminus \sigma(\varrho))} \mathsf{weight}(\mathsf{last}(\varrho), v');$$

- if $\varrho \in \mathsf{Out}(\sigma, \varrho_0) \cap \mathsf{Runs}^\omega$, then $\mathsf{penalty}_{\sigma,\varrho_0}(\varrho) = \lim_{n \to +\infty} \mathsf{penalty}_{\sigma,\varrho_0}(\varrho_{\leq n})$.

The first objective of Player $\Diamond$ is to win the game (*i.e.*, reach ☺), and her second objective is to minimize the penalty. In our formulation of the problem, Player $\Box$ has no formal objective, but her aim is to play against Player $\Diamond$ (this is a zero-sum game), which implicitly means that Player $\Box$ tries to avoid reaching ☺, and if this is not possible, she tries to maximize the penalty before reaching ☺.

We write $\mathsf{opt\_penalty}(\varrho_0)$ for the optimal penalty Player $\Diamond$ can ensure after ϱ_0 while reaching ☺:

$$\mathsf{opt\_penalty}(\varrho_0) = \inf\{\mathsf{penalty}(\sigma', \varrho_0) \mid \sigma' \text{ winning multi-strategy after } \varrho_0\}.$$

It is equal to $+\infty$ if and only if Player $\Diamond$ has no winning multi-strategy after ϱ_0.

The following lemma is rather obvious, and shows that we only need to deal with the optimal penalty from a state.

Lemma 1. *Let* G *be a weighted game, let* ϱ *and* ϱ' *be two runs in* G *such that* $\mathsf{last}(\varrho) = \mathsf{last}(\varrho')$. *Then* $\mathsf{opt\_penalty}(\varrho) = \mathsf{opt\_penalty}(\varrho')$.

Given $\varepsilon \geq 0$, a winning multi-strategy σ is *ε-optimal after* ϱ_0 if $\mathsf{penalty}(\sigma, \varrho_0) \leq \mathsf{opt\_penalty}(\varrho_0) + \varepsilon$. It is *optimal after* ϱ_0 when it is 0-optimal after ϱ_0. If σ is ε-optimal from any winning state, then we say that σ is ε-optimal.

Classical weighted games. This way of associating values to runs and (multi-)strategies is rather non-standard, and usually it is rather a notion of *accumulated cost* along the runs which is considered. It is defined inductively as follows:

- $\mathsf{cost}(v) = 0$ for single-state runs;
- $\mathsf{cost}(\varrho \xrightarrow{e}) = \mathsf{cost}(\varrho) + \mathsf{weight}(e)$ otherwise.

Then again, if σ is a multi-strategy and ϱ_0 is a finite outcome, $\mathsf{cost}(\sigma, \varrho_0) = \sup\{\mathsf{cost}(\varrho) - \mathsf{cost}(\varrho_0) \mid \varrho \in \mathsf{Out}^{\max}(\sigma, \varrho_0)\}$, and notions of ($\varepsilon$-)optimal strategies are defined in the expected way.

Example 1. A weighted game is depicted on Fig. 1. For this example, it can be easily seen that the optimal strategy w.r.t. costs from state a consists in going through b, resulting in a weight of 6.

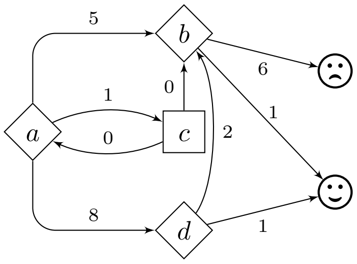


Fig. 1. A weighted game

Regarding penalties and multi-strategies, the situation is more difficult. From state b, there is only one way of winning, with penalty 6 (because the strategy *must* block the transition to the losing state). From d, we have two possible winning multi-strategies: either block the transition to b, with penalty 2, or keep it; in the latter case, we will then have penalty 6 in state d, as explained above. In d, the best multi-strategy thus amounts to blocking the transition to b, so that we can win with penalty 2. Now, from a, it seems natural to try to go winning *via* d. This requires blocking both transitions to b and c, and results in a global penalty of $8(= 5+1+2)$ for winning. However, allowing both transitions to b and d is better, as the global (worst case) penalty in this case is $7(= 1 + 6)$. Note that in that case, it is also possible to allow transition to c for some time, since the loop between a and c will add no extra penalty. But if we allow it forever, it will not be winning, this transition to c has thus to be blocked at some point in order to win.

Computation and decision problems. We let $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ be a weighted game. Given $v \in V$, we will be interested in computing the value $\mathsf{opt\_penalty}(v)$, and if an optimal winning multi-strategy exists, in computing it. We will also be interested in computing for every $\varepsilon > 0$, an ε-optimal winning multi-strategy.

Formally, the optimal reachability problem with penalty we consider is the following: given a weighted game G, a rational number c and a state $v \in V$, does there exist a multi-strategy σ for Player $\Diamond$ such that $\mathsf{penalty}(\sigma, v) \le c$.

2.2 From Penalties to Costs, and Back

Penalties and costs assume very different points of view: in particular, cost-optimality can obviously be achieved with "deterministic" strategies (adding extra outcomes can only increase the overall cost of the strategy), while penalty-optimality generally requires multi-strategies. Still, there exists a tight link between both approaches, which we explain on two examples (Figs. 2 and 3).

Lemma 2. *For every weighted game* $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$*, we can construct an exponential-size weighted game* $\mathsf{G}' = (V'_\Box, V'_\Diamond, E', \mathsf{weight}')$ *such that* $V_\Box \subseteq V'_\Box$*,* $V_\Diamond \subseteq V'_\Diamond$ *and, for any state* $v \in V_\Box \cup V_\Diamond$ *and any bound* c*, Player* $\Diamond$ *has a winning multi-strategy with penalty* c *from* v *in* G *iff she has a winning strategy with cost* c *from* v *in* G'*.*

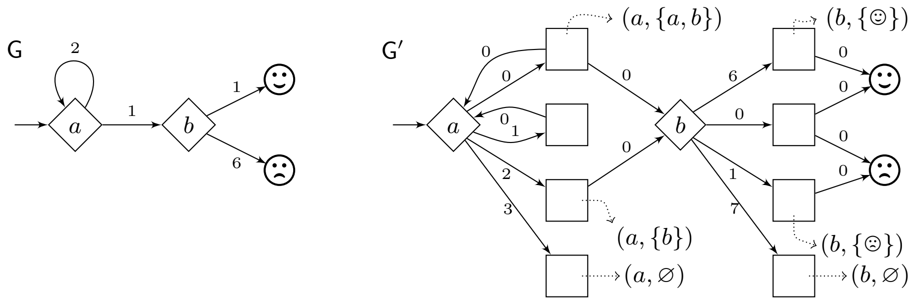


Fig. 2. From penalties (and multi-strategies) to costs (and strategies)

Lemma 3. *For every weighted game* $\mathsf{G}' = (V'_\Box, V'_\Diamond, E', \mathsf{weight}')$*, we can construct a polynomial-size weighted game* $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ *such that* $V'_\Box \subseteq V_\Box$*,* $V'_\Diamond \subseteq V_\Diamond$*, and for any state* $v \in V'_\Box \cup V'_\Diamond$ *and any value* c*, Player* $\Diamond$ *has a winning strategy with cost* c *from* v *in* G' *iff she has a winning multi-strategy with penalty* c *from* v *in* G*.*

3 Optimal Reachability in Penalty Games

Classical weighted games are known to admit memoryless optimal strategies (see *e.g.* [LMO06]). Hence, applying Lemma 2 we know that we can solve the optimal reachability problem with penalty in NP: memoryless multi-strategies are

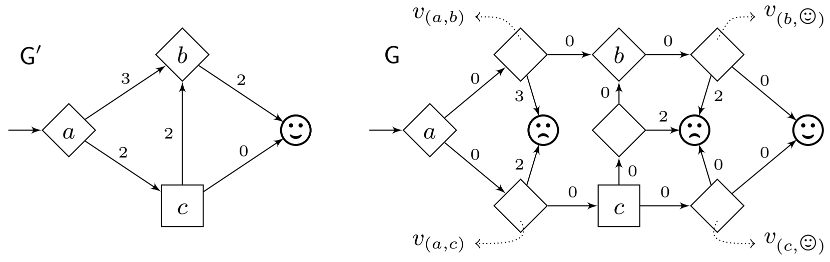


Fig. 3. From costs (and strategies) to penalties (and multi-strategies)

sufficient to win optimally, and we can thus guess a memoryless multi-strategy and check, in polynomial time, that it is winning and has penalty less than the given threshold. This section is devoted to the two-step proof of the following (stronger) result:

Theorem 4. *The optimal reachability problem with penalty can be solved in* PTIME.

In the sequel, we let $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ be a weighted game.

3.1 Construction of an Optimal Winning Multi-strategy

In this section, we give a polynomial-time algorithm for computing an optimal winning multi-strategy (which requires memory). The idea is to inductively compute the penalty for winning in j steps, for each j less than the number of states. This will be sufficient as we know that there exists a memoryless optimal multi-strategy, which wins in $|V|$ from the winning states.

Due to the transformation presented in Lemma 2, there is *a priori* an exponential blowup for computing the best move in one step (because Player $\Diamond$ can select any subset of the outgoing edges of the current state, and will choose 'the best' subset), but we will show that choices satisfy some monotonicity property that will help making the best choice in polynomial time.

For any integer k, we say that a multi-strategy σ is k-step if, for every run ϱ of length (strictly) larger than k with $\mathsf{last}(\varrho) \in V_\Diamond$, we have $\sigma(\varrho) = \varnothing$. For instance, a memoryless winning multi-strategy σ' naturally induces winning multi-strategy (all outcomes of σ' have length no more than $|V|$ and for all the other (useless) runs we can set $\sigma(\varrho) = \varnothing$). We say that a state v is winning in k steps if there is a k-step multi-strategy which is winning from v.

The algorithm will proceed as follows: for every $0 \leq j \leq |V|$, we build a j-step multi-strategy σ_j which will be winning from all states that are winning in j steps, and optimal among all those winning j-step multi-strategies. We also compute, for each state $v \in V$, a value $c_{v,j}$ which is either the penalty of strategy σ_j from v (*i.e.* $\mathsf{penalty}(\sigma_j, v)$), or $+\infty$ in case σ_j is not winning from v.

Since we know that memoryless multi-strategies suffice to win optimally, we conclude that there exists a $|V|$-step multi-strategy, which is winning and optimal, and the multi-strategy $\sigma_{|V|}$ which we build will then be optimal and winning. It follows that $c_{v,|V|}$ will be equal to $\mathsf{opt\_penalty}(v)$.

When $j = 0$, we let $\sigma_0(\varrho) = \varnothing$ for any ϱ ending in a $V_\Diamond$-state. It is the only 0-step multi-strategy, so that it clearly is optimal among these. Clearly, $c_{v,0} = +\infty$ for all $v \neq ☺$, $c_{☺,0} = 0$, and ☺ is the only state from which we can win with a 0-step multi-strategy.

We assume we have built σ_j $(0 \leq j < |V|)$, and we now define σ_{j+1}. Let $\varrho = v_0 \to v_1 \to v_2 \ldots \to v_k$ be a run ending in $V_\Diamond$. If $k \geq j+1$, we let $\sigma_{j+1}(\varrho) = \varnothing$. Otherwise, if $k \geq 1$, we let $\sigma_{j+1}(v_0 \to v_1 \to v_2 \ldots \to v_k) = \sigma_j(v_1 \to v_2 \ldots \to v_k)$. Finally, when $k = 0$ and $\varrho = v$, we let $\{u_1, \ldots, u_p\}$ be the set of successors of v, assuming that they are ordered in such a way that $c_{u_r,j} \leq c_{u_s,j}$ if $r \leq s$. Now, let

$$f_{v,j+1} \colon I \subseteq [\![1,p]\!] \mapsto \sum_{s \notin I} \mathsf{weight}(v, u_s) + \max_{s \in I} c_{u_s,j},$$

and let $I \neq \varnothing$ be a subset of $[\![1,p]\!]$ realizing the minimum of $f_{v,j+1}$ over the non-empty subsets of $[\![1,p]\!]$. Assume that there exist two integers $l < m$ in $[\![1,p]\!]$ such that $l \notin I$ and $m \in I$. Since $u_l \leq u_m$, we have

$$f_{v,j+1}(I \cup \{l\}) - f_{v,j+1}(I) = -\mathsf{weight}(v, u_l).$$

This entails that $I \cup \{l\}$ is also optimal. By repeating the process, we can prove that there exists an interval $[\![1,q]\!]$ realizing the minimum of $f_{v,j+1}$. As a consequence, finding the minimum of $f_{v,j+1}$ can be done in polynomial time (by checking all intervals of the form $[\![1,q]\!]$). We write $T_{v,j+1}$ for a corresponding set of states, whose indices realize the minimum of $f_{v,j+1}$. We then define $\sigma_{j+1}(v) = \{(v,v') \mid v' \in T_{v,j+1}\}$, and $c_{v,j+1} = f_{v,j+1}(T_{v,j+1})$ for all $v \in V_\Diamond$. It is easy to check that $c_{v,j+1} = \mathsf{penalty}(\sigma_{j+1}, v)$ if σ_{j+1} is winning from v, and $c_{v,j+1} = +\infty$ otherwise.

We now prove that for every $0 \leq j \leq |V|$, σ_j is optimal among all j-step winning multi-strategies. Assume that, for some $0 \leq j \leq |V|$, there is a j-step multi-strategy σ' that is winning and strictly better than σ_j from some winning state v. We pick the smallest such index j. We must have $j > 0$ since σ_0 is optimal among the 0-step multi-strategies. Consider the set of successors $\{u_1, \ldots, u_p\}$ of v ordered as above, and let T be the set of indices such that $\sigma'(v) = \{(v, u_t) \mid t \in T\}$. Then after one step, the multi-strategy σ' is $(j-1)$-step and winning from any u_t satisfying $(v, u_t) \in \sigma'(v)$, and its penalty is thus not smaller than that of the multi-strategy σ_{j-1} (by minimality of j, we have $\mathsf{penalty}(\sigma', v \to u_t) \geq c_{u_t,j-1}$). Hence:

$$\mathsf{penalty}(\sigma', v) \geq \sum_{s \notin T} \mathsf{weight}(v, u_s) + \max_{t \in T} c_{u_t,j-1} = f_{v,j}(T)$$

On the other hand, as σ' is strictly better than σ_j we must have

$$\mathsf{penalty}(\sigma', v) < c_{v,j} = f_{v,j}(T_{v,j}) \leq f_{v,j}(T)$$

because $T_{v,j}$ achieves the minimum of $f_{v,j}$. This is a contradiction, and from every state v from which there is a j-step winning multi-strategy, σ_j is winning optimally (in j steps).

As stated earlier, due to the existence of memoryless optimal winning multi-strategies, $|V|$-step multi-strategies are sufficient and $\sigma_{|V|}$ is optimal winning. ⌟

3.2 Deriving a Memoryless Winning Multi-strategy

In this section we compute, from any winning multi-strategy σ, a memoryless winning multi-strategy σ' which has lower penalty for Player $\Diamond$. The idea is to represent σ as a (finite) forest (it is finite because σ is winning) where a node corresponds to a finite outcome, and to select a state v for which σ is not memoryless yet. This state should be chosen carefully[2] so that we will be able to "plug" the subtree (*i.e.*, play the multi-strategy) rooted at some node ending in v at all nodes ending in v while keeping all states winning and while decreasing (or at least leaving unchanged) the penalty of all states. This transformation will be repeated until the multi-strategy is memoryless from all states. That way, if σ was originally optimal, then so will σ' be.

Let Σ be a finite alphabet. A *Σ-forest* is a tuple $\mathcal{T} = (T, R)$ where $T \subseteq \Sigma^+$ is a set of non-empty finite words on Σ (called *nodes*) such that, for each $t \cdot a \in T$ with $a \in \Sigma$ and $t \in \Sigma^+$, it holds $t \in T$ (T is closed by non-empty prefix) ; $R \subseteq \Sigma \cap T$ is the set of roots. Given $a \in \Sigma$, a node t such that $t = u \cdot a$ is called an *occurrence* of a. Given a node $t \in T$, the *depth* of t is $|t| - 1$ (where $|t|$ is the length of t seen as a word on Σ), and its height, denoted $\mathit{height}_{\mathcal{T}}(t)$, is

$$\sup\{|u| \mid u \in \Sigma^* \text{ and } t \cdot u \in T\}.$$

In particular, $\mathit{height}_{\mathcal{T}}(t) = +\infty$ when t is the prefix of an infinite branch in $\mathcal{T}$.

A Σ-tree is a Σ-forest with one single root. Given a forest $\mathcal{T} = (T, R)$ and a node $t \in T$, the *subtree of $\mathcal{T}$ rooted at t* is the tree $\mathcal{S} = (S, \{n\})$ where $n = \mathsf{last}(t)$ and $s \in S$ iff $t \cdot s \in T$.

Let $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ be a weighted game. A winning multi-strategy σ for Player $\Diamond$ in G and a winning state $v \in V$ naturally define a finite V-tree $\mathcal{T}_{\sigma,v}$ with root v: given a state s, a word $t = u \cdot s$ is in $\mathcal{T}_{\sigma,v}$ iff $u \in \mathcal{T}_{\sigma,v}$ and, seeing u as a finite run, we have either $\mathsf{last}(u) = v' \in V_\Diamond$ and $(v', s) \in \sigma(u)$, or $\mathsf{last}(u) = v' \in V_\Box$ and $(v', s) \in E$. In this tree, the height of the root coincides with the length of a longest run generated by the multi-strategy σ from v. Since the multi-strategy σ is winning from v, all branches are finite, and all leaves of $\mathcal{T}_{\sigma,v}$ are occurrences of ☺. The union of all trees $\mathcal{T}_{\sigma,v}$ (for v a winning state) defines a forest $\mathcal{T}_\sigma$.

Conversely, every V-forest $\mathcal{T} = (T, W)$ with $W \subseteq V$ satisfying the following conditions naturally defines a winning multi-strategy $\sigma_{\mathcal{T}}$ (viewing each node $t \in T$ as a run of G):

– if $\mathsf{last}(t) = v' \in V_\Box$, $t \cdot s \in T$ iff $(v', s) \in E$;
– if $\mathsf{last}(t) = v' \in V_\Diamond$ and $t \cdot s \in T$, then $(v', s) \in E$. In that case we set $\sigma_{\mathcal{T}}(t) = \{(v', s) \in E \mid t \cdot s \in T\}$;
– if t is maximal, then $\mathsf{last}(t) =$ ☺.

[2] An appropriate measure will be assigned to every node of the forest.

Lemma 5. *Assume that we are given an optimal winning multi-strategy σ. We can effectively construct in polynomial time a memoryless multi-strategy σ', which is winning and optimal.*

Proof. Assume that W is the set of winning states. Let $\mathcal{T}$ be the forest representing the multi-strategy σ (its set of roots is W). Since σ is winning from every state in W, all branches of the forest are finite. For every node t of $\mathcal{T}$, we define $\gamma_{\mathcal{T}}(t)$ as the *residual penalty* of σ after prefix t. Formally, $\gamma_{\mathcal{T}}(t) = \mathsf{penalty}(\sigma, t)$. Obviously, for all $v \in V$, we have $\mathsf{penalty}(\sigma, v) = \gamma_{\mathcal{T}}(v)$.

We will consider a measure $\mu_{\mathcal{T}}$ on the set of nodes of the forest $\mathcal{T}$ as follows: if t is a node of $\mathcal{T}$, we let $\mu_{\mathcal{T}}(t) = (\gamma_{\mathcal{T}}(t), \mathit{height}_{\mathcal{T}}(t))$.

We say that *no memory is required* for state v in $\mathcal{T}$ if, for every two nodes t and t' that are occurrences of v, the subtree of $\mathcal{T}$ rooted at t and the subtree of $\mathcal{T}$ rooted at t' are identical. Note that in that case, $\mu_{\mathcal{T}}(t) = \mu_{\mathcal{T}}(t')$.

For every $0 \leq i \leq |W|$, we inductively build in polynomial time a forest $\mathcal{T}^i$ and a set $M_i \subseteq W$ containing i elements, such that:

(a) $\mathcal{T}^i$ represents an optimal winning multi-strategy from all the states of W;
(b) for every $v \in M_i$, no memory is required for v in $\mathcal{T}^i$, and for every node t' which is a descendant of some node that is an occurrence of v, letting $v' = \mathsf{last}(t')$, it holds $v' \in M_i$.

Intuitively, each $\mathcal{T}^i$ will be the forest of a winning optimal multi-strategy σ_i, and each M_i will be a set of states from which σ_i is memoryless (*i.e.*, σ_i is memoryless from the states in M_i, and from the states that occur in the outcomes from these states). In the end, the forest $\mathcal{T}^{|W|}$ represents a multi-strategy σ' which is memoryless, optimal and winning from every state of the game. ⌟

4 Discounted and Mean Penalty Games

4.1 Discounted and Mean Penalties of Multi-strategies

We have proposed a way to measure the permissivity of winning strategies in games, by summing penalties for blocking edges in the graph. It can be interesting to consider that blocking an edge *early* in a run is more restrictive than blocking an edge *later*. A classical way to represent this is to consider a *discounted* version of the penalty of a multi-strategy, which we now define.

Discounted penalties of multi-strategies. Let $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ be a weighted game, σ be a winning (w.r.t. the reachability objective) multi-strategy, and ϱ_0 be a finite outcome of σ. Given a discount factor $\lambda \in (0,1)$, the *discounted penalty* of σ after ϱ_0 (w.r.t. λ), denoted $\mathsf{penalty}^\lambda(\sigma, \varrho_0)$, is defined as $\sup\{\mathsf{penalty}^\lambda_{\sigma,\varrho_0}(\varrho) \mid \varrho \in \mathsf{Out}^{\max}_{\mathsf{G}}(\sigma, \varrho_0)\}$, where $\mathsf{penalty}^\lambda_{\sigma,\varrho_0}(\varrho)$ is inductively defined for all $\varrho \in \mathsf{Out}_{\mathsf{G}}(\sigma, \varrho_0)$ as follows:

– $\mathsf{penalty}^\lambda_{\sigma,\varrho_0}(\varrho_0) = 0$;
– if $\mathsf{last}(\varrho) \notin V_\Diamond$ and $(\mathsf{last}(\varrho), v) \in E$, then $\mathsf{penalty}^\lambda_{\sigma,\varrho_0}(\varrho \to v) = \mathsf{penalty}^\lambda_{\sigma,\varrho_0}(\varrho)$;

– if $\mathsf{last}(\varrho) \in V_\Diamond$ and $(\mathsf{last}(\varrho), v) \in \sigma(\varrho)$, then $\mathsf{penalty}^\lambda_{\sigma,\varrho_0}(\varrho \to v)$ is defined as

$$\mathsf{penalty}^\lambda_{\sigma,\varrho_0}(\varrho) \;+\; \lambda^{|\varrho|-|\varrho_0|} \cdot \sum_{(\mathsf{last}(\varrho),v')\in(E\smallsetminus\sigma(\varrho))} \mathsf{weight}(\mathsf{last}(\varrho), v').$$

We also define the discounted penalty along infinite runs, as being the limit (which necessarily exists as $\lambda < 1$) of the penalties along the finite prefixes.

We write $\mathsf{opt\_penalty}^\lambda(\varrho_0)$ for the optimal discounted penalty (w.r.t. λ) Player $\Diamond$ can ensure after ϱ_0 while reaching ☺:

$$\mathsf{opt\_penalty}^\lambda(\varrho_0) = \inf\{\mathsf{penalty}^\lambda(\sigma, \varrho_0) \mid \sigma \text{ winning multi-strategy after } \varrho_0\}.$$

Given $\varepsilon \geq 0$ and $\lambda \in (0, 1)$, a multi-strategy σ is said *ε-optimal* for discount factor λ after ϱ_0 if it is winning after ϱ_0 and

$$\mathsf{penalty}^\lambda(\sigma, \varrho_0) \leq \mathsf{opt\_penalty}^\lambda(\varrho_0) + \varepsilon.$$

Again, optimality is a shorthand for 0-optimality. Finally, a multi-strategy is ε-optimal for discount factor λ if it is ε-optimal for λ from any winning state.

Discounted cost in weighted games. As in Section 2.1, we recall the usual notion cost^λ of discounted cost of runs in a weighted game [ZP96][3]:

– $\mathsf{cost}^\lambda(v) = 0$;
– $\mathsf{cost}^\lambda(\varrho \xrightarrow{e}) = \mathsf{cost}^\lambda(\varrho) + \lambda^{|\varrho|-1} \cdot \mathsf{weight}(e)$;

Then we define $\mathsf{cost}^\lambda(\sigma, \varrho_0) = \sup\{\mathsf{cost}^\lambda(\varrho) \mid \varrho \in \mathsf{Out}^{\max}_{\mathsf{G}}(\sigma, \varrho_0)\}$. Those games are symmetric, and later we will sometimes take the point-of-view of Player $\Box$ whose objective will be to maximize the discounted cost: given a strategy σ for Player $\Box$, we then define $\mathsf{cost}^\lambda(\sigma, \varrho_0) = \inf\{\mathsf{cost}^\lambda(\varrho) \mid \varrho \in \mathsf{Out}^{\max}_{\mathsf{G}}(\sigma, \varrho_0)\}$.

Computation and decision problems. As in the previous section, our aim is to compute (almost-)optimal multi-strategies. The optimal reachability problem with discounted penalty is the following: given a weighted game G, a rational number c, a discount factor $\lambda \in (0, 1)$, and a state $v \in V$, does there exist a multi-strategy σ for Player $\Diamond$ such that $\mathsf{penalty}^\lambda(\sigma, v) \leq c$. The transformations between penalties and costs depicted in Section 2.2 are still possible in the discounted setting. The only point is that in both cases, each single transition gives rise to two consecutive transitions, so that we must consider $\sqrt{\lambda}$ as the new discounting factor[4].

4.2 Some Examples

As far as the existence of an optimal multi-strategy is concerned, the discounted case is more challenging as the results of the previous section do not hold. In particular, we exemplify on Figures 4 and 5 the fact that optimal multi-strategies do not always exist, and when they exist, they cannot always be made memoryless.

[3] Note that we have dropped the normalization factor $(1-\lambda)$, which is only important to relate λ-discounted values to mean values (by making λ tend to 1) [ZP96].

[4] For the reduction of Lemma 3, the penalty is also multiplied by $\sqrt{\lambda}$.

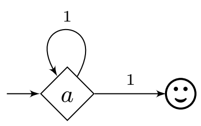


Fig. 4. No optimal discounted multi-strategy

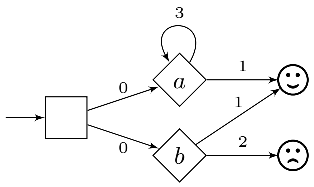


Fig. 5. No memoryless optimal discounted multi-strategy

4.3 A Pair of Memoryless Strategies Is Sufficient

We prove here that there always exist ε-optimal multi-strategies that are made of two memoryless multi-strategies. Roughly, the first multi-strategy aims at lengthening the path (so that the coefficient $\lambda^{|\varrho|}$ will be small) without increasing the penalty, and the second multi-strategy aims at reaching the final state.

To this aim, we need to first study the multi-strategy problem in the setting where there is no reachability objective. Let G be a finite weighted game, $\lambda \in (0,1)$, and $c \in \mathbb{Q}$. The optimal discounted-penalty problem consists in deciding whether there is a multi-strategy for Player $\Diamond$ for which the λ-discounted penalty along any maximal (finite or infinite) outcome is less than or equal to c.

Theorem 6. *The optimal discounted-penalty problem is in* $\mathsf{NP} \cap \mathsf{coNP}$*, and is* PTIME*-hard.*

The proof of this theorem relies on known results in classical discounted games [ZP96, Jur98], uses the transformation of Lemma 2 and monotonicity properties already used in the proof given in section 3.1.

Proof. We let $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ be a finite weighted game with no incoming transitions to ☹, and let $c \in \mathbb{Q}$. Applying the transformation of Lemma 2 to the discounted case, we get an exponential-size weighted game $\mathsf{G}' = (V'_\Box, V'_\Diamond, E', \mathsf{weight}')$ with $V_\Box \subseteq V'_\Box$ and $V_\Diamond = V'_\Diamond$ such that for every $v \in V_\Box \cup V_\Diamond$, Player $\Diamond$ has a winning multi-strategy from v in G with discounted penalty no more than c (for discount λ) iff Player $\Diamond$ has a winning strategy from v in G' with discounted cost no more than c (for discount $\sqrt{\lambda}$).

From [ZP96], Player $\Diamond$ has a memoryless optimal strategy in G'. The NP algorithm is then as follows: guess such a memoryless strategy $\sigma_\Diamond$ for Player $\Diamond$, *i.e.* for every $v \in V_\Diamond$ guess a subset $F \subseteq vE$ and set $\sigma_\Diamond(v) = (v, F)$. Removing from G' transitions that have not been chosen by $\sigma_\Diamond$ yields a polynomial-size graph G'', in which we can compute in polynomial time the maximal discounted cost, which corresponds to $\mathsf{cost}^{\sqrt{\lambda}}(\sigma_\Diamond, v)$. The graph G'' can be computed from G without explicitly building G', so that our procedure runs in polynomial time.

Membership in coNP is harder, and we only give a sketch of proof here. The game G' is memoryless determined [ZP96], which means that for every $c \in \mathbb{Q}$, for every state $v \in V'_\Box \cup V'_\Diamond$, either Player $\Diamond$ has a memoryless strategy $\sigma_\Diamond$ with $\mathsf{cost}^{\sqrt{\lambda}}(\sigma_\Diamond, v) \leq c$, or Player $\Box$ has a memoryless strategy $\sigma_\Box$ with $\mathsf{cost}^{\sqrt{\lambda}}(\sigma_\Box, v) > c$. Our coNP algorithm consists in guessing a memoryless strategy for Player $\Box$ that achieves cost larger than c. However, Player $\Box$ controls exponentially many states in G', so that we will guess a succinct encoding of her strategy, based on the following observation: there is a (preference) order on the states in $V_\Box \cup V_\Diamond$[5] so that, in states of the form (v, F), the optimal strategy for Player $\Box$ consists in playing the "preferred" state of F (w.r.t. the order). In other words, the strategy in those states can be defined in terms of an order on the states, which can be guessed in polynomial time.

Once such a strategy has been chosen non-deterministically, it then suffices to build a polynomial-size graph G'' in which the cost of the strategy $\sigma_\Box$ corresponds to the minimal discounted cost of Player $\Box$ in G'.

Hardness in PTIME directly follows from Lemma 3. ⌟

Remark 1. This problem could be extended with safety condition: the aim is then to minimize the discounted penalty while avoiding some bad states. An easy adaptation of the previous proof yields the very same results for this problem.

Definition 7. *Let σ_1 and σ_2 be two memoryless multi-strategies, and $k \in \mathbb{N}$. The multi-strategy $\sigma = \sigma_1^k \cdot \sigma_2^*$ is defined, for each ϱ such that $\mathsf{last}(\varrho) \in V_\Diamond$, as:*

– *if $|\varrho| < k$, then $\sigma(\varrho) = \sigma_1(\varrho)$;*
– *if $|\varrho| \geq k$, then $\sigma(\varrho) = \sigma_2(\varrho)$.*

Theorem 8. *Let $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ be a finite weighted game with a reachability objective, and $\lambda \in (0, 1)$. Then there exist two memoryless multi-strategies σ_1 and σ_2 such that, for any $\varepsilon > 0$, there is an integer k such that the multi-strategy $\sigma_1^{k'} \cdot \sigma_2^*$ is ε-optimal (w.r.t. λ-discounted penalties) for any $k' \geq k$.*

Proof. This is proved together with the following lemma:

Lemma 9. *Let $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ be a finite weighted game with a reachability objective, $\lambda \in (0, 1)$, and $c \in \mathbb{Q}$. Then (G, λ, c) is a positive instance of the optimal discounted-penalty problem iff for any $\varepsilon > 0$, $(\mathsf{G}, \lambda, c+\varepsilon)$ is a positive instance of the optimal reachability problem with discounted penalty*

Proof. From the remark following the proof of Theorem 6, there is a memoryless optimal multi-strategy σ_1 all of whose maximal outcomes have λ-discounted penalty less than or equal to c, and never visit losing states. Let σ_2 be a memoryless winning multi-strategy for the reachability objective, and let c_2 be the maximal penalty accumulated along an outcome of σ_2. Let $\varepsilon > 0$, and $k \in \mathbb{N}$ such that $\lambda^k \cdot c_2 \leq \varepsilon$. Then for any $k' > k$, the multi-strategy $\sigma_1^{k'} \cdot \sigma_2^*$ is winning, and the λ-discounted penalty of any outcome is at most $c + \lambda^{k'} \cdot c_2 \leq c + \varepsilon$.

Conversely, let $\varepsilon > 0$, and σ be a winning multi-strategy achieving discounted penalty no more than $c + \varepsilon$. Then in particular, σ achieves discounted penalty less than or equal to $c + \varepsilon$ along all of its outcomes, so that $(G, \lambda, c + \varepsilon)$ is a positive instance of the optimal discounted-penalty problem (for any $\varepsilon > 0$). From

[5] Which will be given by ordering the values given by the classical optimality equations [Jur98] in G'.

Theorem 6, this problem admits a (truly) optimal memoryless multi-strategy, so that there must exist a multi-strategy achieving discounted penalty less than or equal to c along all of its outcomes. ⌟

Theorem 10. *The optimal reachability problem with discounted penalty is in* $\mathsf{NP} \cap \mathsf{coNP}$*, and is* PTIME*-hard.*

Remark 2. It can be observed that the results of this section extend to discounted-cost games with reachability objectives (without the exponential gap due to the first transformation of weighted games with penalties). In particular, those games admit almost-optimal strategies made of two memoryless strategies, and the corresponding decision problem is equivalent to classical discounted-payoff games.

4.4 Extension to the Mean Penalty of Multi-strategies

We also define the *mean penalty* of a multi-strategy σ from state v, denoted $\mathsf{mean\_penalty}(\sigma, v)$, as $\sup\{\mathsf{mean\_penalty}_\sigma(\varrho) \mid \varrho \in \mathsf{Out}_\mathsf{G}(\sigma, v),\ \varrho \text{ maximal}\}$, where

$$\mathsf{mean\_penalty}_\sigma(\varrho) = \begin{cases} \frac{\mathsf{penalty}_\sigma(\varrho)}{|\varrho|} & \text{if } |\varrho| < \infty \\ \limsup\limits_{n \to +\infty} \mathsf{mean\_penalty}_\sigma(\varrho_{|\leq n}) & \text{otherwise} \end{cases}$$

where $\varrho_{|\leq n}$ is the prefix of length n of ϱ. The notion of ε-optimality, for $\varepsilon \geq 0$, is defined as previously. Using the same lines of arguments as earlier, we get:

Theorem 11. *Let* $\mathsf{G} = (V_\Box, V_\Diamond, E, \mathsf{weight})$ *be a finite weighted game with reachability objectives, in which all states in* $V_\Box \cup V_\Diamond$ *are winning. There exist two memoryless multi-strategies* σ_1 *and* σ_2 *such that, for any* $\varepsilon > 0$*, there exists* k *so that the multi-strategy* $\sigma_1^{k'} \cdot \sigma_2^*$ *is* ε*-optimal (w.r.t. mean penalties) for any* $k' \geq k$*.*

Theorem 12. *The optimal reachability problem with mean-penalty is in* $\mathsf{NP} \cap \mathsf{coNP}$ *and is* PTIME*-hard.*

Remark 3. Again, this result extends to mean-cost games with reachability objectives, which thus admit almost-optimal strategies made of two memoryless strategies. Surprisingly, the same phenomenon has been shown to occur in mean-payoff parity games [CHJ05], but the corresponding strategy can be made fully optimal thanks to the infiniteness of the outcomes.

5 Conclusion and Future Work

We have proposed an original quantitative approach to the permissivity of (multi-)strategies in two-player games with reachability objectives, through a natural notion of penalty given to the player for blocking edges. We have proven that most permissive strategies exist and can be chosen memoryless in the case where penalties are added up along the outcomes, and proposed a PTIME algorithm for computing such an optimal strategy. When considering discounted sum or mean penalty, we have proved that we must settle for almost-optimal strategies, which are built from two memoryless strategies. The resulting algorithm is in NP ∩ coNP. This is rather surprising as the natural way of encoding multi-strategies in classical weighted games entails an exponential blowup.

Besides the naturalness of multi-strategies, our initial motivation underlying this work (and the aim of our future works) is in the domain of timed games [AMPS98, BCD$^+$07]: in that setting, strategies are often defined as functions from executions to pairs (t, a) where t is a real number and a an action. This way of defining strategies goes against the paradigm of *implementability* [DDR04], as it requires infinite precision. We plan to extend the work reported here to the timed setting, where penalties would depend on the precision needed to apply the strategy. Also, as stated in [CHJ05], we believe that games with mixed objectives are interesting on their own, which gives another direction of research for future work. This catches up with related works on quantitative extensions of ATL.

References

[AMPS98] Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: Proc. IFAC Symposium on System Structure and Control, pp. 469–474. Elsevier, Amsterdam (1998)

[BCD$^+$07] Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)

[BJW02] Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: From parity games to safety games. Inf. Théor. et Applications 36(3), 261–275 (2002)

[CHJ05] Chatterjee, K., Henzinger, T.A., Jurdziński, M.: Mean-payoff parity games. In: Proc. 20th Annual Symposium on Logic in Computer Science (LICS 2005). IEEE Computer Society Press, Los Alamitos (2005)

[DDR04] De Wulf, M., Doyen, L., Raskin, J.-F.: Almost ASAP semantics: From timed models to timed implementations. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 296–310. Springer, Heidelberg (2004)

[HP06] Henzinger, T.A., Prabhu, V.S.: Timed alternating-time temporal logic. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 1–17. Springer, Heidelberg (2006)

[Jur98] Jurdziński, M.: Deciding the winner in parity games is in UP ∩ coUP. Information Processing Letters 68(3), 119–124 (1998)

[LMO06] Laroussinie, F., Markey, N., Oreiby, G.: Model-checking timed ATL for durational concurrent game structures. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 245–259. Springer, Heidelberg (2006)

[Lut08] Luttenberger, M.: Strategy iteration using non-deterministic strategies for solving parity games. Research Report cs.GT/0806.2923, arXiv (2008)

[PR05] Pinchinat, S., Riedweg, S.: You can always compute maximally permissive controllers under partial observation when they exist. In: Proc. 24th American Control Conf. (ACC 2005), pp. 2287–2292 (2005)

[Tho02] Thomas, W.: Infinite games and verification (Extended abstract of a tutorial). In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 58–64. Springer, Heidelberg (2002)

[ZP96] Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theoretical Computer Science 158(1-2), 343–359 (1996)

Contracts for Mobile Processes

Giuseppe Castagna[1] and Luca Padovani[2]

[1] CNRS, Laboratoire Preuves, Programmes et Systèmes, Université Paris Diderot – Paris 7
[2] Istituto di Scienze e Tecnologie dell'Informazione, Università di Urbino

Abstract. Theories identifying well-formed systems of processes—those that are free of communication errors and enjoy strong properties such as deadlock freedom—are based either on session types, which are inhabited by channels, or on contracts, which are inhabited by processes. Current session type theories impose overly restrictive disciplines while contract theories only work for networks with fixed topology. Here we fill the gap between the two approaches by defining a theory of contracts for so-called mobile processes, those whose communications may include delegations and channel references.

1 Introduction

Research on communicating processes has recently focused on the study of static analysis techniques for characterizing those systems that enjoy desirable properties such as the conformance to specifications, the absence of communication errors, and progress. By progress we mean the guarantee that the system will either evolve into a so-called successful state, in which all of its components have completely carried out their task, or that the system will continue to evolve indefinitely, without ever getting stuck.

Two approaches have been studied in great depth: those based on *session types*, and those based on *behavioral contracts*. Session types arise in a type theoretic framework as an evolution of standard channel types [29], by taking into account the fact that the same channel can be used at different times for sending messages of different kind and type. For example, session types defined in [21] describe potentially infinite, dyadic conversations between processes as sequences of messages that are either sent or received. The same approach has been applied to multi-threaded functional programming languages [30], as well as to object-oriented languages [3, 15]. Behavioral contracts arise in a process algebraic framework where the behavior of the component of a system is approximated by means of some term in a process algebra. The contract does not usually reveal how the component is implemented, but rather what its observable behavior is. Theories of contracts for Web services have been introduced in [12] and subsequently extended in [14, 25, 28]. Independently, theories of contracts have been developed for reasoning on Web service choreographies in [6, 7].

Traditionally, the approaches based on session types have always allowed the description of systems where channels and opened sessions can be communicated and delegated just as plain messages. Conversely, all of the works on contracts mentioned above rely on CCS-like formalisms, which makes them suitable for describing systems whose topology is fixed and where the role of each component does not change over time. This work aims at being a first step in filling the gap that concerns the expressiveness of the two approaches and in defining a contract language to describe processes that collaborate not only by exchanging messages, but also by delegating tasks, by dynamically initiating new conversations, by joining existing conversations.

Since we want to describe conversations where exchanged messages may include channels, it is natural to propose a contract language based on the π-calculus. However, quite a few aspects force us to depart from standard presentations. The first aspect regards the interpretation of *actions*. A contract should permit to describe not only the communication of channels, but also of other kinds of messages. Thus we will have actions of the form $c?\mathtt{Int}$ or $c!\mathtt{String}$ for denoting the ability to receive (resp. send) a value of type Int (resp. String). In fact, we will generalize input and output actions so that their subjects are possibly structured *patterns* describing all and only the messages that can be exchanged on a channel at a given time. Since we are interested in describing the observable behavior of a process rather than its internal structure, we will mostly focus on two operators $+$ and $\oplus$ representing external and internal choices respectively. This is consistent with the works on session types and is also the approach taken in some presentations of CCS [17, 20] and of π-calculus [26]. For example, the contract $c?\mathtt{Int}.T + d?\mathtt{String}.S$ describes a process that behaves as T if it receives a number on the channel c, and that behaves as S if it receives a string on the channel d. Dually, the contract $c!\mathtt{Int}.T \oplus d!\mathtt{String}.S$ describes a process that internally decides whether to send an integer or a string, and that behaves as T or as S accordingly. Additionally, we let branching be dependent also on the (type of) exchanged messages, not just on the channel on which communication occurs. This allows us to model behaviors that are affected by the actual values that are communicated, which is just what happens in session type theories with branching and selection constructs whose behavior is driven by labels. For instance, the contract $c?\mathtt{Int}.T + c?\mathtt{String}.S$ describes a process that behaves either as T if it receives an integer value on the channel c or as S if it receives a string value *on the same channel*. Unlike the π-calculus and along the lines of [1], we distinguish several "terminal" behaviors: $\mathbf{0}$ represents a deadlock process that cannot make any further progress; $\mathbf{1}$ represents successful termination; Ω represents an autonomous process that is capable of making indefinite progress.

With respect to current literature on session types we innovate also from the methodological point of view. Current presentations of session type theories begin by defining the semantics of types and then prove that well-typed processes enjoy certain properties. We embrace a *testing* approach [5, 17, 20] and go the other way round: we start by defining the properties we are interested in and then study the types that ensure them. More precisely, we start from the concept of *well-formed system*, which is a parallel composition of participants—each one abstracted by means of its contract—that mutually satisfy each other: no participant ever gets stuck waiting for messages that are never sent or trying to send messages that no one else is willing to receive. Well-formedness gives us the one essential notion that drives the whole theory: a component T is *viable* if there is a well-formed system that includes T. Viability roughly corresponds to the notion of well-typed process in session type theories. Also, well-formedness gives us a semantic way of relating contracts: T and S are *equivalent* if they yield well-formed systems when combined with the same components. The equivalence relation induced by well-formedness can be used for verifying whether a participant respects its specification given as a contract, or for understanding when two participants are equivalent by comparing their contracts, or for querying a database of participants by means of their contract, along the lines of what has been done in the CCS context. Additionally, the very same equivalence relation can be used for assessing properties of participants and systems: a contract T is viable if and only if it is not equivalent to $\mathbf{0}$; a system S is well-formed if $S \simeq \mathbf{1} + S$; a system S has indefinite progress if $S \simeq \Omega + S$.

We can identify three main contributions of this work. First, we propose a natural extension of the contract language presented in [12, 14, 25] for describing processes that communicate channels in addition to plain messages. Second, we define a notion of well-formed system as a system whose stable states, those not allowing any further progress, are such that *all* components have terminated successfully. The resulting equivalence relation is inspired to the classical *must* preorder for mobile processes [5, 26], except that we shift the attention from the success of a particular process (the "test" in the classical framework), to the success of all the components in the system. Well-formedness extends the notion of correct composition in [6, 7] to a name-passing scenario. Finally, we show how contracts provide an alternative and somehow naïve approach to the typing of processes. Indeed we believe that the increasing complexity of session type systems comes from the fact that they are used for guaranteeing properties they were not originally designed for. Session types, which characterize channels, are types with a behavioral flavor. We say "flavor" because, as a matter of facts, channels (like any other value) do not expose any behavior *per se*. Session types do not characterize the behavior of the channels they type; they just *reflect* the behavior of the processes that use those channels. This is evident in the inference rules of every session type system, where a process is *not* associated with a type that abstracts its behavior, but with a set of type assignments that provides a partial and distributed description of how the process independently uses each of its channels. It is thus unsurprising that session types systems guaranteeing global properties—most notably progress—must necessarily rely on draconian restrictions on the usage of sessions (see [4, 13, 18] for a showcase of such conditions). Conversely, contracts faithfully capture the behavior of processes and the temporal dependencies between communications occurring on different channels. This significantly widens the spectrum of systems that are declared well formed.

Overview. We develop our theory of contracts in §2 and put it at work on a core session calculus in §3, where we encode various examples, most of which from existing bibliography. §4 discusses differences between our approach and those with session types, presents more related work and hints at future research. For space reasons proofs, alternative characterizations, decidability, and deduction systems for the relations defined in §2 are omitted and can be found in the extended version available online.

2 Contracts

The syntax of contracts makes use of an infinite set $\mathscr{N}$ of *channel names* ranged over by $a, b, c, \dots$ and of an infinite set $\mathscr{X}$ of *channel variables* ranged over by $x, y, z, \dots$; let $\mathscr{V}$ be the set of *basic values* ranged over by $v, u, \dots$; we let $\alpha, \beta, \dots$ range over elements of $\mathscr{N} \cup \mathscr{X}$ and $m, n, \dots$ range over *messages*, namely elements of $\mathscr{N} \cup \mathscr{V}$. Contracts, ranged over by $T, S, \dots$, action prefixes, ranged over by $\pi, \pi', \dots$, and patterns, ranged over by $f, g, \dots$ are defined by the rules below:

$$\begin{array}{lcl}
T & ::= & \mathbf{0} \mid \mathbf{1} \mid \Omega \mid \pi.T \mid T+T \mid T \oplus T \mid T|T \mid (\nu a)T \\
\pi & ::= & \alpha?f \mid \alpha!f \mid \alpha!(a) \\
f & ::= & \mathbb{0} \mid x \mid m \mid \mathtt{B} \mid (x) \mid f \vee f \mid f \wedge f \mid \neg f
\end{array}$$

The syntax of patterns makes use of a fixed set of basic types ranged over by B such as Int, Bool, Real, ... Patterns describe the (possibly infinite) set of messages that are sent or received by an action occurring in a contract. Their semantics is fairly standard and *pattern matching* rules, described below, derive matching relations of the form $m \in f \rightsquigarrow \sigma$ where $\sigma : \mathscr{X} \to \mathscr{N}$ is a substitution whose domain $\mathtt{dom}(\sigma)$ is always finite:

$$m \in m \rightsquigarrow \emptyset \qquad a \in (x) \rightsquigarrow \{^a/_x\}$$

$$\frac{v : \mathtt{B}}{v \in \mathtt{B} \rightsquigarrow \emptyset} \qquad \frac{m \in f \rightsquigarrow \sigma}{m \in f \vee g \rightsquigarrow \sigma} \qquad \frac{m \in f \rightsquigarrow \sigma \qquad m \in g \rightsquigarrow \sigma'}{m \in f \wedge g \rightsquigarrow \sigma\sigma'} \qquad \frac{m \notin f}{m \in \neg f \rightsquigarrow \emptyset}$$

The empty pattern $\mathbb{0}$ is the one that matches no message. The singleton pattern m matches the message m. The type pattern B matches all and only basic values of type B. The variable pattern x represents a singleton pattern that is going to be instantiated (x must be bound at some outer scope), whereas the capture pattern (x) binds the variable x and matches every channel. The disjunction, conjunction, and negation patterns implement the standard boolean operators and must obey the standard syntactic constraints: the two branches of a disjunction must bind the same variables and the two branches of a conjunction must bind disjoint variables. We write $\mathtt{fv}(f)$ and $\mathtt{bv}(f)$ for the sets of free and bound variables occurring in a pattern. Their definition is standard, here we just recall that $\mathtt{bv}(\neg f) = \emptyset$ regardless of f. Also, we write $f \setminus g$ for $f \wedge \neg g$. Let f be a closed pattern, namely a pattern such that $\mathtt{fv}(f) = \emptyset$; the semantics of f, notation $[\![f]\!]$, is the set of messages that are matched by f, that is $[\![f]\!] = \{m \mid \exists\sigma : m \in f \rightsquigarrow \sigma\}$. We impose an additional constraint on patterns occurring in output actions, which we call output patterns from now on. An output pattern f is *valid* if it matches a *finite* number of names, that is $[\![f]\!] \cap \mathscr{N}$ is finite. Basically this means that a process cannot “guess” channel names. If it sends a channel, then either it is a public channel, or a channel it has received earlier, or it is a fresh channel (pattern validity is formalized in the long version of the paper).

Contracts include three terminal behaviors: $\mathbf{0}$ is the behavior of a deadlocked process that executes no further action; $\mathbf{1}$ is the behavior of the successfully terminated process; Ω is the behavior of a process that can autonomously evolve without any further interaction with the environment. A contract $\pi.T$ describes a process that executes an action π and then behaves according to T. There are three kinds of actions: an input action $\alpha?f$ describes the ability to input any message that matches f on the channel α; output actions $\alpha!f$—i.e., *free* outputs—describe a similar output capability. Actions of the form $\alpha!(a)$—i.e., *bound* outputs—describe the creation and extrusion of a fresh, private channel a, namely the establishment of a connection between the sender of the message and the receiver of the message. The contracts $T + S$ and $T \oplus S$ respectively describe the external and internal choices between the behaviors described by T and S. In an external choice, the process will behave as either T or S according to the environment. In an internal choice, the process will internally and autonomously decide to behave as either T or S. We also include two static operators for combining contracts, but only as a convenient way of describing systems as terms $(\nu a_1)\cdots(\nu a_n)(T_1 \mid T_2 \mid \cdots \mid T_m)$ where T_i is a description of the i-th participant, which is essentially a sequential process, the a_i's are the private channels used by the participants to communicate with each other, and the participants execute in parallel. It can be shown that restriction and parallel composition are redundant and their effects can be expressed by suitable combinations of actions (including bound output actions) and the two choice operators.

To cope with infinite behaviors we use a technique we introduced in [14] and allow contracts (but not patterns) to be infinite, provided that they satisfy the following conditions: (1) their syntax tree must be regular, namely it must be made of a finite number of different subtrees; (2) the syntax tree must contain a finite number of occurrences of the parallel composition operator and of the restriction operator. The reason of such a choice is that we want to account for infinite behaviors in a syntax-independent way. In process algebra literature infinite behaviours are obtained by adding syntax for defining recursive processes: usually these appear in the form of recursively defined processes $\mathtt{rec}\,X.T$, or of starred processes T^*, or of a separate set of mutually recursive declarations. These are nothing but finite representations of the infinite (syntax) trees obtained by unfolding or expanding them. We believe that it is far simpler and enlightening to get rid of syntactic constraints and work directly on infinite trees by relying on the theory developed by Bruno Courcelle [16]. In our theory we do not consider every possible infinite tree but just those that satisfy the two conditions above. The first condition – regularity – powers down the expressiveness of the language to include only and all contracts that can be generated by the `rec`-expressions or mutually recursive declarations customary in the process algebra literature (it is well-known that in a nondeterministic setting the $*$ operator is not as expressive as recursive definitions). Nevertheless all results stated in this work hold also for non regular trees, except the decidability ones of course. The second condition, which requires that only finitely many restrictions and parallel compositions occur in a contract, ensures that every contract describes a finite-state system (and, incidentally, keeps the whole theory decidable). Once the results are established for infinite trees, then it is straightforward to transpose them to any concrete syntax chosen and verify the consequences of this choice (see [13, 14] for a more detailed discussion on similar restrictions).

We give contracts the labeled operational semantics defined by the rules below, plus the symmetric of the rules for the binary operators. Labels are generated by the grammar $\mu ::= \checkmark \mid c?m \mid c!m \mid c!(a)$ and we use standard definitions $\mathtt{bn}(\cdot)$ and $\mathtt{fn}(\cdot)$ of bound and free names for labels and contracts:

$$\Omega \longrightarrow \Omega \qquad \mathbf{1} \xrightarrow{\checkmark} \mathbf{1} \qquad T \oplus S \longrightarrow T \qquad c!m.T \xrightarrow{c!m} T \qquad \frac{m \in f \rightsquigarrow \emptyset \qquad m \neq f}{c!f.T \longrightarrow c!m.T}$$

$$\frac{a \neq c}{c!(a).T \xrightarrow{c!(a)} T} \qquad \frac{m \in f \rightsquigarrow \sigma}{c?f.T \xrightarrow{c?m} T\sigma} \qquad \frac{T \xrightarrow{\mu} T'}{T + S \xrightarrow{\mu} T'} \qquad \frac{T \longrightarrow T'}{T + S \longrightarrow T' + S}$$

$$\frac{T \longrightarrow T'}{T \mid S \longrightarrow T' \mid S} \qquad \frac{T \xrightarrow{\mu} T' \qquad \mu \neq \checkmark \qquad \mathtt{bn}(\mu) \cap \mathtt{fn}(S) = \emptyset}{T \mid S \xrightarrow{\mu} T' \mid S} \qquad \frac{T \xrightarrow{\checkmark} T' \qquad S \xrightarrow{\checkmark} S'}{T \mid S \xrightarrow{\checkmark} T' \mid S'}$$

$$\frac{T \xrightarrow{c!m} T' \qquad S \xrightarrow{c?m} S'}{T \mid S \longrightarrow T' \mid S'} \qquad \frac{T \xrightarrow{c!(a)} T' \qquad S \xrightarrow{c?a} S' \qquad a \notin \mathtt{fn}(S)}{T \mid S \longrightarrow (\nu a)(T' \mid S')}$$

$$\frac{T \xrightarrow{\mu} T' \qquad a \notin \mathtt{fn}(\mu) \cup \mathtt{bn}(\mu)}{(\nu a)T \xrightarrow{\mu} (\nu a)T'} \qquad \frac{T \xrightarrow{c!a} T' \qquad a \neq c}{(\nu a)T \xrightarrow{c!(a)} T'} \qquad \frac{T \longrightarrow T'}{(\nu a)T \longrightarrow (\nu a)T'}$$

The contract Ω provides an unbound number of internal transitions, while no transition stems from $\mathbf{0}$. The contract $\mathbf{1}$ emits a label $\checkmark$ denoting successful termination and reduces to itself. The internal choice $T \oplus S$ may silently reduce to either T or S. A contract $c!f.T$ first silently chooses one particular message m that matches f, and then emits it. The operational semantics requires that no name is captured (the substitution resulting from matching must be $\emptyset$) but this follows from validity of output patterns. A contract $c!(a).T$ emits a fresh ($a \neq c$) channel name a and reduces to T. A contract $c?f.T$ is capable of receiving any message m that matches f and reduces to T where all the captured channel names in m are substituted for the corresponding capture variables. We omit the precise definition of substitution, which is standard except that it applies also to free variables occurring in patterns. An external choice $T+S$ offers all visible actions that are offered by either T or S. Both the external choice and the parallel composition are preserved under internal choices of their component contracts. In particular, the rule for $+$ indicates that this operator is a truly external choice. Any visible action emitted by either T or S is also emitted by their parallel composition, provided that the action is different from $\checkmark$ and that no capture of free names occurs. The parallel composition of two contracts is successfully terminated only if both contracts are. Then we have the usual synchronization rules for parallel composition, where pairs of dual actions react and give rise to an internal action. Finally, visible transitions whose labels have names that differ from a restricted one are propagated outside restrictions, and so are invisible transitions. The output of a channel name becomes a connection whenever it escapes its restriction. In the following we write $T \not\longrightarrow$ (respectively $T \overset{\mu}{\not\longrightarrow}$) if there is no T' such that $T \longrightarrow T'$ (respectively $T \xrightarrow{\mu} T'$); if $T \not\longrightarrow$, then we say that T is *stable*; we write $\Longrightarrow$ for the reflexive, transitive closure of $\longrightarrow$; we write $\overset{\mu}{\Longrightarrow}$ for $\Longrightarrow\xrightarrow{\mu}\Longrightarrow$.

Remark 1. Our syntax is redundant insofar as $\mathbf{0}$ and Ω can be encoded respectively as the infinite processes (solution of the equations) $X = X+X$ and $X = X \oplus X$. Both are regular and finite state and, according to the LTS, the former cannot perform any transition while the latter can only perform internal transitions and rewrite to itself. ■

Remark 2. Patterns allow us to capture different flavors of the π-calculus: standard π-calculus is obtained by using variable and singleton patterns in outputs and capture patterns in inputs. Adding singleton patterns in inputs gives us the π-calculus with matching ($[x=y]T$ can be encoded as $(\nu c)(c!x \mid c?y.T)$ for c not free in T); negation adds mismatch (($[x \neq y]T \equiv (\nu c)(c!x \mid c?\neg y.T)$); the polyadic π-calculus is obtained by adding the product constructor to patterns. ■

Example 1. The contract

$$T_{\text{Seller}} \overset{\text{def}}{=} \texttt{store}?(x).x?\texttt{String}.(x!\texttt{error}.\mathbf{1} \oplus x!\texttt{Int}.(x?\texttt{ok}.\texttt{ship}!x.\mathbf{1} + x?\texttt{quit}.\mathbf{1}))$$

describes the behavior of a “seller” process that accepts connections on a public channel `store`. The conversation continues on the private channel x sent by the “buyer”: the seller waits for the name of an item, and it sends back either the value `error` indicating that the item is not in the catalog, or the price of the item. In this case the seller waits for a decision from the buyer. If the buyer answers `ok`, the seller delegates some “shipper” process to conclude the transaction, by sending it the private channel x via the public channel `ship`. If the buyer answers `quit`, the transaction terminates immediately. ■

We now formalize the notion of *well-formed system* and we do so in two steps. First we characterize the compliance of a component T with respect to another component S, namely the fact that the component T is capable of correct termination if composed with another component that behaves according to S; then, we say that a system is well formed if every component of the system is compliant with the rest of the system.

Definition 1 (compliance). *Let* $\#$ *be the least symmetric relation such that* $\checkmark \# \checkmark$, $c!m \# c?m$, *and* $c!(a) \# c?a$. *Let* $[T \mid S]$ *stand for the system* $T \mid S$ *possibly enclosed within restrictions. We say that* T *is* compliant with S, *notation* $T \lhd S$, *if* $T \mid S \Longrightarrow [T' \mid S']$ *and* $T' \not\longrightarrow$ *implies that there exist* μ_1 *and* μ_2 *such that* $\mu_1 \# \mu_2$ *and* $T' \xrightarrow{\mu_1}$ *and* $S' \overset{\mu_2}{\Longrightarrow}$.

According to the definition, T is compliant with S if *every* computation of $T \mid S$ leading to a state where the residual of T is stable is such that either the residual of T has successfully terminated and the residual of S will eventually terminate successfully, or the two residuals can eventually synchronize. The transition labeled by μ_2 is weak because the synchronization may only be available after some time. Notwithstanding this, the availability of μ_2 is guaranteed because compliance quantifies over *every* possible computation. Observe that Ω is compliant with every S (but not viceversa). This is obvious, since Ω denotes indefinite progress without any support from the environment.

Compliance roughly corresponds to the notion of "passing a test" in the classical testing framework [5, 20, 26]: T compliant with S is like saying that S *must* pass the test T. There is an important difference though: in the classical framework, divergence is a catastrophic event that may prevent the test from reaching a successful state. This is sometimes justified as the fact that a diverging component may eat up all the computational power of a system, making the rest of the system starve for progress. In a setting where processes run on different machines/cores this justification is not applicable. Actually, divergence turns out to characterize good systems, those that do have progress. In particular, divergence of the test T is ignored, since it implies that T is making progress autonomously; divergence of the contract S being tested is ignored, in the sense that all the visible actions it provides are guaranteed. For example, we have $c?a.\mathbf{1} \lhd c!a.\mathbf{1} + \Omega$. In this sense, $+$ is a "strongly external" choice (if compared to the external choice in [20, 26]) since it guarantees the visible actions in the converging branches.

Definition 2 (well-formed system). *Let* $\prod_{i\in\{1,\dots,n\}} T_i$ *stand for the system* $T_1 \mid \cdots \mid T_n$, *where* $\prod_{i\in\emptyset} = \mathbf{1}$ *by definition. Let* $S \equiv \prod_{i\in\{1,\dots,n\}} T_i$. *We say that the system* S *is* well formed *if* $T_k \lhd \prod_{i\in\{1,\dots,n\}\setminus\{k\}} T_i$ *for every* $1 \le k \le n$. *We say that* T *and* S *are* dual, *notation* $T \bowtie S$, *if* $T \mid S$ *is a well-formed system.*

For example, we have that $c?a.\mathbf{1} + c?b.\mathbf{1} \mid c!a.\mathbf{1} \oplus c!b.\mathbf{1}$ is well formed but $c?a.\mathbf{1} \mid c!a.\mathbf{1} \oplus c!b.\mathbf{1}$ is not. Similarly, $c!\mathtt{Int}.\mathbf{1} \mid c?\mathtt{Int}.\mathbf{1} + c?\neg\mathtt{Int}.\mathbf{0}$ is well formed but $c!\mathtt{Int} \vee a.\mathbf{1} \mid c?\mathtt{Int}.\mathbf{1} + c?\neg\mathtt{Int}.\mathbf{0}$ is not because $c!\mathtt{Int} \vee a.\mathbf{1} \mid c?\mathtt{Int}.\mathbf{1} + c?\neg\mathtt{Int}.\mathbf{0} \Longrightarrow \mathbf{1} \mid \mathbf{0}$. The systems $\mathbf{1}$ and Ω are trivially well formed: the former has terminated, and hence is compliant with the rest of the system, which is empty and consequently terminated as well; the latter is compliant with every system, and thus with the empty system as well.

Duality is the symmetric version of compliance, namely it considers the success of the test as well as of the contract being tested. Technically this corresponds to restricting the set of tests (or observers) of a contract T to those contracts that not only are satisfied by T but that additionally satisfy T.

Example 2. The contracts

$$\begin{aligned} T_{\text{Buyer}} &\stackrel{\text{def}}{=} \texttt{store}!(c).c!\texttt{String}.(c?\texttt{error}.\mathbf{1} + c?\texttt{Int}.c!\texttt{ok}.c!\texttt{String}.\mathbf{1}) \\ T_{\text{Shipper}} &\stackrel{\text{def}}{=} \texttt{ship}?(x).x?\texttt{String}.\mathbf{1} \end{aligned}$$

describe the behavior of a client that is always willing to buy the requested item regardless of its price, and of a shipper that asks the buyer's address before terminating successfully. The system $T_{\text{Buyer}} \mid T_{\text{Shipper}} \mid T_{\text{Seller}}$ is well formed since the only maximal computation starting from these contracts ends up in $\mathbf{1} \mid \mathbf{1} \mid \mathbf{1}$. ■

Well-formedness is a property of whole systems, but it is of little help when we want to reason about the single components of a system. For example, the contract Ω by itself is a well-formed system and any system in which one of its components has contract $\mathbf{0}$ is clearly ill formed. The contract T_{Buyer} in Example 2 is not a well-formed system by itself, yet it is very different from $\mathbf{0}$: there exist components that, combined with T_{Buyer}, make a well-formed system, while this is impossible for $\mathbf{0}$. In this sense we say that T_{Buyer} is *viable* and $\mathbf{0}$ is not.

Definition 3 (viability). *We say that T is* viable*, notation $T^{\bowtie}$, if $T \bowtie S$ for some S.*

Our notion of viability roughly corresponds to the notion of *well-typed process* in theories of session types. There is a fundamental difference though: a locally well-typed process (in the sense of session types) yields a well-typed system when completed by *any* context that is itself well-typed, whereas for a contract to be viable it suffices to find *one particular* context that can yield a well-formed system. This intuition suggests that well-typedness of a process is a much stricter requirement than viability, and explains why the guarantee of global properties such as system well-formedness comes at the cost of imposing severe constraints on the behavior of the single components. As an example that shows the difference between well-typedness and viability, consider the contracts $T \stackrel{\text{def}}{=} c?a.\mathbf{1} + c?b.\mathbf{0}$ and $S \stackrel{\text{def}}{=} c?a.\mathbf{0} + c?b.\mathbf{1}$. Both are viable when taken in isolation, but their composition is not: an hypothetical third component interacting with $T \mid S$ cannot send $c!b$ because T might read it and reduce to $\mathbf{0}$, and symmetrically it cannot send $c!a$ because S might read it and reduce to $\mathbf{0}$.

Duality induces a semantic notion of equivalence between contracts. Informally, two contracts are equivalent if they have the same duals.

Definition 4 (subcontract). *We say that T is a* subcontract *of S, notation $T \preceq S$, if $T \bowtie R$ implies $S \bowtie R$ for every R. Let $\approx$ be the equivalence relation induced by $\preceq$.*

For example $T \oplus S \preceq T$, namely it is safe to replace a process with a more deterministic one. Then we have $T \preceq \mathbf{1} + T$ and $T \preceq \Omega + T$, namely it is safe to replace a process with another one that, in addition to exposing the behavior of the original process, is also able to immediately terminate with success or is immediately able to make autonomous progress. Observe that $\mathbf{0}$, $\mathbf{1}$, and Ω are pairwise different: $\mathbf{0}$ is the least element of $\preceq$ and $T \approx \mathbf{0}$ means that there is no context that can guarantee progress to T, hence the process with contract T is ill-typed; $\mathbf{1} \not\approx \Omega$ because $\mathbf{1} \bowtie \mathbf{1}$ but $\mathbf{1} \not\bowtie \Omega$. Indeed $\mathbf{1}$ denotes eventual termination, while Ω denotes indefinite progress.

Just as for the π-calculus, it is easy to see that $\preceq$ is a precongruence for action prefixes without bound variables, and we have that $T\sigma \preceq S\sigma$ for every σ such that $\mathtt{dom}(\sigma) = \mathtt{bv}(f)$ implies $c?f.T \preceq c?f.S$. Additionally, $\preceq$ is a precongruence with respect to $\oplus$: it suffices to observe that $R \bowtie T \oplus S$ if and only if $R \bowtie T$ and $R \bowtie S$. It is equally easy to see however that $\preceq$ is not a precongruence with respect to $+$, because of the relation $\mathbf{0} \preceq T$. For example $\mathbf{0} \preceq c?a.\mathbf{0}$, but $c?a.\mathbf{1} + \mathbf{0} \not\preceq c?a.\mathbf{1} + c?a.\mathbf{0}$. This makes it difficult to axiomatize $\preceq$, since it is not possible to replace equals for equals in arbitrary contexts. Furthermore, the usefulness of the relation $\mathbf{0} \preceq T$ is questionable, since it allows the replacement of a deadlocked process with anything else. But if the process is already deadlocked, for sure the system it lives in is ill-formed from the start and it makes little sense to require that the system behaves well after the upgrade. Thus, we will also consider the closure of $\preceq$ with respect to external choice.

Definition 5 (strong subcontract relation). *Let $\sqsubseteq$ be the largest relation included in $\preceq$ that is a precongruence with respect to $+$, namely $T \sqsubseteq S \overset{\text{def}}{\iff} T + R \preceq S + R$ for every R. We write $\simeq$ for the equivalence relation induced by $\sqsubseteq$.*

Unlike $\preceq$, we have $\mathbf{0} \not\sqsubseteq T$ in general, but $\pi.\mathbf{0} \sqsubseteq \pi.T$ does hold. In fact, it is interesting to investigate whether the loss of the law $\mathbf{0} \preceq T$ in particular, and the use of $\sqsubseteq$ instead of $\preceq$ in general, have any significant impact on the theory. The following result shows that this is not the case:

Theorem 1. *If $T^{\bowtie}$ and $T \preceq S$, then $T \sqsubseteq S$.*

Namely, $\preceq$ and $\sqsubseteq$ *may* differ only when the $\preceq$-smaller contract is not viable. In practice, the use of $\sqsubseteq$ over $\preceq$ has no impact: the relation $T \sqsubseteq \mathbf{0}$ completely characterizes non-viable contracts and upgrading, specialization, and searching based on the subcontract relation make sense only when the smaller contract is viable. In fact, $\sqsubseteq$ allows us to reason on the properties of a process by means of its contract:

Proposition 1. *The following properties hold: (1) $T^{\bowtie}$ iff $T \not\sqsubseteq \mathbf{0}$; (2) $\mathbf{1} + T \sqsubseteq T$ iff $T \Longrightarrow T'$ implies $T' \overset{\checkmark}{\Longrightarrow}$; (3) $\Omega + T \sqsubseteq T$ iff $T \Longrightarrow T'$ implies $T' \longrightarrow$.*

Item (1) characterizes viable contracts, and hence describe processes that are well typed. Item (2) characterizes those contracts that, when reaching a stable state, are in a successful state; consequently, the property gives us a sufficient (but not necessary) condition for well-formedness. Item (3) characterizes those contracts that never reach a stable state, and hence describe processes that are capable of making indefinite progress.

3 Typing a Core Language of Sessions

One possible way to assess the expressiveness of the contract language would be to provide a contract-based type system for one of the latest session type calculi in the literature. Unfortunately, this would spoil the simplicity of our approach: existing session type calculi are of increasing complexity and include ad-hoc operators designed to adapt session types to new usage scenarios. Typing such operators with our types would require twisted encodings, bringing back that very kind of complexity that our contracts aim to avoid. For these reasons we prefer to introduce yet another session core calculus, use it to encode examples defined in other papers to motivate the introduction of restrictions and specialized constructs, and finally show that our types allow us to prove progress for these examples (and for more complex ones that escape current session type systems) without resorting to ad-hoc linguistic constructions.

The calculus we propose here—just as a proof-of-concept, not as an object of study—is a streamlined version of several calculi introduced in the literature (eg, [4, 18, 21]) and is defined by the following productions, where t is either `Bool` or `Int` and e ranges over unspecified expressions on these types:

$$\begin{array}{rcl} P & ::= & 0 \mid \alpha!(a).P \mid \alpha!\langle e\rangle.P \mid \alpha?(x:t).P \mid \alpha!(\!|\alpha|\!).P \mid \alpha?(\!|x|\!).P \\ & \mid & \alpha\triangleleft\ell.P \mid \alpha\triangleright\{\ell_i:P_i\}_{i\in I} \mid P|P \mid (\nu a)P \mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P \end{array}$$

For a reader knowledgeable of session types literature the syntax above needs no explanation. It boils down to a π-calculus that explicitly differentiates (channel) names, ranged over by $a,b,c,\dots$ from variables, ranged over by $x,y,z,\dots$ (only the former can be restricted, only the latter can be abstracted) and enriched with specific constructions for sessions, namely actions $\alpha!(\!|\alpha|\!)$ and $\alpha?(\!|x|\!)$ for session delegation[1] and actions $\alpha\triangleleft\ell$ and $\alpha\triangleright\{\ell_i:P_i\}_{i\in I}$ for label-based session branching. The calculus also includes both bound and free outputs, the former being used for session connection, the latter for communication. Infinite behaviour is obtained by considering terms coinductively generated by the productions, in the same way as we did for contracts and with the same restrictions.The semantics of the calculus is defined by an LTS whose most important rules are (see the online extended version for all rules):

$$c!(\!|a|\!).P \xrightarrow{c!a} P \qquad c?(\!|x|\!).P \xrightarrow{c?a} P\{^a/_x\} \qquad c\triangleleft\ell.P \xrightarrow{c!\ell} P$$

$$\frac{a\neq c}{c!(a).P \xrightarrow{c!(a)} P} \qquad \frac{e\downarrow v}{c!\langle e\rangle.P \xrightarrow{c!v} P} \qquad \frac{v:t}{c?(x:t).P \xrightarrow{c?v} P\{^v/_x\}} \qquad \frac{k\in I}{c\triangleright\{\ell_i:P_i\}_{i\in I} \xrightarrow{c?\ell_k} P_k}$$

The typing relation for the process calculus is coinductively defined by the following rules (where $\texttt{Ch}\equiv\neg\neg(x)$ is the type of all channel names and the various ℓ's are reserved names that cannot be restricted or appear as subject of a communication).

$$\text{END}\ \ \Gamma\vdash 0:\mathbf{1} \qquad \text{NAME}\ \ \Gamma\vdash a:\texttt{Ch} \qquad \text{VAR}\ \ \Gamma\vdash x:\Gamma(x)$$

$$\text{F-OUTPUT}\ \frac{\Gamma\vdash\alpha:\texttt{Ch}\qquad\Gamma\vdash e:t\qquad\Gamma\vdash P:T}{\Gamma\vdash\alpha!e.P:\alpha!t.T} \qquad \text{INPUT}\ \frac{\Gamma\vdash\alpha:\texttt{Ch}\qquad\Gamma,x:t\vdash P:T}{\Gamma\vdash\alpha?(x:t).P:\alpha?t.T}$$

$$\text{B-OUTPUT}\ \frac{\Gamma\vdash\alpha:\texttt{Ch}\qquad\Gamma\vdash P:T}{\Gamma\vdash\alpha!(a).P:\alpha!(a).T} \qquad \text{C-SEND}\ \frac{\Gamma\vdash\alpha,\beta:\texttt{Ch}\qquad\Gamma\vdash P:T}{\Gamma\vdash\alpha!(\!|\beta|\!).P:\alpha!\beta.T} \qquad \text{C-RECV}\ \frac{\Gamma\vdash\alpha:\texttt{Ch}\qquad\Gamma,x:\texttt{Ch}\vdash P:T}{\Gamma\vdash\alpha?(\!|x|\!).P:\alpha?(x).T}$$

[1] In our case one should rather speak of “session forwarding” or “session sharing” since, contrarily to session type systems, we do not enforce channel bi-linearity.

$$\text{CHOICE}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma \vdash P : T}{\Gamma \vdash \alpha \triangleleft \ell.P : \alpha!\ell.T} \qquad \text{BRANCH}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma \vdash P_i : T_i}{\Gamma \vdash \alpha \triangleright \{\ell_i : P_i\}_{i\in I} : \sum_{i\in I} \alpha?\ell_i.T_i}$$

$$\text{NEW}\ \frac{\Gamma \vdash P : T}{\Gamma \vdash (\nu a)P : (\nu a)T} \qquad \text{PAR}\ \frac{\Gamma \vdash P : T \qquad \Gamma \vdash Q : S}{\Gamma \vdash P \,|\, Q : T \,|\, S} \qquad \text{IF}\ \frac{\Gamma \vdash e : \mathtt{Bool} \quad \Gamma \vdash P : T \quad \Gamma \vdash Q : S}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q : T \oplus S}$$

The rules are straightforward since they perform a very simple abstraction on the content of communications. Although this typing permits a quite liberal usage of channels, it is sufficient to verify the progress property, as stated by the following theorem:

Theorem 2 (progress). *We say that a process* has succeeded *whenever it contains no action. If* $\vdash P : T$ *and* $\mathbf{1} + T \sqsubseteq T$ *and* $P \stackrel{\tau}{\Longrightarrow} Q \stackrel{\tau}{\not\longrightarrow}$*, then* Q *has succeeded.*

In words, for any residual Q of a process whose contract T satisfies $\mathbf{1} + T \sqsubseteq T$, if Q cannot make further progress ($Q \stackrel{\tau}{\not\longrightarrow}$), then it contains no further actions (i.e., it is a possibly restricted parallel composition of null processes 0) that is to say that all its components have successfully terminated.

We devote the rest of the section to the encoding of a few examples in our process language, some are taken from existing bibliography, others are new. The motivating example of [18] is (in our syntax) the following pair of processes

$$P = a?(\!|x|\!).b?(\!|y|\!).x!3.x?(z : \mathtt{Int}).y!\mathtt{true}.0$$
$$Q = a!(c).b!(d).c?(z : \mathtt{Int}).d?(z' : \mathtt{Bool}).c!5.0$$

The two processes initiate two sessions on (public) channels a and b and they exchange two integers over the former and a boolean value over the latter. The parallel composition of these two processes deadlocks. This composition type-checks in simple session type theories that verify the order of messages in each single session. The system in [18] ensures progress and thus rejects this composition. Progress is enforced by establishing a partial order on sessions and requiring that the usage of session channels respects this order. So in [18] the process P can be composed with $Q' = a!(c).b!(d).c?(z : \mathtt{Int}).c!5.d?(z' : \mathtt{Bool}).0$ since in both processes the communication on b follows all communications on a. In our system the two processes have respectively type:

$$T_P = a?(x).b?(y).x!\mathtt{Int}.x?\mathtt{Int}.y!\mathtt{Bool}.\mathbf{1} \qquad T_Q = a!(c).b!(d).c?\mathtt{Int}.d?\mathtt{Bool}.c!\mathtt{Int}.\mathbf{1}$$

We have $T_P \,|\, T_Q \sqsubseteq \mathbf{0}$, hence the composition does not satisfy progress. In fact we can tell something more: $T_P \,|\, T_Q$ is not viable, namely there is no process, that composed with P and Q, allows us to obtain a well-formed system. As in [18] our system detects that $P \,|\, Q'$ is well formed, since $\mathbf{1} + (T_P \,|\, T_{Q'}) \sqsubseteq T_P \,|\, T_{Q'}$ (we let the reader figure out $T_{Q'}$). Furthermore while the system in [18] rejects the composition of Q with $P' = a?(\!|x|\!).b?(\!|y|\!).x!3.y!\mathtt{true}.x?(z : \mathtt{Int}).0$, since it is not possible to order the actions of the two sessions, our system instead proves progress for this composition too.

Other restrictions introduced for session types to enforce progress are useless in our framework. In particular, we do not require linearity of session channels: a process can still use a channel that it has "delegated" without necessarily jeopardizing progress.

Consider the following two processes $P' = a?(\!|x|\!).b?(\!|y|\!).x!(\!|y|\!).x?(z:\texttt{Int}).y!\texttt{true}.0$ and $Q' = a!(c).b!(d).c?(\!|z|\!).c!5.z?(z':\texttt{Bool}).0$. Although P' uses y after having delegated it, the contract T of $P' \mid Q'$ satisfies $\mathbf{1}+T \sqsubseteq T$. This freedom from linearity constraints instantly enables multi-party sessions, without the need of any dedicated syntax. Consider for instance the "Two Buyers" protocol of [22], which describes a conversation between a seller and two buyers, the first buyer being the initiator of the multi-party session. It can be rendered in our language as follows

$$\begin{aligned}
\text{Seller} &= a?(\!|x|\!).x?(\mathit{title}:\texttt{String}).x!\langle \mathit{quote}\rangle.x!\langle \mathit{quote}\rangle.\\
&\qquad x \triangleright \{\texttt{ok}: x?(\mathit{addr}:\texttt{String}).x!\langle \mathit{date}\rangle.0,\ \texttt{quit}:0\}\\
\text{Buyer1} &= a!(c).a!(\!|c|\!).c!\langle\text{"War and Peace"}\rangle.c?(\mathit{quote}:\texttt{Int}).b!(d).d!\langle \mathit{quote}/2\rangle.0\\
\text{Buyer2} &= a?(\!|x|\!).x?(\mathit{quote}:\texttt{Int}).b?(\!|y|\!).y?(\mathit{contrib}:\texttt{Int}).\texttt{if}\ \mathit{quote}-\mathit{contrib}\leq 99\\
&\qquad \texttt{then}\ x \triangleleft \texttt{ok}.x!\langle \mathit{address}\rangle.x?(d:\texttt{Date}).0\ \texttt{else}\ x\triangleleft\texttt{quit}.0
\end{aligned}$$

Buyer1 initiates the session on the public channel a by broadcasting twice the session channel c. This channel is received by Seller and Buyer2, used by Buyer1 to request a title to Seller, and used by Seller to send the price to both buyers and to conclude the transaction with Buyer2. As for [22] the communication between the two buyers takes place on a separate private channel y initiated on the public channel b. The three agents are really the same as the corresponding ones in [22] except that (i) connections are dyadic and do not have to explicitly state the name of the intended partner and (ii) Seller is not aware of the private channel used by the buyers to communicate together. As [22] our type system: (1) it ensures that the composition of the above agents is well typed since their contracts are

$$\begin{aligned}
\text{Seller} &: a?(x).x?\texttt{String}.x!\texttt{Int}.x!\texttt{Int}.(x?\texttt{ok}.x?\texttt{String}.x!\texttt{Date}.\mathbf{1}+x?\texttt{quit}.\mathbf{1})\\
\text{Buyer1} &: a!(c).a!(\!|c|\!).c!\texttt{String}.c?\texttt{Int}.b!(d).d!\texttt{Int}.\mathbf{1}\\
\text{Buyer2} &: a?(x).x?\texttt{Int}.b?(y).y?\texttt{Int}.(x!\texttt{ok}.x!\texttt{String}.x?\texttt{Date}.\mathbf{1}\oplus x!\texttt{quit}.\mathbf{1})
\end{aligned}$$

and the parallel composition T of these contracts satisfies the law $\mathbf{1}+T \sqsubseteq T$ and (2) it would reject the protocol if the channel c were also used for the inter-buyer communication. Of course the property $\mathbf{1}+T \sqsubseteq T$ ensures that the system formed by the three agents has also progress, as does the system of [4] which extends [22] with progress.

The full expressive power of contracts emerges when specifying recursive protocols. To illustrate the point we encode a well-known variant of the Diffie-Hellman protocol, the Authenticated Group Key Agreement protocol A-GDH.2, defined in [2]. This protocol allows a group of n processes $P_1 \dots P_n$ to share a common authenticated key s_n. The protocol assumes the existence of n channels $c_1 \dots c_n$ each c_i being shared between P_{i-1} and P_i (for the sake of the simplicity we use a bootstrapping process P_0 absent in the original presentation). The original algorithm also assumes that the process P_n shares a secret key K_i with process P_i for $i \in [0,n)$. We spice up the algorithm so that the sharing of these keys is implemented via a private channel that is delegated at each step along the participants:

$$\begin{aligned}
P_0 &= c^1!\langle \mathsf{d}\rangle.c^1?(\!|y|\!).y\triangleleft\texttt{ok}.0\\
P_i &= c^i?(x:\mathsf{data}).c^{i+1}!\langle f(x)\rangle.c^{i+1}?(\!|y|\!).y\triangleleft\texttt{key}.y!\langle \mathsf{K}_i\rangle.y?(s:\texttt{Int}).c^i!(\!|y|\!).0 \qquad {\scriptstyle i\in[1,n)}\\
P_n &= c^n?(x:\mathsf{data}).c^n!(c).Q \qquad \text{with } Q = c\triangleright\{\texttt{ok}:0\,,\ \texttt{key}: c?(k:\texttt{Int}).c!\langle g(x,k)\rangle.Q\}
\end{aligned}$$

In a nutshell, P_0 starts the protocol by sending some data d to P_1; each intermediate process P_i receives some data x from P_{i-1}, uses this data to compute new data $f(x)$ that it forwards to P_{i+1}; the terminal process P_n receives the data x from P_{n-1}, generates a new private channel c used for retrieving from every process P_i the key K_i and sending back an integer $g(x,\mathsf{K}_i)$ (from which P_i can deduce the key $\mathsf{s_n}$). Each intermediate process delegates the channel y to its predecessor and P_0 notifies to P_n the successful termination of the protocol (refer to Figures 2 and 3 of [2] for precise definitions of f and g).

The algorithm is quite complex to analyse for at least two reasons: (i) P_n is recursively defined since it does not know *a priori* how many processes take part in the protocol (actually it statically knows just the existence of channel c^n), and (ii) all the shared keys are transmitted over the same channel c (generated in the second action of P_n) which constitutes a potential source of interference that may hinder progress.

All the session type systems that enforce progress we are aware of fail to type check the above protocol. In particular the progress type systems for dyadic sessions [18] and multi-party ones [4] fail to type P_i because, as long as the c^i are considered private channels, it performs an output on a channel c^i in the continuation of the reception of a delegation; also, the protocol ends by emitting `ok` on the channel y, but every P_i process uses c^i before and after a synchronization on y, so it is not possible to find a well-founded order on y and the various c^i's since there is a double alternation of actions on y and c^i. Likewise, conversation types [11] can type all the processes of the protocol but progress cannot be ensured because there is no well-founded order for channels. As a matter of facts, in all these works progress is enforced by the techniques introduced by Kobayashi [23, 24] where types are inhabited by channels on which a well-founded usage (capability/obligation, in Kobayashi's terminology) order can be established. By inhabiting types by processes we escape the need of such an order and thus can type the processes of the protocol.

Classical Sessions Typing. The type system presented earlier in the section is enough to ensure the progress property stated in Theorem 2. However, in some contexts it may be desirable to enforce a stricter typing discipline in order to impose a particular communication model. Let us clarify this point with an example. Consider the system $c?(x:\mathtt{Int}).0 \mid c?(x:\mathtt{Bool}).0 \mid c!\langle 3\rangle.0 \mid c!\langle \mathtt{true}\rangle.0$ which is composed of four processes, two of which are sending on the channel c messages with different types (`Int` and `Bool`) while the remaining processes are waiting for two messages on the channel c (of type `Int` and `Bool`). Its contract is $S \stackrel{\text{def}}{=} c?\mathtt{Int}.\mathbf{1} \mid c?\mathtt{Bool}.\mathbf{1} \mid c!\mathtt{Int}.\mathbf{1} \mid c!\mathtt{Bool}.\mathbf{1}$ which satisfies $\mathbf{1}+S \sqsubseteq S$. Namely, the above system is free from communication errors despite it contains processes that are sending and receiving messages of different types on the same channel at the same time. Technically this happens because the operational semantics of contracts (and of processes) does not distinguish between *communication* and *matching*. In practice, the typing rules we have given reflect a particular communication model: a receiver waits until a message *of the expected type* is available. Thus, a well-formed system is such that any process that is blocked waiting for a message will *eventually* read a message of the expected type, and any process that is blocked trying to send a message will *eventually* deliver it to someone who is able to handle it.

This communication model is not the only reasonable one, especially in a distributed setting where asynchrony decouples communication from the ability to inspect the content of messages. In this setting, it could be reasonable to assume that a process waiting for messages sent on some channel c should be ready to handle *any* message sent on that channel at that particular time. Somewhat surprisingly, this communication model can be implemented without any change to the operational semantics of contracts and processes, but merely adjusting a few typing rules, those that deal with input actions:

$$\text{INPUT}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma, x : t \vdash P : T}{\Gamma \vdash \alpha?(x:t).P : \alpha?t.T + \alpha?\neg t.\mathbf{0}} \qquad \text{C-RECV}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma, x : \mathtt{Ch} \vdash P : T}{\Gamma \vdash \alpha?(\!|x|\!).P : \alpha?(x).T + \alpha?\neg\mathtt{Ch}.\mathbf{0}}$$

$$\text{BRANCH}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma \vdash P_i : T_i}{\Gamma \vdash \alpha \rhd \{\ell_i : P_i\}_{i\in I} : \sum_{i\in I} \alpha?\ell_i.T_i + \alpha?\bigwedge_{i\in I}\neg\ell_i.\mathbf{0}}$$

Intuitively, the contract of a process waiting for messages states that the process is capable of receiving *any* message, but only those of some particular type will allow the process to continue. If the process receives a message that it is not able to handle, the process deadlocks, therefore compromising well-formedness of the system it belongs to. With these typing rules in place, we are making the assumption that at each step of a computation any channel is implicitly associated with a unique type of its messages (i.e., a set of labels, a set of values, a finite set of channel names, or a combination of these in case the language provides for boolean operators over types) and that every process that at that step waits for messages on that channel must be able to handle *every* message that is or may be sent on it. With the modified typing rules, the above system is typed by the contract $S' \overset{\text{def}}{=} c?\mathtt{Int}.\mathbf{1} + c?\neg\mathtt{Int}.\mathbf{0} \mid c?\mathtt{Bool}.\mathbf{1} + c?\neg\mathtt{Bool}.\mathbf{0} \mid c!\mathtt{Int}.\mathbf{1} \mid c!\mathtt{Bool}.\mathbf{1}$ and it is immediate to see that this system no longer enjoys progress, since $S' \sqsubseteq \mathbf{0}$.

Interestingly, this modified typing discipline gives rise to the same subtyping relation as the one defined by Gay and Hole for session types [19]. In particular, we have contravariance for outputs, covariance for inputs, and width subtyping for branching. More precisely if we define the subtyping relation for patterns (and thus for types) as $f \leq g \overset{\text{def}}{=} [\![f]\!] \subseteq [\![g]\!]$, then it is not difficult to verify the soundness of the rules

$$\frac{f \leq g \qquad T \sqsubseteq S}{c!g.T \sqsubseteq c!f.S} \qquad \frac{f \leq g \qquad T \sqsubseteq S}{c?f.T + c?\neg f.\mathbf{0} \sqsubseteq c?g.S + c?\neg g.\mathbf{0}}$$

The two rules state that the implicit type of a channel (in the sense of session types: we do not assign any type to channels) can be contravariantly specialized for emission actions (F-OUTPUT), (C-SEND), (CHOICE) and covariantly specialized for reception actions (INPUT), (C-RECV), (BRANCH). For instance, when in the system above we deduce for the process $c?(x:t).P$ the contract $c?t.T + c?\neg t.\mathbf{0}$ we are implicitly assuming that the channel c in the system of Gay and Hole would have type $?[t].T'$ for some T' (which would be the projection of T over c); the rule on the right hand side states that it can be safely replaced by a channel whose (implicit) type is $?[s].T'$, provided that $t \leq s$.

The language proposed in this section is just an example of how our contracts can be used. But one can, and indeed should, imagine to use them to type advanced process calculi with, for instance, type driven synchronization (e.g., the language PiST for sessions defined in [13]) or first-class processes (contracts being assigned to processes, a typed calculus with higher-order processes looks as a natural next step).

4 Related Work and Conclusions

The latest works on session types witness a general trend to use more and more informative types, a trend that makes these approaches closer to the techniques of specification refinement. Here we push this trend to an extreme. Contracts are behavioral types that accurately capture the behavior of participants in a conversation by providing a relatively shallow abstraction over processes that respect them. We are at the edge between behavioral types, specification refinements, and abstract interpretation: contracts record the flow of communications only when channels are passed around and they abstract communication content into patterns; values are abstracted into possibly infinite set of values (i.e., types) and names into finite sets of names (i.e., valid output patterns). Inasmuch as shallow this abstraction is, it is enough to define a theory of contracts (whose comparison with testing theories was discussed in Sections 1 and 2) that allows us to reason *effectively* (see the online extended version for decidability results) about the correctness of systems and about the safe substitutability and well-typedness of components of a system: it makes us switch from undecidability to decidability. A similar approach is followed by [9], where the content of messages can be made opaque and thus abstracted; this is closer to, though grosser than, the refinement approach. The crucial difference between our approach and all other theories of contracts, [9] included, is that we keep track of names passed around in communications.

We discussed technical differences between contracts and session types all the presentation long. The key point is that contracts record the overall behavior of a process, while session types project this behavior on the various private channels the process uses. Providing partial views of the behavior makes session types more readable and manageable. At the same time it hinders their use in enforcing global properties—notably, progress—whence the need for awkward restrictions such as channel linearity, controlled nesting, scarce session interleaving, and global order on channel usage. In practice, the two approaches have both pros and cons. The contract approach is "optimistic" in that a process is considered well typed as long as there exists at least one context that composed with it yields a well-formed system; session types, instead, account for the nastier possible context. Thus, the contract-based approach widens the spectrum of well-typed (viable) processes but, as we have seen, the composition of viable processes is not necessarily viable. This implies that to prove viability of a parallel composition of processes our approach requires a global system analysis that effectively enumerates all control-flow paths, whereas session types allow each process component to be verified independently. However, if a process is not viable, then it is easy to exhibit a trace of actions that leads the process into an error state by looking at its contract. The session-based approach restricts the set of well-typed processes, but makes it easy to compose them. However, if the type checker detects an ill-typed process, it may be unable to provide any sensible information to help the programmer fix the problem.

An interesting, related approach is the one of Conversation Types [11]. Conversation Types are close to contracts, inasmuch as types provide a global and unique description of the processes involved in a composition. The difference resides mainly in the formalism used to describe the behavior: contracts stick as close as possible to the π-calculus, while Conversation Types draw their inspiration from the structural features of spatial logic [10] and Boxed Ambients [8] by organizing behavior around places of conversation (which thus generalize sessions) and describing communications relatively to them (local vs. external). As in our case the approach of Conversation Types is optimistic and does not require awkward usage conditions: for instance Conversation Types allow processes to still use a channel after having delegated it, as for contracts. Conversation Types do not ensure progress, which is instead enforced via an auxiliary deduction system that exploits an order relation on channels. In this respect Conversation Types seem more basic than contracts. The authors say that the Conversation Calculus they type can be seen as a π-calculus with labels and patterns, as our calculus is. It will be interesting to check whether/how contracts can type the conversation calculus and deduce an in-depth comparison of the two approaches. We leave this as future work.

A problem that is connected with but orthogonal to the ones studied here is the global specification of choreographies. Contracts are subjective descriptions of (components of) systems, but cannot be used to give a global specification of a choreography to which every acceptable implementation (decomposition) must conform. As a matter of facts, in our theory a closed, well-formed system is typed by either $\mathbf{1}$ or Ω. There exist two main proposals of languages for high-level specification of the structure of conversation within a choreography against which the components of the choreography must be validated. The first proposal, issued from the literature of session types, is based on the definition of "global types" that describe the structure of conversation by listing for each session the sender, receiver and content of each communication [22, 27]. The second proposal is directly embedded into conversation types, since they describe the global structure of the conversation by providing a set of traces in which internal transitions are labeled by the synchronization that generated them. Whether these two approaches fit our setting is matter of future research.

Alternative characterizations of viability and of the subcontract relations, as well as the proof system for $\sqsubseteq$, shed light on important aspects of the theory, yet we had to omit them because of the page limit: they can be found in the online extended version. We are currently extending the completeness proof of the deduction system to an algorithm for deciding $\sqsubseteq$. In this respect, the constraint of working with regular (hence finite-state) contracts plays a crucial role and may prove fundamental in comparing the expressive power of our theory with alternative theories that share common goals. The exact implications of this constraint are currently unclear and subject of in-depth investigations.

Acknowledgments. Work partially supported by the project ANR-08-EMER-004-04. We thank Mariangiola Dezani-Ciancaglini, Hugo Vieira, and Nobuko Yoshida for insightful discussions.

References

1. Aceto, L., Hennessy, M.: Termination, deadlock, and divergence. J. ACM 39(1) (1992)
2. Ateniese, G., Steiner, M., Tsudik, G.: Authenticated group key agreement and friends. In: CCS 1998: 5th ACM conference on Computer and Communications Security (1998)

3. Bettini, L., Capecchi, S., Dezani-Ciancaglini, M., Giachino, E., Venneri, B.: Session and Union Types for Object Oriented Programming. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 659–680. Springer, Heidelberg (2008)
4. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
5. Boreale, M., De Nicola, R.: Testing equivalence for mobile processes. Inf. Comput. 120(2), 279–303 (1995)
6. Bravetti, M., Zavattaro, G.: Contract based multi-party service composition. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 207–222. Springer, Heidelberg (2007)
7. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
8. Bugliesi, M., Castagna, G., Crafa, S.: Access control for mobile agents: The Calculus of Boxed Ambients. ACM TOPLAS 26(1), 57–124 (2004)
9. Buscemi, M.G., Melgratti, H.: Abstract processes in orchestration languages. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 301–315. Springer, Heidelberg (2009)
10. Caires, L.: Spatial-behavioral types for concurrency and resource control in distributed systems. TCS 402(2-3), 120–141 (2008)
11. Caires, L., Vieira, H.: Conversation types. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 285–300. Springer, Heidelberg (2009)
12. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A Formal Account of Contracts for Web Services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Heidelberg (2006)
13. Castagna, G., Dezani-Ciancaglini, M., Giachino, E., Padovani, L.: Foundation of session types (unpublished manuscript) (available on line) (January 2009)
14. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for Web Services. ACM TOPLAS 31(5) (2009)
15. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N.: Asynchronous Session Types and Progress for Object-Oriented Languages. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, Springer, Heidelberg (2007)
16. Courcelle, B.: Fundamental properties of infinite trees. Theor. Comput. Sci. 25, 95–169 (1983)
17. De Nicola, R., Hennessy, M.: CCS without τ's. In: Ehrig, H., Levi, G., Montanari, U. (eds.) CAAP 1987 and TAPSOFT 1987. LNCS, vol. 249, pp. 138–152. Springer, Heidelberg (1987)
18. Dezani-Ciancaglini, M., de' Liguoro, U., Yoshida, N.: On progress for structured communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)
19. Gay, S., Hole, M.: Subtyping for session types in the π-calculus. Acta Inf. 42(2-3) (2005)
20. Hennessy, M.: Algebraic Theory of Processes. Foundation of Computing. MIT Press, Cambridge (1988)
21. Honda, K., Vasconcelos, V., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, p. 122. Springer, Heidelberg (1998)
22. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008: the 35th annual ACM Symp. on Principles of Programming Languages (2008)
23. Kobayashi, N.: A type system for lock-free processes. Inf. Comput. 177(2), 122–159 (2002)
24. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)

25. Laneve, C., Padovani, L.: The *must* preorder revisited – An algebraic theory for web services contracts. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)
26. Hennessy, M.: A fully abstract denotational semantics for the pi-calculus. Theor. Comput. Sci. 278(37), 53–89 (2002)
27. Mostrous, D., Yoshida, N., Honda, K.: Global principal typing in partially commutative asynchronous sessions. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 316–332. Springer, Heidelberg (2009)
28. Padovani, L.: Contract-directed synthesis of simple orchestrators. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 131–146. Springer, Heidelberg (2008)
29. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science 6(5), 409–453 (1996)
30. Vasconcelos, V.T., Ravara, A., Gay, S.J.: Session types for functional multithreading. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 497–511. Springer, Heidelberg (2004)

Power of Randomization in Automata on Infinite Strings

Rohit Chadha, A. Prasad Sistla, and Mahesh Viswanathan

[1] Univ. of Illinois, Urbana-Champaign
[2] Univ. of Illinois, Chicago

Abstract. Probabilistic Büchi Automata (PBA) are randomized, finite state automata that process input strings of infinite length. Based on the threshold chosen for the acceptance probability, different classes of languages can be defined. In this paper, we present a number of results that clarify the power of such machines and properties of the languages they define. The broad themes we focus on are as follows. We precisely characterize the complexity of the emptiness, universality, and language containment problems for such machines, answering canonical questions central to the use of these models in formal verification. Next, we characterize the languages recognized by PBAs topologically, demonstrating that though general PBAs can recognize languages that are not regular, topologically the languages are as simple as ω-regular languages. Finally, we introduce Hierarchical PBAs, which are syntactically restricted forms of PBAs that are tractable and capture exactly the class of ω-regular languages.

1 Introduction

Automata on infinite (length) strings have played a central role in the specification, modeling and verification of non-terminating, reactive and concurrent systems [8, 10, 17, 20, 21]. However, there are classes of systems whose behavior is probabilistic in nature; the probabilistic behavior being either due to the employment of randomization in the algorithms executed by the system or due to other uncertainties in the system, such as failures, that are modeled probabilistically. While Markov Chains and Markov Decision Processes have been used to model such behavior in the formal verification community [15], both these models do not adequately capture *open*, *reactive* probabilistic systems that continuously accept inputs from an environment. The most appropriate model for such systems are probabilistic automata on infinite strings, which are the focus of study in this paper.

Probabilistic Büchi Automata (PBA) have been introduced in [3] to capture such computational devices. These automata generalize probabilistic finite automata (PFA) [12, 14, 16] from finite length inputs to infinite length inputs. Informally, PBA's are like finite-state automata except that they differ in two respects. First, from each state and on each input symbol, the PBA may roll a dice to determine the next state. Second, the notion of acceptance is different because PBAs are probabilistic in nature and have infinite length input strings. The behavior of a PBA on a given infinite input string can be captured by an infinite Markov chain that defines a probability measure on the space of runs/executions of the machine on the given input. Like Büchi automata, a run is considered to be accepting if some accepting state occurs infinitely often, and therefore, the probability of acceptance of the input is defined to be the measure of all accepting runs on the given input. There are two possible languages that one can associate with a PBA $\mathcal{B}$ [2, 3] — $\mathcal{L}_{>0}(\mathcal{B})$ (called *probable semantics*) consisting of all strings whose probability of acceptance is non-zero, and $\mathcal{L}_{=1}(\mathcal{B})$ (called *almost sure semantics*) consisting all strings whose probability of acceptance is 1. Based on these two languages, one can define two classes of languages — $\mathbb{L}(\mathrm{PBA}^{>0})$, and $\mathbb{L}(\mathrm{PBA}^{=1})$ which are the collection of all languages (of infinite length strings) that can be accepted by some PBA with respect to probable, and almost sure semantics, respectively. In this paper we study the expressive power of, and decision problems for these classes of languages.

We present a number of new results that highlight three broad themes. First, we establish the precise complexity of the canonical decision problems in verification, namely, emptiness, universality, and language containment, for the classes $\mathbb{L}(\mathrm{PBA}^{>0})$ and $\mathbb{L}(\mathrm{PBA}^{=1})$. For the decision problems, we focus our attention on RatPBAs which are PBAs in which all transition probabilities are rational. First we show the problem of checking emptiness of the language $\mathcal{L}_{=1}(\mathcal{B})$ for a RatPBA $\mathcal{B}$ is **PSPACE**-complete, which substantially improves the result of [2] where it was shown to be decidable in **EXPTIME** and conjectured to be **EXPTIME**-hard. This upper bound is established by observing that the complement of the language $\mathcal{L}_{=1}(\mathcal{B})$ is recognized by a special PBA $\mathcal{M}$ (with probable semantics) called a *finite state probabilistic monitor* (FPM) [4, 6] and then exploiting a result in [6] that shows that the language of an FPM is non-empty if and only if there is an *ultimately periodic word* in the language. This observation of the existence of ultimately periodic words does not carry over to the class $\mathbb{L}(\mathrm{PBA}^{>0})$. However, we show that $\mathcal{L}_{>0}(\mathcal{B})$, for a RatPBA $\mathcal{B}$, is non-empty iff it contains a *strongly asymptotic word*, which is a generalization of ultimately periodic word. This allows us to show that the emptiness problem for $\mathbb{L}(\mathrm{PBA}^{>0})$, though undecidable as originally shown in [2], is Σ_2^0-complete, where Σ_2^0 is a set in the second level of the arithmetic hierarchy. Next we show that the universality problems for $\mathbb{L}(\mathrm{PBA}^{>0})$ and $\mathbb{L}(\mathrm{PBA}^{=1})$ are also Σ_2^0-complete and **PSPACE**-complete, respectively. Finally, we show that for both $\mathbb{L}(\mathrm{PBA}^{>0})$ and $\mathbb{L}(\mathrm{PBA}^{=1})$, the language containment problems are Σ_2^0-complete. This is a surprising observation — given that emptiness and universality are both in **PSPACE** for $\mathbb{L}(\mathrm{PBA}^{=1})$, one would expect language containment to be at least decidable.

The second theme brings to sharper focus the correspondence between nondeterminism and probable semantics, and between determinism and almost sure semantics, in the context of automata on infinite words. This correspondence was hinted at in [2]. There it was observed that $\mathbb{L}(\mathrm{PBA}^{=1})$ is a strict subset of $\mathbb{L}(\mathrm{PBA}^{>0})$ and that while Büchi, Rabin and Streett acceptance conditions all yield the same class of languages under the probable semantics, they yield different classes of languages under the almost sure semantics. These observations mirror the situation in non-probabilistic automata — languages recognized by deterministic Büchi automata are a strict subset of the class of languages recognized by nondeterministic Büchi automata, and while Büchi, Rabin and Streett acceptances are equivalent for nondeterministic machines, Büchi acceptance is strictly weaker than Rabin and Streett for deterministic machines. In this paper we further strenghten this correspondence through a number of results on the closure properties as well as the topological structure of $\mathbb{L}(\mathrm{PBA}^{>0})$ and $\mathbb{L}(\mathrm{PBA}^{=1})$.

First we consider closure properties. It was shown in [2] that the class $\mathbb{L}(\mathrm{PBA}^{>0})$ is closed under all the Boolean operations (like the class of languages recognized by nondeterministic Büchi automata) and that $\mathbb{L}(\mathrm{PBA}^{=1})$ is not closed under complementation. We show that $\mathbb{L}(\mathrm{PBA}^{=1})$ is, however, closed under intersection and union, just like the class of languages recognized by deterministic Büchi automata. We also show that every language in $\mathbb{L}(\mathrm{PBA}^{>0})$ is a Boolean combination of languages in $\mathbb{L}(\mathrm{PBA}^{=1})$, exactly like every ω-regular language (or languages recognized by nondeterministic Büchi machines) is a Boolean combination of languages recognized by deterministic Büchi machines. Next, we characterize the classes topologically. There is natural topological space on infinite length strings called the *Cantor topology* [18]. We show that, like ω-regular languages, all the classes of languages defined by PBAs lie in very low levels of this Borel hierarchy. We show that $\mathbb{L}(\mathrm{PBA}^{=1})$ is strictly contained in $\mathcal{G}_\delta$, just like the class of languages recognized by deterministic Büchi is strictly contained in $\mathcal{G}_\delta$. From these results, it follows that $\mathbb{L}(\mathrm{PBA}^{>0})$ is in the Boolean closure of $\mathcal{G}_\delta$ much like the case for ω-regular languages.

The last theme identifies syntactic restrictions on PBAs that capture regularity. Much like PFAs for finite word languages, PBAs, though finite state, allow one to recognize non-regular languages. It has been shown [2, 3] that both $\mathbb{L}(\mathrm{PBA}^{>0})$ and $\mathbb{L}(\mathrm{PBA}^{=1})$ contain non-ω-regular languages. A question initiated in [3] was to identify restrictions on PBAs that ensure that PBAs have the same expressive power as finite-state (non-probabilistic) machines. One such restriction was identified in [3], where it was shown that *uniform* PBAs with respect to the probable semantics capture exactly the class of ω-regular languages. However, the uniformity condition identified by Baier et. al. was semantic in nature. In this paper, we identify one simple syntactic restriction that capture regularity both for probable semantics and almost sure semantics. The restriction we consider is that of a hierarchical structure. A *Hierarchical* PBA *(*HPBA*)* is a PBA whose states are partitioned into different levels such that, from any state q, on an input symbol a, at most one transition with non-zero probability goes to a state at the same level as q and all others go to states at higher level. We show that HPBA with respect to probable semantics define exactly the class of ω-regular languages, and with respect to almost sure semantics define exactly the class of ω-regular languages in $\mathbb{L}(\mathrm{PBA}^{=1})$, namely, those recognized by deterministic Büchi automata. Next, HPBAs not only capture the notion of regularity, they are also very tractable. We show that the emptiness and universality problems for HPBA with probable semantics are **NL**-complete and **PSPACE**-complete, respectively; for almost sure semantics, emptiness is **PSPACE**-complete and universality is **NL**-complete. This is interesting because this is the exact same complexity as that for (non-probabilistic) Büchi automata. In contrast, the emptiness problem for uniform PBA has been shown to be in **EXPTIME** and co-**NP**-hard [3]; thus, they seem to be less tractable than HPBA.

The rest of the paper is organized as follows. After discussing closely related work, we start with some preliminaries (in Section 2) before introducing PBAs. We present our results about the probable semantics in Section 3, and almost sure semantics in

Section 4. Hierarchical PBAs are introduced in Section 5, and conclusions are presented in Section 6. In the interest of space, some proofs have been deferred to [5].

Related Work. Probabilistic Büchi automata (PBA), introduced in [3], generalize the model of Probabilistic Finite Automata [12, 14, 16] to consider inputs of infinite length. In [3], Baier and Größer only considered the probable semantics for PBA. They also introduced the model of uniform PBAs to capture ω-regular languages and showed that the emptiness problem for such machines is in **EXPTIME** and co-**NP**-hard. The almost sure semantics for PBA was first considered in [2] where a number of results were established. It was shown that $\mathbb{L}(\mathrm{PBA}^{>0})$ are closed under all Boolean operations, $\mathbb{L}(\mathrm{PBA}^{=1})$ is strictly contained in $\mathbb{L}(\mathrm{PBA}^{>0})$, the emptiness problem for $\mathbb{L}(\mathrm{PBA}^{>0})$ is undecidable, and the emptiness problem of $\mathbb{L}(\mathrm{PBA}^{=1})$ is in **EXPTIME**. We extend and sharpen the results of this paper. In a series of previous papers [4, 6], we considered a special class of PBAs called FPMs (Finite state Probabilistic Monitors) whose accepting set of states consists of all states excepting a rejecting state which is also absorbing. There we proved a number of results on the expressiveness and decidability/complexity of problems for FPMs. We draw on many of these observations to establish new results for the more general model of PBAs.

2 Preliminaries

We assume that the reader is familiar with arithmetical hierarchy. The set of natural numbers will be denoted by $\mathbb{N}$, the closed unit interval by $[0, 1]$ and the open unit interval by $(0, 1)$. The power-set of a set X will be denoted by 2^X.

Sequences. Given a finite set S, $|S|$ denotes the cardinality of S. Given a sequence (finite or infinite) $\kappa = s_0, s_1, \ldots$ over S, $|\kappa|$ will denote the length of the sequence (for infinite sequence $|\kappa|$ will be ω), and $\kappa[i]$ will denote the ith element s_i of the sequence. As usual S^* will denote the set of all finite sequences/strings/words over S, S^+ will denote the set of all finite non-empty sequences/strings/words over S and S^ω will denote the set of all infinite sequences/strings/words over S. Given $\eta \in S^*$ and $\kappa \in S^* \cup S^\omega$, $\eta\kappa$ is the sequence obtained by concatenating the two sequences in order. Given $\mathsf{L}_1 \subseteq \Sigma^*$ and $\mathsf{L}_2 \subseteq \Sigma^\omega$, the set $\mathsf{L}_1\mathsf{L}_2$ is defined to be $\{\eta\kappa \mid \eta \in \mathsf{L}_1 \text{ and } \kappa \in \mathsf{L}_2\}$. Given natural numbers $i, j \leq |\kappa|$, $\kappa[i : j]$ is the finite sequence $s_i, \ldots s_j$, where $s_k = \kappa[k]$. The set of *finite prefixes* of κ is the set $\mathit{Pref}(\kappa) = \{\kappa[0, j] \mid j \in \mathbb{N}, j \leq |\kappa|\}$.

Languages of infinite words. A language L of infinite words over a finite alphabet Σ is a subset of Σ^ω. (Please note we restrict only to finite alphabets). A set of languages of infinite words over Σ is said to be a class of languages of infinite words over Σ. Given a class $\mathcal{L}$, the Boolean closure of $\mathcal{L}$, denoted $\mathsf{BCl}(\mathcal{L})$, is the smallest class containing $\mathcal{L}$ that is closed under the Boolean operations of complementation, union and intersection.

Automata and ω-regular Languages. A *finite automaton on infinite words*, $\mathcal{A}$, over a (finite) alphabet Σ is a tuple (Q, q_0, F, Δ), where Q is a finite set of states, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and F defines the accepting condition. The nature of F depends on the type of automaton we are considering; for a *Büchi automaton* $F \subseteq Q$, while for a *Rabin automaton* F is a finite subset of $2^Q \times 2^Q$.

If for every $q \in Q$ and $a \in \Sigma$, there is exactly one q' such that $(q, a, q') \in \Delta$ then $\mathcal{A}$ is called a *deterministic* automaton. Let $\alpha = a_0, a_1, \ldots$ be an infinite string over Σ. A *run* r *of* $\mathcal{A}$ on α is an infinite sequence $s_0, s_1, \ldots$ over Q such that $s_0 = q_0$ and for every $i \geq 0$, $(s_i, a_i, s_{i+1}) \in \Delta$. The notion of an *accepting run* depends on the type of automaton we consider. For a Büchi automaton r is accepting if some state in F appears infinitely often in r. On the other hand for a Rabin automaton, r is accepting if it satisfies the *Rabin acceptance condition*— there is some pair $(B_i, G_i) \in F$ such that all the states in B_i appear only finitely many times in r, while at least one state in G_i appears infinitely many times. The automaton $\mathcal{A}$ *accepts* the string α if it has an accepting run on α. The *language accepted (recognized) by* $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of strings that $\mathcal{A}$ accepts. A language $\mathsf{L} \subseteq \Sigma^\omega$ is called *ω-regular* iff there is some Büchi automata $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathsf{L}$. In this paper, given a fixed alphabet Σ, we will denote the class of ω-regular languages by $\mathsf{Regular}$. It is well-known that unlike the case of finite automata on finite strings, deterministic Büchi automata are less powerful than nondeterministic Büchi automata. On the other hand, nondeterministic Rabin automata and deterministic Rabin automata have the expressive power and they recognize exactly the class $\mathsf{Regular}$. Finally, we will sometimes find it convenient to consider automata $\mathcal{A}$ that do not have finitely many states. We will say that a language L is *deterministic* iff it can be accepted by a deterministic Büchi automaton that does not necessarily have finitely many states. We denote by $\mathrm{Deterministic}$ the collection of all deterministic languages. Please note that the class $\mathrm{Deterministic}$ strictly contains the class of languages recognized by finite state deterministic Büchi automata. The following are well-known results [13, 18].

Proposition 1. $\mathsf{L} \in \mathsf{Regular} \cap \mathrm{Deterministic}$ iff there is a finite state deterministic Büchi automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathsf{L}$. Furthermore, $\mathsf{Regular} \cap \mathrm{Deterministic} \subsetneq \mathsf{Regular}$ and $\mathsf{Regular} = \mathsf{BCl}(\mathsf{Regular} \cap \mathrm{Deterministic})$.

Topology on infinite strings. The set Σ^ω comes equipped with a natural topology called the *Cantor topology*. The collection of open sets is the collection $\mathcal{G} = \{\mathsf{L}\Sigma^\omega \mid \mathsf{L} \subseteq \Sigma^+\}$[1]. The collection of closed sets, $\mathcal{F}$, is the collection of *prefix-closed sets* — L is prefix-closed if for every infinite string α, if every prefix of α is a prefix of some string in L, then α itself is in L. In the context of verification of reactive systems, closed sets are also called *safety languages* [1, 11]. One remarkable result in automata theory is that the class of languages $\mathcal{G}_\delta$ coincides exactly with the class of languages recognized by infinite-state deterministic Büchi automata [13, 18]. This combined with the fact that the class of ω-regular languages is the Boolean closure of ω-regular deterministic Büchi automata yields that the class of ω-regular languages is strictly contained in $\mathsf{BCl}(\mathcal{G}_\delta)$ which itself is strictly contained in $\mathcal{G}_{\delta\sigma} \cap \mathcal{F}_{\sigma\delta}$ [13, 18].

Proposition 2. $\mathcal{G}_\delta = \mathrm{Deterministic}$, and $\mathsf{Regular} \subsetneq \mathsf{BCl}(\mathcal{G}_\delta) \subsetneq \mathcal{G}_{\delta\sigma} \cap \mathcal{F}_{\sigma\delta}$.

2.1 Probabilistic Büchi Automata

We shall now recall the definition of probabilistic Büchi automata given in [3]. Informally, PBA's are like finite-state deterministic Büchi automata except that the transition

[1] This topology is also generated by the metric $d : \Sigma^\omega \times \Sigma^\omega \to [0, 1]$ where $d(\alpha, \beta)$ is 0 iff $\alpha = \beta$; otherwise it is $\frac{1}{2^i}$ where i is the smallest integer such that $\alpha[i] \neq \beta[i]$.

function from a state on a given input is described as a probability distribution that determines the probability of the next state. PBAs generalize the probabilistic finite automata (PFA) [12, 14, 16] on finite input strings to infinite input strings. Formally,

Definition: A *finite state probabilistic Büchi automata (*PBA*)* over a finite alphabet Σ is a tuple $\mathcal{B} = (Q, q_s, Q_f, \delta)$ where Q is a finite set of *states*, $q_s \in Q$ is the *initial state*, $Q_f \subseteq Q$ is the set of *accepting/final states*, and $\delta : Q \times \Sigma \times Q \rightarrow [0, 1]$ is the *transition relation* such that for all $q \in Q$ and $a \in \Sigma$, $\sum_{q' \in Q} \delta(q, a, q') = 1$. In addition, if $\delta(q, a, q')$ is a rational number for all $q, q' \in Q, a \in \Sigma$, then we say that $\mathcal{M}$ is a rational probabilistic Büchi automata (RatPBA).

Notation: The transition function δ of PBA $\mathcal{B}$ on input a can be seen as a square matrix δ_a of order $|Q|$ with the rows labeled by "current" state, columns labeled by "next state" and the entry $\delta_a(q, q')$ equal to $\delta(q, a, q')$. Given a word $u = a_0 a_1 \dots a_n \in \Sigma^+$, δ_u is the matrix product $\delta_{a_0}\delta_{a_1}\dots\delta_{a_n}$. For an empty word $\epsilon \in \Sigma^*$ we take δ_ϵ to be the identity matrix. Finally for any $Q_0 \subseteq Q$, we say that $\delta_u(q, Q_0) = \sum_{q' \in Q_0} \delta_u(q, q')$. Given a state $q \in Q$ and a word $u \in \Sigma^+$, $\mathsf{post}(q, u) = \{q' \mid \delta_u(q, q') > 0\}$.

Intuitively, the PBA starts in the initial state q_s and if after reading $a_0, a_1 \dots, a_n$ results in state q, then it moves to state q' with probability $\delta_{a_{i+1}}(q, q')$ on symbol a_{i+1}. Given a word $\alpha \in \Sigma^\omega$, the PBA $\mathcal{B}$ can be thought of as a infinite state Markov chain which gives rise to the standard σ-algebra defined using cylinders and the standard probability measure on Markov chains [9, 19]. We denote this measure by $\mu_{\mathcal{B},\alpha}$. A *run* of the PBA $\mathcal{B}$ is an infinite sequence $\rho \in Q^\omega$. A run ρ is *accepting* if $\rho[i] \in Q_f$ for infinitely many i. A run ρ is said to be *rejecting* if it is not accepting. The set of accepting runs and the set of rejecting runs are measurable [19]. Given a word α, the measure of the set of accepting runs is said to be the *probability of accepting* α and is henceforth denoted by $\mu^{acc}_{\mathcal{B},\,\alpha}$; and the measure of the set of rejecting runs is said to be the *probability of rejecting* α and is henceforth denoted by $\mu^{rej}_{\mathcal{B},\,\alpha}$. Clearly $\mu^{acc}_{\mathcal{B},\,\alpha} + \mu^{rej}_{\mathcal{B},\,\alpha} = 1$. Following, [2, 3], a PBA $\mathcal{B}$ on alphabet Σ defines two *semantics*:

– $\mathcal{L}_{>0}(\mathcal{B}) = \{\alpha \in \Sigma^\omega \mid \mu^{acc}_{\mathcal{B},\,\alpha} > 0\}$, henceforth referred to as the *probable semantics* of $\mathcal{B}$, and
– $\mathcal{L}_{=1}(\mathcal{B}) = \{\alpha \in \Sigma^\omega \mid \mu^{acc}_{\mathcal{B},\,\alpha} = 1\}$, henceforth referred to as the *almost-sure semantics* of $\mathcal{B}$.

This gives rise to the following classes of languages of infinite words.

Definition: Given a finite alphabet Σ, $\mathbb{L}(\mathrm{PBA}^{>0}) = \{\mathsf{L} \subseteq \Sigma^\omega \mid \exists \mathrm{PBA}\ \mathcal{B}.\ \mathsf{L} = \mathcal{L}_{>0}(\mathcal{B})\}$ and $\mathbb{L}(\mathrm{PBA}^{=1}) = \{\mathsf{L} \subseteq \Sigma^\omega \mid \exists \mathrm{PBA}\ \mathcal{B}.\ \mathsf{L} = \mathcal{L}_{=1}(\mathcal{B})\}$.

Remark: Given $x \in [0, 1]$, one can of course, also define the languages $\mathcal{L}_{>x}(\mathcal{B}) = \{\alpha \in \Sigma^\omega \mid \mu^{acc}_{\mathcal{B},\,\alpha} > x\}$ and $\mathcal{L}_{\geq x}(\mathcal{B}) = \{\alpha \in \Sigma^\omega \mid \mu^{acc}_{\mathcal{B},\,\alpha} \geq x\}$. The exact value of x is not important and thus one can also define classes $\mathbb{L}(\mathrm{PBA}^{>\frac{1}{2}})$ and $\mathbb{L}(\mathrm{PBA}^{\geq\frac{1}{2}})$.

Probabilistic Rabin automaton. Analogous to the definition of a PBA and RatPBA, one can define a Probabilistic Rabin automaton PRA and RatPRA [2, 7]; where instead of using a set of final states, a set of pairs of subsets of states is used. A run in that case is said to be accepting if it satisfies the Rabin acceptance condition. It is shown in [2, 7] that PRAs have the same expressive power under both probable and almost-sure semantics. Furthermore, it is shown in [2, 7] that for any PBA $\mathcal{B}$, there is PRA $\mathcal{R}$ such that a word α is accepted by $\mathcal{R}$ with probability 1 iff α is accepted by $\mathcal{B}$ with probability > 0. All other words are accepted with probability 0 by $\mathcal{R}$.

Proposition 3 ([2]). For any PBA $\mathcal{B}$ there is a PRA $\mathcal{R}$ such that $\mathcal{L}_{>0}(\mathcal{B}) = \mathcal{L}_{>0}(\mathcal{R}) = \mathcal{L}_{=1}(\mathcal{R})$ and $\mathcal{L}_{=0}(\mathcal{B}) = \mathcal{L}_{=0}(\mathcal{R})$. Furthermore, if $\mathcal{B}$ is a RatPBA then $\mathcal{R}$ can be taken to be RatPRA and the construction of $\mathcal{R}$ is recursive in this case.

Finite probabilistic monitors (FPM)s. We identify one useful syntactic restriction of PBAs, called *finite probabilistic monitors* (FPM)s. In an FPM, all the states are accepting except a special absorbing *reject* state. We studied them extensively in [4, 6].

Definition: A PBA $\mathcal{M} = (Q, q_s, Q_f, \delta)$ on Σ is said to be an FPM if there is a state $q_r \in Q$ such that $q_r \neq q_s$, $Q_f = Q \setminus \{q_r\}$ and $\delta(q_r, a, q_r) = 1$ for each $a \in \Sigma$. The state q_r said to be the *reject* state of $\mathcal{M}$. If in addition $\mathcal{M}$ is a RatPBA, we say that $\mathcal{M}$ is a rational finite probabilistic monitor (RatFPM).

3 Probable Semantics

In this section, we shall study the expressiveness of the languages contained in $\mathbb{L}(\mathrm{PBA}^{>0})$ as well as the complexity of deciding emptiness and universality of $\mathcal{L}_{>0}(\mathcal{B})$ for a given RatPBA $\mathcal{B}$. We assume that the alphabet Σ is fixed and contains at least two letters.

3.1 Expressiveness

It was already shown in [3] that the class of ω-regular languages is strictly contained in the class $\mathbb{L}(\mathrm{PBA}^{>0})$ and that $\mathbb{L}(\mathrm{PBA}^{>0})$ is closed under the Boolean operations of complementation, finite intersection and finite union. We will now show that even though the class $\mathbb{L}(\mathrm{PBA}^{>0})$ strictly contains ω-regular languages, it is not topologically harder. More precisely, we will show that for any PBA $\mathcal{B}$, $\mathcal{L}_{>0}(\mathcal{B})$ is a $\mathsf{BCl}(\mathcal{G}_\delta)$-set. The proof of this fact relies on two facts. The first is that just as the class of ω-regular languages is the Boolean closure of the class of ω-regular recognized by deterministic Büchi automata, the class $\mathbb{L}(\mathrm{PBA}^{>0})$ coincides with the Boolean closure of the class $\mathbb{L}(\mathrm{PBA}^{=1})$. This is the content of the following theorem whose proof is of independent interest and shall be used later in establishing that the containment of languages of two PBAs under almost-sure semantics is undecidable (see Theorem 4).

Theorem 1. $\mathbb{L}(\mathrm{PBA}^{>0}) = \mathsf{BCl}(\mathbb{L}(\mathrm{PBA}^{=1}))$.

Proof. First observe that it was already shown in [2] that $\mathbb{L}(\mathrm{PBA}^{=1}) \subseteq \mathbb{L}(\mathrm{PBA}^{>0})$. Since $\mathbb{L}(\mathrm{PBA}^{>0})$ is closed under Boolean operations, we get that $\mathsf{BCl}(\mathbb{L}(\mathrm{PBA}^{=1})) \subseteq \mathbb{L}(\mathrm{PBA}^{>0})$. We have to show the reverse inclusion.

It suffices to show that given a PBA $\mathcal{B}$, the language $\mathcal{L}_{>0}(\mathcal{B}) \in \mathsf{BCl}(\mathbb{L}(\mathrm{PBA}^{=1}))$. Fix $\mathcal{B}$. Recall that results of [2, 7] (see Proposition 3) imply that there is a probabilistic Rabin automaton (PRA) $\mathcal{R}$ such that 1) $\mathcal{L}_{>0}(\mathcal{B}) = \mathcal{L}_{=1}(\mathcal{R}) = \mathcal{L}_{>0}(\mathcal{R})$ and 2)

$\mathcal{L}_{=0}(\mathcal{B}) = \mathcal{L}_{=0}(\mathcal{R})$. Let $\mathcal{R} = (Q, q_s, F, \delta)$ where $F \subseteq 2^Q \times 2^Q$ is the set of the Rabin pairs. Assuming that F consists of n-pairs, let $F = ((B_1, G_1), \ldots, (B_n, G_n))$.

Given an index set $\mathcal{I} \subseteq \{1, \ldots, n\}$, let $\mathsf{Good}_{\mathcal{I}} = \cup_{r \in \mathcal{I}} G_r$. Let $\mathcal{R}_{\mathcal{I}}$ be the PBA obtained from $\mathcal{R}$ by taking the set of final states to be $\mathsf{Good}_{\mathcal{I}}$. In other words, $\mathcal{R}_{\mathcal{I}} = (Q, q_s, \mathsf{Good}_{\mathcal{I}}, \delta)$. Given $\mathcal{I} \subseteq \{1, \ldots, n\}$ and an index $j \in \mathcal{I}$, let $\mathsf{Bad}_{\mathcal{I},j} = B_j \cup \cup_{r \in \mathcal{I}, r \neq j} G_r$. Let $\mathcal{R}^j_{\mathcal{I}}$ be the PBA obtained from $\mathcal{R}$ by taking the set of final states to be $\mathsf{Bad}_{\mathcal{I},j}$, *i.e.*, $\mathcal{R}^j_{\mathcal{I}} = (Q, q_s, \mathsf{Bad}_{\mathcal{I},j}, \delta)$. The result follows from the following claim.

Claim:

$$\mathcal{L}_{>0}(\mathcal{B}) = \bigcup_{\mathcal{I} \subseteq \{1,\ldots,n\}, j \in \mathcal{I}} \mathcal{L}_{=1}(\mathcal{R}_{\mathcal{I}}) \cap (\Sigma^\omega \setminus \mathcal{L}_{=1}(\mathcal{R}^j_{\mathcal{I}})).$$

The proof of the claim is detailed in [5]. □

The second component needed for showing that $\mathbb{L}(\mathrm{PBA}^{>0}) \subseteq \mathsf{BCl}(\mathcal{G}_\delta)$ is the fact that for any PBA $\mathcal{B}$ and $x \in [0,1]$, the language $\mathcal{L}_{\geq x}(\mathcal{B})$ is a $\mathcal{G}_\delta$-set.

Lemma 1. For any PBA $\mathcal{B}$ and $x \in [0,1]$, $\mathcal{L}_{\geq x}(\mathcal{B})$ is a $\mathcal{G}_\delta$ set.

Using Lemma 1, one immediately gets that $\mathbb{L}(\mathrm{PBA}^{>0}) \subseteq \mathsf{BCl}(\mathcal{G}_\delta)$. Even though PBAs accept non-ω-regular languages, they cannot accept all the languages in $\mathsf{BCl}(\mathcal{G}_\delta)$.

Lemma 2. $\mathsf{Regular} \subsetneq \mathbb{L}(\mathrm{PBA}^{>0}) \subsetneq \mathsf{BCl}(\mathcal{G}_\delta)$.

Remark: Please note that Lemma 1 can be used to show that the classes $\mathbb{L}(\mathrm{PBA}^{>\frac{1}{2}})$ and $\mathbb{L}(\mathrm{PBA}^{\geq\frac{1}{2}})$ are also contained within the first few levels of Borel hierarchy. However, we can show that no version of Theorem 1 holds for those classes. More precisely, $\mathbb{L}(\mathrm{PBA}^{>\frac{1}{2}}) \not\subseteq \mathbb{L}(\mathrm{PBA}^{\geq\frac{1}{2}})$ and $\mathbb{L}(\mathrm{PBA}^{\geq\frac{1}{2}}) \not\subseteq \mathbb{L}(\mathrm{PBA}^{>\frac{1}{2}})$. These results are out of the scope of this paper.

3.2 Decision Problems

Given a RatPBA $\mathcal{B}$, the problems of emptiness and universality of $\mathcal{L}_{>0}(\mathcal{B})$ are known to be undecidable [2]. We sharpen this result by showing that the problem is Σ^0_2-complete. This is interesting in the light of the fact that problems on infinite string automata that are undecidable tend to typically lie in the analytical hierarchy, and not in the arithmetic hierarchy.

Before we proceed with the proof of the upper bound, let us recall an important property of finite-state Büchi automata [13, 18]. The language recognized by a finite-state Büchi automata $\mathcal{A}$ is non-empty iff there is a final state q_f of $\mathcal{A}$, and finite words u and v such that q_f is reachable from the initial state on input u, and q_f is reachable from the state q_f on input v. This is equivalent to saying that any non-empty ω-regular language contains an ultimately periodic word. We had extended this observation to FPMs in [4, 6]. In particular, we had shown that the language $\mathcal{L}_{>x}(\mathcal{M}) \neq \emptyset$ for a given $\mathcal{M}$ iff there exists a set of final states C of $\mathcal{M}$ and words u and v such that the probability of reaching C from the initial state on input u is $> x$ and for each state $q \in \mathsf{C}$ the probability of reaching C from q on input v is 1. This immediately implies that if $\mathcal{L}_{>x}(\mathcal{M})$ is non-empty then $\mathcal{L}_{>x}(\mathcal{M})$ must contain an ultimately periodic word. In contrast, this fact does not hold for non-empty languages in $\mathbb{L}(\mathrm{PBA}^{>0})$. In fact, Baier and Größer [3], construct a PBA $\mathcal{B}$ such that $\mathcal{L}_{>0}(\mathcal{B})$ does not contain any ultimately periodic word.

However, we will show that even though the probable semantics may not contain an ultimately periodic, they nevertheless are restrained in the sense that they must contain *strongly asymptotic* words. Given a PBA $\mathcal{B} = (Q, q_s, Q_f, \delta)$ and a set C of states of $\mathcal{B}$, a word $\alpha \in \Sigma^\omega$ is said to be *strongly asymptotic with respect to* $\mathcal{B}$ *and* C if there is an infinite sequence $i_1 < i_2 < \dots.$ such that 1) $\delta_{\alpha[0:i_1]}(q_s, C) > 0$ and 2) all $j > 0$, for all $q \in C$, the probability of being in C from q after passing through a final state on the finite input string $\alpha[i_j, i_{j+1}]$ is strictly greater than $1 - \frac{1}{2^j}$. A word α is said to be *strongly asymptotic with respect to* $\mathcal{B}$ if there is some C such that α is strongly asymptotic with respect to $\mathcal{B}$ and C . The following notations shall be useful.

Notation: Let $\mathcal{B} = (Q, q_s, Q_f, \delta)$. Given $C \subseteq Q$, $q \in C$ and a finite word $u \in \Sigma^+$, let $\delta_u^{Q_f}(q, C)$ be the probability that the PBA $\mathcal{B}$, when started in state q, on the input string u, is in some state in C at the end of u after passing through a final state. Let $Reach(\mathcal{B}, C, x)$ denote the predicate that for some finite non-empty input string u, the probability of being in C having started from the initial state q_s is $> x$, i.e., $Reach(\mathcal{B}, C, x) = \exists u \in \Sigma^+ . \delta_u(q_s, C) > x$.

The asymptotic sequence property is an immediate consequence of the following Lemma.

Lemma 3. Let $\mathcal{B} = (Q, q_s, Q_f, \delta)$. For any $x \in [0, 1)$, $\mathcal{L}_{>x}(\mathcal{B}) \neq \emptyset$ iff $\exists C \subseteq Q$ such that $Reach(\mathcal{B}, C, x)$ is true and $\forall j > 0$ there is a finite non-empty word u_j such that $\forall q \in C.\ \delta_{u_j}^{Q_f}(q, C) > (1 - \frac{1}{2^j})$.

Proof. The ($\Leftarrow$)-direction is proved in [5]. We outline here the proof of ($\Rightarrow$)-direction. The missing parts of the proof will be cast in terms of claims which are proved in [5].

Assume that $\mathcal{L}_{>x}(\mathcal{B}) \neq \emptyset$. Fix an infinite input string $\gamma \in \mathcal{L}_{>x}(\mathcal{B})$. Recall that the probability measure generated by γ and $\mathcal{B}$ is denoted by $\mu_{\mathcal{B},\gamma}$. For the rest of this proof we will just write μ for $\mu_{\mathcal{B},\gamma}$.

We will call a non-empty set of states C *good* if there is an $\epsilon > 0$, a measurable set $\mathsf{Paths} \subseteq Q^\omega$ of runs, and an infinite sequence of natural numbers $i_1 < i_2 < i_3 < \dots$ such that following conditions hold.

- $\mu(\mathsf{Paths}) \geq x + \epsilon$;
- For each $j > 0$ and each run ρ in Paths, we have that
  1. $\rho[0] = q_s$, $\rho[i_j] \in C$ and
  2. at least one state in the finite sequence $\rho[i_j, i_{j+1}]$ is a final state.

We say that a good set C is *minimal* if C is good but for each $q \in C$, the set $C \setminus \{q\}$ is not good. Clearly if there is a good set of states then there is also a minimal good set of states.

Claim:

- There is a good set of states C.

– Let C be a minimal good set of states. Fix ϵ, Paths and the sequence $i_1 < i_2 < \ldots$ which witness the fact that C is good set of states. For each $q \in C$ and each $j > 0$, let $\mathsf{Paths}_{j,q}$ be the subset of Paths such that each run in Paths passes through q at point i_j, *i.e.*, $\mathsf{Paths}_{j,q} = \{\rho \in \mathsf{Paths} \mid \rho[i_j] = q\}$. Then there exists a $p > 0$ such that $\mu(\mathsf{Paths}_{j,q}) \geq p$ for each $q \in C$ and each $j > 0$.

Now, fix a minimal set of good states C. Fix ϵ, Paths and the sequence $i_1 < i_2 < \ldots$ which witness the fact that C is a good set of states. We claim that C is the required set of states. As $\mu(\mathsf{Paths}) \geq x + \epsilon$ and for each $\rho \in \mathsf{Paths}$, $\rho[i_1] \in C$, it follows immediately that $Reach(\mathcal{B}, C, x)$. Assume now, by way of contradiction, that there exists a $j_0 > 0$ such that for each finite word u, there exists a $q \in C$ such that $\delta_u^{Q_f}(q, C) \leq 1 - \frac{1}{2^{j_0}}$. Fix j_0. Also fix $p > 0$ be such that $\mu(\mathsf{Paths}_{j,q}) \geq p$ for each j and $q \in C$, where $\mathsf{Paths}_{j,q}$ is the subset of Paths such that each run in $\mathsf{Paths}_{j,q}$ passes through q at point i_j; the existence of p is guaranteed by the above claim.

We first construct a sequence of sets $L_i \subseteq Q^+$ as follows. Let $L_1 \subseteq Q^+$ be the set of finite words on states of Q of length i_1+1 such that each word in L_1 starts with the state q_s and ends in a state in C. Formally $L_1 = \{\eta \subseteq Q^+ \mid |\eta| = i_1+1, \eta[0] = q_s \text{ and } \eta[i_1] \in C\}$. Assume that L_r has been constructed. Let $L_{r+1} \subseteq Q^+$ be the set of finite words on states of Q of length $i_{r+1} + 1$ such that each word in L_{r+1} has a prefix in L_r, passes through a final state in between i_r and i_{r+1}, and ends in a state in C. Formally, $L_{r+1} = \{\eta \subseteq Q^+ \mid |\eta| = i_{r+1} + 1, \eta[0 : i_r] \in L_r, \exists i.(i_r < i < i_{r+1} \ \wedge \ \rho[i] \in Q_f)\}$.

Note that $L_r \Sigma^\omega$ is a decreasing sequence of measurable subsets and $\mathsf{Paths} \subseteq \cap_{r>1} L_r \Sigma^\omega$. Now, it is easy to see from the choice of j_0 and p that $\mu(L_{r+1}\Sigma^\omega) \leq \mu(L_r\Sigma^\omega) - \frac{p}{2^{j_0}}$. This, however, implies that there is a r_0 such that $\mu(L_{r_0}\Sigma^\omega) < 0$. A contradiction. □

Lemma 3 implies that checking the non-emptiness of $\mathcal{L}_{>0}(\mathcal{B})$ for a given a RatPBA $\mathcal{B}$ is in Π_2^0. We can exhibit that non-emptiness checking is Π_2^0-hard also. Since the class $\mathbb{L}(\text{PBA}^{>0})$ is closed under complementation and the complementation procedure is recursive [2] for RatPBAs, we can conclude that checking universality of $\mathcal{L}_{>0}(\mathcal{B})$ is also Σ_2^0-complete. The same bounds also apply to checking language containment under probable semantics. Note that these problems were already shown to undecidable in [2], but the exact complexity was not computed therein.

Theorem 2. Given a RatPBA, $\mathcal{B}$, the problems 1) deciding whether $\mathcal{L}_{>0}(\mathcal{B}) = \emptyset$ and 2) deciding whether $\mathcal{L}_{>0}(\mathcal{B}) = \Sigma^\omega$, are Σ_2^0-complete. Given another RatPBA, $\mathcal{B}'$, the problem of deciding whether $\mathcal{L}_{>0}(\mathcal{B}) \subseteq \mathcal{L}_{>0}(\mathcal{B}')$ is also Σ_2^0-complete.

Remark: Lemma 3 can be used to show that emptiness-checking of $\mathcal{L}_{>\frac{1}{2}}(\mathcal{B})$ for a given RatPBA $\mathcal{B}$ is in Σ_2^0. In contrast, we had shown in [6] that the problem of deciding whether $\mathcal{L}_{>\frac{1}{2}}(\mathcal{M}) = \Sigma^\omega$ for a given FPM $\mathcal{M}$ lies beyond the arithmetical hierarchy.

4 Almost-Sure Semantics

The class $\mathbb{L}(\text{PBA}^{=1})$ was first studied in [2], although they were not characterized topologically. In this section, we study the expressiveness and complexity of the class

$\mathbb{L}(\mathrm{PBA}^{=1})$. We will also demonstrate that the class $\mathbb{L}(\mathrm{PBA}^{=1})$ is closed under finite unions and intersections. As in the case of probable semantics, we assume that the alphabet Σ is fixed and contains at least two letters.

4.1 Expressiveness

Lemma 1 already implies that topologically, the class $\mathbb{L}(\mathrm{PBA}^{=1}) \subseteq \mathcal{G}_\delta$. Recall that $\mathcal{G}_\delta$ coincides exactly with the class of languages recognizable with infinite-state deterministic Büchi automata (see Section 2). Thanks to Theorem 1 and Lemma 2, it also follows immediately that the inclusion $\mathbb{L}(\mathrm{PBA}^{=1}) \subseteq \mathcal{G}_\delta$ is strict (otherwise we will have $\mathbb{L}(\mathrm{PBA}^{>0}) = \mathsf{BCl}(\mathbb{L}(\mathrm{PBA}^{=1})) = \mathsf{BCl}(\mathcal{G}_\delta)$). The fact that every language $\mathbb{L}(\mathrm{PBA}^{=1})$ is contained in $\mathcal{G}_\delta$ implies immediately that there are ω-regular languages not in $\mathbb{L}(\mathrm{PBA}^{=1})$. That there are ω-regular languages not in $\mathbb{L}(\mathrm{PBA}^{=1})$ was also proved in [2], although the proof therein is by explicit construction of an ω-regular language which is then shown to be not in $\mathbb{L}(\mathrm{PBA}^{=1})$. Our topological characterization of the class $\mathbb{L}(\mathrm{PBA}^{=1})$ has the advantage that we can characterize the intersection $\mathsf{Regular} \cap \mathbb{L}(\mathrm{PBA}^{=1})$ exactly: $\mathsf{Regular} \cap \mathbb{L}(\mathrm{PBA}^{=1})$ is the class of ω-regular languages that can be recognized by a finite-state deterministic Büchi automaton.

Proposition 4. For any PBA $\mathcal{B}$, $\mathcal{L}_{=1}(\mathcal{B})$ is a $\mathcal{G}_\delta$ set. Furthermore, $\mathsf{Regular} \cap \mathbb{L}(\mathrm{PBA}^{=1}) = \mathsf{Regular} \cap \mathrm{Deterministic}$ and $\mathsf{Regular} \cap \mathrm{Deterministic} \subsetneq \mathbb{L}(\mathrm{PBA}^{=1}) \subsetneq \mathcal{G}_\delta = \mathrm{Deterministic}$.

An immediate consequence of the characterization of the intersection $\mathsf{Regular} \cap \mathrm{Deterministic}$ is that the class $\mathbb{L}(\mathrm{PBA}^{=1})$ is not closed under complementation as the class of ω-regular languages recognized by deterministic Büchi automata is not closed under complementation. That the class $\mathbb{L}(\mathrm{PBA}^{=1})$ is not closed under complementation is also observed in [2], and is proved by constructing an explicit example. However, even though the class $\mathbb{L}(\mathrm{PBA}^{=1})$ is not closed under complementation, we have a "partial" complementation operation— for any PBA $\mathcal{B}$ there is another PBA $\mathcal{B}'$ such that $\mathcal{L}_{>0}(\mathcal{B}')$ is the complement of $\mathcal{L}_{=1}(\mathcal{B})$. This also follows from the results of [2] as they showed that $\mathbb{L}(\mathrm{PBA}^{=1}) \subseteq \mathbb{L}(\mathrm{PBA}^{>0})$ and $\mathbb{L}(\mathrm{PBA}^{>0})$ is closed under complementation. However our construction has two advantages: 1) it is much simpler than the one obtained by the constructions in [2], and 2) the PBA $\mathcal{B}'$ belongs to the restricted class of finite probabilistic monitors FPMs (see Section 2 for definition of FPMs). This construction plays a critical role in our complexity analysis of decision problems.

Lemma 4. For any PBA $\mathcal{B}$, there is an FPM $\mathcal{M}$ such that $\mathcal{L}_{=1}(\mathcal{B}) = \Sigma^\omega \setminus \mathcal{L}_{>0}(\mathcal{M})$.

Proof. Let $\mathcal{B} = (Q, q_s, Q_f, \delta)$. We construct $\mathcal{M}$ as follows. First we pick a new state q_r, which will be the reject state of the FPM $\mathcal{M}$. The set of states of $\mathcal{M}$ would be $Q \cup \{q_r\}$. The initial state of $\mathcal{M}$ will be q_s, the initial state of $\mathcal{B}$. The set of final states of $\mathcal{M}$ will be Q, the set of states of $\mathcal{B}$. The transition relation of $\mathcal{M}$ would be defined as follows. If q is not a final state of $\mathcal{B}$ then the transition function would be the same as for $\mathcal{B}$. If q is an final state of $\mathcal{B}$ then $\mathcal{M}$ will transit to the reject state with probability $\frac{1}{2}$ and with probability $\frac{1}{2}$ continue as in $\mathcal{B}$. Formally, $\mathcal{M} = (Q \cup \{q_r\}, q_s, Q, \delta_{\mathcal{M}})$ where $\delta_{\mathcal{M}}$ is defined as follows. For each $a \in \Sigma$, $q, q' \in Q$,

– $\delta_{\mathcal{M}}(q, a, q_r) = \frac{1}{2}$ and $\delta_{\mathcal{M}}(q, a, q') = \frac{1}{2}\delta(q, a, q')$ if $q \in Q_f$,
– $\delta_{\mathcal{M}}(q, a, q_r) = 0$ and $\delta_{\mathcal{M}}(q, a, q') = \delta(q, a, q')$ if $q \in Q \setminus Q_f$,
– $\delta_{\mathcal{M}}(q_r, a, q_r) = 1$.

It is easy to see that a word $\alpha \in \Sigma^\omega$ is rejected with probability 1 by $\mathcal{M}$ iff it is accepted with probability 1 by $\mathcal{B}$. The result now follows. □

The "partial" complementation operation has many consequences. One consequence is that the class $\mathbb{L}(\mathrm{PBA}^{=1})$ is closed under union. The class $\mathbb{L}(\mathrm{PBA}^{=1})$ is easily shown to be closed under intersection. Hence for closure properties, $\mathbb{L}(\mathrm{PBA}^{=1})$ behave like deterministic Büchi automata. Please note that closure properties were not studied in [2].

Corollary 1. The class $\mathbb{L}(\mathrm{PBA}^{=1})$ is closed under finite union and finite intersection.

4.2 Decision Problems

The problem of checking whether $\mathcal{L}_{=1}(\mathcal{B}) = \emptyset$ for a given RatPBA $\mathcal{B}$ was shown to be decidable in **EXPTIME** in [2], where it was also conjectured to be **EXPTIME**-complete. The decidability of the universality problem was left open in [2]. We can leverage our "partial" complementation operation to show that a) the emptiness problem is in fact **PSPACE**-complete, thus tightening the bound in [2] and b) the universality problem is also **PSPACE**-complete.

Theorem 3. Given a RatPBA $\mathcal{B}$, the problem of deciding whether $\mathcal{L}_{=1}(\mathcal{B}) = \emptyset$ is **PSPACE**-complete. The problem of deciding whether $\mathcal{L}_{=1}(\mathcal{B}) = \Sigma^\omega$ is also **PSPACE**-complete.

Proof. (**Upper bounds**.) We first show the upper bounds. The proof of Lemma 4 shows that for any RatPBA $\mathcal{B}$, there is a RatFPM $\mathcal{M}$ constructed in polynomial time such that $\mathcal{L}_{=1}(\mathcal{B}) = \Sigma^\omega \setminus \mathcal{L}_{>0}(\mathcal{M})$. $\mathcal{L}_{=1}(\mathcal{B})$ is empty (universal) iff $\mathcal{L}_{>0}(\mathcal{M})$ is universal (empty respectively). Now, we had shown in [4, 6] that given a RatFPM $\mathcal{M}$, the problems of checking emptiness and universality of $\mathcal{L}_{>0}(\mathcal{M})$ are in **PSPACE**, thus giving us the desired upper bounds.
(**Lower bounds**.) We had shown in [4, 6] that given a RatFPM $\mathcal{M}$, the problems of deciding the emptiness and universality of $\mathcal{L}_{>0}(\mathcal{M})$ are **PSPACE**-hard respectively. Given a RatFPM $\mathcal{M} = (Q, q_s, Q_0, \delta)$ with q_r as the absorbing reject state, consider the PBA $\overline{\mathcal{M}} = (Q, q_s, \{q_r\}, \delta)$ obtained by considering the unique reject state of $\mathcal{M}$ as the only final state of $\overline{\mathcal{M}}$. Clearly we have that $\mathcal{L}_{>0}(\mathcal{M}) = \Sigma^\omega \setminus \mathcal{L}_{=1}(\overline{\mathcal{M}})$. Thus $\mathcal{L}_{>0}(\mathcal{M})$ is empty (universal) iff $\mathcal{L}_{=1}(\overline{\mathcal{M}})$ is universal (empty respectively). The result now follows. □

Even though the problems of checking emptiness and universality of almost-sure semantics of a RatPBA are decidable, the problem of deciding language containment under almost-sure semantics turns out to be undecidable, and is indeed as hard as the problem of deciding language containment under probable semantics (or, equivalently, checking emptiness under probable semantics).

Theorem 4. Given RatPBAs, $\mathcal{B}_1$ and $\mathcal{B}_2$, the problem of deciding whether $\mathcal{L}_{=1}(\mathcal{B}_1) \subseteq \mathcal{L}_{=1}(\mathcal{B}_2)$ is Σ^0_2-complete.

5 Hierarchical PBAs

We shall now identify a simple syntactic restriction on PBAs which under probable semantics coincide exactly with ω-regular languages and under almost-sure semantics coincide exactly with ω-regular deterministic languages. These restricted PBAs shall be called *hierarchical* PBAs.

Intuitively, a hierarchical PBA is a PBA such that the set of its states can be stratified into (totally) ordered levels. From a state q, for each letter a, the machine can transition with non-zero probability to at most one state in the same level as q, and all other probabilistic transitions go to states that belong to a higher level. Formally,

Definition: Given a natural number k, a PBA $\mathcal{B} = (Q, q_s, Q, \delta)$ over an alphabet Σ is said to be a *k-level hierarchical PBA (k-*PBA*)* if there is a function $\mathsf{rk} : Q \to \{0, 1, \ldots, k\}$ such that the following holds.

> Given $j \in \{0, 1, \ldots, k\}$, let $Q_j = \{q \in Q \mid \mathsf{rk}(Q) = j\}$. For every $q \in Q$ and $a \in \Sigma$, if $j_0 = \mathsf{rk}(q)$ then $\mathsf{post}(q, a) \subseteq \cup_{j_0 \leq \ell \leq k} Q_\ell$ and $|\mathsf{post}(q, a) \cap Q_{j_0}| \leq 1$.

The function rk is said to be a *compatible ranking function* of $\mathcal{B}$ and for $q \in Q$ the natural number $\mathsf{rk}(q)$ is said to be the *rank* or *level* of q. $\mathcal{B}$ is said to be a *hierarchical* PBA *(*HPBA*)* if $\mathcal{B}$ is k-hierarchical for some k. If $\mathcal{B}$ is also a RatPBA, we say that $\mathcal{B}$ is a rational hierarchical PBA (RatHPBA).

We can define classes analogous to $\mathbb{L}(\mathrm{PBA}^{>0})$ and $\mathbb{L}(\mathrm{PBA}^{=1})$; and we shall call them $\mathbb{L}(\mathrm{HPBA}^{>0})$ and $\mathbb{L}(\mathrm{HPBA}^{=1})$ respectively.

Before we proceed to discuss the probable and almost-sure semantics for HPBAs, we point out two interesting facts about hierarchical HPBAs. First is that for the class of ω-regular deterministic languages, HPBAs like non-deterministic Büchi automata can be exponentially more succinct.

Lemma 5. Let $\Sigma = \{a, b, c\}$. For each $n \in \mathbb{N}$, there is a ω-regular deterministic property $\mathsf{L}_n \subseteq \Sigma^\omega$ such that i) any deterministic Büchi automata for L_n has at least $O(2^n)$ number of states, and ii) there are HPBAs $\mathcal{B}_n$ s.t. $\mathcal{B}_n$ has $O(n)$ number of states and $\mathsf{L}_n = \mathcal{L}_{=1}(\mathcal{B}_n)$.

The second thing is that even though HPBAs yield only ω-regular languages under both almost-sure semantics and probable semantics, we can recognize non-ω-regular languages with cut-points.

Lemma 6. There is a HPBA $\mathcal{B}$ such that both $\mathcal{L}_{\geq \frac{1}{2}}(\mathcal{B})$ and $\mathcal{L}_{> \frac{1}{2}}(\mathcal{B})$ are not ω-regular.

Remark: We will see shortly that the problems of deciding emptiness and universality for a HPBA turn out to be decidable under both probable and almost-sure semantics. However, with cut-points, they turn out to be undecidable. The latter, however, is out of scope of this paper.

5.1 Probable Semantics

We shall now show that the class $\mathbb{L}(\mathrm{HPBA}^{>0})$ coincides with the class of ω-regular languages under probable semantics. In [3], a restricted class of PBAs called uniform

PBAs was identified that also accept exactly the class of ω-regular languages. We make a couple of observations, contrasting our results here with theirs. First the definition of uniform PBA was semantic (i.e., the condition depends on the acceptance probability of infinitely many strings from different states of the automaton), whereas HPBA are a syntactic restriction on PBA. Second, we note that the definitions themselves are incomparable in some sense; in other words, there are HPBAs which are not uniform, and vice versa. Finally, HPBAs appear to be more tractable than uniform PBAs. We show that the emptiness problem for $\mathbb{L}(\mathrm{HPBA}^{>0})$ is **NL**-complete. In contrast, the same problem was demonstrated to be in **EXPTIME** and co-**NP**-hard [3].

Our main observation is the Hierarchical PBAs capture exactly the class of ω-regular languages.

Theorem 5. $\mathbb{L}(\mathrm{HPBA}^{>0}) = \mathsf{Regular}$.

We will show that the problem of deciding whether $\mathcal{L}_{>0}(\mathcal{B})$ is empty for hierarchical RatPBA's is **NL**-complete while the problem of deciding whether $\mathcal{L}_{>0}(\mathcal{B})$ is universal is **PSPACE**-complete. Thus "algorithmically", hierarchical PBAs are much "simpler" than both PBAs and uniform PBAs. Note that the emptiness and universality problem for finite state Büchi-automata are also **NL**-complete and **PSPACE**-complete respectively.

Theorem 6. Given a RatHPBA, $\mathcal{B}$, the problem of deciding whether $\mathcal{L}_{>0}(\mathcal{B}) = \emptyset$ is **NL**-complete. The problem of deciding whether $\mathcal{L}_{>0}(\mathcal{B}) = \Sigma^\omega$ is **PSPACE**-complete.

5.2 Almost-Sure Semantics

For a hierarchical PBA, the "partial" complementation operation for almost-sure semantics discussed in Section 4 yields a hierarchical PBA. Therefore using Theorem 5, we immediately get that a language $\mathcal{L} \in \mathbb{L}(\mathrm{HPBA}^{=1})$ is ω-regular. Thanks to the topological characterization of $\mathbb{L}(\mathrm{HPBA}^{=1})$ as a sub-collection of deterministic languages, we get that $\mathbb{L}(\mathrm{HPBA}^{=1})$ is exactly the class of languages recognized by deterministic finite-state Büchi automata.

Theorem 7. $\mathbb{L}(\mathrm{HPBA}^{=1}) = \mathsf{Regular} \cap \mathrm{Deterministic}$.

The "partial" complementation operation also yields the complexity of emptiness and universality problems.

Theorem 8. Given a RatHPBA, $\mathcal{B}$, the problem of deciding whether $\mathcal{L}_{=1}(\mathcal{B}) = \emptyset$ is **PSPACE**-complete. The problem of deciding whether $\mathcal{L}_{=1}(\mathcal{B}) = \Sigma^\omega$ is **NL**-complete.

6 Conclusions

In this paper, we investigated the power of randomization in finite state automata on infinite strings. We presented a number of results on the expressiveness and decidability problems under different notions of acceptance based on the probability of acceptance. In the case of decidability, we gave tight bounds for both the universality and emptiness problems. As part of future work, it will be interesting to investigate the power of randomization in other models of computations on infinite strings such as pushdown automata etc. Since the universality and emptiness problems are PSPACE-complete for almost-sure semantics, their application to practical systems needs further enquiry.

Acknowledgements

Rohit Chadha was supported in part by NSF grants CCF04-29639 and NSF CCF04-48178. A. Prasad Sistla was supported in part by NSF CCF-0742686. Mahesh Viswanathan was supported in part by NSF CCF04-48178 and NSF CCF05-09321.

References

1. Alpern, B., Schneider, F.: Defining liveness. Information Processing Letters 21, 181–185 (1985)
2. Baier, C., Bertrand, N., Größer, M.: On decision problems for probabilistic büchi automata. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 287–301. Springer, Heidelberg (2008)
3. Baier, C., Größer, M.: Recognizing ω-regular languages with probabilistic automata. In: Proceedings of the IEEE Symposium on Logic in Computer Science, pp. 137–146 (2005)
4. Chadha, R., Sistla, A.P., Viswanathan, M.: On the expressiveness and complexity of randomization in finite state monitors. In: LICS 2008- 23rd Annual IEEE Symposium on Logic in Computer Science, pp. 18–29. IEEE Computer Society, Los Alamitos (2008)
5. Chadha, R., Sistla, A.P., Viswanathan, M.: On the expressiveness and complexity of randomization in finite state monitors. Journal of the ACM (to appear), `http://www.cs.uiuc.edu/homes/rch/TechReports/MonitoringTechReport.pdf`
6. Condon, A., Lipton, R.J.: On the complexity of space bounded interactive proofs (extended abstract). In: 30th Annual Symposium on Foundations of Computer Science, pp. 462–467 (1989)
7. Groesser, M.: Reduction Methods for Probabilistic Model Checking. PhD thesis, TU Dresden (2008)
8. Holzmann, G.J., Peled, D.: The state of spin. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102. Springer, Heidelberg (1996)
9. Kemeny, J., Snell, J.: Denumerable Markov Chains. Springer, Heidelberg (1976)
10. Kurshan, R.P.: Computer Aided Verification of the Coordinated Processes: The Automata Theoretic Approach. Princeton University Press, Princeton (1994)
11. Lamport, L.: Logical foundation, distributed systems- methods and tools for specification, vol. 190. Springer, Heidelberg (1985)
12. Paz, A.: Introduction to Probabilistic Automata. Academic Press, London (1971)
13. Perrin, D., Pin, J.-E.: Infinite Words: Automata, Semigroups, Logic and Games. Elsevier, Amsterdam (2004)
14. Rabin, M.O.: Probabilitic automata. Information and Control 6(3), 230–245 (1963)
15. Rutten, J.M., Kwiatkowska, M., Norman, G., Parker, D.: Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems. AMS (2004)
16. Salomaa, A.: Formal Languages. Academic Press, London (1973)
17. Sistla, A.P.: Theoretical Issues in the Design and Verification of Distributed Systems. PhD thesis, Harvard University (1983)
18. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, vol. B, pp. 133–192 (1990)
19. Vardi, M.: Automatic verification of probabilistic concurrent systems. In: 26th annual Symposium on Foundations of Computer Science, pp. 327–338. IEEE Computer Society Press, Los Alamitos (1985)
20. Vardi, M., Wolper, P.: An automata theoretic approach to automatic program verification. In: Proceedings of the first IEEE Symposium on Logic in Computer Science (1986)
21. Vardi, M., Wolper, P., Sistla, A.P.: Reasoning about infinite computation paths. In: Proceedings of the 24th IEEE Symposium on Foundations of Computer Science (1983)

Probabilistic Weighted Automata[⋆]

Krishnendu Chatterjee[1], Laurent Doyen[2,⋆⋆], and Thomas A. Henzinger[3]

[1] Institute of Science and Technology (IST), Austria
[2] Université Libre de Bruxelles (ULB), Belgium
[3] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. Nondeterministic weighted automata are finite automata with numerical weights on transitions. They define quantitative languages L that assign to each word w a real number $L(w)$. The value of an infinite word w is computed as the maximal value of all runs over w, and the value of a run as the supremum, limsup, liminf, limit average, or discounted sum of the transition weights.

We introduce probabilistic weighted automata, in which the transitions are chosen in a randomized (rather than nondeterministic) fashion. Under almost-sure semantics (resp. positive semantics), the value of a word w is the largest real v such that the runs over w have value at least v with probability 1 (resp. positive probability).

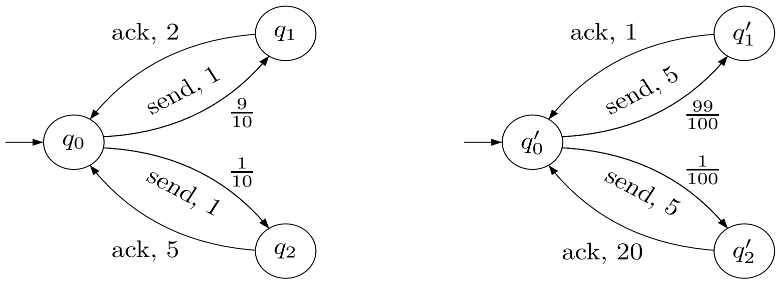
We study the classical questions of automata theory for probabilistic weighted automata: emptiness and universality, expressiveness, and closure under various operations on languages. For quantitative languages, emptiness and universality are defined as whether the value of some (resp. every) word exceeds a given threshold. We prove some of these questions to be decidable, and others undecidable. Regarding expressive power, we show that probabilities allow us to define a wide variety of new classes of quantitative languages, except for discounted-sum automata, where probabilistic choice is no more expressive than nondeterminism. Finally, we give an almost complete picture of the closure of various classes of probabilistic weighted automata for the following pointwise operations on quantitative languages: max, min, sum, and numerical complement.

1 Introduction

In formal design, specifications describe the set of correct behaviors of a system. An implementation satisfies a specification if all its behaviors are correct. If we view a behavior as a word, then a specification is a language, i.e., a set of words. Languages can be specified using finite automata, for which a large number of results and techniques are known; see [20, 24]. We call them *boolean languages*

[⋆] This research was supported in part by the Swiss National Science Foundation under the Indo-Swiss Joint Research Programme, by the European Network of Excellence on Embedded Systems Design (ArtistDesign), by the European projects Combest, Quasimodo, and Gasics, by the PAI program Moves funded by the Belgian Federal Government, and by the CFV (Federated Center in Verification) funded by the F.R.S.-FNRS.

[⋆⋆] Postdoctoral researcher of the F.R.S.-FNRS.



(a) Low reliability but cheap. (b) High reliability but expensive.

Fig. 1. Two specifications of a channel

because a given behavior is either good or bad according to the specification. Boolean languages are useful to specify functional requirements.

In a generalization of this approach, we consider *quantitative languages* L, where each word w is assigned a real number $L(w)$. The value of a word can be interpreted as the amount of some resource (e.g., memory or power) needed to produce it, or as a quality measurement for the corresponding behavior [5, 6]. Therefore, quantitative languages are useful to specify nonfunctional requirements such as resource constraints, reliability properties, or levels of quality (such as quality of service). Note that a boolean language L is a special case of quantitative language that assigns value 1 to the words in L and value 0 to the words not in L.

Quantitative languages can be defined using nondeterministic weighted automata, i.e., finite automata with numerical weights on transitions [13, 17]. In [7], we studied quantitative languages of infinite words and defined the value of an infinite word w as the maximal value of all runs of an automaton over w (if the automaton is nondeterministic, then there may be many runs over w). The value of a run r is a function of the infinite sequence of weights that appear along r. There are several natural functions to consider, such as Sup, LimSup, LimInf, limit average, and discounted sum of weights. For example, peak power consumption can be modeled as the maximum of a sequence of weights representing power usage; energy use, as a discounted sum; average response time, as a limit average [4, 5].

In this paper, we present *probabilistic* weighted automata as a new model defining quantitative languages. In such automata, nondeterministic choice is replaced by probability distributions on successor states. The value of an infinite word w is defined to be the maximal value v such that the set of runs over w with value at least v has either positive probability (*positive semantics*), or probability 1 (*almost-sure semantics*). This simple definition combines in a general model the natural quantitative extensions of logics and automata [7, 14, 15], and the probabilistic models of automata for which boolean properties have been studied [2, 3, 22]. Note that the probabilistic Büchi and coBüchi automata of [2] are a special case of probabilistic weighted automata with weights 0 and 1 only (and the value of an infinite run computed as LimSup or LimInf, respectively). While quantitative objectives are standard in the branching-time context of stochastic games [5, 11, 16, 18, 19, 23], we are not aware of any model combining probabilities and weights in the linear-time context of words and languages, though such a model is very natural for the specification of quantitative properties. Consider the specification of two types of communication channels given in Fig. 1. One has low cost (sending costs 1 unit) and low reliability (a failure occurs in 10% of the cases and entails an increased cost for the operation), while the second is expensive (sending costs 5 units), but the reliability is high (though the cost of a failure is prohibitive). In the figure, we omit the self-loops with cost 0 in state q_0 and q_0' over *ack*, and in q_1, q_2, q_1', q_2' over *send*. Natural questions can be formulated in this framework, such as whether the average cost of every word $w \in \{send, ack\}^\omega$ is really smaller in the low-cost channel, or to construct a probabilistic weighted automaton that assigns to each infinite word $w \in \{send, ack\}^\omega$ the minimum of the average cost of the two types of channels (the answers are yes for both the questions for Fig. 1). In this paper, we attempt a comprehensive study of such fundamental questions, about the expressive power, closure properties, and decision problems for probabilistic weighted automata. We focus on the positive and the almost-sure semantics. In future work, we will consider another semantics where the value of a word w is defined to be the expectation of the values of the runs over w.

First, we compare the expressiveness of the various classes of probabilistic and nondeterministic weighted automata over infinite words. For LimSup, LimInf, and limit average, we show that a wide variety of new classes of quantitative languages can be defined using probabilities, which are not expressible using nondeterminism. Our results rely on reachability properties of closed recurrent sets in Markov chains. For discounted sum, we show that probabilistic weighted automata under the positive semantics have the same expressive power as nondeterministic weighted automata, while under the almost-sure semantics, they have the same expressive power as weighted automata with universal branching, where the value of a word is the minimal (instead of maximal) value of all runs. The question of whether the positive semantics of weighted limit-average automata is more expressive than nondeterminism remains open.

Second, we give an almost complete picture of the closure of probabilistic weighted automata under the pointwise operations of maximum, minimum, and sum for quantitative languages. We also consider the numerical complement L^c of a quantitative language L defined by $L^c(w) = 1 - L(w)$ for all words w.[1] Note that maximum and minimum provide natural generalization of the classical union and intersection operations on boolean languages, and they define the least upper bound and greatest lower bound for the pointwise natural order on quantitative languages (where $L_1 \leq L_2$ if $L_1(w) \leq L_2(w)$ for all words w). The numerical complement applied to boolean languages also defines the usual complement operation.

[1] One can define $L^c(w) = k - L(w)$ for any constant k without changing our results.

Closure under max trivially holds for the positive semantics, and closure under min for the almost-sure semantics. Only LimSup-automata under positive semantics and LimInf-automata under almost-sure semantics are closed under all four operations; these results extend corresponding results for the boolean case [1]. To establish the closure properties of limit-average automata, we characterize the expected limit-average reward of Markov chains. Our characterization answers all closure questions except for the language sum in the case of positive semantics, which we leave open. Note that expressiveness results and closure properties are tightly connected. For instance, because they are closed under max, the LimInf-automata with positive semantics are no more expressive than to LimInf-automata with almost-sure semantics and to LimSup-automata with positive semantics; and because they are not closed under complement, the LimSup-automata with almost-sure semantics and LimInf-automata with positive semantics have incomparable expressive powers.

Third, we investigate the emptiness and universality problems for probabilistic weighted automata, which ask to decide if some (resp. all) words have a value above a given threshold. Using our expressiveness results, as well as [1, 9], we establish some decidability and undecidability results for Sup, LimSup, and LimInf automata; in particular, emptiness and universality are undecidable for LimSup-automata with positive semantics and for LimInf-automata with almost-sure semantics, while the question is open for the emptiness of LimInf-automata with positive semantics and for the universality of LimSup-automata with almost-sure semantics. We also prove the decidability of emptiness for probabilistic discounted-sum automata with positive semantics, while the universality problem is as hard as for nondeterministic discounted-sum automata, for which no decidability result is known. We leave open the case of limit average. Due to lack of space, we omit detailed proofs; they can be found in [10].

2 Definitions

A *quantitative language* over a finite alphabet Σ is a function $L : \Sigma^\omega \to \mathbb{R}$. A boolean language (or a set of infinite words) is a special case where $L(w) \in \{0, 1\}$ for all words $w \in \Sigma^\omega$. Nondeterministic weighted automata define the value of a word as the maximal value of a run [7]. In this paper, we study probabilistic weighted automata as generator of quantitative languages.

Value functions. We consider the following value functions $\mathsf{Val} : \mathbb{Q}^\omega \to \mathbb{R}$ to define quantitative languages. Given an infinite sequence $v = v_0 v_1 \dots$ of rational numbers, define

- $\mathsf{Sup}(v) = \sup\{v_n \mid n \geq 0\}$;
- $\mathsf{LimSup}(v) = \limsup_{n\to\infty} v_n = \lim_{n\to\infty} \sup\{v_i \mid i \geq n\}$;
- $\mathsf{LimInf}(v) = \liminf_{n\to\infty} v_n = \lim_{n\to\infty} \inf\{v_i \mid i \geq n\}$;
- $\mathsf{LimAvg}(v) = \liminf_{n\to\infty} \frac{1}{n} \cdot \sum_{i=0}^{n-1} v_i$;

– for $0 < \lambda < 1$, $\mathsf{Disc}_\lambda(v) = \sum_{i=0}^{\infty} \lambda^i \cdot v_i$.

Given a finite set S, a *probability distribution* over S is a function $f : S \to [0,1]$ such that $\sum_{s \in S} f(s) = 1$. We denote by $\mathcal{D}(S)$ the set of all probability distributions over S.

Probabilistic weighted automata. A *probabilistic weighted automaton* is a tuple $A = \langle Q, \rho_I, \Sigma, \delta, \gamma \rangle$, where

– Q is a finite set of states;
– $\rho_I \in \mathcal{D}(Q)$ is the initial probability distribution;
– Σ is a finite alphabet;
– $\delta : Q \times \Sigma \to \mathcal{D}(Q)$ is a probabilistic transition function;
– $\gamma : Q \times \Sigma \times Q \to \mathbb{Q}$ is a weight function.

The automaton A is *deterministic* if $\rho_I(q_I) = 1$ for some $q_I \in Q$, and for all $q \in Q$ and $\sigma \in \Sigma$, there exists $q' \in Q$ such that $\delta(q, \sigma)(q') = 1$.

A *run* of A over a finite (resp. infinite) word $w = \sigma_1 \sigma_2 \dots$ is a finite (resp. infinite) sequence $r = q_0 \sigma_1 q_1 \sigma_2 \dots$ of states and letters such that (*i*) $\rho_I(q_0) > 0$, and (*ii*) $\delta(q_i, \sigma_{i+1})(q_{i+1}) > 0$ for all $0 \le i < |w|$. We denote by $\gamma(r) = v_0 v_1 \dots$ the sequence of weights that occur in r where $v_i = \gamma(q_i, \sigma_{i+1}, q_{i+1})$ for all $0 \le i < |w|$. The probability of a finite run $r = q_0 \sigma_1 q_1 \sigma_2 \dots \sigma_k q_k$ over a finite word $w = \sigma_1 \dots \sigma_k$ is $\mathbb{P}^A(r) = \rho_I(q_0) . \prod_{i=1}^{k} \delta(q_{i-1}, \sigma_i)(q_i)$. For a finite run r, let $\mathsf{Cone}(r)$ denote the set of infinite runs r' such that r is a prefix of r'. The set of cones forms the basis for the Borel sets for runs. For each $w \in \Sigma^\omega$, the function $\mathbb{P}^A(\cdot)$ defines a unique probability measure over Borel sets of runs of A over w.

Given a value function $\mathsf{Val} : \mathbb{Q}^\omega \to \mathbb{R}$, we say that the probabilistic Val-automaton A generates the quantitative languages defined for all words $w \in \Sigma^\omega$ by $L_A^{=1}(w) = \sup\{\eta \mid \mathbb{P}^A(\{r \in \mathsf{Run}^A(w) \text{ such that } \mathsf{Val}(\gamma(r)) \ge \eta\}) = 1\}$ under the *almost-sure* semantics, and $L_A^{>0}(w) = \sup\{\eta \mid \mathbb{P}^A(\{r \in \mathsf{Run}^A(w) \text{ such that } \mathsf{Val}(\gamma(r)) \ge \eta\}) > 0\}$ under the *positive* semantics. In classical (non-probabilistic) semantics, the value of a word is defined either as the maximal value of the runs (i.e., $L_A^{\max}(w) = \sup\{\mathsf{Val}(\gamma(r)) \mid r \in \mathsf{Run}^A(w)\}$ for all $w \in \Sigma^\omega$) and the automaton is then called *nondeterministic*, or as the minimal value of the runs, and the automaton is then called *universal* [8]. Note that the above four semantics coincide for deterministic weighted automata (because then every word has exactly one run), and that Büchi and coBüchi automata [2] are special cases of respectively LimSup- and LimInf-automata, where all weights are either 0 or 1.

Reducibility. A class $\mathcal{C}$ of weighted automata is *reducible* to a class $\mathcal{C}'$ of weighted automata if for every $A \in \mathcal{C}$ there exists $A' \in \mathcal{C}'$ such that $L_A = L_{A'}$, i.e., $L_A(w) = L_{A'}(w)$ for all words w. Reducibility relationships for (non)deterministic weighted automata are given in [7].

Composition. Given two quantitative languages $L, L' : \Sigma^\omega \to \mathbb{R}$, we denote by $\max(L, L')$ (resp. $\min(L, L')$ and $L + L'$) the quantitative language that assigns $\max\{L(w), L'(w)\}$ (resp. $\min\{L(w), L'(w)\}$ and $L(w) + L'(w)$) to each word

$w \in \Sigma^\omega$. The language $1 - L$ is called the *complement* of L. The max, min and complement operators for quantitative languages generalize respectively the union, intersection and complement operator for boolean languages. The closure properties of (non)deterministic weighted automata are given in [9].

Notation. The first letter in acronyms for classes of automata can be N(ondeterministic), D(eterministic), U(niversal), Pos for the language in the positive semantics, or As for the language in the almost-sure semantics. For $X, Y \in \{\text{N, D, U, Pos, As}\}$, we sometimes use the notation $^{\text{X}}_{\text{Y}}$ for classes of automata where the X and Y versions are reducible to each other. For Büchi and coBüchi automata, we use the classical acronyms NBW, DBW, NCW, etc. When the type of an automaton A is clear from the context, we often denote its language simply by $L_A(\cdot)$ or even $A(\cdot)$, instead of $L_A^{=1}$, $L_A^{\max}$, etc.

Remark. We sometimes use automata with weight functions $\gamma : Q \to \mathbb{Q}$ that assign a weight to states instead of transitions. This is a convenient notation for weighted automata in which from each state, all outgoing transitions have the same weight. In pictorial descriptions of probabilistic weighted automata, the transitions are labeled with probabilities, and states with weights.

3 Expressive Power of Probabilistic Weighted Automata

We complete the picture given in [7] about reducibility for nondeterministic weighted automata, by adding the relations with probabilistic automata. The results for LimInf, LimSup, and LimAvg are summarized in Fig. 2s, and for Sup- and Disc-automata in Theorems 1 and 6.

As for probabilistic automata over finite words, the quantitative languages definable by probabilistic and (non)deterministic Sup-automata coincide.

Theorem 1. DSup *is as expressive as* PosSup *and* AsSup.

In many of our results, we use the following definitions and properties related to Markov chains. A *Markov chain* $M = (S, E, \delta)$ consists of a finite set S of states, a set E of edges, and a probabilistic transition function $\delta : S \to \mathcal{D}(S)$. For all $s, t \in S$, there is an edge $(s, t) \in E$ iff $\delta(s)(t) > 0$. A *closed recurrent set* C of states in M is a bottom strongly connected set of states in the graph (S, E). The proof of the Lemma 1 relies on the following basic properties [21]. Lemma 1 will be used in the proof of some of the following results.

1. *Property 1.* Given a Markov chain M, and a start state s, with probability 1, the set of closed recurrent states is reached from s in finite time. Hence for any $\epsilon > 0$, there exists k_0 such that for all $k > k_0$, for all starting state s, the set of closed recurrent states are reached with probability at least $1 - \epsilon$ in k steps.
2. *Property 2.* If a closed recurrent set C is reached, and the limit of the expectation of the average weights of C is α, then for all $\epsilon > 0$, there exists a k_0 such that for all $k > k_0$ the expectation of the average weights for k steps is at least $\alpha - \epsilon$.

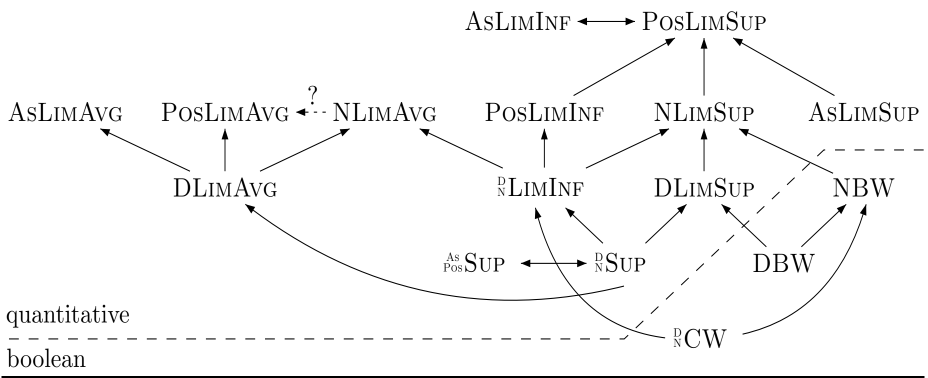


Fig. 2. Reducibility relation. $\mathcal{C}$ is reducible to $\mathcal{C}'$ if $\mathcal{C} \to \mathcal{C}'$. Classes that are not connected by an arrow are incomparable. Reducibility for the dashed arrow is open. The Disc-automata are incomparable with the automata in the figure. Their reducibility relation is given in Theorem 6.

Lemma 1. *Let A be a probabilistic weighted automaton with alphabet $\Sigma = \{a, b\}$. Consider the Markov chain arising from A on input b^ω (we refer to this as the b-Markov chain) and the a-Markov chain is defined symmetrically. The following assertions hold:*

1. *If for all closed recurrent sets C in the b-Markov chain, the expected limit-average value is at least 1, then there exists j such that for all closed recurrent sets arising from A on input $(b^j \cdot a)^\omega$ the expected limit-average reward is positive.*
2. *If for all closed recurrent sets C in the b-Markov chain, the expected limit-average value is at most 0, then there exists j such that for all closed recurrent sets arising from A on input $(b^j \cdot a)^\omega$ the expected limit-average reward is strictly less than 1.*
3. *If for all closed recurrent sets C in the b-Markov chain, the expected limit-average value is at most 0,and if for all closed recurrent sets C in the a-Markov chain, the expected limit-average value is at most 0, then there exists j such that for all closed recurrent sets arising from A on input $(b^j \cdot a^j)^\omega$ the expected limit-average reward is strictly less than 1/2.*

Proof. We present the proof of the first part. Let β be the maximum absolute value of the weights of A. From any state $s \in A$, there is a path of length at most n to a closed recurrent set C in the b-Markov chain, where n is the number of states of A. Hence if we choose $j > n$, then any closed recurrent set in the Markov chain arising on the input $(b^j \cdot a)^\omega$ contains closed recurrent sets of the b-Markov chain. For $\epsilon > 0$, there exists k_ϵ such that from any state $s \in A$, for all $k > k_\epsilon$, on input b^k from s, the closed recurrent sets of the b-Markov chain is reached with probability at least $1 - \epsilon$ (by property 1). If all closed recurrent

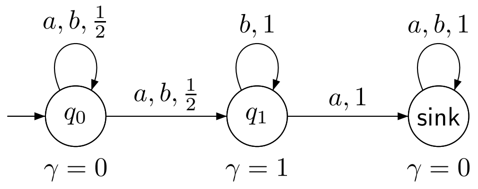


Fig. 3. PosLimAvg for Lemma 2

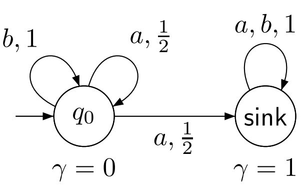


Fig. 4. AsLimAvg for Lemma 3

sets in the b-Markov chain have expected limit-average value at least 1, then by property 2 it follows that for all $\epsilon > 0$, there exists l_ϵ such that for all $l > l_\epsilon$, from all states s of a closed recurrent set on the input b^l the expected average of the weights is at least $1 - \epsilon$, (i.e., expected sum of the weights is $l - l \cdot \epsilon$). Consider $0 < \epsilon \leq \min\{1/4, 1/(20 \cdot \beta)\}$, we choose $j = k + l$, where $k = k_\epsilon > 0$ and $l > \max\{l_\epsilon, k\}$. Observe that by our choice $j + 1 \leq 2l$. Consider a closed recurrent set in the Markov chain on $(b^j \cdot a)^\omega$ and we obtain a lower bound on the expected average reward as follows: with probability $1 - \epsilon$ the closed recurrent set of the b-Markov chain is reached within k steps, and then in the next l steps at the expected sum of the weights is at least $l - l \cdot \epsilon$, and since the worst case weight is $-\beta$ we obtain the following bound on the expected sum of the rewards:

$$(1 - \epsilon) \cdot (l - l \cdot \epsilon) - \epsilon \cdot \beta \cdot (j + 1) \geq \frac{l}{2} - \frac{l}{10} = \frac{2l}{5}$$

Hence the expected average reward is at least $1/5$ and hence positive. ■

3.1 Probabilistic LimAvg-Automata

We consider the alphabet $\Sigma = \{a, b\}$ and we define the boolean language L_F of finitely many a's, i.e., $L_F(w) = 1$ if $w \in \Sigma^\omega$ consists of finitely many a's, and $L_F(w) = 0$ otherwise. We also consider the language L_I of words with infinitely many a's, i.e., the complement of L_F.

Lemma 2. *Consider the language L_F of finitely many a's. The following assertions hold.*

1. *There is no* NLimAvg *that specifies L_F.*
2. *There exists a* PosLimAvg *that specifies L_F (see Fig. 3).*
3. *There is no* AsLimSup *that specifies L_F.*

Proof. We present the proof of the third part. Assume that there exists an AsLimAvg automaton A that specifies L_F. Consider the Markov chain M that arises from A if the input is only b (i.e., on b^ω), we refer to it as the b-Markov chain. If there is a closed recurrent set C in M that can be reached from the starting state in A (reached by any sequence of a and b's in A), then the expected limit-average reward in C must be at least 1 (otherwise, if there is a closed

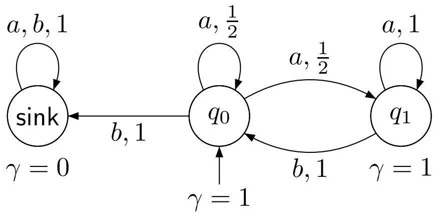


Fig. 5. A probabilistic weighted automaton (PosLimAvg, PosLimSup, or PosLimInf) for Lemma 4

recurrent set C in M with limit-average reward less than 1, we can construct a finite word w that will reach C with positive probability in A, and then follow w by b^ω yielding $A(w \cdot b^\omega) < 1$). Thus any closed recurrent set in M has limit-average reward at least 1 and by Lemma 1 there exists j such that the $A((b^j \cdot a)^\omega) > 0$. It follows that A cannot specify L_F. ■

Lemma 3. *Consider the language L_I of infinitely many a's. The following assertions hold.*

1. *There is no* NLimAvg *that specifies L_I.*
2. *There is no* PosLimAvg *that specifies L_I.*
3. *There exists an* AsLimAvg *that specifies L_I (see Fig. 4).*

Lemma 4. *There exists a language L such that: (a) there exists a* PosLimAvg, *a* PosLimSup *and a* PosLimInf *that specifies L (see Fig. 5); and (b) there is no* NLimAvg, *no* NLimSup *and no* NLimInf *that specifies L.*

The next theorem summarizes the results for limit-average automata obtained in this section.

Theorem 2. AsLimAvg *is incomparable in expressive power with* PosLimAvg *and* NLimAvg, *and* NLimAvg *is not as expressive as* PosLimAvg.

Open question. Whether NLimAvg is reducible to PosLimAvg or NLimAvg is incomparable to PosLimAvg (i.e., whether there is a language expressible by NLimAvg but not by PosLimAvg) remains open.

3.2 Probabilistic LimInf- and LimSup-Automata

To compare the expressiveness of probabilistic LimInf- and LimSup-automata, we use and extend results from [1, 7], Lemma 3 and 4, and the notion of *determinism in the limit* [12, 25]. A nondeterministic weighted automaton A is *deterministic in the limit* if for all states s of A with weight greater than the minimum weight, all states t reachable from s have deterministic transitions.

Lemma 5. *For every* NLimSup *A, there exists a* NLimSup *B that is deterministic in the limit and specifies the same quantitative language.*

Lemma 5 is useful to translate a nondeterministic automaton into a probabilistic one with positive semantics. The next lemma presents the results about reducibility of liminf automata.

Lemma 6. *The following assertions hold: (a) both* AsLimInf *and* PosLimInf *are as expressive as* NLimInf*; (b) there exists an* AsLimInf *that specifies the language* L_I*, there is no* NLimInf *and there is no* PosLimInf *that specifies* L_I*; (c)* AsLimInf *is as expressive as* PosLimInf*.*

As a corollary of Lemma 4 and Lemma 6, we get the following theorem.

Theorem 3. AsLimInf *is strictly more expressive than* PosLimInf*; and* PosLimInf *is strictly more expressive than* NLimInf*.*

The following lemma presents the results about reducibility of limsup automata.

Lemma 7. *The following assertions hold: (a)* NLimSup *and* PosLimSup *are not as expressive as* AsLimSup*; (b)* PosLimSup *is as expressive as* NLimSup*; (c)* PosLimSup *is as expressive as* AsLimSup*; (d)* AsLimSup *is not as expressive as* NLimSup*.*

Theorem 4. AsLimSup *and* NLimSup *are incomparable in expressive power, and* PosLimSup *is strictly more expressive than* AsLimSup *and* NLimSup*.*

The above theorem summarizes the reducibility results for limsup automata. Finally, we establish the reducibility relation between probabilistic LimSup- and LimInf-automata.

Theorem 5. AsLimInf *and* PosLimSup *have the same expressive power;* AsLimSup *and* PosLimInf *have incomparable expressive power.*

Proof. This result is an easy consequence of the fact that an automaton interpreted as AsLimInf specifies the complement of the language of the same automaton interpreted as PosLimSup (and similarly for AsLimSup and PosLimInf), and from the fact that AsLimInf and PosLimSup are closed under complement, while AsLimSup and PosLimInf are not (see Lemma 13). ■

3.3 Probabilistic Disc-Automata

For probabilistic discounted-sum automata, the nondeterministic and the positive semantics have the same expressive power. Intuitively, this is because the run with maximal value can be approached arbitrarily close by a finite run, and therefore the set of infinite runs sharing that finite run as a prefix has positive probability. This also shows that the positive semantics does not depend on the actual values of the probabilities, but only on whether they are positive or not. Analogous results hold for the universal semantics and the almost-sure semantics.

Theorem 6. *The following assertions hold: (a)* NDisc *and* PosDisc *have the same expressive power; (b)* UDisc *and* AsDisc *have the same expressive power.*

Proof. (a) Let $A = \langle Q, \rho_I, \Sigma, \delta_A, \gamma \rangle$ be a NDisc, and let $v_{\min}, v_{\max}$ be its minimum and maximum weights respectively. Consider the PosDisc $B = \langle Q, \rho_I, \Sigma, \delta_B, \gamma \rangle$ where $\delta_B(q, \sigma)$ is the uniform probability distribution over the set of states q' such that $(q, \sigma, \{q'\}) \in \delta_A$. Let $r = q_0 \sigma_1 q_1 \sigma_2 \dots$ be a run of A (over $w = \sigma_1 \sigma_2 \dots$) with value η. For all $\epsilon > 0$, we show that $\mathbb{P}^B(\{r \in \mathsf{Run}^B(w) \mid \mathsf{Val}(\gamma(r)) \geq \eta - \epsilon\}) > 0\}$. Let $n \in \mathbb{N}$ such that $\frac{\lambda^n}{1-\lambda} \cdot (v_{\max} - v_{\min}) \leq \epsilon$, and let $r_n = q_0 \sigma_1 q_1 \sigma_2 \dots \sigma_n q_n$. The discounted sum of the weights in r_n is at least $\eta - \frac{\lambda^n}{1-\lambda} \cdot (v_{\max})$. The probability of the set of runs over w that are continuations of r_n is positive, and the value of all these runs is at least $\eta - \frac{\lambda^n}{1-\lambda} \cdot (v_{\max} - v_{\min})$, and therefore at least $\eta - \epsilon$. This shows that $L_B(w) \geq \eta$, and thus $L_B(w) \geq L_A(w)$. Note that $L_B(w) \leq L_A(w)$ since there is no run in A (nor in B) over w with value greater than $L_A(w)$. Hence $L_B = L_A$.

Now, we prove that PosDisc is reducible to NDisc. Given a PosDisc $B = \langle Q, \rho_I, \Sigma, \delta_B, \gamma \rangle$, we construct a NDisc $A = \langle Q, \rho_I, \Sigma, \delta_A, \gamma \rangle$ where $(q, \sigma, \{q'\}) \in \delta_A$ if and only if $\delta_B(q, \sigma)(q') > 0$, for all $q, q' \in Q$, $\sigma \in \Sigma$. By analogous arguments as in the first part of the proof, it is easy to see that $L_B = L_A$.

(b) The complement of the quantitative language specified by an UDisc (resp. AsDisc) can be specified by a NDisc (resp. PosDisc). Then, the result follows from Part *a*) (essentially, given an UDisc, we obtain easily a NDisc for the complement, then an equivalent PosDisc, and finally an AsDisc for the complement of the complement, i.e., the original quantitative language). ■

4 Closure Properties of Probabilistic Weighted Automata

We consider the closure properties of the probabilistic weighted automata under the operations max, min, complement, and sum.

Closure under max **and** min**.** The closure under max holds for the positive semantics (and under min for the almost-sure semantics) using initial non-determinism (Lemma 8), while a synchronized product can be used for AsLimSup and PosLimInf (Lemma 9). In Lemma 10, we use the closure under intersection of probabilistic Büchi automata [2], and the closure under max of PosLimSup.

Lemma 8. PosLimSup, PosLimInf, *and* PosLimAvg *are closed under* max*; and* AsLimSup, AsLimInf, *and* AsLimAvg *are closed under* min.

Lemma 9. AsLimSup *is closed under* max*;* PosLimInf *is closed under* min.

Lemma 10. PosLimSup *is closed under* min*;* AsLimInf *is closed under* max.

The closure properties of LimAvg-automata in the positive semantics rely on the following lemma.

Table 1. Closure properties and decidability of emptiness and universality

		max	min	comp.	sum	emptiness	universality
positive	PosSup	✓	✓	×	✓	✓	✓
	PosLimSup	✓	✓	✓	✓	×	×
	PosLimInf	✓	✓	×	✓	✓	✓
	PosLimAvg	✓	×	×	?	?	?
	PosDisc	✓	×	×	✓	✓	? (1)
almost-sure	AsSup	✓	✓	×	✓	✓	✓
	AsLimSup	✓	✓	×	✓	✓	✓
	AsLimInf	✓	✓	✓	✓	×	×
	AsLimAvg	×	✓	×	×	?	?
	AsDisc	×	✓	×	✓	? (1)	✓

The universality problem for NDisc can be reduced to (1).
It is not known whether this problem is decidable.

Lemma 11. *Consider the alphabet $\Sigma = \{a, b\}$, and consider the languages L_a and L_b that assign the long-run average number of a's and b's, respectively. Then the following assertions hold:*

1. *There is no* PosLimAvg *that specifies the language $L_m = \min\{L_a, L_b\}$.*
2. *There is no* PosLimAvg *that specifies the language $L^* = 1 - \max\{L_a, L_b\}$.*

Lemma 12. PosLimAvg *is not closed under* min, *and* AsLimAvg *is not closed under* max.

Proof. The result for PosLimAvg follows from Lemma 11. We show that AsLimAvg is not closed under max. Consider the alphabet $\Sigma = \{a, b\}$ and the quantitative languages L_a and L_b that assign the long-run average number of a's and b's, respectively. There exist DLimAvg (and hence AsLimAvg) to specify L_a and L_b. We show that $L_m = \max(L_a, L_b)$ cannot be specified by an AsLimAvg. By contradiction, assume that A is an AsLimAvg with set of states Q that specifies L_m. Consider any closed recurrent set of the a-Markov chain of A. The expected limit-average of the weights of the recurrent set must be 1, as if we consider the word $w^* = w_C \cdot a^\omega$ where w_C is a finite word to reach C in A, the value of w^* in L_m is 1. Hence, the limit-average of the weights of all the reachable a-closed recurrent set C in A is 1.

Given $\epsilon > 0$, there exists j_ϵ such that the following properties hold:

1. from any state of A, given the word a^{j_ϵ} with probability $1 - \epsilon$ an a-closed recurrent set is reached (by property 1 for Markov chains);
2. once an a-closed recurrent set is reached, given the word a^{j_ϵ}, (as a consequence of property 2 for Markov chains) we can show that the following properties hold: (a) the expected average of the weights is at least $j_\epsilon \cdot (1 - \epsilon)$, and (b) the probability distribution of the states is with ϵ of the probability distribution of the states for the word $a^{2 \cdot j_\epsilon}$ (this holds as the probability distribution of states on words a^j converges to the probability distribution of states on the word a^ω).

Let $\beta > 1$ be a number that is greater than the absolute maximum value of weights in A. We chose $\epsilon > 0$ such that $\epsilon < \frac{1}{40\cdot\beta}$. Let $j = 2 \cdot j_\epsilon$ (such that j_ϵ satisfies the properties above). Consider the word $(a^j \cdot b^{3j})^\omega$ and the answer by A must be $\frac{3}{4}$, as $L_m((a^j \cdot b^{3j})^\omega) = \frac{3}{4}$. Consider the word $\widehat{w} = (a^{2j} \cdot b^{3j})^\omega$ and consider a closed recurrent set in the Markov chain obtain from A on $\widehat{w}$. We obtain the following lower bound on the expected limit-average of the weights: (a) with probability at least $1-\epsilon$, after $j/2$ steps, a-closed recurrent sets are reached; (b) the expected average of the weights for the segment between a^j and a^{2j} is at least $j \cdot (1-\epsilon)$; and (c) the difference in probability distribution of the states after a^j and a^{2j} is at most ϵ. Since the limit-average of the weights of $(a^j \cdot b^{3j})^\omega$ is $\frac{3}{4}$, the lower bound on the limit-average of the weights is as follows

$$\begin{aligned}(1-3\cdot\epsilon)\cdot(\tfrac{3\cdot j + j\cdot(1-\epsilon)}{5j}) - 3\cdot\epsilon\cdot\beta &= (1-\epsilon)\cdot(\tfrac{4}{5}-\tfrac{\epsilon}{5}) - 3\cdot\epsilon\cdot\beta \\ &\geq \tfrac{4}{5} - \epsilon - 3\cdot\epsilon\cdot\beta \geq \tfrac{4}{5} - 4\cdot\epsilon\cdot\beta \\ &\geq \tfrac{4}{5} - \tfrac{1}{10} \geq \tfrac{7}{10} > \tfrac{3}{5}.\end{aligned}$$

It follows that $A((a^{2j} \cdot b^{3j})^\omega) > \frac{3}{5}$. This contradicts that A specifies L_m. ∎

Closure under complement and sum. We now consider closure under complement and sum.

Lemma 13. PosLimSup *and* AsLimInf *are closed under complement; all other classes of probabilistic weighted automata are not closed under complement.*

Proof. We give the proof for limit average. The fact that PosLimAvg is not closed under complement follows from Lemma 11. We now show that AsLimAvg is not closed under complement. Consider the DLimAvg A over alphabet $\Sigma = \{a, b\}$ that consists of a single self-loop state with weight 1 for a and 0 for b. Notice that $A(w.a^\omega) = 1$ and $A(w.b^\omega) = 0$ for all $w \in \Sigma^*$. To obtain a contradiction, assume that there exists a AsLimAvg B such that $B = 1 - A$. For all finite words $w \in \Sigma^*$, let $B(w)$ be the expected average weight of the finite run of B over w. Fix $0 < \epsilon < \frac{1}{2}$. For all finite words w, there exists a number n such that the average number of a's in $w.b^n$ is at most ϵ, and there exists a number m such that $B(w.a^m) \leq \epsilon$ (since $B(w.a^\omega) = 0$). Hence, we can construct a word $w = b^{n_1}a^{m_1}b^{n_2}a^{m_2}\dots$ such that $A(w) \leq \epsilon$ and $B(w) \leq \epsilon$. Since $B = 1 - A$, this implies that $1 \leq 2\epsilon$, a contradiction. ∎

Lemma 14. *The* Sup-, LimSup-, LimInf-, *and* Disc-*automata are closed under sum under both the positive and almost-sure semantics.* AsLimAvg *is not closed under sum.*

Theorem 7. *The closure properties for probabilistic weighted automata under max, min, complement, and sum are summarized in Table 1.*

Open question. Whether PosLimAvg is closed under sum remains open.

5 Decision Problems

We conclude the paper with some decidability and undecidability results for classical decision problems about quantitative languages (see Table 1). Most of them are direct corollaries of the results in [1]. Given a weighted automaton A and a rational number $\nu \in \mathbb{Q}$, the *quantitative emptiness problem* asks whether there exists a word $w \in \Sigma^\omega$ such that $L_A(w) \geq \nu$, and the *quantitative universality problem* asks whether $L_A(w) \geq \nu$ for all words $w \in \Sigma^\omega$.

Theorem 8. *The emptiness and universality problems for* PosSup*,* AsSup*,* AsLimSup*, and* PosLimInf *are decidable.*

Theorem 9. *The emptiness and universality problems for* PosLimSup *and* AsLimInf *are undecidable.*

Finally, by Theorem 6 and the decidability of emptiness for NDisc, we get the following result.

Theorem 10. *The emptiness problem for* PosDisc *and the universality problem for* AsDisc *are decidable.*

Note that by Theorem 6, the universality problem for NDisc (which is not know to be decidable) can be reduced to the universality problem for PosDisc and to the emptiness problem for AsDisc.

Language inclusion. Given two weighted automata A and B, the *quantitative language-inclusion problem* asks whether for all words $w \in \Sigma^\omega$ we have $L_A(w) \geq L_B(w)$ and the *quantitative language-equivalence problem* asks whether for all words $w \in \Sigma^\omega$ we have $L_A(w) = L_B(w)$. It follows from our results that the language-inclusion problem is decidable for PosSup and AsSup, and is undecidable for PosLimSup and AsLimInf. The decidability of language inclusion for PosLimInf and AsLimSup remains open; the problem is also open for the respective boolean cases (i.e., for PosCW and AsBW). The decidability of language inclusion for PosLimAvg, AsLimAvg, PosDisc, and AsDisc also remains open as either the universality or the emptiness problem (or both) remain open in the respective cases. The situation for language equivalence is the same as for language inclusion.

References

1. Baier, C., Bertrand, N., Größer, M.: On decision problems for probabilistic Büchi automata. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 287–301. Springer, Heidelberg (2008)
2. Baier, C., Größer, M.: Recognizing omega-regular languages with probabilistic automata. In: Proc. of LICS: Logic in Comp. Science, pp. 137–146. IEEE, Los Alamitos (2005)
3. Blondel, V.D., Canterini, V.: Undecidable problems for probabilistic automata of fixed dimension. Theory Comput. Syst. 36(3), 231–245 (2003)

4. Chakrabarti, A., Chatterjee, K., Henzinger, T.A., Kupferman, O., Majumdar, R.: Verifying quantitative properties using bound functions. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 50–64. Springer, Heidelberg (2005)
5. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
6. Chatterjee, K., de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: Compositional quantitative reasoning. In: Proc. of QEST, pp. 179–188. IEEE Computer Society Press, Los Alamitos (2006)
7. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 385–400. Springer,Heidelberg (2008)
8. Chatterjee, K., Doyen, L., Henzinger, T.A.: Alternating weighted automata. In: Kutyłowski, M., Charatonik, W., Gębala, M. (eds.) FCT 2009. LNCS, vol. 5699, pp. 3–13. Springer, Heidelberg (2009)
9. Chatterjee, K., Doyen, L., Henzinger, T.A.: Expressiveness and closure properties for quantitative languages. In: Proc. of LICS: Logic in Computer Science. IEEE Comp. Soc. Press, Los Alamitos (to appear, 2009)
10. Chatterjee, K., Doyen, L., Henzinger, T.A.: Probabilistic weighted automata (2009), http://infoscience.epfl.ch/record/133665
11. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Quantitative stochastic parity games. In: Proc. of SODA, pp. 114–123. ACM Press, New York (2004)
12. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
13. Culik II, K., Karhumäki, J.: Finite automata computing real functions. SIAM J. Comput. 23(4), 789–814 (1994)
14. de Alfaro, L., Faella, M., Stoelinga, M.: Linear and branching metrics for quantitative transition systems. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 97–109. Springer, Heidelberg (2004)
15. Droste, M., Kuske, D.: Skew and infinitary formal power series. Theor. Comput. Sci. 366(3), 199–227 (2006)
16. Ehrenfeucht, A., Mycielski, J.: Positional strategies for mean payoff games. Int. Journal of Game Theory 8(2), 109–113 (1979)
17. Ésik, Z., Kuich, W.: An algebraic generalization of omega-regular languages. In: Fiala, J., Koubek, V., Kratochvíl, J. (eds.) MFCS 2004. LNCS, vol. 3153, pp. 648–659. Springer, Heidelberg (2004)
18. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer, Heidelberg (1997)
19. Gimbert, H.: Jeux positionnels. PhD thesis, Université Paris 7 (2006)
20. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
21. Kemeny, J.G., Snell, J.L., Knapp, A.W.: Denumerable Markov Chains. D. Van Nostrand Company (1966)
22. Rabin, M.O.: Probabilistic automata. Information and Control 6(3), 230–245 (1963)
23. Shapley, L.S.: Stochastic games. Proc. of the National Acadamy of Science USA 39, 1095–1100 (1953)
24. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
25. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state programs. In: Proc. of FOCS, pp. 327–338. IEEE, Los Alamitos (1985)

Partially-Commutative Context-Free Processes*

Wojciech Czerwiński[1], Sibylle Fröschle[2], and Sławomir Lasota[1]

[1] Institute of Informatics, University of Warsaw
[2] University of Oldenburg

Abstract. Bisimulation equivalence is decidable in polynomial time for both sequential and commutative normed context-free processes, known as BPA and BPP, respectively. Despite apparent similarity between the two classes, different techniques were used in each case. We provide one polynomial-time algorithm that works in a superclass of both normed BPA and BPP. It is derived in the setting of *partially-commutative context-free processes*, a new process class introduced in the paper. It subsumes both BPA and BPP and seems to be of independent interest.

We investigate the bisimulation equivalence of the *context-free processes*, i.e., the process graphs defined by a context-free grammar in Greibach normal form. In process algebra, there are two essentially different ways of interpreting such grammars, depending on whether the concatenation is understood as the sequential or parallel composition of processes. These two process classes are known as BPA (Basic Process Algebra) and BPP (Basic Parallel Processes) [1].

The bisimulation equivalence is decidable both in BPA and BPP [2,7]. Under the assumption of *normedness* the polynomial-time algorithms exist [5,6]. These surprising results were obtained basing on the strong *unique decomposition property* enjoyed by both classes. Despite the apparent similarity of BPA and BPP, the algorithms are fundamentally different; cf. [1] (Chapt. 9, p. 573):

> "These algorithms are both based on an exploitation of the decomposition properties enjoyed by normed transition systems; however, despite the apparent similarity of the two problems, different methods appear to be required."

In [4] a decision procedure was given for normed PA, a superclass of both BPA and BPP. It is however very complicated and has doubly exponential complexity. In [8] a polynomial-time algorithm was proposed for the normed BPA vs. BPP problem. It transforms a BPP process into BPA, if possible, and then refers to a BPA algorithm.

This paper contains a polynomial-time algorithm for a superclass of normed BPA and BPP. The algorithm simultaneously applies to BPA and BPP, thus confirming the similarity of the two classes. Our contributions are as follows.

In Section 1 we introduce a new class of *partially-commutative* context-free processes, called BPC, build on the underlying concept of *(in)dependence* of

* The first and the last author acknowledge a partial support by Polish government grants no. N206 008 32/0810 and N N206 356036.

elementary processes. BPA (no independence) and BPP (full independence) are special cases. Our main motivation was to introduce a common setting for both BPA and BPP; however, the BPC class seems to be of independent interest and may be applied, e.g., as an abstraction in program analysis.

Our first main result is the proof of the unique decomposition property for normed BPC with a transitive dependence relation (Thm 1 in Section 2).

Then in Sections 3–5 we work out our second main result: a polynomial-time algorithm for the bisimulation equivalence in the *feasible* fragment of normed BPC, to be explained below. It clearly subsumes both normed BPA and BPP but allows also for expressing, e.g., a parallel composition of inter-dependent BPA processes. It seems thus suitable for applications, e.g., for modeling of multi-core recursive programs. We sketch the main technical points below.

Recall the classical idea of approximating the bisimulation equivalence from above, by consecutive refinements $R \mapsto R \cap \exp(R)$, where $\exp(R)$ denotes the bisimulation expansion wrt. the relation R. The crucial idea underlying the BPP algorithm [6] was to ensure that the approximant R is always a congruence, and to represent it by a finite *base*; the latter requires a further additional refinement step. Our starting point was an insight of [3] that this latter step yields *the greatest norm-reducing bisimulation* contained in $R \cap \exp(R)$.

The feasibility condition requires the bisimulation expansion to preserve congruences. It appears sufficient for the above scheme to work, after a suitable adaptation, in the general setting of BPC. Roughly speaking, we demonstrate in particular that the BPP algorithm works, surprisingly, for BPA just as well!

Our algorithm efficiently processes both multisets and strings over the set of elementary processes, of pessimistically exponential size. One of technical contributions of this paper is to devise a way of combining the BPP base refinement of [6] with the BPA procedure based on compressed string algorithms [10].

1 Partially Commutative Context-Free Processes

BPA and BPP are defined by a context-free grammar in Greibach normal form. The former class is built on sequential composition of processes, while the latter one on parallel composition. Thus BPA processes are elements of the free monoid generated by non-terminal symbols; and BPP processes correspond to the free commutative monoid. Our aim in this section is to define a process class corresponding to the free *partially-commutative* monoid.

A grammar in Greibach normal form consists of a set of non-terminal symbols $\mathtt{V}$, which we call *variables* or *elementary processes*, and a finite set of productions, which we call *rules*, of the form

$$X \xrightarrow{a} \alpha, \tag{1}$$

where $\alpha \in \mathtt{V}^*$, $X \in \mathtt{V}$, and a is an alphabet letter. Additionally assume a symmetric irreflexive relation $\mathtt{I} \subseteq \mathtt{V} \times \mathtt{V}$, called the *independence relation*. For convenience we will also use the *dependence relation* $\mathtt{D} \subseteq \mathtt{V} \times \mathtt{V}$ defined as $\mathtt{D} = (\mathtt{V} \times \mathtt{V}) \setminus \mathtt{I}$. $\mathtt{D}$ is thus symmmetric and reflexive.

The independence induces an equivalence in $\mathtt{V}^*$ in a standard way: two strings over $\mathtt{V}$ are equivalent, if one can transform one into another by a sequence of transpositions of independent variables. Formally, the equivalence $\sim_{\mathtt{I}} \subseteq \mathtt{V}^* \times \mathtt{V}^*$ is the reflexive-transitive closure of the relation containing all pairs $(wXYv, wYXv)$, for $w, v \in \mathtt{V}^*$, $(X,Y) \in \mathtt{I}$; or equivalently, $\sim_{\mathtt{I}}$ is the smallest congruence in $\mathtt{V}^*$ relating all pairs (XY, YX) where $(X,Y) \in \mathtt{I}$. We work in the monoid $\mathtt{V}_{\mathtt{I}}^{\diamondsuit} = \mathtt{V}^*/{\sim_{\mathtt{I}}}$ from now on; we call $\mathtt{V}_{\mathtt{I}}^{\diamondsuit}$ the *free partially-commutative monoid* generated by $\mathtt{I}$. The subscript is usually omitted when $\mathtt{I}$ is clear from the context. Elements of $V^{\diamondsuit}$ will be called *partially-commutative processes*, or *processes* in short, and usually denoted by Greek letters α, β, ... Composition of α and β in $\mathtt{V}^{\diamondsuit}$ is written $\alpha\beta$. Empty process (identity in $\mathtt{V}^{\diamondsuit}$) will be written as ϵ. Our development is based on the decision to interpret right-hand sides α of productions (1) as elements of $\mathtt{V}^{\diamondsuit}$, instead of as words or multisets over $\mathtt{V}$. The induced class of processes we will call BPC (Basic Partially-Commutative algebra).

Formally, a *BPC process definition* Δ consists of a finite set $\mathtt{V}$ of variables, a finite alphabet $\mathcal{A}$, an independence relation $\mathtt{I} \subseteq \mathtt{V} \times \mathtt{V}$, and a finite set of rules (1), where $\alpha \in \mathtt{V}^{\diamondsuit}$, $X \in \mathtt{V}$, and $a \in \mathcal{A}$. The induced transition system has processes as states, and transitions derived according to the following rule:

$$X\beta \xrightarrow{a} \alpha\beta \quad \text{whenever} \quad (X \xrightarrow{a} \alpha) \in \Delta,\ \beta \in \mathtt{V}^{\diamondsuit}.$$

Note that when $(X,Y) \in \mathtt{I}$ it may happen that $X\beta = Y\beta'$ in $\mathtt{V}^{\diamondsuit}$. In such case the rules of both X and Y contribute to the transitions of $X\beta = Y\beta'$. Particular special cases are BPA ($\mathtt{I}$ is empty), and BPP ($\mathtt{D}$ is the identity relation).

Example 1. Let $\mathtt{I}$ contain the pairs (B,C), (T,C), (B,U), (T,U), and the symmetric ones. In the transition system induced by the rules:

$$\begin{array}{llll} P \xrightarrow{a} WBCT & W \xrightarrow{a} WBC & T \xrightarrow{t} \epsilon & B \xrightarrow{b} \epsilon \\ & W \xrightarrow{s} U & U \xrightarrow{u} \epsilon & C \xrightarrow{c} \epsilon \end{array}$$

there are, among the others, the following transitions:

$$P \xrightarrow{a^3} W(BC)^3T \xrightarrow{s} U(BC)^3T \sim_{\mathtt{I}} B^3TUC^3 \xrightarrow{b^3} TUC^3 \xrightarrow{tu} C^3 \xrightarrow{c^3} \epsilon.$$

Definition 1. A *bisimulation* is any binary relation R over processes such that $R \subseteq \exp(R)$, where $\exp(R)$, the *bisimulation expansion wrt. R*, contains all pairs (α, β) of processes such that for all $a \in \mathcal{A}$:

1. whenever $\alpha \xrightarrow{a} \alpha'$, there is β' with $\beta \xrightarrow{a} \beta'$ and $(\alpha', \beta') \in R$,
2. the symmetric condition holds,

The *bisimulation equivalence*, written as $\sim$, is the union of all bisimulations.

An equivalence $\approx \subseteq \mathtt{V}^{\diamondsuit} \times \mathtt{V}^{\diamondsuit}$ is a congruence if it is preserved by composition: $\alpha \approx \alpha'$ and $\beta \approx \beta'$ implies $\alpha\beta \approx \alpha'\beta'$. Bisimulation equivalence is a congruence both in BPA and BPP; however it needs not be so in BPC, as the following simple example shows:

Example 2. Consider $\mathtt{D} = \{(A, B), (B, A)\}$ (plus identity pairs) and the rules below; $AB \not\sim A'B'$, despite that $A \sim A'$ and $B \sim B'$:

$$A \xrightarrow{a} \epsilon \qquad A' \xrightarrow{a} \epsilon \qquad B \xrightarrow{b} \epsilon \qquad B' \xrightarrow{b} \epsilon.$$

2 The Unique Decomposition for Normed Processes

Assume that Δ is *normed*, i.e., for every variable $X \in \mathtt{V}$, there is a sequence of transitions $X \xrightarrow{a_1} \alpha_1 \dots \xrightarrow{a_k} \alpha_k = \epsilon$ leading to the empty process ϵ. The length of the shortest such sequence is *the norm* of X, written $|X|$. Norm extends additively to all processes. By the very definition of norm, a transition may decrease norm by at most 1. Those transitions that do decrease norm will be called *norm-reducing* (n-r-transitions, in short). The rules of Δ that induce such transitions will be called norm-reducing as well.

We will need a concept of norm-reducing bisimulation (n-r-bisimulation, in short), i.e., a bisimulation over the transition system restricted to only norm-reducing transitions. The appropriate norm-reducing expansion wrt. R will be written as n-r-exp(R). Every bisimulation is a n-r-bisimulation (as a norm-reducing transition must be matched in a bisimulation by a norm-reducing one) but the converse does not hold in general.

Proposition 1. *Each n-r-bisimulation, and hence each bisimulation, is norm-preserving, i.e., whenever α and β are related then $|\alpha| = |\beta|$.*

Assume from now on that variables $\mathtt{V} = \{X_1, \dots, X_n\}$ are ordered according to non-decreasing norm: $|X_i| \leq |X_j|$ whenever $i < j$. We write $X_i < X_j$ if $i < j$. Note that $|X_1|$ is necessarily 1, and that norm of a variable is at most exponential wrt. the size of Δ, understood as the sum of lengths of all rules.

We write $X^{\diamondsuit}$, for a subset $X \subseteq \mathtt{V}$ of variables, to mean the free partially-commutative monoid generated by X and the independence relation restricted to pairs from X. Clearly, $X^{\diamondsuit}$ inherits composition and identity from $\mathtt{V}^{\diamondsuit}$.

Let $\equiv$ be an arbitrary norm-preserving congruence in $\mathtt{V}^{\diamondsuit}$. Intuitively, an elementary process X_i is *decomposable* if $X_i \equiv \alpha\beta$ for some $\alpha, \beta \neq \epsilon$. Note that $|\alpha|$, $|\beta| < |X_i|$ then. For technical convenience we prefer to apply a slightly different definition. We say that X_i is *decomposable* wrt. $\equiv$, if $X_i \equiv \alpha$ for some process $\alpha \in \{X_1, \dots, X_{i-1}\}^{\diamondsuit}$; otherwise, X_i is called *prime* wrt. $\equiv$. In particular, X_1 is always prime.

Denote by $\mathtt{P}$ the set of primes wrt. $\equiv$. It is easy to show by induction on norm that for each process α there is some $\gamma \in \mathtt{P}^{\diamondsuit}$ with $\alpha \equiv \gamma$; in such case γ is called a *prime decomposition* of α. Note that a prime decomposition of X_i is either X_i itself, or it belongs to $\{X_1, \dots, X_{i-1}\}^{\diamondsuit}$. We say that $\equiv$ has the *unique decomposition property* if each process has precisely one prime decomposition. While the set $\mathtt{P}$ of primes depends on the chosen ordering of variables (in case $X_i \equiv X_j$, $i \neq j$), the unique decomposition property does not.

In general $\sim$, even if it is a congruence, needs not to have the unique decomposition property, as the following example shows:

Example 3. Let $\mathtt{I} = \{(B, C), (C, B)\}$ and the rules be as follows:

$$A \xrightarrow{a} B \qquad A' \xrightarrow{a} C \qquad B \xrightarrow{b} \epsilon \qquad C \xrightarrow{c} \epsilon.$$

Consider two equivalent processes $AC \sim A'B$. As all four variables are prime wrt. $\sim$, we have thus a process with two different prime decompositions wrt. $\sim$.

The situation is not as bad as Example 3 suggests. We can prove the unique decomposition property (Thm 1 below) assumed that $\mathtt{D}$ is transitive (hence $\mathtt{D}$ is an equivalence). Abstraction classes of $\mathtt{D}$ we call *threads*. Intuitively, a process (i.e., an abstraction class of $\sim_{\mathtt{I}}$) may be seen as a collection of strings over $\mathtt{V}$, one for each thread. For convenience, each of the strings will be called thread as well. This concrete representation will be extensively exploited in the sequel.

For $\alpha, \gamma \in \mathtt{V}^{\diamondsuit}$ we say that α *masks* γ if any thread nonempty in γ is also nonempty in α. A binary relation $\approx$ is called:

– *strongly right-cancellative* if whenever $\alpha\gamma \approx \beta\gamma$ then $\alpha \approx \beta$;
– *right-cancellative* if whenever $\alpha\gamma \approx \beta\gamma$ and both α and β mask γ then $\alpha \approx \beta$.

Proposition 2. *If a congruence has the unique decomposition property then it is strongly right-cancellative.*

Proposition 3. *If $\approx$ is strongly right-cancellative then exp$(\approx)$ and n-r-exp$(\approx)$ are right-cancellative.*

A counterexample to the unique decomposition property of $\equiv$, if any, is a pair (α, β) of processes from $\mathtt{P}^{\diamondsuit}$ with $\alpha \equiv \beta$, $\alpha \neq \beta$. If there is a counterexample, there is one of minimal norm; we call it a *minimal $\equiv$-counterexample.* A congruence $\equiv$ is *weakly right-cancellative* if whenever $\alpha\gamma \equiv \beta\gamma$ and both α and β mask γ and $(\alpha\gamma, \beta\gamma)$ is a minimal $\equiv$-counterexample then $\alpha \equiv \beta$.

Theorem 1. *Assume $\mathtt{D}$ to be transitive. Then each weakly right-cancellative congruence that is a n-r-bisimulation has the unique decomposition property.*

It is a generalization of the unique decomposition in BPA and BPP (cf. [5] and [6], resp.), in two respects. Firstly, we consider a wider class of processes. Secondly, we treat every n-r-bisimulation that is a weak right-cancellative congruence, while in the two cited papers the result was proved only for the bisimulation equivalence $\sim$ (cf. Prop. 10 in Section 3.2, where we consider the unique decomposition property of $\sim$). The rest of this section is devoted to the proof.

Proof of Thm 1. Fix a weakly right-cancellative congruence $\equiv$ that is a n-r-bisimulation; $\equiv$ is thus norm-preserving. Let $\mathtt{P} \subseteq \mathtt{V}$ denote primes wrt. $\equiv$, ordered consistently with the ordering $\leq$ of $\mathtt{V}$. For the sake of contradiction, suppose that the unique decomposition property does not hold, and consider a minimal $\equiv$-counterexaple (α, β).

We say that a transition of one of α, β is *matched* with a transition of the other if the transitions are equally labelled and the resulting processes are related by $\equiv$.

Clearly, each norm-reducing transition of one of α, β may be matched with a transition of the other. Due to the minimality of the counterexample (α, β), *any* prime decompositions of the resulting processes, say α' and β', are necessarily identical. For convenience assume that each right-hand side of $\varDelta$ was replaced by a prime decomposition wrt. $\equiv$. Thus α', β' must be identical.

Let t be the number of threads and let $\mathtt{V} = \mathtt{V}_1 \cup \ldots \cup \mathtt{V}_t$ be the partition of $\mathtt{V}$ into threads. A process α restricted to the ith thread we denote by $\alpha_i \in \mathtt{V}_i{}^*$. Hence $\alpha = \alpha_1 \ldots \alpha_t$ and the order of composing the processes α_i is irrelevant.

A (n-r-)transition of α, or β, is always a transition of the first variable in some α_i, or β_i; such variables we call *active*. Our considerations will strongly rely on the simple observation: a n-r-transition of an active variable X may 'produce' only variables of strictly smaller norm than X, thus smaller than X wrt. $\leq$.

In Claims 1–6, to follow, we gradually restrict the possible form of (α, β).

Claim 1. For each $i \leq t$, one of α_i, β_i is a suffix of the other.

Proof. Suppose that some thread i does not satisfy the requirement, and consider the longest common suffix γ of α_i and β_i. Thus γ is masked in α and β. As $\equiv$ is weakly right-cancellative, γ must be necessarily empty – otherwise we would obtain a smaller counterexample. Knowing that the last letters of α_i and β_i, say P_α, P_β, are different, we perform a case-analysis to obtain a contradiction. The length of a string w is written $\|w\|$.

CASE 1: $\|\alpha_i\| \geq 2$, $\|\beta_i\| \geq 2$. After performing any pair of matching n-r-transitions, the last letters P_α, P_β will still appear in the resulting processes α', β', thus necessarily $\alpha' \neq \beta'$ – a contradiction to the minimality of (α, β).
CASE 2: $\|\alpha_i\| = 1$, $\|\beta_i\| \geq 2$ (or a symmetric one). Thus $\alpha_i = P_\alpha$. As P_α is prime, some other thread is necessarily nonempty in α. Perform any n-r-transition from that other thread. Irrespective of a matching move in β, the last letters P_α and P_β still appear in the resulting processes – a contradiction.
CASE 3: $\|\alpha_i\| = \|\beta_i\| = 1$. Thus $\alpha_i = P_\alpha$, $\beta_i = P_\beta$. Similarly as before, some other thread must be nonempty both in α and β. Asssume wlog. $|P_\alpha| \geq |P_\beta|$. Perform any n-r-transition in α from a thread different than i. Irrespective of a matching move in β, in the resulting processes α', β' the last letter P_α in $\alpha'{}_i$ is different from the last letter (if any) in $\beta'{}_i$ – a contradiction. □

Claim 2. For each $i \leq t$, either $\alpha_i = \beta_i$, or $\alpha_i = \epsilon$, or $\beta_i = \epsilon$.

Proof. By minimality of (α, β). If α_i, say, is a proper suffix of β_i, then a n-r-transition of α_i may not be matched in β. □

A thread i is called *identical* if $\alpha_i = \beta_i \neq \epsilon$.

Claim 3. A n-r-transition of one of α, β from an identical thread may be matched only with a transition from the same thread.

Proof. Consider an identical thread i. A n-r-transition of α_i decreases $|\alpha_i|$. By minimality of (α, β), $|\beta_i|$ must be decreased as well. □

Claim 4. There is no identical thread.

Proof. Assume thread i is identical. Some other thread j is not as $\alpha \neq \beta$; wlog. assume $|\alpha_j| > |\beta_j|$, using Claim 1. Consider a n-r-transition of the active variable in $\alpha_i = \beta_i$ that maximises the increase of norm on thread j. This transition, performed in α, may not be matched in β, due to Claim 3, so that the norms of α_j and β_j become equal. □

Claim 5. One of α, β, say α, has only one nonempty thread.

Proof. Consider the greatest (wrt. $\leq$) active variable and assume wlog. that it appears in α. We claim α has only one nonempty thread. Indeed, if some other thread is nonempty, a n-r-transition of this thread can not be matched in β. □

Let α_i be the only nonempty thread in α, and let P_i be the active variable in that thread, $\alpha_i = P_i\gamma_i$. The process γ_i is nonempty by primality of P_i.

Claim 6. $|P_i|$ is greater than norm of any variable appearing in γ_i.

Proof. Consider any thread $\beta_j = P_j\gamma_j$ nonempty in β. We know that $|P_i| \geq |P_j|$. As the thread i is empty in β, the norm of P_j must be sufficiently large to "produce" all of γ_i in one n-r-transition, i.e., $|P_j| > |\gamma_i|$. Thus $|P_i| > |\gamma_i|$. □

Now we easily derive a contradiction. Knowing that P_i has the greatest norm in α, consider the processes $P_i\alpha \equiv P_i\beta$, and an arbitrary sequence of $|P_i|+1$ norm-reducing transitions from $P_i\beta$. We may assume that this sequence does not touch P_i as $|\beta| = |\alpha| > |P_i|$. Let β' be the resulting process, and let α' denote the process obtained by performing some matching transitions from $P_i\alpha = P_iP_i\gamma_i$. The variable P_i may not appear in α' while it clearly appears in β'. Thus $\alpha' \equiv \beta'$, $\alpha' \neq \beta'$ and $|\alpha'| = |\beta'|$ is smaller than $|\alpha| = |\beta|$ – a contradiction to the minimality of the counterexample (α, β). This completes the proof of the theorem.

3 The Algorithm

From now on we only consider normed BPC process definitions Δ with a transitive dependence relation $\mathtt{D}$. Such Δ is called *feasible* if it satifies the following:

Assumption 1 (Feasibility). Whenever $\equiv$ is a congruence, then the relations $\equiv \cap \exp(\equiv)$ and $\equiv \cap$ n-r-$\exp(\equiv)$ are congruences as well.

Clearly, not all normed BPC process definitions are feasible (cf., e.g., Example 2 and the smallest congruence $\equiv$ such that $A \equiv A'$ and $B \equiv B'$).

Proposition 4. *Every normed BPA or BPP process definition is feasible.*

We prefer to separate description of the algorithm from the implementation details. In this section we provide an outline of the algorithm only. In Section 4 we explain how each step can be implemented. Without further refinement, this would give an exponential-time procedure. Finally, in Section 5 we provide the polynomial-time implementation of crucial subroutines. Altogether, Sections 3–5 contain the proof of our main result:

Theorem 2. *The bisimulation equivalence $\sim$ is decidable in polynomial time for feasible BPC process definitions.*

The algorithm will compute a finite representation of $\sim$. From now on let Δ be a fixed feasible process definition with variables $\mathtt{V}$ and dependence $\mathtt{D}$; we also fix an ordering of variables $\mathtt{V} = \{X_1, \ldots, X_n\}$.

3.1 Bases

A *base* will be a finite representation of a congruence having the unique decomposition property. A base $\mathtt{B} = (\mathtt{P}, \mathtt{E})$ consists of a subset $\mathtt{P} \subseteq \mathtt{V}$ of variables, and a set $\mathtt{E}$ of equations $(X_i = \alpha)$ with $X_i \notin \mathtt{P}$, $\alpha \in (\mathtt{P} \cap \{X_1, \ldots, X_{i-1}\})^{\diamondsuit}$ and $|X_i| = |\alpha|$. We assume that there is precisely one equation for each $X_i \notin \mathtt{P}$.

An equation $(X_i = \alpha) \in \mathtt{E}$ is thought to specify a decomposition of a variable $X_i \notin \mathtt{P}$ in $\mathtt{P}^{\diamondsuit}$. Put $d_{\mathtt{B}}(X_i) = \alpha$ if $(X_i = \alpha) \in \mathtt{E}$ and $d_{\mathtt{B}}(X_i) = X_i$ if $X_i \in \mathtt{P}$. We want $d_{\mathtt{B}}$ to unambiguously extend to all processes as a homomorphism from $\mathtt{V}^{\diamondsuit}$ to $\mathtt{P}^{\diamondsuit}$. This is only possible when, intuitively, decompositions of independent variables are independent. Formally, we say that a base $\mathtt{B}$ is $\mathtt{I}$-*preserving* if whenever $(X_i, X_j) \in \mathtt{I}$, and $d_{\mathtt{B}}(X_i) = \alpha$, $d_{\mathtt{B}}(X_j) = \beta$, then $\alpha\beta = \beta\alpha$ in $\mathtt{P}^{\diamondsuit}$.

The elements of $\mathtt{P}$ are, a priori, arbitrarily chosen, and not to be confused with the primes wrt. a given congruence. However, an $\mathtt{I}$-preserving base $\mathtt{B}$ naturally induces a congruence $=_{\mathtt{B}}$ on $\mathtt{V}^{\diamondsuit}$: $\alpha =_{\mathtt{B}} \beta$ iff $d_{\mathtt{B}}(\alpha) = d_{\mathtt{B}}(\beta)$. It is easy to verify that primes wrt. $=_{\mathtt{B}}$ are precisely variables from $\mathtt{P}$ and that $=_{\mathtt{B}}$ has the unique decomposition property. Conversely, given a congruence $\equiv$ with the latter property, one easily obtains a base $\mathtt{B}$: take primes wrt. $\equiv$ as $\mathtt{P}$, and the (unique) prime decompositions of decomposable variables as $\mathtt{E}$. $\mathtt{B}$ is guaranteed to be $\mathtt{I}$-preserving, by the uniqueness of decomposition of $XY \equiv YX$, for $(X, Y) \in \mathtt{I}$. As these two transformations are mutually inverse, we have just shown:

Proposition 5. *A norm-preserving congruence in* $\mathtt{V}^{\diamondsuit}$ *has unique decomposition property iff it equals* $=_{\mathtt{B}}$*, for an* $\mathtt{I}$*-preserving base* $\mathtt{B}$.

This allows us, in particular, to speak of *the base of* a given congruence, if it exhibits the unique decomposition property; and to call elements of $\mathtt{P}$ *primes.*

3.2 Outline of the Algorithm

For an equivalence $\equiv$ over processes, let gnrb($\equiv$) denote the greatest n-r-bisimulation that is contained in $\equiv$, defined as the union of all n-r-bisimulations contained in $\equiv$. It admits the following fix-point characterization:

Proposition 6. $(\alpha, \beta) \in$ *gnrb*$(\equiv)$ *iff* $\alpha \equiv \beta$ *and* $(\alpha, \beta) \in$ *n-r-exp*$($*gnrb*$(\equiv))$.

Proposition 7. *(i) if* $\equiv$ *is a congruence then gnrb*$(\equiv)$ *is a congruence as well. (ii) if* $\equiv$ *is right-cancellative then gnrb*$(\equiv)$ *is weakly right-cancellative.*

Proof. (i) gnrb($\equiv$) is the intersection of the descending chain of relations $\equiv_1 := \equiv \cap$ n-r-exp($\equiv$), $\equiv_2 := \equiv_1 \cap$ n-r-exp($\equiv_1$), $\ldots$. Due to Assumption 1 each $\equiv_i$ is a congruence, hence gnrb($\equiv$) is a congruence too.

(ii) Consider a minimal gnrb($\equiv$)-counterexample $(\alpha\gamma, \beta\gamma)$ such that γ is masked both by α and β. Hence each n-r-transition of α (β) is matched by a transition of β (α); the resulting processes have the same prime decompositions, due to minimality of $(\alpha\gamma, \beta\gamma)$, and thus are related by gnrb($\equiv$). This proves that $(\alpha, \beta) \in$ n-r-exp(gnrb($\equiv$)). Due to right-cancellativity of $\equiv$ we have also $\alpha \equiv \beta$. Now by the *if* implication of Prop. 6 we deduce $(\alpha, \beta) \in$ gnrb($\equiv$). $\square$

Here is the overall idea. We start with the initial congruence $=_{\mathtt{B}}$ that relates processes of equal norm, and then perform the fixpoint computation by refining $=_{\mathtt{B}}$ until it finally stabilizes. The initial approximant has the unique decomposition property. To ensure that all consecutive approximants also have the property, we apply the refinement step: $=_{\mathtt{B}} \mapsto$ gnrb($=_{\mathtt{B}} \cap$ exp($=_{\mathtt{B}}$)). By Assumption 1, $=_{\mathtt{B}} \cap$ exp($=_{\mathtt{B}}$) is a congruence, and by Prop. 2 and 3 it is right-cancellative. Thus Prop. 7 applies to gnrb($=_{\mathtt{B}} \cap$ exp($=_{\mathtt{B}}$)) and in consequence of Thm. 1 we get:

Proposition 8. *gnrb*($=_{\mathtt{B}} \cap$ *exp*($=_{\mathtt{B}}$)) *is a congruence with the unique decomposition property.*

Outline of the algorithm:

(1) Compute the base $\mathtt{B}$ of 'norm equality'.
(2) If $=_{\mathtt{B}}$ is a bisimulation then halt and return $\mathtt{B}$.
(3) Otherwise, compute the base of the congruence gnrb($=_{\mathtt{B}} \cap$ exp($=_{\mathtt{B}}$)).
(4) Assign this new base to $\mathtt{B}$ and go to step (2).

This scheme is a generalization of the BPP algorithm [6]. As our setting is more general, the implementation details, to be given in the next sections, will be necessarily more complex than in [6]. However, termination and correctness may be proved without inspecting the details of implementation:

Proposition 9 (termination). *The number of iterations is smaller than n.*

Proof. In each iteration the current relation $=_{\mathtt{B}}$ gets strictly finer (if $\mathtt{B}$ did not change in one iteration, then $=_{\mathtt{B}}$ would necessarily be a bisimulation). Therefore all prime variables stay prime, and at least one non-prime variable becomes prime. To prove this suppose the contrary. Consider the smallest X_i wrt. $\leq$ such that its prime decomposition changes during the iteration. X_i has thus two different prime decompositions (wrt. the 'old' relation $=_{\mathtt{B}}$), a contradiction. $\square$

Proposition 10 (correctness). *The algorithm computes the base of $\sim$.*

Proof. The invariant $\sim \subseteq =_{\mathtt{B}}$ is preserved by each iteration. The opposite inclusion $=_{\mathtt{B}} \subseteq \sim$ follows when $=_{\mathtt{B}}$ is a bisimulation. $\square$

The unique decomposition property of $\sim$ is thus only a corollary, as we did not have to prove it prior to the design of the algorithm! A crucial discovery is that the unique decomposition must only hold for the relations $\mathrm{gnrb}(=_{\mathtt{B}} \cap \exp(=_{\mathtt{B}}))$ and that these relations play a prominent role in the algorithm (cf. [3]).

4 Implementation

Step (1) is easy: recalling that $|X_1| = 1$, initialize $\mathtt{B}$ by $\mathtt{P} := \{X_1\}$, $\mathtt{E} := \{X_i = {X_1}^{|X_i|} : i = 2 \ldots n\}$. On the other hand implementations of steps (2) and (3) require some preparation. We start with a concrete characterization of $\mathtt{I}$-preserving bases $\mathtt{B} = (\mathtt{P}, \mathtt{E})$. Distinguish *monic threads* as those containing precisely one prime variable. $\mathtt{B}$ is called pure if for each decomposition $(X_i = \alpha) \in \mathtt{E}$, α contains only variables from the thread of X_i and from (other) monic threads.

Proposition 11. *A base* $\mathtt{B}$ *is* $\mathtt{I}$*-preserving if and only if it is pure.*

Proof. The *if* implication is immediate: if $\mathtt{B}$ is pure and the decompositions $\alpha = d_{\mathtt{B}}(X_i)$, $\beta = d_{\mathtt{B}}(X_j)$ of independent variables X_i, X_j both contain a prime from some thread, then the thread is necessarily monic. Thus $\alpha\beta = \beta\alpha$. This includes the case when any of X_i, X_j is prime. The *only if* implication is shown as follows. Consider a decomposition $(X_i = \alpha) \in \mathtt{E}$ and any prime X_j, appearing in α, from a thread different than that of X_i. As $\mathtt{B}$ is $\mathtt{I}$-preserving, $X_j\alpha = \alpha X_j$. Hence α, restricted to the thread of X_j, must be a monomial ${X_j}^k$. As X_j was chosen arbitrary, we deduce that this thread must be a monic one. □

Let $\mathtt{B} = (\mathtt{P}, \mathtt{E})$ be a pure base. Two variables $X_i, X_j \notin \mathtt{P}$ are *compatible* if either they are independent, or $(X_i, X_j) \in \mathtt{D}$, $(X_i = \alpha)$, $(X_j = \beta) \in \mathtt{E}$ and α and β contain primes from the same threads. That is, α contains a prime from a thread iff β contains a prime from that thread. Note that it must be the same prime only in case of a (necessarily monic) thread different from the thread of X_i and X_j. $\mathtt{B}$ is compatible if all pairs of non-prime variables are compatible.

Proposition 12. *Let* $\mathtt{B}$ *be pure. If* $=_{\mathtt{B}}$ *is a n-r-bisimulation then* $\mathtt{B}$ *is compatible.*

Proof. Assume $(X_i, X_j) \in \mathtt{D}$, $(X_i = \alpha)$ and $(X_j = \beta) \in \mathtt{E}$. For the sake of contradiction, suppose that some thread t is nonempty in α but empty in β: $\alpha_t \neq \epsilon$, $\beta_t = \epsilon$. We know that $(X_j X_i, \beta\alpha) \in \text{n-r-exp}(=_{\mathtt{B}})$. Hence a norm-reducing transition of $(\beta\alpha)_t = \alpha_t$ is matched by a transition of X_j, so that the decompositions of the two resulting processes are equal. In the decomposition of the left process the norm of thread t is at least $|\alpha_t|$, as X_i was not involved in the transition. On the right side, the norm decreased due to the fired transition, and is thus smaller than $|\alpha_t|$ – a contradiction. □

Step (2) may be implemented using the fact stated below. It is essentially an adaptation of the property of *Caucal base* [1] to the setting of partially-commutative processes, but the proof requires more care than in previously studied settings.

Proposition 13. *Let* $\mathtt{B} = (\mathtt{P}, \mathtt{E})$ *be pure and compatible. Then* $=_{\mathtt{B}}$ *is a bisimulation if and only if* $(X, \alpha) \in exp(=_{\mathtt{B}})$ *for each* $(X = \alpha) \in \mathtt{E}$*.*

Proof. We only need to consider the *if* direction. For any pair α, β of processes such that $\alpha =_{\mathtt{B}} \beta$, we should show that $(\alpha, \beta) \in \exp(=_{\mathtt{B}})$. Let $\gamma := d_{\mathtt{B}}(\alpha) = d_{\mathtt{B}}(\beta)$ be the prime decomposition of α and β. It is sufficient to prove that $(\alpha, \gamma) \in \exp(=_{\mathtt{B}})$, as $\exp(=_{\mathtt{B}})$ is symmetric and transitive. We will analyse the possible transitions of α and γ, knowing that all decompositions in $\mathtt{E}$ are pure.

First consider the possible transitions of α. Let X_i be an active variable in α, i.e., $\alpha = X_i \alpha'$ for some α', and let $\delta := d_{\mathtt{B}}(X_i)$. Then γ may be also split into $\gamma = \delta\gamma'$, where $\gamma' = d_{\mathtt{B}}(\alpha')$. Thus, any transition of α may be matched by a transition of γ, as we know, by assumption, that $(X_i, \delta) \in \exp(=_{\mathtt{B}})$.

Now we consider the possible transitions of γ. Let a prime X_j be active in γ. Choose a variable X_i such that X_j appears in the decomposition $\delta = d_{\mathtt{B}}(X_i)$. Due to compatibility of $\mathtt{B}$ we may assume that the chosen X_i is active in α, i.e., $\alpha = X_i \alpha'$, for some α'. Similarly as above we have $\gamma = \delta\gamma'$. A transition of X_j is necessarily a transition of δ, hence may be matched by a transition of X_i, by the assumption that $(X_i, \delta) \in \exp(=_{\mathtt{B}})$. □

Proposition 14. *The base* $\mathtt{B}$ *is pure and compatible in each iteration.*

Proof. Initially $\mathtt{B}$ is pure and compatible. After each iteration $\mathtt{B}$, being the base of $\mathrm{gnrb}(=_{\mathtt{B}} \cap \exp(=_{\mathtt{B}}))$, is pure by Prop. 11, 8 and 5, and compatible by Prop. 12. □

Therefore in step (2), the algorithm only checks the condition of Prop. 13.

Implementation of step (3). We compute the base $\mathtt{B}' = (\mathtt{P}', \mathtt{E}')$ of the greatest n-r-bisimulation contained in $=_{\mathtt{B}} \cap \exp(=_{\mathtt{B}})$. As only norm-reducing transitions are concerned, the base is obtained in a sequence of consecutive extensions, by inspecting the variables according to their ordering, as outlined below.

In the following let $\equiv$ denote the relation $=_{\mathtt{B}} \cap \exp(=_{\mathtt{B}})$. The algorithm below is an implementation of the fix-point characterization of $\mathrm{gnrb}(\equiv)$ (cf. Prop. 6).

Implementation of step (3):

> Start with the set $\mathtt{P}' = \{X_1\}$ of primes and the empty set $\mathtt{E}'$ of decompositions. Then for $i := 2, \ldots, n$ do the following:
>
> > Check if there is some $\alpha \in \mathtt{P}'^{\diamondsuit}$ such that
> >
> > (a) $(X_i, \alpha) \in$ n-r-$\exp(=_{\mathtt{B}'})$, and (b) $X_i \equiv \alpha$.
> >
> > If one is found, add $(X_i = \alpha)$ to $\mathtt{E}'$. Otherwise, add X_i to $\mathtt{P}'$ and thus declare X_i prime in $\mathtt{B}'$.

Before explaining how searching for a decomposition α of X_i is implemented, we consider the correctness issue.

Proposition 15 (correctness of step (3)). *The base* $\mathtt{B}'$ *computed in step (3) coincides with the base of* $gnrb(\equiv)$.

Now we return to the implementation of step (3). Seeking $\alpha \in (\mathtt{P}')^{\diamondsuit}$ appropriate for the decomposition of X_i is performed by an exhaustive check of all 'candidates' computed according to the procedure described below. The computation implements a necessary condition for (a) to hold: if $(X_i, \alpha) \in$ n-r-exp$(=_{\mathtt{B}'})$ then α is necessarily among the candidates.

Computing candidates α*:*

Fix an arbitrarily chosen norm-reducing rule $X_i \xrightarrow{a} \beta$ (hence $\beta \in \{X_1, \ldots, X_{i-1}\}^{\diamondsuit}$) and let $\beta' := d_{\mathtt{B}'}(\beta)$ be a decomposition of β wrt. $\mathtt{B}'$ (hence $\beta' \in (\mathtt{P}')^{\diamondsuit}$). For any $j < i$ such that $X_j \in \mathtt{P}'$, for any norm reducing rule $X_j \xrightarrow{a} \gamma$, do the following: let $\gamma' := d_{\mathtt{B}'}(\gamma)$; if $\beta' = \gamma'\gamma''$, for some γ'', then let $\alpha := X_j\gamma'' \in (\mathtt{P}')^{\diamondsuit}$ be a candidate.

We will write $\gamma \leq_{\mathtt{B}'} \beta$ to mean that $d_{\mathtt{B}'}(\gamma)$ is a prefix of $d_{\mathtt{B}'}(\beta)$.

5 Polynomial-Time Implementation

The algorithm performs various manipulations on processes. The most important are the following 'subroutines', invoked a polynomial number of times:

(i) Given α, $\beta \in \mathtt{V}^{\diamondsuit}$ and $\mathtt{B}$, check if $\alpha =_{\mathtt{B}} \beta$.
(ii) Given α, $\beta \in \mathtt{V}^{\diamondsuit}$ and $\mathtt{B}$, check if $\alpha \leq_{\mathtt{B}} \beta$. If so, compute γ such that $\alpha\gamma =_{\mathtt{B}} \beta$.

Recall that the processes involved in the algorithm are tuples of strings over prime variables, one string for each thread, of pessimistically exponential length and norm. We need thus to consider two inter-related issues:

– a succint representation of processes in polynomial space; and
– polynomial-time implementations of all manipulations, including subroutines (i) and (ii), that preserve the succint representation of manipulated data.

The special case of BPP is straightforward: the commutative processes are essentially multisets, may be thus succintly represented by storing exponents in binary, and effectively manipulated in polynomial time.

In the general case of BPC, and even in BPA, we need a more elaborate approach. To get a polynomial-time implementation, we need to use a method of 'compressed' representation of strings. Moreover, all the operations performed will have to be implemented on compressed representations, without 'decompressing' to the full exponential-size strings. After preparatory Section 5.1, in Section 5.2 we explain how to implement steps (1)–(3) in polynomial time.

5.1 Compression by an Acyclic Morphism

Let $\mathtt{A}$ be a finite alphabet and $\mathtt{S} = \{z_1, \ldots, z_m\}$ a finite set of non-terminal symbols. An *acyclic morphism* is a mapping $h : \mathtt{S} \to (\mathtt{S} \cup \mathtt{A})^*$ such that

$$h(z_i) \in (\mathtt{A} \cup \{z_1, \ldots, z_{i-1}\})^*.$$

We assume thus a numbering of symbols such that in string $h(z_i)$, only z_j with smaller index $j < i$ are allowed. Due to this acyclicity requirement, h induces a monoid morphism $h^* : \mathtt{S}^* \to \mathtt{A}^*$, as the limit of compositions $h, h^2 = h \circ h, \ldots$. Formally, $h^*(z_i) = h^k(z_i)$, for the smallest k with $h^k(z_i) \in \mathtt{A}^*$. Then the extension of h^* to all strings in $\mathtt{S}^*$ is as usual. Therefore each symbol z_i *represents* a nonempty string over A. Its length $\|h^*(z_i)\|$ may be exponentially larger than the size of h, defined as the sum of lengths of all strings $h(z_i)$.

Action of h^* on a symbol z may be presented by a finite tree, that we call the *derivation tree* of z. The root is labeled by z. If a node is labeled by some z', then the number of its children is equal to the length of $h(z')$. Their labels are consecutive letters from $h(z')$ and their ordering is consistent with the ordering of letters in $h(z')$. Nodes labeled by an alphabet letter are leaves. By acyclicity of h the tree is necessarily finite; the labels of its leaves store $h^*(z)$.

Lemma 1 ([9]). *Given an acyclic morphism h and two symbols $z, z' \in \mathtt{S}$, one may answer in polynomial-time (wrt. the size of h) the following questions:*

- *is $h^*(z) = h^*(z')$?*
- *is $h^*(z)$ a prefix of $h^*(z')$?*

A relevant parameter of a symbol z, wrt. an acyclic morphism h, is its *depth*, written $\mathtt{depth}_h(z)$, and defined as the longest path in the derivation tree of z. A depth of h, $\mathtt{depth}(h)$, is the greatest depth of a symbol.

An acyclic morphism h is *binary* if $\|h(z_i)\| \leq 2$, for all $z_i \in \mathtt{S}$. Any acyclic morphism h may be transformed to the equivalent binary one: replace each $h(z_i)$ of length greater than 2 with a balanced binary tree, using $\|h(z_i)\| - 2$ auxiliary symbols. Note that the depth d of h may increase to $d \log d$.

Lemma 2. *Given a binary acyclic morphism h, a symbol $z \in \mathtt{S}$, and $k < \|h^*(z)\|$, one may compute in polynomial-time an acyclic morphism h' extending h, such that one of new symbols of h' represents the suffix of $h^*(z)$ of length k, and* $\mathtt{size}(h') \leq \mathtt{size}(h) + \mathcal{O}(\mathtt{depth}_h(z))$ *and* $\mathtt{depth}(h') \leq \mathtt{depth}(h)$.

Proof. By inspecting the path in the derivation tree of z leading to the "cutting point", that is, to the first letter of the suffix of length k. For each symbol y appearing on this path, we will add its copy $\widetilde{y}$ to $\mathtt{S}$. Our intention is that $\widetilde{y}$ represents a suitable suffix of $h^*(y)$. This is achieved as follows. Assume that $h(y) = y_1 y_2$. If the path to the cutting point traverses y and y_1, we define h for $\widetilde{y}$ as: $h(\widetilde{y}) = \widetilde{y_1} y_2$. Otherwise, if the path traverses y_2, we put $h(\widetilde{y}) = \widetilde{y_2}$.

The total overhead in increase of size of h is constant. Hence by repeating this procedure along the whole path from the root, labelled by z, to the cutting point, the size of h will increase by $\mathcal{O}(\mathtt{depth}_h(z))$. Clearly, $\widetilde{z}$ represents the required suffix of $h^*(z)$. The new acyclic morphism is an extension of h, in the sense that value of h^* is preserved for all symbols that were previously in $\mathtt{S}$. □

5.2 Representation of a Base by an Acyclic Morphism

We focus on the case of BPA first, $\mathtt{V}^{\diamondsuit} = \mathtt{V}^*$. Extension to BPC, being straightforward, is discussed at the end of this section. The complexity considerations are wrt. the size N of Δ, i.e., the sum of lengths of all rules. The subroutines (i) and (ii) may be implemented in polynomial time due to Lemma 1 and 2.

In each iteration of the algorithm, a base $\mathtt{B} = (\mathtt{P}, \mathtt{E})$ will be represented succinctly by a binary acyclic morphism h. For each $X_i \notin \mathtt{P}$, the right-hand side of its decomposition $(X_i = \alpha_i) \in \mathtt{E}$ will be represented by a designated symbol $x_i \in \mathtt{S}$. Thus $d_{\mathtt{B}}(X_i) = \alpha_i = h^*(x_i)$. The set $\mathtt{P}$ of primes will be the alphabet.

In the initial step (1), cf. Section 4, it is easy to construct such h of size $\mathcal{O}(N)$ and depth $\mathcal{O}(N \log N)$. Implementation of step (2) will be similar to checking the conditions (a) and (b) in step (3) (cf. Sect. 4), to be described now.

Given a 'compressed' representation h of $\mathtt{B}$, we now show how to construct a representation h' of $\mathtt{B}'$ in each execution of step (3) of the algorithm; h' will be of size $\mathcal{O}(N^2 \log N)$ and depth $\mathcal{O}(N \log N)$. The alphabet $\mathtt{A}$ will be $\mathtt{P}'$.

We construct h' by consecutive extensions, according to step (3) of the algorithm. Initially, h' is empty and $\mathtt{A} = \{X_1\}$. For the extension step, suppose that each non-prime $X_j \notin \mathtt{P}'$, $j < i$, has already a designated symbol x_j in h' that represents α_j, where $(X_j = \alpha_j) \in \mathtt{E}'$. The most delicate point is the size of the representation of a 'candidate' α in step (3), as the decomposition α_i of X_i, if any, is finally found among the candidates.

Let $X_i \xrightarrow{a} \beta$ be a chosen norm-reducing rule. Replace occurences of non-prime X_j's in β by the corresponding x_j's. Extend h' by $h'(y_i) = \beta$, for a fresh auxiliary symbol y_i, and transform h' to the binary form (we say that we *encode* the rule in h'). This increases the size of h' by $\mathcal{O}(N)$ and its depth by at most $\log N$. To compute a representation of a candidate α, we extend h' similarly as above, to encode a rule $X_j \xrightarrow{a} \gamma$, by $h'(y_j) := \gamma$. Then we apply Lemma 1 to check whether $h^*(y_j)$ is a prefix of $h^*(y_i)$, and if so, apply Lemma 2 to y_i, so that the newly added symbol $\widetilde{y_i}$ represents the required suffix of $h^*(x_i)$. This increases the size of h' by $\mathcal{O}(\mathtt{depth}(h'))$ and keeps its previous depth. Finally, we compose X_j, represented by x_j, with $\widetilde{y_i}$: $h(x_i) = x_j \widetilde{y_i}$, according to step (3) of the algorithm. The cumulative increase of size and depth, after at most N repetitions of the above procedure, fits the required bounds, $\mathcal{O}(N^2 \log N)$ and $\mathcal{O}(N \log N)$, on the size and depth of h', respectively.

To test the conditions (a) and (b) we invoke the subroutine (i) for the successors of X_i and the candidate α. This involves encoding the rules of X_i and X_j in h', in the similar way as above. The condition (b) refers to $\mathtt{B}$, so we need to merge h' with h. As the total number of candidates is polynomial, it follows that the whole algorithm runs in polynomial time.

Remark 1. The input process definition may be well given in a compressed form, i.e., by an acyclic morphism representing the right-hand sides of all rules.

Implementation for BPC. The only difference is that there may be more threads than one. Hence instead of single symbol x_i, representing decomposition of X_i, we need to have a tuple of symbols, $x_i^1, \ldots, x_i^t$, where t is the number of threads, to represent the content of each thread separately. The overall idea is that the algorithm should work thread-wise, i.e., process separately the strings appearing in each thread. E.g., the subroutines (i) and (ii) may be implemented analogously as for BPA, by referring to Lemma 1 and 2 for each thread.

6 Conclusions

We have provided an evidence that the bisimulation equivalence in both normed BPA and BPP can be solved by essentially the same polynomial-time algorithm.

The algorithm works correctly in a feasible fragment of normed BPC. An interesting open question remains whether the procedure may be extended to work for all of BPC with transitive $\mathtt{D}$. This would probably require a quite different method, as the core ingredient of the approach of this paper is that the bisimulation equivalence is a congruence, which is not the case in general. Another open question is whether our setting can solve the normed BPA vs. BPP problem (disjoint union of normed BPA and BPP needs not be feasible, cf. Example 2).

It also remains to investigate possible applications of the BPC framework as an abstraction of programs, e.g., of multi-core computations.

Concerning the expressibility, normed BPC class and normed PA seem to be incomparable, even with respect to the trace equivalence (e.g., the process in Example 1 is not expressible in normed PA).

References

1. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification of infinite structures. In: Handbook of Process Algebra, pp. 545–623. Elsevier, Amsterdam (2001)
2. Christensen, S., Hüttel, H., Stirling, C.: Bisimulation equivalence is decidable for all context-free processes. Inf. Comput. 121(2), 143–148 (1995)
3. Fröschle, S., Lasota, S.: Normed processes, unique decomposition, and complexity of bisimulation equivalences. In: Proc. INFINITY 2006. ENTCS (2006) (to appear)
4. Hirshfeld, Y., Jerrum, M.: Bisimulation equivalence is decidable for normed process algebra. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 412–421. Springer, Heidelberg (1999)
5. Hirshfeld, Y., Jerrum, M., Moller, F.: A polynomial algorithm for deciding bisimilarity of normed context-free processes. Theor. Comput. Sci. 158(1-2), 143–159 (1996)
6. Hirshfeld, Y., Jerrum, M., Moller, F.: A polynomial time algorithm for deciding bisimulation equivalence of normed Basic Parallel Processes. Mathematical Structures in Computer Science 6, 251–259 (1996)
7. Jančar, P.: Bisimilarity of Basic Parallel Processes is PSPACE-complete. In: Proc. LICS 2003, pp. 218–227 (2003)
8. Jančar, P., Kot, M., Sawa, Z.: Normed BPA vs. Normed BPP revisited. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 434–446. Springer, Heidelberg (2008)
9. Karpiński, M., Rytter, W., Shinohara, A.: An efficient pattern-matching algorithm for strings with short descriptions. Nord. J. Comput. 4(2), 172–186 (1997)
10. Lasota, S., Rytter, W.: Faster algorithm for bisimulation equivalence of normed context-free processes. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 646–657. Springer, Heidelberg (2006)

Testing Finitary Probabilistic Processes
(Extended Abstract)

Yuxin Deng[1,⋆], Rob van Glabbeek[2,4], Matthew Hennessy[3,⋆⋆], and Carroll Morgan[4,⋆⋆⋆]

[1] Shanghai Jiao Tong University, China
[2] NICTA, Sydney, Australia
[3] Trinity College Dublin, Ireland
[4] University of New South Wales, Sydney, Australia

Abstract. We provide both modal- and relational characterisations of may- and must-testing preorders for recursive CSP processes with divergence, featuring probabilistic as well as nondeterministic choice. May testing is characterised in terms of simulation, and must testing in terms of failure simulation. To this end we develop weak transitions between probabilistic processes, elaborate their topological properties, and express divergence in terms of partial distributions.

1 Introduction

It has long been a challenge for theoretical computer scientists to provide a firm mathematical foundation for process-description languages that incorporate both nondeterministic and probabilistic behaviour in such a way that processes are semantically distinguished just when they can be told apart by some notion of testing.

In our earlier work [1, 3] a semantic theory was developed for one particular language with these characteristics, a finite process calculus called pCSP: nondeterminism is present in the form of the standard choice operators inherited from CSP [7], that is $P \sqcap Q$ and $P \;\Box\; Q$, while probabilistic behaviour is added via a new choice operator $P \;{}_p{\oplus}\; Q$ in which P is chosen with probability p and Q with probability $1{-}p$. The intensional behaviour of a pCSP process is given in terms of a probabilistic labelled transition system [3, 14], or pLTS, a generalisation of labelled transition systems [12]. In a pLTS the result of performing an action in a given state results in a *probability distribution* over states, rather than a single state; thus the relations $s \xrightarrow{\alpha} t$ in an LTS are replaced by relations $s \xrightarrow{\alpha} \Delta$, with Δ a distribution. Closed pCSP expressions P are interpreted as probability distributions $[\![P]\!]$ in the associated pLTS. Our semantic theory [1, 3] naturally generalises the two preorders of standard testing theory [5] to pCSP:

- $P \sqsubseteq_{\text{pmay}} Q$ indicates that Q is at least as good as P from the point of view of *possibly* passing probabilistic tests; and
- $P \sqsubseteq_{\text{pmust}} Q$ indicates instead that Q is at least as good as P from the point of view of *guaranteeing* the passing of probabilistic tests.

⋆ Deng was supported by the National Natural Science Foundation of China (60703033).
⋆⋆ Hennessy gratefully acknowledges the financial support of Science Foundation Ireland.
⋆⋆⋆ Morgan acknowledges the support of ARC Discovery Grant DP0879529.

The most significant result of [1] was an alternative characterisation of these preorders as particular forms of coinductively defined *simulation* relations, $\sqsubseteq_S$ and $\sqsubseteq_{FS}$, over the underlying pLTS. We also provided a characterisation in terms of a modal logic.

The object of the current paper is to extend the above results to a version of pCSP with recursive process descriptions: we add a construct $\mathsf{rec}\, x.\, P$ for recursion, and extend the intensional semantics of [1] in a straightforward manner. We restrict ourselves to *finitary* pCSP processes, those having finitely many states and displaying finite branching.

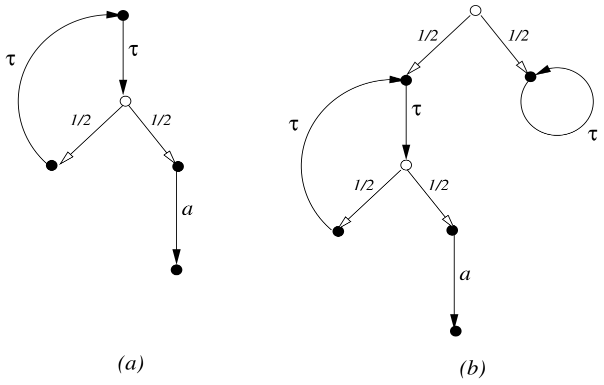


Fig. 1. The pLTSs of processes Q_1 and Q_2

The simulation relations $\sqsubseteq_S$ and $\sqsubseteq_{FS}$ in [1] were defined in terms of weak transitions $\overset{\hat{\tau}}{\Longrightarrow}$ between distributions, obtained as the transitive closure of a relation $\xrightarrow{\hat{\tau}}$ between distributions that allows one part of a distribution to make a τ-move with the other part remaining in place. This definition is however inadequate for processes that can do an unbounded number of τ-steps. The problem is highlighted by the process $Q_1 = \mathsf{rec}\, x.\, (\tau.x \;{}_{\frac{1}{2}}\!\oplus\; a.\, \mathbf{0})$ illustrated in Figure 1(a). Process Q_1 is indistinguishable, using tests, from the simple process $a.\, \mathbf{0}$: we have $Q_1 \simeq_{\text{pmay}} a.\, \mathbf{0}$ and $Q_1 \simeq_{\text{pmust}} a.\, \mathbf{0}$. This is because the process Q_1 will eventually perform the action a with probability 1. However, the action $[\![a.\, \mathbf{0}]\!] \xrightarrow{a} [\![\mathbf{0}]\!]$ can not be simulated by a corresponding move $[\![Q_1]\!] \overset{\hat{\tau}}{\Longrightarrow}\xrightarrow{a}$. No matter which distribution Δ we obtain from executing a finite sequence of internal moves $[\![Q_1]\!] \overset{\hat{\tau}}{\Longrightarrow} \Delta$, still part of it is unable to subsequently perform the action a.

To address this problem we propose a new relation $\Delta \Longrightarrow \Theta$, that indicates that Θ can be derived from Δ by performing an unbounded sequence of internal moves; we call Θ a *weak derivative* of Δ. For example $[\![a.\, \mathbf{0}]\!]$ will turn out to be a weak derivative of $[\![Q_1]\!]$, $[\![Q_1]\!] \Longrightarrow [\![a.\, \mathbf{0}]\!]$, via the infinite sequence of internal moves

$$[\![Q_1]\!] \xrightarrow{\tau} [\![Q_1 \;{}_{\frac{1}{2}}\!\oplus\; a.\, \mathbf{0}]\!] \xrightarrow{\tau} [\![Q_1 \;{}_{\frac{1}{2^2}}\!\oplus\; a.\, \mathbf{0}]\!] \xrightarrow{\tau} \ldots [\![Q_1 \;{}_{\frac{1}{2^n}}\!\oplus\; a.\, \mathbf{0}]\!] \xrightarrow{\tau} \ldots .$$

One of our contributions here is the significant use of "sub distributions" that sum to *no more than* one [8, 11]. For example, the empty subdistribution ε elegantly represents the chaotic behaviour of processes that in CSP and in must-testing semantics is tantamount to divergence, because we have $\varepsilon \xrightarrow{\alpha} \varepsilon$ for any action α, and a process like $\mathsf{rec}\, x.\, x$ that diverges via an infinite τ path gives rise to the weak transition $\mathsf{rec}\, x.\, x \Longrightarrow \varepsilon$. So the process $Q_2 = Q_1 \;{}_{\frac{1}{2}}\!\oplus\; \mathsf{rec}\, x.\, x$ illustrated in Figure 1(b) will enable the weak transition $[\![Q_2]\!] \Longrightarrow \frac{1}{2}[\![a.\, \mathbf{0}]\!]$, where intuitively the latter is a proper subdistribution mapping the state $a.\, \mathbf{0}$ to the probability $\frac{1}{2}$. Our weak transition relation $\Longrightarrow$ can be regarded as an extension of the *weak hyper-transition* from [10] to partial distributions; the latter, although defined in a very different way, can be represented in terms of ours by requiring weak derivatives to be total distributions.

We end this introduction with a brief glimpse at our proof strategy. In [1] the characterisations for finite pCSP processes were obtained using a probabilistic extension of the Hennessy-Milner logic [12]. Moving to recursive processes, we know that process behaviour can be captured by a finite modal logic only if the underlying LTS is finitely branching, or at least image-finite [12]. Thus to take advantage of a finite probabilistic HML we need a property of pLTSs corresponding to finite branching in LTSs: this is topological compactness, whose relevance we now sketch.

Subdistributions over (derivatives of) finitary pCSP processes inherit the standard (complete) Euclidean metric. One of our key results is that

Theorem 1. For every finitary pCSP process P, the set $\{\,\Delta \mid [\![P]\!] \Longrightarrow \Delta\,\}$ is convex and compact.

Indeed, using techniques from Markov Decision Theory [13] we can show that the potentially uncountable set $\{\,\Delta \mid [\![P]\!] \Longrightarrow \Delta\,\}$ is nevertheless the convex closure of a *finite* set of subdistributions, from which Theorem 1 follows.

This key result allows an *inductive* characterisation of the simulation preorders $\sqsubseteq_S$ and $\sqsubseteq_{FS}$, here defined using our novel weak derivation relation $\Longrightarrow$. We first construct a sequence of approximations $\sqsubseteq_S^k$ for $k \geq 0$ and, using Theorem 1, we prove

Theorem 2. For every finitary pCSP process P, and for every $k \geq 0$, the set $\{\,\Delta \mid [\![P]\!] \sqsubseteq_S^k \Delta\,\}$ is convex and compact.

This in turn enables us to use the *Finite Intersection Property* of compact sets to prove

Theorem 3. For finitary pCSP processes we have $P \sqsubseteq_S Q$ iff $P \sqsubseteq_S^k Q$ for all $k \geq 0$.

Our main characterisation results can then be obtained by extending the probabilistic modal logic used in [1], so that for example

– it characterises $\sqsubseteq_S^k$ for every $k \geq 0$, and therefore it also characterises $\sqsubseteq_S$
– every probabilistic modal formula can be captured by a may-test.

Similar results accrue for must testing and the new failure simulation preorder $\sqsubseteq_{FS}$: details are given in Section 6.

Due to lack of space, we omit proofs, and most examples: they are reported in [2].

2 The Language pCSP

Let Act be a set of visible actions which a process can perform, and let Var be an infinite set of variables. The language pCSP of probabilistic CSP processes is given by the following two-sorted syntax, in which $p \in [0,1]$, $a \in \mathsf{Act}$ and $A \subseteq \mathsf{Act}$:

$$\begin{aligned} P &::= S \mid P\,{}_p{\oplus}\,P \\ S &::= \mathbf{0} \mid x \in \mathsf{Var} \mid a.P \mid P \sqcap P \mid S \,\Box\, S \mid S \,|_A\, S \mid \mathsf{rec}\,x.\,P\,. \end{aligned}$$

This is essentially the finite language of [1, 3] plus the recursive construct $\mathsf{rec}\,x.\,P$ in which x is a variable and P a term. The notions of free- and bound variables are

$$a.P \xrightarrow{a} [\![P]\!] \qquad\qquad \mathsf{rec}\, x.\, P \xrightarrow{\tau} [\![P[x \mapsto \mathsf{rec}\, x.\, P]]\!]$$

$$P \sqcap Q \xrightarrow{\tau} [\![P]\!] \qquad\qquad P \sqcap Q \xrightarrow{\tau} [\![Q]\!]$$

$$\frac{s_1 \xrightarrow{a} \Delta}{s_1 \,\Box\, s_2 \xrightarrow{a} \Delta} \qquad\qquad \frac{s_2 \xrightarrow{a} \Delta}{s_1 \,\Box\, s_2 \xrightarrow{a} \Delta}$$

$$\frac{s_1 \xrightarrow{\tau} \Delta}{s_1 \,\Box\, s_2 \xrightarrow{\tau} \Delta \,\Box\, s_2} \qquad\qquad \frac{s_2 \xrightarrow{\tau} \Delta}{s_1 \,\Box\, s_2 \xrightarrow{\tau} s_1 \,\Box\, \Delta}$$

$$\frac{s_1 \xrightarrow{\alpha} \Delta \quad \alpha \notin A}{s_1 \mid_A s_2 \xrightarrow{\alpha} \Delta \mid_A s_2} \qquad\qquad \frac{s_2 \xrightarrow{\alpha} \Delta \quad \alpha \notin A}{s_1 \mid_A s_2 \xrightarrow{\alpha} s_1 \mid_A \Delta}$$

$$\frac{s_1 \xrightarrow{a} \Delta_1, \; s_2 \xrightarrow{a} \Delta_2 \; a \in A}{s_1 \mid_A s_2 \xrightarrow{\tau} \Delta_1 \mid_A \Delta_2}$$

Fig. 2. Operational semantics of pCSP

standard; by $Q[x \mapsto P]$ we indicate substitution of term P for variable x in Q, with renaming if necessary. We write pCSP for the set of closed P-terms defined by this grammar, and sCSP for its *state-based* subset of closed S-terms.

Following [1, 3], we interpret the language as a *probabilistic labelled transition system*. A (discrete) probability *subdistribution* over a set S is a function $\Delta : S \to [0,1]$ with $\sum_{s \in S} \Delta(s) \leq 1$; the *support* of such a Δ is $\lceil \Delta \rceil := \{ s \in S \mid \Delta(s) > 0 \}$, and its *mass* $|\Delta|$ is $\sum_{s \in \lceil \Delta \rceil} \Delta(s)$. A subdistribution is a (total, or full) *distribution* if $|\Delta| = 1$. The point distribution $\overline{s}$ assigns probability 1 to s and 0 to all other elements of S, so that $\lceil \overline{s} \rceil = \{s\}$. With $\mathcal{D}_{sub}(S)$ we denote the set of subdistributions over S, and with $\mathcal{D}(S)$ its subset of full distributions.

Let $\{\Delta_k \mid k \in K\}$ be a set of subdistributions, possibly infinite. Then $\sum_{k \in K} \Delta_k$ is the real-valued function in $S \to \mathbb{R}$ defined by $(\sum_{k \in K} \Delta_k)(s) := \sum_{k \in K} \Delta_k(s)$. This is a partial operation on subdistributions because for some state s the sum of $\Delta_k(s)$ might exceed 1. If the index set is finite, say $\{1..n\}$, we often write $\Delta_1 + \ldots + \Delta_n$. For p a real number from $[0,1]$ we use $p \cdot \Delta$ to denote the subdistribution given by $(p \cdot \Delta)(s) := p \cdot \Delta(s)$. Finally we use ε to denote the everywhere-zero subdistribution that thus has empty support. These operations on subdistributions do not readily adapt themselves to distributions; yet if that $\sum_{k \in K} p_k = 1$ for some collection of $p_k \geq 0$, and the Δ_k are distributions, then so is $\sum_{k \in K} p_k \cdot \Delta_k$. In general when $0 \leq p \leq 1$ we write $x \,{}_p\!\oplus\, y$ for $p \cdot x + (1-p) \cdot y$ where that makes sense, so that for example $\Delta_1 \,{}_p\!\oplus\, \Delta_2$ is always defined, and is full if Δ_1 and Δ_2 are.

The expected value $\sum_{s \in S} \Delta(s) \cdot f(s)$ over a distribution Δ of a bounded non-negative function f to the reals or tuples of them is written $\mathrm{Exp}_\Delta(f)$, and the image of a distribution Δ through a function f is written $\mathrm{Img}_f(\Delta)$ — the latter is the distribution over the range of f given by $\mathrm{Img}_f(\Delta)(t) := \sum_{f(s)=t} \Delta(s)$.

Definition 1. A *probabilistic labelled transition system* (pLTS) is a triple $\langle S, L, \to \rangle$, where

(i) S is a set of states,
(ii) L is a set of transition labels,
(iii) relation $\to$ is a subset of $S \times L \times \mathcal{D}(S)$.

A (non-probabilistic) labelled transition system (LTS) may be viewed as a degenerate pLTS — one in which only point distributions are used. As with LTSs, we write $s \xrightarrow{\alpha} \Delta$ for $(s, \alpha, \Delta) \in \rightarrow$, as well as $s \xrightarrow{\alpha}$ for $\exists \Delta : s \xrightarrow{\alpha} \Delta$ and $s \rightarrow$ for $\exists \alpha : s \xrightarrow{\alpha}$. A pLTS is *finitely branching* if the set $\{\Delta \mid s \xrightarrow{\alpha} \Delta,\ \alpha \in L\}$ is finite for all states s; if moreover S is finite, then the pLTS is *finitary*. A subdistribution Δ in an arbitrary pLTS is *finitary* if restricting the state set to the states reachable from Δ yields a finitary sub-pLTS.

The operational semantics of pCSP is defined by a particular pLTS $\langle \mathsf{sCSP}, \mathsf{Act}_\tau, \rightarrow \rangle$ in which sCSP is the set of states and $\mathsf{Act}_\tau := \mathsf{Act} \cup \{\tau\}$ is the set of transition labels; we let a range over Act and α over Act_τ. We interpret pCSP processes P as distributions $[\![P]\!] \in \mathcal{D}(\mathsf{sCSP})$ via the function $[\![\_]\!] : \mathsf{pCSP} \rightarrow \mathcal{D}(\mathsf{sCSP})$ defined by

$$[\![s]\!] \;:=\; \overline{s} \quad \text{for } s \in \mathsf{sCSP}, \quad \text{and} \quad [\![P\,{}_p\!\oplus Q]\!] \;:=\; [\![P]\!]\,{}_p\!\oplus [\![Q]\!]\,.$$

The relations $\xrightarrow{\alpha}$ are defined in Figure 2 which extends the rules used in [1, 3] for finite processes with a new rule for recursion. External choice and parallel composition use an abbreviation for distributing an operator over a distribution: for example $\Delta \,\Box\, s$ is the distribution given by $(\Delta \,\Box\, s)(t) := \Delta(s')$ if t is $s' \,\Box\, s$ and 0 otherwise. We sometimes write $\tau.P$ for $P \sqcap P$, thus giving $\tau.P \xrightarrow{\tau} [\![P]\!]$.

Note that this pLTS is finitely branching and for each $P \in \mathsf{pCSP}$ the distribution $[\![P]\!]$ has finite support. However, it is possible for there to be infinitely many states reachable from $[\![P]\!]$. If only finitely many states are reachable from $[\![P]\!]$, then P is called *finitary*.

3 Testing Probabilistic Processes

We follow the approach of [1, 3] to the testing of probabilistic processes. A *test* is simply a process from the language pCSP except that it may use extra visible actions $\omega_i \notin \mathsf{Act}_\tau$, which are assumed to be fresh, for reporting success. Given a set of test actions Ω, we write pCSP^Ω for the set of pCSP expressions using actions from $\Omega \cup \mathsf{Act}_\tau$, and sCSP^Ω for the set of state-based pCSP^Ω expressions. To apply test T to process P we form the process $T \,|_{\mathsf{Act}}\, P$ in which *all* visible actions except the ω_i must synchronise, leaving only actions τ and ω_i, and as in [1, 3] we extract testing outcomes from them. However, as processes $T \,|_{\mathsf{Act}}\, P$ are in general not of finite depth, we can no longer do this inductively. Below we outline two alternative methods that for finitary systems will turn out to be equivalent. The first one is slightly easier to explain, whereas the second one extends the work of [4, 14, 15] and is needed in establishing our results.

3.1 Extremal Testing

For the first method we assume that tests may use only a *single* success action ω. We view the unit interval $[0, 1]$ ordered in the standard manner as a complete lattice; this induces a complete lattice on the set of functions $\mathsf{sCSP}^\Omega \rightarrow [0, 1]$. Now consider the function $\mathcal{R}_{\min} : (\mathsf{sCSP}^\Omega \rightarrow [0, 1]) \rightarrow (\mathsf{sCSP}^\Omega \rightarrow [0, 1])$ defined by

$$\mathcal{R}_{\min}(f)(s) := \begin{cases} 1 & \text{if } s \xrightarrow{\omega} \\ 0 & \text{if } s \not\rightarrow \\ \min\{\, \mathrm{Exp}_\Delta(f) \mid s \xrightarrow{\alpha} \Delta \,\} & \text{otherwise.} \end{cases}$$

In a similar fashion we define the function $\mathcal{R}_{\max}$ which uses $\max$ in place of $\min$. Both these functions are monotonic, and therefore have least fixed points which we call $\mathbb{V}_{\min}$, $\mathbb{V}_{\max}$ respectively.

Now for a test T and a process P we have two ways of defining the outcome of the application of T to P:

$$\mathcal{A}^{\mathrm{e}}_{\min}(T,P) := \mathrm{Exp}_{[\![T|_{\mathsf{Act}}P]\!]}(\mathbb{V}_{\min})$$
$$\mathcal{A}^{\mathrm{e}}_{\max}(T,P) := \mathrm{Exp}_{[\![T|_{\mathsf{Act}}P]\!]}(\mathbb{V}_{\max})\ .$$

Here $\mathcal{A}^{\mathrm{e}}_{\min}(T,P)$ returns a single probability p, estimating the minimum probability of success; it is a pessimistic estimate. On the other hand $\mathcal{A}^{\mathrm{e}}_{\max}(T,P)$ is optimistic, in that it gives the maximum probability of success.

Definition 2. The *may*- and *must* preorders are given by
- $P \sqsubseteq^{e}_{\mathrm{pmay}} Q$ if for every test T we have $\mathcal{A}^{\mathrm{e}}_{\max}(T,P) \leq \mathcal{A}^{\mathrm{e}}_{\max}(T,Q)$
- $P \sqsubseteq^{e}_{\mathrm{pmust}} Q$ if for every test T we have $\mathcal{A}^{\mathrm{e}}_{\min}(T,P) \leq \mathcal{A}^{\mathrm{e}}_{\min}(T,Q)$.

3.2 Resolution-Based Testing

In the second method we use Ω-*tests* for any given collection Ω of success actions disjoint from Act_τ; here ω will be a variable ranging over the individual success actions of Ω. We calculate the result of applying test T to process P in terms of the *resolutions* of the combined process $T\ |_{\mathsf{Act}}\ P$, where intuitively a resolution represents a *run* of a process and, as such, gives exactly one probability for each success action. So in general the application of T to P will yield a *set of vectors* of probabilities.

We define the resolutions of a process $T\ |_{\mathsf{Act}}\ P$ in terms of the distribution $[\![T\ |_{\mathsf{Act}}\ P]\!]$ in the pLTS $\langle \mathsf{sCSP}^\Omega\ |_{\mathsf{Act}}\ \mathsf{sCSP}, \Omega_\tau, \rightarrow\rangle$ obtained by restricting attention to states of the form $t\ |_{\mathsf{Act}}\ s$ with $t \in \mathsf{sCSP}^\Omega$ and $s \in \mathsf{sCSP}$. Note that all transitions in this pLTS have labels τ or $\omega \in \Omega$. Following [1, 3, 5, 15], and unlike [4, 14], this paper employs *state-based* testing [1, 4], meaning that transitions $s \xrightarrow{\omega} \Delta$ are merely expedients to mark the state s as an ω-success state — the target distribution Δ is wholly ignored. Hence the pLTS can also be regarded as having just τ-labels and moreover state markers $\omega \in \Omega$. Intuitively, a resolution of a distribution in such a pLTS is obtained by pruning away multiple τ-transitions from a state until only a single choice remains, possibly introducing some linear combinations in the process.

Definition 3. A pLTS $\langle R, L, \rightarrow\rangle$ is *deterministic* if for every $r \in R$ and every $\alpha \in L$ there is at most one $\Theta \in \mathcal{D}_{sub}(R)$ such that $r \xrightarrow{\alpha} \Theta$.

A *resolution* of a subdistribution $\Delta \in \mathcal{D}_{sub}(S)$ in a pLTS $\langle S, \Omega_\tau, \rightarrow\rangle$ is a triple $\langle R, \Theta, \rightarrow'\rangle$ where $\langle R, \Omega_\tau, \rightarrow'\rangle$ is a deterministic pLTS and $\Theta \in \mathcal{D}_{sub}(R)$, such that there exists a *resolving function* $f \in R \rightarrow S$ satisfying
1. $\mathrm{Img}_f(\Theta) = \Delta$
2. if $r \xrightarrow{\alpha}{}' \Theta'$ for $\alpha \in \Omega_\tau$ then $f(r) \xrightarrow{\alpha} \mathrm{Img}_f(\Theta')$
3. if $f(r) \xrightarrow{\alpha}$ for $\alpha \in \Omega_\tau$ then $r \xrightarrow{\alpha}{}'$.

By analogy with the functions $\mathcal{R}_{\min}$ and $\mathcal{R}_{\max}$ of Section 3.1, we define the function $\mathcal{R} : (R \rightarrow [0,1]^\Omega) \rightarrow (R \rightarrow [0,1]^\Omega)$ for a deterministic pLTS $\langle R, \Omega_\tau, \rightarrow\rangle$ as

$$\mathcal{R}(f)(r)(\omega) := \begin{cases} 1 & \text{if } r \xrightarrow{\omega} \\ 0 & \text{if } r \stackrel{\omega}{\not\rightarrow} \text{ and } r \stackrel{\tau}{\not\rightarrow} \\ \mathrm{Exp}_\Delta(f)(\omega) & \text{if } r \stackrel{\omega}{\not\rightarrow} \text{ and } r \xrightarrow{\tau} \Delta. \end{cases}$$

Once more this function has a least fixed point, which we denote by $\mathbb{V}_{\langle R,\Omega_\tau,\rightarrow\rangle}$.

Now let $\mathcal{A}^\Omega(T,P)$ denote the set of vectors

$$\{\,\mathrm{Exp}_\Theta(\mathbb{V}_{\langle R,\Omega_\tau,\rightarrow\rangle}) \;\mid\; \langle R,\Theta,\rightarrow\rangle \text{ is a resolution of } [\![T \mid_{\mathsf{Act}} P]\!]\,\}\;.$$

We compare two vectors of probabilities component-wise, and two sets of vectors of probabilities via the Hoare- and Smyth preorders:

$$\begin{array}{lll} X \le_{\mathrm{Ho}} Y & \text{iff} & \forall x \in X : \exists y \in Y : x \le y \\ X \le_{\mathrm{Sm}} Y & \text{iff} & \forall y \in Y : \exists x \in X : x \le y\;. \end{array}$$

Definition 4. Given two pCSP processes P and Q,

– $P \sqsubseteq^{\Omega}_{\text{pmay}} Q$ if for every Ω-test T, we have $\mathcal{A}^\Omega(T,P) \le_{\mathrm{Ho}} \mathcal{A}^\Omega(T,Q)$
– $P \sqsubseteq^{\Omega}_{\text{pmust}} Q$ if for every Ω-test T, we have $\mathcal{A}^\Omega(T,P) \le_{\mathrm{Sm}} \mathcal{A}^\Omega(T,Q)$.

These preorders are abbreviated to $P \sqsubseteq_{\text{pmay}} Q$ and $P \sqsubseteq_{\text{pmust}} Q$ when $|\Omega|= 1$.

3.3 Equivalence of Testing Methods

In this section we compare the two approaches of testing introduced in the previous two subsections. First of all, we recall the result from [4] which says that when testing finitary processes it suffices to use a single success action rather than multiple ones[1].

Theorem 4. For finitary processes:

$$P \sqsubseteq^{\Omega}_{\text{pmay}} Q \;\text{ iff }\; P \sqsubseteq_{\text{pmay}} Q \qquad \text{and} \qquad P \sqsubseteq^{\Omega}_{\text{pmust}} Q \;\text{ iff }\; P \sqsubseteq_{\text{pmust}} Q.$$

The following theorem states that, for finitary processes, extremal testing yields the same preorders as resolution-based testing with a single success action.

Theorem 5. For finitary processes

$$P \sqsubseteq^{e}_{\text{pmay}} Q \;\text{ iff }\; P \sqsubseteq_{\text{pmay}} Q \qquad \text{and} \qquad P \sqsubseteq^{e}_{\text{pmust}} Q \;\text{ iff }\; P \sqsubseteq_{\text{pmust}} Q.$$

Neither result in Theorem 5 is true in the general (non-finitary) case, as counterexamples in [2, App. A] demonstrate. Although Theorem 4 suggests that we could have avoided multiple success actions in the resolution-based definition of testing, our completeness proof (Theorem 15) makes essential use of a countable set of them.

4 A Novel Approach to Weak Derivations

In this section we develop a new definition of what it means for a recursive process to evolve by silent activity into another process; it allows the simulation and failure-simulation preorders of [1] to be adapted to characterise the testing preorders for at least

[1] The result in [4] is stated for *action-based* testing, meaning that it is the actual execution of a success action rather than reaching a success state that constitutes success, but, as mentioned in the conclusion of [4], it also holds in our current state-based setting.

finitary probabilistic processes. The key technical generalisation is the *subdistributions* that enable us to express divergence very conveniently[2].

In a pLTS actions are only performed by states, in that actions are given by relations from states to distributions. But pCSP processes in general correspond to distributions over states, so in order to define what it means for a process to perform an action we need to *lift* these relations so that they also apply to (sub)distributions.

Definition 5. Let $(S, L, \rightarrow)$ be a pLTS and $\mathcal{R} \subseteq S \times \mathcal{D}_{sub}(S)$ be a relation from states to subdistributions. Then $\overline{\mathcal{R}} \subseteq \mathcal{D}_{sub}(S) \times \mathcal{D}_{sub}(S)$ is the smallest relation that satisfies

(1) $s \mathrel{\mathcal{R}} \Theta$ implies $\overline{s} \mathrel{\overline{\mathcal{R}}} \Theta$, and
(2) (Linearity) $\Delta_i \mathrel{\overline{\mathcal{R}}} \Theta_i$ for $i \in I$ implies $(\sum_{i\in I} p_i \cdot \Delta_i) \mathrel{\overline{\mathcal{R}}} (\sum_{i\in I} p_i \cdot \Theta_i)$ for any $p_i \in [0,1]$ with $\sum_{i\in I} p_i \leq 1$.

This applies when the relation is $\xrightarrow{\alpha}$ for $\alpha \in \mathsf{Act}_\tau$, where we also write $\xrightarrow{\alpha}$ for $\overline{\xrightarrow{\alpha}}$. Thus as source of a relation $\xrightarrow{\alpha}$ we now also allow distributions, and even subdistributions. A subtlety of this approach is that for any action α, we have $\varepsilon \xrightarrow{\alpha} \varepsilon$ simply by taking $I = \emptyset$ or $\sum_{i\in I} p_i = 0$ in Definition 5. That will turn out to make ε especially useful for modelling the "chaotic" aspects of divergence, in particular that in the must-case a divergent process can mimic any other.

We now formally define the notation of weak derivatives.

Definition 6. Suppose we have subdistributions $\Delta, \Delta_k^{\rightarrow}, \Delta_k^{\times}$, for $k \geq 0$, with the following properties:

$$\begin{aligned} \Delta &= \Delta_0^{\rightarrow} + \Delta_0^{\times} \\ \Delta_0^{\rightarrow} &\xrightarrow{\tau} \Delta_1^{\rightarrow} + \Delta_1^{\times} \\ &\vdots \\ \Delta_k^{\rightarrow} &\xrightarrow{\tau} \Delta_{k+1}^{\rightarrow} + \Delta_{k+1}^{\times} \, . \\ &\vdots \end{aligned}$$

Then we call $\Delta' := \sum_{k=0}^{\infty} \Delta_k^{\times}$ a *weak derivative* of Δ, and write $\Delta \Longrightarrow \Delta'$ to mean that Δ can make a *weak τ move* to its derivative Δ'.

It is easy to check that $\sum_{k=0}^{\infty} \Delta_k^{\times}$ is indeed a subdistribution, whereas in general it is not a full distribution: for instance we have $[\![\mathsf{rec}\, x.\, x]\!] \Longrightarrow \varepsilon$. By setting appropriate $\Delta_k^{\times}$'s to ε we see that $\Delta (\xrightarrow{\tau})^{*} \Phi$, where * denotes reflexive and transitive closure, implies $\Delta \Longrightarrow \Phi$. It is also easy to check that on recursion-free pCSP the relation $\Longrightarrow$ agrees with the one defined in [1, 3] by means of transitive closure. Moreover the standard notion of *divergence*, the ability of a subdistribution Δ to perform an infinite sequence of τ transitions, is neatly captured by the relation $\Delta \Longrightarrow \varepsilon$.

Example 1. Consider the (infinite) collection of states s_k and probabilities p_k for $k \geq 2$ such that

$$s_k \xrightarrow{\tau} [\![a.\mathbf{0}]\!]\ {}_{p_k}\!\oplus \overline{s_{k+1}} \ ,$$

[2] Subdistributions' nice properties with respect to divergence are due to their being equivalent to the discrete probabilistic powerdomain over a flat domain [8].

where we choose p_k so that starting from any s_k the probability of eventually taking a left-hand branch, and so reaching $[\![a.\mathbf{0}]\!]$ ultimately, is just $1/k$ in total. Thus p_k must satisfy $1/k = p_k + (1-p_k)/(k+1)$, whence by arithmetic we have that $p_k := 1/k^2$ will do. Therefore in particular $\overline{s_2} \Longrightarrow \frac{1}{2}[\![a.\mathbf{0}]\!]$, with the remaining $\frac{1}{2}$ lost in divergence.

Definition 7. Let Δ and its variants be subdistributions in a pLTS $\langle S, \mathsf{Act}_\tau, \rightarrow\rangle$.

- For $a \in \mathsf{Act}$ write $\Delta \overset{a}{\Longrightarrow} \Delta'$ whenever $\Delta \Longrightarrow \Delta^{\text{pre}} \xrightarrow{a} \Delta^{\text{post}} \Longrightarrow \Delta'$. Extend this to Act_τ by allowing as a special case that $\overset{\tau}{\Longrightarrow}$ is simply $\Longrightarrow$, i.e. including identity (rather than requiring at least one $\xrightarrow{\tau}$).
- For $A \subseteq \mathsf{Act}$ and $s \in S$ write $s \overset{A}{\not\longrightarrow}$ if $s \overset{\alpha}{\not\longrightarrow}$ for every $\alpha \in A \cup \{\tau\}$; write $\Delta \overset{A}{\not\longrightarrow}$ if $s \overset{A}{\not\longrightarrow}$ for every $s \in \lceil\Delta\rceil$.
- More generally write $\Delta \Longrightarrow\overset{A}{\not\longrightarrow}$ if $\Delta \Longrightarrow \Delta^{\text{pre}}$ for some Δ^{pre} such that $\Delta^{\text{pre}} \overset{A}{\not\longrightarrow}$.

For example, in Figure 1 we have $[\![Q_1]\!] \overset{a}{\Longrightarrow} [\![\mathbf{0}]\!]$, because $[\![Q_1]\!] \Longrightarrow [\![a.\mathbf{0}]\!] \xrightarrow{a} [\![\mathbf{0}]\!]$.

5 Some Properties of Weak Derivations in Finitary pLTSs

In this section we expose some less obvious properties of weak derivations from states in finitary pLTSs, relating to their behaviour at infinity; they underpin many results in the next section. One important property is that the set of weak derivations from a single starting point is *compact* in the sense (from analysis) of being bounded and containing all its limit points, where, in turn, limits depend on a Euclidean-style metric defining the distance between two distributions in a straightforward way. The other property is "distillation of divergence", allowing us to find in any weak derivation that partially diverges (by no matter how small an amount) a point at which the divergence is "distilled" into a state which wholly diverges.

Both properties depend on our working within *finitary* pLTSs — that is, ones in which the state space is finite and the (unlifted) transition relation is finite-branching.

5.1 Finite Generability and Closure

In a finitary pLTS, by definition the sets $\{\Delta \mid s \xrightarrow{\alpha} \Delta\}$ are finite, for every s and α. This of course is no longer true for the lifted relations $\xrightarrow{\alpha}$ over subdistributions; nevertheless, the sets $\{\Delta \mid \overline{s} \xrightarrow{\alpha} \Delta\}$ and their weak counterparts $\{\Delta \mid \overline{s} \overset{\alpha}{\Longrightarrow} \Delta\}$ can be finitely represented. Below, we focus on the set $\{\Delta \mid \overline{s} \Longrightarrow \Delta\}$.

Definition 8. A *static derivative policy* (SDP) for a pLTS $\langle S, \mathsf{Act}_\tau, \rightarrow\rangle$ is a partial function $\mathsf{pp} : S \rightharpoonup \mathcal{D}(S)$ such that if pp is defined at s then $s \xrightarrow{\tau} \mathsf{pp}(s)$.

Intuitively a policy pp decides for each state, once and for all, which of the available τ-choices to take, if any: since it either chooses a specific transition, or inaction (by being undefined), it does not interpolate via a convex combination of two different transitions; and since it is a function of the state, it makes the same choice on every visit.

The great importance for us of SDP's is that they give a particularly simple characterisation of weak derivatives, provided the state-space is finite and the pLTS is finitely branching. This is essentially a result of Markov Decision Processes [13], which we translate into our context. We first introduce a notion of SDP-derivatives by adapting Definition 6.

Definition 9 (SDP-derivatives). Let pp be a SDP. We write $\Delta \Longrightarrow_{\mathsf{pp}} \Delta'$ if $\Delta \Longrightarrow \Delta'$ and the following holds (using the notation of Def. 6 and writing Δ_k for $\Delta_k^{\rightarrow} + \Delta_k^{\times}$):

$$\begin{aligned} \Delta_k^{\times}(s) &= \begin{cases} 0 & \text{if } \mathsf{pp} \text{ defined at } s \\ \Delta_k(s) & \text{otherwise} \end{cases} \\ \Delta_{k+1} &= \sum\{\Delta_k(s)\cdot \mathsf{pp}(s) \mid s \in \lceil \Delta_k \rceil \text{ and } \mathsf{pp} \text{ defined at } s\}. \end{aligned}$$

Intuitively, $\Delta \Longrightarrow_{\mathsf{pp}} \Delta'$ means that Δ' is the single derivative of Δ that results from using policy pp to construct the weak transition $\Delta \Longrightarrow \Delta'$. Note that, for a given SDP pp, the relation $\Longrightarrow_{\mathsf{pp}}$ is actually a function; moreover in a finitary pLTS the set of all possible SDPs is finite, due to the constraints of Definition 8.

Theorem 6 (Finite generability). Let s be a state in a finitary pLTS $\langle S, \mathsf{Act}_\tau, \rightarrow\rangle$. Then $s \Longrightarrow \Delta$ for some $\Delta \in \mathcal{D}_{sub}(S)$ iff there is a finite index set I, probabilities p_i summing to 1 and static derivative policies pp_i with $s \Longrightarrow_{\mathsf{pp}_i} \Delta_i$ for each i, such that $\Delta = \sum_{i\in I} p_i \cdot \Delta_i$.

Since the convex closure of a finite set of points is always compact, we obtain

Corollary 1. For any state s in a finitary pLTS the set $\{\Delta \mid s \Longrightarrow \Delta\}$ is convex and compact.

A similar result is obtained by Desharnais, Gupta, Jagadeesan & Panagaden [6].

Although the pLTS $\langle \mathsf{sCSP}, \mathsf{Act}_\tau, \rightarrow\rangle$ is not finitary, the interpretation $[\![P]\!] \in \mathcal{D}(\mathsf{sCSP})$ of a finitary pCSP process P can also be understood to be a distribution in a finitary pLTS, namely the restriction of $\langle \mathsf{sCSP}, \mathsf{Act}_\tau, \rightarrow\rangle$ to the states reachable from $[\![P]\!]$. Using this, Corollary 1 leads to the essential Theorem 1, referred to in the introduction.

5.2 Distillation of Divergence

Although it is possible to have processes that diverge with some probability strictly between zero and one, in a finitary pLTS we can *distill* divergence in the sense that for many purposes we can limit our analyses to processes that either wholly diverge (can do so with probability one) or wholly converge (can diverge only with probability zero). This property is based on the zero-one law for finite-state probabilistic systems, relevant aspects of which we present in this sub-section.

We first note that static derivative policies obey the following zero-one law.

Theorem 7 (Zero-one law). If for a static derivative policy pp over a finite-state pLTS there is for some s a derivation $s \Longrightarrow_{\mathsf{pp}} \Delta$ with $|\Delta| < 1$ then in fact for some (possibly different) state s_ε we have $s_\varepsilon \Longrightarrow_{\mathsf{pp}} \varepsilon$.

Based on Theorems 6 and 7, the following property of weak derivations can now be established.

Theorem 8 (Distillation of divergence). For any s, Δ in a finitary pLTS with $s \Longrightarrow \Delta$ there is a probability p and full distributions $\Delta_1, \Delta_\varepsilon$ such that $s \Longrightarrow (\Delta_1\ {}_p\!\oplus\ \Delta_\varepsilon)$ and $\Delta = p\cdot\Delta_1$ and $\Delta_\varepsilon \Longrightarrow \varepsilon$.

6 Failure Simulation Is Sound and Complete for Must Testing

In this section we define the failure-simulation preorder and show that it is sound and complete for the must-testing preorder. The following presentation is an enhancement of our earlier definition in [1].

Definition 10 (Failure-Simulation Preorder). Define $\sqsupseteq_{FS}$ to be the largest relation in $\mathcal{D}_{sub}(S) \times \mathcal{D}_{sub}(S)$ such that if $\Delta \sqsupseteq_{FS} \Theta$ then

1. whenever $\Delta \overset{\alpha}{\Longrightarrow} (\sum_i p_i \Delta'_i)$, for $\alpha \in \mathsf{Act}_\tau$ and certain p_i with $(\sum_i p_i) \leq 1$, then there are $\Theta'_i \in \mathcal{D}_{sub}(S)$ with $\Theta \overset{\alpha}{\Longrightarrow} (\sum_i p_i \Theta'_i)$ and $\Delta'_i \sqsupseteq_{FS} \Theta'_i$ for each i
2. and whenever $\Delta \Longrightarrow\overset{A}{\not\longrightarrow}$ then also $\Theta \Longrightarrow\overset{A}{\not\longrightarrow}$.

Naturally $\Theta \sqsubseteq_{FS} \Delta$ just means $\Delta \sqsupseteq_{FS} \Theta$. For pCSP processes P and Q and any preorder $\sqsubseteq \subseteq \mathcal{D}_{sub}(\mathsf{sCSP}) \times \mathcal{D}_{sub}(\mathsf{sCSP})$ we write $P \sqsubseteq Q$ for $[\![P]\!] \sqsubseteq [\![Q]\!]$.

Although the regularity of Definition 10 is appealing — for example it is trivial to see that $\sqsubseteq_{FS}$ is reflexive and transitive, as it should be — in practice, for specific processes, it is easier to work with a characterisation of the failure-simulation preorder in terms of a relation between *states* and subdistributions.

Definition 11 (Failure Similarity). Let $\lhd_{FS}$ be the largest relation in $S \times \mathcal{D}_{sub}(S)$ such that if $s \lhd_{FS} \Theta$ then

1. whenever $\overline{s} \Longrightarrow \varepsilon$ then also $\Theta \Longrightarrow \varepsilon$,
2. whenever $s \xrightarrow{\alpha} \Delta'$, for $\alpha \in \mathsf{Act}_\tau$, then there is a Θ' with $\Theta \overset{\alpha}{\Longrightarrow} \Theta'$ and $\Delta' \overline{\lhd_{FS}} \Theta'$
3. and whenever $s \overset{A}{\not\longrightarrow}$ then $\Theta \Longrightarrow\overset{A}{\not\longrightarrow}$.

As an example, in Figure 1 it is straightforward to exhibit failure simulations to prove both $[\![Q_1]\!] \overline{\lhd_{FS}} [\![a.\mathbf{0}]\!]$ and the converse $[\![a.\mathbf{0}]\!] \overline{\lhd_{FS}} [\![Q_1]\!]$, the essential ingredient being the weak move $[\![Q_1]\!] \overset{a}{\Longrightarrow} [\![\mathbf{0}]\!]$. Likewise, we have $a.\mathbf{0} \lhd_{FS} [\![Q_1 \;{}_{\frac{1}{2}}{\oplus}\; \mathsf{rec}\, x.\, x]\!]$, the additional ingredient being $\mathbf{0} \lhd_{FS} \varepsilon$.

The next result shows how the failure-simulation preorder can alternatively be defined in terms of failure similarity. This is actually how we defined it in [1].

Theorem 9. For finitary $\Delta, \Theta \in \mathcal{D}_{sub}(S)$ we have $\Delta \sqsupseteq_{FS} \Theta$ just when there is a $\Theta^\natural$ with $\Theta \Longrightarrow \Theta^\natural$ and $\Delta \overline{\lhd_{FS}} \Theta^\natural$.

The proof of this theorem depends crucially on Theorems 1 and 8. The restriction to finitary subdistributions is essential, as in [2, App. A] we provide a counterexample to the general case. It is in terms of this characterisation that we establish soundness and completeness of the failure-simulation preorder with respect to the must-testing preorder; consequently we have these results for finitary processes only.

Theorem 10 (Precongruence). If P_1, P_2, Q_1 and Q_2 are finitary pCSP processes with $P_1 \sqsupseteq_{FS} Q_1$ and $P_2 \sqsupseteq_{FS} Q_2$ then we have $\alpha.P_1 \sqsupseteq_{FS} \alpha.Q_1$ for any $\alpha \in \mathsf{Act}_\tau$, as well as $P_1 \odot P_2 \sqsupseteq_{FS} Q_1 \odot Q_2$ for $\odot$ any of the operators $\sqcap$, $\Box$, ${}_p{\oplus}$ and $|_A$.

The proof of this precongruence property involves a significant complication: in order to relate two processes we have to demonstrate that if the first diverges then so does the second. This affects particularly the proof that $\sqsupseteq_{FS}$ is preserved by the parallel operator $|_A$. The approach we use involves first characterising divergence coinductively and then applying a novel coinductive proof technique.

Theorem 11 (Soundness and Completeness). For finitary pCSP processes P and Q we have $P \sqsubseteq_{FS} Q$ iff $P \sqsubseteq_{\text{pmust}} Q$.

Soundness, that $\sqsubseteq_{FS} \subseteq \sqsubseteq_{\text{pmust}}$, is a relatively easy consequence of $\sqsubseteq_{FS}$ being a precongruence (Theorem 10). The completeness proof (that $\sqsubseteq_{\text{pmust}} \subseteq \sqsubseteq_{FS}$) is much more complicated and proceeds in three steps, which we detail below. First we provide a characterisation of the preorder relation $\sqsubseteq_{FS}$ by finite approximations. Secondly, using this, we develop a modal logic which can be used to characterise the failure-simulation preorder on finitary processes. Finally, we adapt the results of [1] to show that the modal formulae can in turn be characterised by tests. From this, completeness follows.

6.1 Inductive Characterisation

The relation $\triangleleft_{FS}$ of Definition 11 is given coinductively: it is the largest fixpoint of an equation $\mathcal{R}= \mathcal{F}(\mathcal{R})$. An alternative approach is to use that $\mathcal{F}(-)$ to define $\triangleleft_{FS}$ as a limit of approximants:

Definition 12. For every $k \geq 0$ we define the relations $\triangleleft_{FS}^k \subseteq S \times \mathcal{D}_{sub}(S)$ as follows:

(i) $\triangleleft_{FS}^0 \ := \ S \times \mathcal{D}_{sub}(S)$
(ii) $\triangleleft_{FS}^{k+1} := \ \mathcal{F}(\triangleleft_{FS}^k)$

Finally let $\triangleleft_{FS}^\infty := \bigcap_{k=0}^\infty \triangleleft_{FS}^k$. Furthermore, for every $k \geq 0$ let $\Delta \sqsupseteq_{FS}^k \Theta$ if there exists a $\Theta \Longrightarrow \Theta^\natural$ with $\Delta \ \overline{\triangleleft_{FS}^k} \ \Theta^\natural$, and let $\sqsupseteq_{FS}^\infty$ denote $\bigcap_{k=0}^\infty \sqsupseteq_{FS}^k$.

Theorem 12. For finitary pCSP processes P and Q we have $P \sqsupseteq_{FS}^\infty Q$ iff $P \sqsupseteq_{FS} Q$.

To show this theorem, we need to use two key results, Propositions 1 and 2 below. We say a relation $\mathcal{R} \subseteq S \times \mathcal{D}(S)$ is convex (resp. compact) whenever the set $\{\Delta \mid s \ \mathcal{R} \ \Delta\}$ is convex (resp. compact) for every $s \in S$.

Proposition 1. In a finitary pLTS, the relation $\triangleleft_{FS}^k$ is convex and compact, for every $k \geq 0$.

The proof of this property heavily relies on Corollary 1.

Proposition 2. Suppose $\mathcal{R}^k \subseteq S \times \mathcal{D}_{sub}(S)$ is a sequence of convex and compact relations such that $\mathcal{R}^{k+1} \subseteq \mathcal{R}^k$. Then $(\bigcap_{k=0}^\infty \overline{\mathcal{R}^k}) \subseteq \overline{(\bigcap_{k=0}^\infty \mathcal{R}^k)}$.

This proposition is proved using the Finite Intersection Property of compact sets [9].

6.2 A Modal Logic

Let $\mathcal{F}$ be the set of modal formulae defined inductively as follows:

– $\mathbf{div}, \top \in \mathcal{F}$
– $\mathbf{ref}(A) \in \mathcal{F}$ when $A \subseteq$ Act,
– $\langle a \rangle \varphi \in \mathcal{F}$ when $\varphi \in \mathcal{F}$ and $a \in$ Act,
– $\varphi_1 \wedge \varphi_2 \in \mathcal{F}$ when $\varphi_1, \varphi_2 \in \mathcal{F}$,
– $\varphi_1 \, {}_p{\oplus} \, \varphi_2 \in \mathcal{F}$ when $\varphi_1, \varphi_2 \in \mathcal{F}$ and $p \in [0, 1]$.

This generalises the modal language used in [1] by the addition of the new constant **div**, representing the ability of a process to diverge.

Relative to a given pLTS $\langle S, \mathsf{Act}_\tau, \rightarrow\rangle$ the *satisfaction relation* $\models \subseteq \mathcal{D}_{sub}(S) \times \mathcal{F}$ is given by:

- $\Delta \models \top$ for any $\Delta \in \mathcal{D}_{sub}(S)$,
- $\Delta \models \mathbf{div}$ iff $\Delta \Longrightarrow \varepsilon$,
- $\Delta \models \mathbf{ref}(A)$ iff $\Delta \Longrightarrow \not\xrightarrow{A}$,
- $\Delta \models \langle a\rangle\varphi$ iff there is a Δ' with $\Delta \stackrel{a}{\Longrightarrow} \Delta'$ and $\Delta' \models \varphi$,
- $\Delta \models \varphi_1 \wedge \varphi_2$ iff $\Delta \models \varphi_1$ and $\Delta \models \varphi_2$,
- $\Delta \models \varphi_1\, {}_p\!\oplus\, \varphi_2$ iff there are $\Delta_1, \Delta_2 \in \mathcal{D}_{sub}(S)$ with $\Delta_1 \models \varphi_1$ and $\Delta_2 \models \varphi_2$, such that $\Delta \Longrightarrow \Delta_1\, {}_p\!\oplus\, \Delta_2$.

We write $\Delta \sqsupseteq^{\mathcal{F}} \Theta$ when $\Delta \models \varphi$ implies $\Theta \models \varphi$ for all $\varphi \in \mathcal{F}$, and can verify that $\sqsupseteq_{FS}$ is sound for $\sqsupseteq^{\mathcal{F}}$. In establishing the converse, we mimic the development in Section 7 of [1] by designing *characteristic formulae* which capture the behaviour of states in a pLTS. But here the behaviour is not characterised relative to $\lhd_{FS}$, but rather to the sequence of approximating relations $\lhd_{FS}^k$.

Definition 13. In a finitary pLTS $\langle S, \mathsf{Act}_\tau, \rightarrow\rangle$, the k^{th} *characteristic formulae* $\varphi_s^k, \varphi_\Delta^k$ of states $s \in S$ and subdistributions $\Delta \in \mathcal{D}_{sub}(S)$ are defined inductively as follows:

- $\varphi_s^0 = \top$ and $\varphi_\Delta^0 = \top$,
- $\varphi_s^{k+1} = \mathbf{div}$, provided $\overline{s} \Longrightarrow \varepsilon$,
- $\varphi_s^{k+1} = \mathbf{ref}(A) \wedge \bigwedge_{s \xrightarrow{a} \Delta} \langle a\rangle \varphi_\Delta^k$ where $A = \{a \in \mathsf{Act} \mid s \not\xrightarrow{a}\}$, provided $s \not\xrightarrow{\tau}$,
- $\varphi_s^{k+1} = \bigwedge_{s \xrightarrow{a} \Delta} \langle a\rangle \varphi_\Delta^k \wedge \bigwedge_{s \xrightarrow{\tau} \Delta} \varphi_\Delta^k$ otherwise,
- and $\varphi_\Delta^{k+1} = (\bigoplus_{s \in \lceil\Delta\rceil} \frac{\Delta(s)}{\lceil\Delta\rceil} \cdot \varphi_s^{k+1})\ {}_{\lceil\Delta\rceil}\!\oplus\ (\mathbf{div})$.

The next result relates the k^{th} characteristic formulae to the k^{th} failure similarity.

Proposition 3. For $k \geq 0$ we have

(i) $\Theta \models \varphi_s^k$ implies $s \lhd_{FS}^k \Theta$,
(ii) $\Theta \models \varphi_\Delta^k$ implies $\Theta \sqsupseteq_{FS}^k \Delta$.

Using Proposition 3 we obtain a logical characterisation of $\sqsupseteq_{FS}^\infty$ (and hence of $\sqsupseteq_{FS}$):

Theorem 13. For finitary pCSP processes P and Q we have $P \sqsupseteq^{\mathcal{F}} Q$ iff $P \sqsupseteq_{FS}^\infty Q$.

6.3 Characteristic Tests for Formulae

The import of Theorems 12 and 13 is that we can obtain completeness of the failure-simulation preorder with respect to the must-testing preorder by designing for each formula φ a test which in some sense characterises the property that a process satisfies φ. This was achieved for the pLTS generated by the recursion-free fragment of pCSP in Section 8 of [1]. Here we have generalised this technique to the pLTS generated by the set of finitary pCSP terms. The crucial property is stated as follows.

Theorem 14. For every formula $\varphi \in \mathcal{F}$ there exists a pair (T_φ, v_φ) with T_φ an Ω-test and $v_\varphi \in [0,1]^\Omega$ such that $\Delta \models \varphi$ if and only if $\exists o \in \mathcal{A}^\Omega(T_\varphi, \Delta) : o \leq v_\varphi$. Test T_φ is called a *characteristic test* of φ and v_φ is its *target value*.

This property can be shown by exploiting several characteristics of the testing function $\mathcal{A}^{\Omega}(-,-)$; unlike in [1] these cannot be obtained inductively. The most complicated one is the following.

Proposition 4. If $o \in \mathcal{A}^{\Omega}(T_1 \sqcap T_2, \Delta)$ then there are a $q \in [0,1]$ and $\Delta_1, \Delta_2 \in \mathcal{D}_{sub}(\mathsf{sCSP})$ such that $\Delta \Longrightarrow q\cdot\Delta_1 + (1-q)\cdot\Delta_2$ and $o = q\cdot o_1 + (1-q)\cdot o_2$ for certain $o_i \in \mathcal{A}^{\Omega}(T_i, \Delta_i)$.

From Theorem 14 we obtain that the must-testing preorder is at least as discriminating as the logical preorder:

Theorem 15. Let P and Q be pCSP processes. If $P \sqsupseteq^{\Omega}_{\text{pmust}} Q$ then $P \sqsupseteq^{\mathcal{F}} Q$.

The completeness result in Theorem 11 follows by combining Theorems 15, 13 and 12.

7 Simulation Is Sound and Complete for May Testing

We define a simulation preorder that can be shown sound and complete for may testing following the same strategy as for failure simulation and must testing, except that we restrict our treatment to full distributions, a simpler domain. This is possible because in may testing an infinite τ-path is not treated specially — it engages in no visible actions; in must testing, infinite τ-paths potentially can do anything (chaos).

Definition 14 (Simulation Preorder). Let $\sqsubseteq_S$ be the largest relation in $\mathcal{D}(S) \times \mathcal{D}(S)$ such that if $\Delta \sqsubseteq_S \Theta$ then

whenever $\Delta \overset{\alpha}{\Longrightarrow} (\sum_i p_i\Delta'_i)$, for $\alpha \in \mathsf{Act}_\tau$ and certain p_i with $(\sum_i p_i) \leq 1$, then there are $\Theta'_i \in \mathcal{D}(S)$ with $\Theta \overset{\alpha}{\Longrightarrow} (\sum_i p_i\Theta'_i)$ and $\Delta'_i \sqsubseteq_S \Theta'_i$ for each i.

The technical development from this point on is similar to that given in Section 6. For the modal logic, we use the set of formulae obtained from $\mathcal{F}$ by skipping the $\mathbf{div}$ and $\mathbf{ref}(A)$ clauses. However the satisfaction relation used for this sub-logic is radically different from that given in Section 6.2, because here the interpretation is relative to full distributions. Nevertheless we still obtain the counterparts of Theorems 12, 13 and 15.

Theorem 16 (Soundness and Completeness). For finitary pCSP processes P and Q we have $P \sqsubseteq_{\text{pmay}} Q$ if and only if $P \sqsubseteq_S Q$.

8 Conclusion and Related Work

In this paper we continued our previous work [1, 3, 4] in our quest for a testing theory for processes which exhibit both nondeterministic and probabilistic behaviour. We have generalised our results in [1] of characterising the may preorder as a simulation relation and the must preorder as a failure-simulation relation, from finite processes to finitary processes. To do this it was necessary to investigate fundamental structural properties of derivation sets (finite generability) and similarities (infinite approximations), which are of independent interest. The use of Markov Decision Processes and Zero-One laws was essential in obtaining our results.

Segala [14] defined two preorders called trace distribution precongruence ($\sqsubseteq_{TD}$) and failure distribution precongruence ($\sqsubseteq_{FD}$). He proved that the former coincides with an action-based version of $\sqsubseteq^{\Omega}_{\mathrm{pmay}}$ and that for "probabilistically convergent" systems the latter coincides with an action-based version of $\sqsubseteq^{\Omega}_{\mathrm{pmust}}$. The condition of probabilistic convergence amounts in our framework to the requirement that for $\Delta \in \mathcal{D}(S)$ and $\Delta \Longrightarrow \Delta'$ we have $|\Delta'| = 1$. In [10] it has been shown that $\sqsubseteq_{TD}$ coincides with a notion of simulation akin to $\sqsubseteq_S$. Other probabilistic extensions of simulation occurring in the literature are reviewed in [1, 3].

References

1. Deng, Y., van Glabbeek, R.J., Hennessy, M., Morgan, C.C.: Characterising testing preorders for finite probabilistic processes. Logical Methods in Computer Science 4(4:4) (2008)
2. Deng, Y., van Glabbeek, R.J., Hennessy, M., Morgan, C.C.: Testing finitary probabilistic processes. Full version of this extended abstract (2009), http://www.cse.unsw.edu.au/~rvg/pub/finitary.pdf
3. Deng, Y., van Glabbeek, R.J., Hennessy, M., Morgan, C.C., Zhang, C.: Remarks on testing probabilistic processes. ENTCS 172, 359–397 (2007)
4. Deng, Y., van Glabbeek, R.J., Morgan, C.C., Zhang, C.: Scalar outcomes suffice for finitary probabilistic testing. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 363–378. Springer, Heidelberg (2007)
5. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34, 83–133 (1984)
6. Desharnais, J., Gupta, V., Jagadeesan, R., Panagaden, P.: Weak bisimulation is sound and complete for PCTL*. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 355–370. Springer, Heidelberg (2002)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
8. Jones, C.: Probabilistic Non-determinism. Ph.D. Thesis, University of Edinburgh (1990)
9. Lipschutz, S.: Schaum's outline of theory and problems of general topology. McGraw-Hill, New York (1965)
10. Lynch, N., Segala, R., Vaandrager, F.W.: Observing branching structure through probabilistic contexts. SIAM Journal on Computing 37(4), 977–1013 (2007)
11. McIver, A.K., Morgan, C.C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer, Heidelberg (2005)
12. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
13. Puterman, M.: Markov Decision Processes. Wiley, Chichester (1994)
14. Segala, R.: Testing probabilistic automata. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 299–314. Springer, Heidelberg (1996)
15. Yi, W., Larsen, K.G.: Testing probabilistic and nondeterministic processes. In: Proc. PSTV 1992. IFIP Transactions C-8, pp. 47–61. North-Holland, Amsterdam (1992)

A Demonic Approach to Information in Probabilistic Systems

Josée Desharnais, François Laviolette, and Amélie Turgeon

Dép. d'informatique et de génie logiciel, Université Laval
Québec Canada G1V 0A6
{Firstname.Lastname}@ift.ulaval.ca

Abstract. This paper establishes a Stone-type duality between specifications and infLMPs. An infLMP is a probabilistic process whose transitions satisfy super-additivity instead of additivity. Interestingly, its simple structure can encode a mix of probabilistic and non-deterministic behaviors. Our duality shows that an infLMP can be considered as a demonic representative of a system's information. Moreover, it carries forward a view where states are less important, and events, or properties, become the main characters, as it should be in probability theory. Along the way, we show that bisimulation and simulation are naturally interpreted in this setting, and we exhibit the interesting relationship between infLMPs and the usual probabilistic modal logics.

1 Introduction

The analysis of probabilistic systems has been the subject of active research in the last decade, and many formalisms have been proposed to model them: Probabilistic Labelled Transition Systems [1], Probabilistic Automata [2], Labelled Markov Processes (LMPs) [3]. In all these models, states are the central focus, even if the analysis must rely on probability theory, where one usually deals with events, or *sets of states*. A recent investigation [4] showed that bisimulation, the usual notion of equivalence between probabilistic processes, could be defined in terms of events. Moreover, it is well known that bisimulation (for LMPs, let say) is characterized by a simple logic $\mathcal{L}_\vee$ (Section 2.2): two states are bisimilar if and only if they satisfy exactly the same formulas of that logic. Since formulas can be seen as sets of states, they are ideal candidates for events. More precisely, any LMP with σ-algebra Σ can be associated to a morphism

$$[\![\cdot]\!] : \mathcal{L}_\vee \to \Sigma$$

where the image of a formula is the (measurable) set of states that satisfy it. We are interested in what we can say of the converse: can a probabilistic process be defined by the set of its logical properties only? Indeed even if Σ is a structure of sets, we can abstract it as a σ-complete Boolean algebra and we can ask the question: when is it the case that a given function $\hat{\mu} : \mathcal{L}_\vee \to \mathcal{A}$ for an arbitrary σ-complete Boolean algebra $\mathcal{A}$ corresponds to some LMP whose σ-algebra is isomorphic to $\mathcal{A}$ and whose semantics accords with $\hat{\mu}$? This opens the way to working with probabilistic processes in an abstract way, that is, without any explicit mention of the state space, manipulating properties only. In other words, this opens the way to a Stone-type duality theory for these processes.

Kozen [5] has already discussed such a duality for probabilistic programs. Duality notions appear in a variety of areas, allowing one to get back and forth from a concrete model to an "abstract version" of it. In this paper we propose a notion of "abstract" pointless probabilistic processes that can be viewed as the probabilistic counterpart of the theory of Frames (or Locales), that is, the theory of "pointless" topological spaces [6]. In the latter, topological spaces are abstracted by complete Boolean algebras (more precisely by complete Heyting algebras), whereas, in our framework, the probabilistic processes state spaces will be abstracted by σ-complete Boolean algebras. The encoding of transition probabilities will be done through the logic properties.

Consequently, we will define an abstract probabilistic process as a morphism $\hat{\mu} : \mathcal{L}_\vee \to \mathcal{A}$. This idea has been proposed for LMPs by Danos, Panangaden and two of us [4]. Unfortunately, it is very difficult to obtain a duality for LMPs. One of the problems is the reconstruction function that, from an abstract LMP, should output a "concrete" one. More specifically, we do not know any category of abstract LMPs that would be functorially linked to the concrete one. LMPs are not the adequate level of generality because they carry, through their probabilistic transition functions, more information than just the properties that they satisfy. In this paper, we introduce infLMPs which happen to be essentially partial specifications of LMPs, and therefore represent the suitable level of generality for a Stone-type duality theory.

InfLMPs are obtained from standard probabilistic transition systems, like LMPs, by relaxing the standard σ-additivity axiom of probability measures with a super-additivity axiom: $p(s, A \cup B) \geq p(s, A) + p(s, B)$ for disjoint A, B. This allows us to give a demonic interpretation of information because infLMPs are underspecified. Indeed, a state s from which we only know that with action l either event A or B will happen (with A, B disjoint), can be encoded as an infLMP by $p_l(s, A) = p_l(s, B) = 0$ and $p_l(s, A \cup B) = 1$. It is widely acknowledged that it is not appropriate to model this situation by giving equal probabilities to A and B. Indeed, no justification allows one to choose a particular distribution. The super-additivity relaxation permits to easily maintain all possible distributions, by leaving unknown information unspecified. We are in presence of a demonic approach to information in probabilistic systems: for example, even if $p_l(s, B) = 0$, it does not necessarily mean that B is impossible to reach, but rather that in the presence of a demonic adversary, it will indeed be impossible. The notion of bisimulation smoothly transfers from LMPs to infLMPs. Logical characterization theorems do not transfer directly because, as we will show, infLMPs can encode both probabilistic and non-deterministic processes. This was a pleasing surprise that was awaiting us along the way.

2 Background

A *measurable space* is a pair (S, Σ) where S is any set and $\Sigma \subseteq 2^S$ is a *σ-algebra* over S, that is, a set of subsets of S containing S and closed under countable intersection and complement. Well-known examples are $[0,1]$ and $\mathbb{R}$ equipped with their respective *Borel* σ-algebras, written $\mathcal{B}$, generated by intervals. When we do not specify the σ-algebra over $\mathbb{R}$ or one of its intervals, we assume the Borel one. For $R \subseteq S \times S$, a set $A \subseteq S$ is R-closed if $R(A) := \{s \in S \mid \exists a \in A.(a,s) \in R\} \subseteq A$. A map f between two measurable spaces (S,Σ) and (S',Σ') is said to be *measurable* if for all $A' \in \Sigma'$, $f^{-1}(A') \in \Sigma$. A necessary and sufficient criterion for measurability of a map $p : (S,\Sigma) \to ([0,1],\mathcal{B})$ is that the condition be satisfied for $[r,1]$, for any rational r. Finally, countable suprema of real valued measurable functions are also measurable [7].

A *subprobability measure* on (S,Σ) is a map $p : \Sigma \to [0,1]$, such that for any countable collection (A_n) of pairwise disjoint sets, $p(\cup_n A_n) = \sum_n p(A_n)$.

2.1 Labelled Markov Processes (LMPs)

LMPs are probabilistic processes whose state space can be uncountable. In the finite case, they are also known as Probabilistic Labelled Transition Systems [1]. We fix a finite alphabet of actions L.

Definition 1 ([3]). *A* labelled Markov process (LMP) *is a triple* $\mathcal{S} = (S,\Sigma,h : L \times S \times \Sigma \to [0,1])$ *where* (S,Σ) *is a measurable space and for all* $a \in L$, $s \in S$, $A \in \Sigma$: $h_a(s,\cdot)$ *is a subprobability on* (S,Σ), *and* $h_a(\cdot,A) : (S,\Sigma) \to ([0,1],\mathcal{B})$ *is measurable.*

The category ***LMP*** *has LMPs as objects and zigzag morphisms as morphisms. A measurable surjective map* $f : S \twoheadrightarrow S'$ *is called a* zigzag *morphism if* $\forall a \in L$, $s \in S$, $A' \in \Sigma'$: $h_a(s, f^{-1}(A')) = h'_a(f(s), A')$.

$h_a(s,A)$ is the probability that being at s and receiving action a, the LMP will jump in A. Examples of finite LMPs are $\mathcal{T}_1$ and $\mathcal{T}_2$ of Fig. 1. Alternatively, LMPs can be expressed as coalgebras of the Giry monad [8,3]. We will assume the structure of any LMP $\mathcal{S}$ to be (S,Σ,h), letting primes and subscripts propagate.

LMPs depart from usual Markov chains in that transitions also depend on an auxiliary set of actions, but most importantly, because one thinks of them differently. They are interactive processes and therefore one is interested in what an interacting user can deduce about them, as in non-deterministic process algebras [9]. The following observational relation is a natural extension of the usual notion of bisimulation that one encounters in non probabilistic processes as well as in finite probabilistic processes. Following Danos et al. [4], we call it state bisimulation, a denomination that emphasizes the fact that the relation is based on states, as opposed to events.

Definition 2 ([10]). *Given an LMP* $\mathcal{S}$, *a* state bisimulation *relation* R *is a binary relation on* S *such that whenever* $(s,t) \in R$ *and* $C \in \Sigma$ *is* R*-closed, then for all* $a \in L$, $h_a(s,C) = h_a(t,C)$. *We say that* s *and* t *are* state bisimilar *if there is a bisimulation* R *such that* $(s,t) \in R$. *A state bisimulation between two LMPs* $\mathcal{S}$ *and* $\mathcal{S}'$ *is a state bisimulation between their disjoint union* $\mathcal{S} + \mathcal{S}'$.

This definition is intuitive, but *event* bisimulation has a better probability theory taste, where states are often irrelevant and where one focusses on events instead.

Note that for countable or analytic processes, these definitions are equivalent [11]. We say that $\mathcal{C} \subseteq \Sigma$ is stable if for all $C \in \mathcal{C}$, $a \in L$, $q \in [0,1] \cap \mathbb{Q}$, $\langle a \rangle_q C := \{s \in S \mid h_a(s, C) \geq q\} \in \mathcal{C}$. By measurability of h, this is always in Σ.

Definition 3 ([11]). *An* event bisimulation *on an LMP (S, Σ, h) is a stable sub-σ-algebra Λ of Σ, or equivalently, (S, Λ, h) is an LMP. An event bisimulation between two LMPs is one on their disjoint union. Equivalently, an* event bisimulation *on $\mathcal{S}$ is a morphism in the category* ***LMP*** *to some $\mathcal{S}'$. An event bisimulation between $\mathcal{S}_1$ and $\mathcal{S}_2$ is a cospan $\mathcal{S}_1 \twoheadrightarrow \mathcal{S}' \twoheadleftarrow \mathcal{S}_2$.*

Co-simulation morphisms, a relaxation of zigzag morphisms, send any state to a state that it *simulates*. We elide the general definition of simulation [10] because we only need the particular one obtained through co-simulation morphisms. Intuitively, a state simulates another one if it can mimic its behaviour with greater or equal probability. Co-simulation morphisms differ from simulation morphisms [10], by the direction of the inequation and the surjectivity requirement. They are useful when one wants to establish that a process obtained by some quotient operation is simulated by another one.

Definition 4 ([12]). *A* co-simulation morphism *is a surjective map $q : S \to S'$ such that $\forall a \in L$, $s \in S$, $A' \in \Sigma'$: $h_a(s, q^{-1}(A')) \geq h'_a(q(s), A')$.*

In Fig. 1, there is an obvious co-simulation morphism from $\mathcal{T}_2$ to $\mathcal{S}_1$.

2.2 Temporal Properties

The following well-known logic grammars will define properties of LMPs:

$$\mathcal{L}: \quad \varphi := \top \mid \varphi \wedge \varphi \mid \langle a \rangle_q \varphi$$

$$\mathcal{L}_\vee: \quad \varphi := \mathcal{L} \mid \varphi \vee \varphi \qquad \mathcal{L}_\neg: \quad \varphi := \mathcal{L} \mid \neg\varphi \qquad \mathcal{L}_\infty: \quad \varphi := \mathcal{L}_\neg \mid \bigvee_{i=1}^{\infty} \varphi_i$$

with $q \in \mathbb{Q} \cap [0,1]$. Note that the three first logics are countable.

Definition 5. *Given an LMP $\mathcal{S}$, the semantics of $\mathcal{L}$ is inductively defined as the map $[\![.]\!]_{\mathcal{S}} : \mathcal{L} \to \Sigma$ as follows: $[\![\top]\!]_{\mathcal{S}} := S$, $[\![\varphi_0 \wedge \varphi_1]\!]_{\mathcal{S}} := [\![\varphi_0]\!]_{\mathcal{S}} \cap [\![\varphi_1]\!]_{\mathcal{S}}$, $[\![\langle a \rangle_r \varphi]\!]_{\mathcal{S}} := \langle a \rangle_r [\![\varphi]\!]_{\mathcal{S}} = \{s \in S \mid h_a(s, [\![\varphi]\!]_{\mathcal{S}}) \geq r\}$.*

This map is easily shown to be measurable by structural induction [3] (and hence writing $h_a(s, [\![\varphi]\!]_{\mathcal{S}})$ is indeed legal). The semantics of $\mathcal{L}_\vee$, $\mathcal{L}_\neg$ and $\mathcal{L}_\infty$ are easily derived. Note the abuse of notation $\langle a \rangle_q X$ which is used both as a formula if X is one, and as an operation returning a set when X is a set.

Logics induce an equivalence on states and processes, as follows.

Definition 6. *Let $\mathcal{L}_*$ be some logic: two states are* $\mathcal{L}_*$-equivalent*, if they satisfy exactly the same formulas of $\mathcal{L}_*$ Two LMPs are $\mathcal{L}_*$-equivalent if every state of any of them is $\mathcal{L}_*$-equivalent to some state of the other.*

Despite the fact that $\mathcal{L}$ is a very simple logic, two states of any LMPs are event bisimilar if and only if they satisfy exactly the same formulas. Indeed, one can

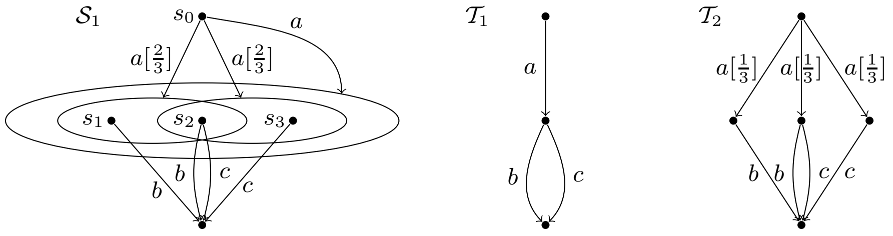


Fig. 1. A typical infLMP $\mathcal{S}_1$ and two LMP implementations of $\mathcal{S}_1$. An arrow labelled with action l and value r represents an l-transition of probability r; in picture representations, we omit r when it is 1 and we omit transitions of zero probability. For example, state s_0 is pictured to be able to jump with probability 1 to the three middle states (represented by the biggest ellipse), with probability $\frac{2}{3}$ to the first two and also to the last two. The probability from s_0 to any single state is 0. We also omit transitions to sets whose value is governed by super additivity, like $h_a(s_0, S_1) \geq h_a(s_0, \{s_1, s_2, s_3\}) = 1$.

decide whether two states are bisimilar and effectively construct a distinguishing formula in case they are not [3]. This theorem is refered to as the *logical characterization* theorem, and, for state bisimulation, it works only when the state space is an analytic space. This is one situation where event bisimulation appears to be more natural since its logical characterization has no restriction.

3 InfLMPs

In our quest for a Stone-type duality, we needed a model that would be in correspondence with sets of properties. It turns out that the appropriate one is very simple; we call it infLMP because, as we will see, it encodes infimum requirements. InfLMPs are LMPs whose transition functions are super-additive, a weaker condition than additivity. Thus, LMPs are infLMPs. Recall that a set function p is super-additive if $p(A \cup B) \geq p(A) + p(B)$ for disjoint A, B. Super-additivity implies $p(\varnothing) = 0$ and monotonicity: $A \subseteq B \Rightarrow p(A) \leq p(B)$.

Definition 7. *An* infLMP *is a triple* $\mathcal{S} = (S, \Sigma, h : L \times S \times \Sigma \to [0, 1])$ *where* (S, Σ) *is a measurable space, and for all* $a \in L$, $s \in S$, $A \in \Sigma$: $h_a(\cdot, A)$ *is measurable and* $h_a(s, \cdot)$ *is super-additive. The category* ***infLMP*** *is a super category of* ***LMP*** *: it has infLMPs as objects and morphisms are defined as in* ***LMP***.

Many known results on LMPs do not use additivity and hence they carry on for infLMPs trivially. Event and state bisimulation are defined exactly as for LMPs, and hence, LMPs viewed as infLMPs are related in the same way through bisimulation. Definition of logics semantics are also unchanged.

Example 1. A typical example of infLMP is process $\mathcal{S}_1$ of Fig. 1. One can check that $h_a(s_0, \cdot)$ is super-additive, but it is not a subprobability measure, as $0 = h_a(s_0, \{s_1\}) + h_a(s_0, \{s_2\})$ is strictly less than $h_a(s_0, \{s_1, s_2\}) = \frac{2}{3}$.

Such a weakening of the subprobability condition is not dramatic. Transitions can still be composed using Choquet integral, which only requires monotonicity of the set functions involved [13,14]. We prefer super-additivity over monotonicity because we want to interpret the values of transitions as probabilities.

The following example shows how infLMPs can be interpreted as underspecified processes, or as specifications of LMPs. We will prove later on (Theorem 6) that indeed, any set of positive formulas can be represented by an infLMP.

Example 2. In $\mathcal{S}_1$ of Fig. 1, some exact probabilities are not known: how the $\frac{2}{3}$ probability would distribute between s_1 and s_2 is not specified. The two LMPs $\mathcal{T}_1$ and $\mathcal{T}_2$ of Fig. 1 are possible implementations of $\mathcal{S}_1$. In the first one, the value of $\frac{2}{3}$ is sent in full to the state that can make both a b-transition and a c-transition. In the other one a minimal probability is sent to this state and the remaining is distributed to the left-most and right-most states. A reader familiar with simulation can check that $\mathcal{S}_1$ is *simulated by* both processes, as both can perform transitions with equal or higher probabilities to simulating states.

This example exhibits two important points. The first one is that when we say that an infLMP represents properties for an LMP, we have in mind positive properties, that is, properties of $\mathcal{L}$ or $\mathcal{L}_\vee$, involving no negation. Consequently, probability transitions are interpreted as lower bound requirements: a probability of zero in an infLMP could possibly be positive in a realization of this infLMP. The second point is that, as we claimed in the introduction, infLMPs can model not only underspecified LMPs, but also a kind of non deterministic processes. Indeed, state s_0 of process $\mathcal{S}_1$ is a non-deterministic state: the super-additive set function $h_a(s_0,\cdot)$ models all subprobability measures that are greater than or equal to it on every set. Any distribution giving probability $q \in [\frac{1}{3}, 1]$ for the transition to s_2 and at least $\max(\frac{2}{3} - q, 0)$ to s_1 and to s_3 in such a way that the probability to $\{s_1, s_2, s_3\}$ is 1, would implement this specification.

An implementation should be defined as an LMP that *simulates* the infLMP specification. However, since we do not want to enter the details of simulation, and since interpreting infLMPs is not the primary goal of this paper, we prefer to keep the notion of implementation or realization at an informal level.

The two following results show that zigzag morphisms link processes in a strong way. The second implies that bisimilar processes satisfy the same formulas.

Proposition 1. *Let $f : \mathcal{S} \twoheadrightarrow \mathcal{S}'$ be a morphism of infLMPs. If $\mathcal{S}$ is an LMP, so is $\mathcal{S}'$.*

Proposition 2 ([3]). *Let $f : \mathcal{S} \twoheadrightarrow \mathcal{S}'$ be a morphism of infLMPs, then for all $\phi \in \mathcal{L}_\infty$, $s \in S$: $s \in [\![\phi]\!]_\mathcal{S} \Leftrightarrow f(s) \in [\![\phi]\!]_{\mathcal{S}'}$ or equivalently: $[\![\phi]\!]_\mathcal{S} = f^{-1}([\![\phi]\!]_{\mathcal{S}'})$.*

An interesting feature of infLMPs is the relation between logical equivalence and bisimulation. It is quite different from what happens on LMPs for which even the simplest logic $\mathcal{L}$ characterizes bisimulation.

Theorem 1. *The $\mathcal{L}_*$-equivalences on infLMPs for $\mathcal{L}_* \in \{\mathcal{L}, \mathcal{L}_\vee, \mathcal{L}_\neg\}$ are strictly weaker than bisimulation, but $\mathcal{L}_\infty$ characterizes event bisimulation for infLMPs.*

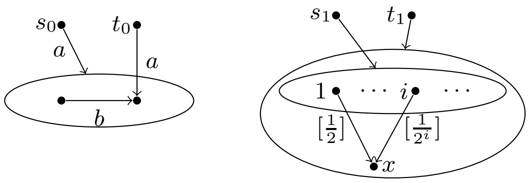


Fig. 2. For infLMPs, $\neg$ and $\vee^\infty$ are necessary for bisimulation

Proof (Sketch). The second claim is trivial because $\{[\![\phi]\!] \mid \phi \in \mathcal{L}_\infty\}$ is a stable σ-algebra. States s_0 and t_0 of Fig. 2 prove that negation is necessary. They satisfy the same formulas of $\mathcal{L}_\vee$, but they are distinguished by formula $\langle a\rangle_1(\neg\langle b\rangle_{\frac{1}{2}}\top)$. The right part of Fig. 2 shows that infinite disjunction is necessary. The state space is $\mathbb{N}_+\cup\{s_1, t_1, x\}$. States s_1 and t_1 both have probability 1 to $\mathbb{N}_+\cup\{x\}$, and s_1 also has probability 1 to $\mathbb{N}_+$. States s_1 and t_1 are $\mathcal{L}_\neg$-equivalent but they are distinguished by $\langle a\rangle_1(\vee_{i=1}^{\infty}\langle a\rangle_{\frac{1}{2^i}}\top)$. ∎

This result is not surprising since infLMPs encode non determinism: it is well known that logical characterization of bisimulation needs negation and infinite disjunction when non determinism and probabilities cohabite. However, since all mentioned logics characterize bisimulation for LMPs, we have the following.

Corollary 1. *Every equivalence class of infLMPs with respect to each of the logics $\mathcal{L}$, $\mathcal{L}_\vee$ and $\mathcal{L}_\neg$ contains at most one LMP, up to event bisimulation.*

Although quite natural, this result raises one question: if only one LMP is in the same equivalence class as an infLMP, how can the latter be realized by more than one LMP? The answer is: because a specification is a lower bound requirement, and hence a realization is *not* necessarily equivalent to its specification. There are specifications that cannot be realized exactly, such as process $\mathcal{S}_1$ of Fig. 1. In this process, s_0 has probability 0 to s_2, but any LMP implementation, which has to satisfy additivity, will have a probability of at least $\frac{1}{3}$ to some state that can make both a b and a c-transition. Thus, any realization of s_0 will satisfy the formula $\langle a\rangle_{\frac{1}{3}}(\langle b\rangle_1\top \wedge \langle c\rangle_1\top)$, as do $\mathcal{T}_1$ and $\mathcal{T}_2$.

3.1 Related Models

Abstractions. In order to combat the state explosion problem, Fecher et al. [16] and Chadha et al. [17] propose to abstract probabilistic processes by grouping states. In the former, this is done by giving intervals to probability transitions, whereas in the latter, a tree-like partial order is chosen on top of the state space. With intervals, one looses the link between individual transitions which is preserved in our setting. Abstract processes of Chadha et al. are closer to infLMPs but they carry less information since they do not rely on satisfied properties but on the quality of the chosen partial order.

PreLMPs InfLMPs have a structure close to the one of preLMPs; these emerged as formula based approximations of LMPs [12]: given an LMP and a set of formulas, the goal was to define an approximation of the LMP that satisfied the given formulas. This looks close to what we want since infLMPs represent the properties that an LMP satisfies. However the set functions of preLMPs must not only be super-additive, but also co-continuous: $\forall \downarrow A_n \in \Sigma : p(\cap A_n) = \inf_n p(A_n)$. Such a condition is too strong in our context: co-continuity cannot be obtained from a set of formulas only, and it is incompatible with lower bounds. In the work we mention above, the transition functions of the preLMP are defined using the subprobability measure of the LMP (which are already co-continuous, of course), and this is crucial in obtaining their co-continuity nature.

Non determinism. When investigating on the duality presented in Section 5, we first thought that preLMPs would be our abstract processes. It turns out that not only infLMPs are the right framework, but they represent a promising alternative to processes mixing non determinism and probabilities, because of their simplicity and expressiveness.

We briefly show how infLMPs compare to existing models, but we leave for future work a deeper investigation of related questions. Models of non-deterministic probabilistic processes *a la Segala* ([2,15]) usually describe transitions as a subset of $S \times L \times Distr(S)$, where $Distr(S)$ is a set of probability distributions over S. Thereafter, a scheduler, which aims at resolving non determinism, is defined as a map that, when given an execution from the initial state to some state s, returns a distribution on the transitions at s. More specifically, this describes a randomized scheduler. In our model, given an execution ending in s and an action, the scheduler could simply pick up one distribution among the ones that are greater than or equal to the super-additive set function at s (and in addition, give a distribution on the action label set). InfLMPs thus appear to be quite simpler since a simple super-additive set function encode an uncountable number of possible distributions (as does s_0 of $\mathcal{S}_1$). Of course, a deeper understanding has to be undertaken to properly compare those models with regards to their power and efficiency. Note yet that by taking the infimum over transitions of a process a la Segala, we obtain an infLMP that encodes at least all the behaviours of the original processes.

4 Abstract Processes

There is a strong correspondence between σ-algebras and σ-complete Boolean algebras. A Boolean algebra $\mathcal{A}$ is called a *σ-algebra of sets* if $\mathcal{A} \subseteq \mathcal{P}(X)$ for some $X \in \mathcal{A}$ (the greatest element), and $\mathcal{A}$ is closed under complementation and countable intersections. Every Boolean algebra carries an intrinsic partial order defined as $A \preccurlyeq B \Leftrightarrow A \wedge B = A$. Thus, a σ-algebra is just a σ-complete Boolean algebra of sets, where $\preccurlyeq$ is set inclusion $\subseteq$.

As announced in the introduction, we now propose a generalization of our notion of infLMPs to a more abstract, purely algebraic setting, without any references to the notion of states. Hence, instead of referring to a set of states and a σ-algebra, the underlying space of an abstract infLMP will be some σ-complete Boolean algebra $\mathcal{A}$ (we say a σ-BA from now on). Any infLMP can be associated to a morphism $[\![\cdot]\!] : \mathcal{L}_\vee \to \mathcal{A}$. However, what can we say of the converse? When is it the case that a given function $\hat{\mu} : \mathcal{L}_\vee \to \mathcal{A}$ for an arbitrary σ-BA $\mathcal{A}$ corresponds to some infLMP whose σ-algebra is isomorphic to $\mathcal{A}$ and whose semantics accords with $\hat{\mu}$? In other words, do we have a Stone-type duality between infLMPs and their abstract counterparts? To obtain this notion of duality, we need in the infLMP framework to construct an axiomatization of abstract objects $\hat{\mu} : \mathcal{L}_\vee \to \mathcal{A}$ that guarantees that :

(1) a representation theorem ensures that each $\mathcal{A}$ is isomorphic to a σ-algebra;
(2) $\hat{\mu}$ is coherent with logic operators and induces a super-additive set function.

4.1 A Suitable Representation Theorem

Recall that the well-known Stone's representation theorem asserts that any Boolean algebra is isomorphic to an algebra of sets (consisting of the clopen sets of a topological space). This is not true for σ-BA (when considered with homomorphisms that preserve countable meet and join). The generalization of Stone's theorem for σ-BA is known as the Loomis-Sikorski's theorem [18], and states that for any σ-BA $\mathcal{A}$, there is a σ-ideal $\mathcal{N}$ such that $\mathcal{A}/\mathcal{N}$ is isomorphic to an algebra of sets. Such a result is very difficult to use here, for two reasons. First, the construction of the algebra of sets that arises from Loomis-Sikorski's theorem is not unique and far from being as simple as the construction associated to Stone's theorem. Second, the ideal $\mathcal{N}$ is difficult to interpret in the concrete counterpart. One possibility would be to introduce the notion of negligible events and to relax accordingly the notion of bisimulation (see [4]), but it is not enough and we prefer to keep the structure simple. Indeed, as we will show below in Corollary 2, we are able to circumvent all those difficulties by simply restricting the possible $\mathcal{A}$ to ω-distributive σ-BA.

Definition 8. *A σ-BA $\mathcal{A}$ is* ω-distributive *if, for any countable set I and for every family $(a_{ij})_{i\in I, j=1,2}$ in $\mathcal{A}$, $\wedge_{i\in I}(a_{i1} \vee a_{i2}) = \vee\{\wedge_{i\in I} a_{if(i)} : f \in 2^I\}$.*

Note that if we restrict I to finite sets only, we retrieve the standard distributivity law, and that a σ-algebra is always ω-distributive.

The two following results will provide the essential link between ω-distributivity and the representation we are seeking.

Theorem 2 ([18, 24.5]). *For $\mathcal{A}$ a σ-BA generated by at most ω elements, the following conditions are equivalent:*

- *$\mathcal{A}$ is isomorphic to a σ-algebra*
- *$\mathcal{A}$ is ω-distributive*
- *$\mathcal{A}$ is atomic.*

Theorem 3 ([18, 24.6]). *For $\mathcal{A}$ a σ-BA, $\mathcal{A}$, is ω-distributive if and only if every sub-σ-BA generated by at most ω elements is isomorphic to a σ-algebra.*

Corollary 2. *A σ-BA $\mathcal{A}$ is ω-distributive if and only if for any $\hat{\mu}:\mathcal{L}_\vee \to \mathcal{A}$, the sub-$\sigma$-BA of $\mathcal{A}$ generated by $\hat{\mu}(\mathcal{L}_\vee)$, noted $\sigma(\hat{\mu}(\mathcal{L}_\vee))$, is isomorphic to a σ-algebra $\hat{\Sigma}$. Moreover, the underlying set of $\hat{\Sigma}$ is the set of all atoms of $\sigma(\hat{\mu}(\mathcal{L}_\vee))$.*

This result shows that, provided that our logic is countable (as is $\mathcal{L}_\vee$), the condition of ω-distributivity is not only sufficient to get a representation theorem, but also necessary. The following Lemma will be useful in Section 5.

Lemma 1. *Let (S,Σ) and (S',Σ') be two measurable spaces. If Σ' is ω-generated and separates points of S', then for any σ-BA morphism $\rho:\Sigma'\to\Sigma$, there exists a unique measurable function $f:(S,\Sigma)\to(S',\Sigma')$ such that $f^{-1}=\rho$.*

4.2 A Suitable Axiomatic for $\hat{\mu}$

As for the guaranty of our Condition (2), note that to ease intuition, it is useful to think of the image of $\hat{\mu}(\phi)$ as a set of states that satisfy ϕ since we are indeed looking for the existence of an infLMP that accords with $\hat{\mu}$. In the following, we extract the desired necessary conditions.

From now on, we fix $\mathcal{A}=(A,\vee,\wedge,-,\mathbf{0},\mathbf{1})$. By analogy with set theory, we say that $\hat{\mu}(\phi)$ and $\hat{\mu}(\psi)$ are disjoint if $\hat{\mu}(\phi)\wedge\hat{\mu}(\psi)=\mathbf{0}$ and we denote by $\sigma(\hat{\mu}(\mathcal{L}_\vee))$ the smallest σ-BA that contains $\{\hat{\mu}(\phi)\mid\phi\in\mathcal{L}_\vee\}$.

We begin by defining a condition on $\hat{\mu}$ that will represent super-additivity.

Definition 9. *We say that $\hat{\mu}:\mathcal{L}_\vee\to\mathcal{A}$ is* super-additive *if for all countable families of pairwise disjoint $\hat{\mu}(\phi_i)$ such that $\wedge\hat{\mu}(\langle a\rangle_{q_i}\phi_i)\neq\mathbf{0}$, then $\sum q_i\leq 1$ and for all $\varphi\in\mathcal{L}_\vee$ where $\vee\hat{\mu}(\phi_i)\preccurlyeq\hat{\mu}(\varphi)$ we have $\wedge\hat{\mu}(\langle a\rangle_{q_i}\phi_i)\preccurlyeq\hat{\mu}(\langle a\rangle_{\sum q_i}\varphi)$.*

The condition of super-additivity makes sure that we will not have a superset with a smaller value than the sum of its disjoint subsets; it is illustrated in Fig. 3. The condition $\sum_i q_i\leq 1$ is for the formula $\langle a\rangle_{\sum q_i}\varphi$ to exist in $\mathcal{L}_\vee$.

Theorem 4. *Let $\hat{\mu}:\mathcal{L}_\vee\to\mathcal{A}$ where $\hat{\mu}:=[\![\cdot]\!]_\mathcal{S}$ for an infLMP $\mathcal{S}$. Then*
1. *$\hat{\mu}$ respects $\top$, $\wedge$ and $\vee$*
2. *if $r\leq q$, then $\hat{\mu}(\langle a\rangle_r\phi)\succcurlyeq\hat{\mu}(\langle a\rangle_q\phi)$*
3. *$\hat{\mu}$ is super-additive as per Definition 9.*

We now have conditions on $\mathcal{A}$ and $\hat{\mu}$ that are satisfied by the semantic map $[\![\cdot]\!]$ of any infLMP. We claim that these conditions will be sufficient to generate an infLMP, and hence we use them to define the category of abstract infLMPs.

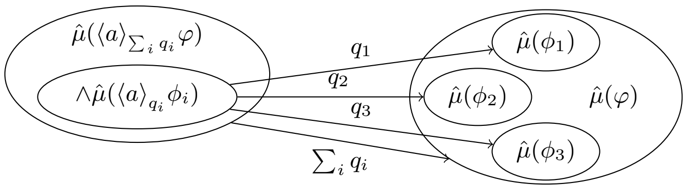


Fig. 3. Super-additivity of $\hat{\mu}$

$$\begin{tikzcd}
\mathcal{L}_\vee \arrow[r, "\hat\mu"] \arrow[dr, "\hat\mu'"'] & \sigma(\hat\mu(\mathcal{L}_\vee)) & \subseteq \mathcal{A} \\
& \sigma(\hat\mu'(\mathcal{L}_\vee)) \arrow[u, hook, "\rho"'] & \subseteq \mathcal{A}'
\end{tikzcd}
\qquad
\begin{tikzcd}
& \mathcal{A} & \\
\mathcal{L}_\vee \arrow[ur, "\hat\mu_\mathcal{A}"] \arrow[rr, "\hat\mu_\mathcal{X}"] \arrow[dr, "\hat\mu_\mathcal{B}"'] & & \mathcal{X} \arrow[ul, hook] \arrow[dl] \\
& \mathcal{B} &
\end{tikzcd}$$

Fig. 4. Commutative diagrams for definition of ρ and bisimulation

Definition 10. *An* abstract infLMP *is a function $\hat\mu : \mathcal{L}_\vee \to \mathcal{A}$ where $\mathcal{A}$ is an ω-distributive σ-BA and where $\hat\mu$ respects the conditions of Theorem 4. The category **infLMPa** has abstract infLMP as objects and a morphism from $\mathcal{L}_\vee \to \mathcal{A}'$ to $\mathcal{L}_\vee \to \mathcal{A}$ is a monomorphism $\rho : \sigma(\hat\mu'(\mathcal{L}_\vee)) \hookrightarrow \sigma(\hat\mu(\mathcal{L}_\vee))$ of σ-BA that makes the left diagram of Fig. 4 commute.*

The commutativity condition ensures that ρ will respect the logic. Thinking of members of $\mathcal{A}$ and $\mathcal{A}'$ as sets of states, this means that for any formula, states that satisfy a formula are sent to states that satisfy the same formula.

These ideas could be extended to other equivalences, by replacing $\mathcal{L}_\vee$ by other suitable logics like $\mathcal{L}$ and $\mathcal{L}_\neg$. We rejected $\mathcal{L}_\neg$ because we are basically interested in infLMPs as underspecifications of LMPs. We choose $\mathcal{L}_\vee$ because definitions are smoother and it keeps more information than $\mathcal{L}$.

5 From infLMP to infLMPa and Back

We will define functors between categories **infLMP** and **infLMPa**.

Definition 11. *The contravariant functor $\mathcal{F}$:* **infLMP** $\to$ **infLMPa** *is:*

– *$\mathcal{F}(S, \Sigma, h) = [\![\cdot]\!]_\mathcal{S} : \mathcal{L}_\vee \to \Sigma$.*
– *for $f\colon (S, \Sigma, h) \twoheadrightarrow (S', \Sigma', h')$, $\mathcal{F}(f)$ is the restriction of f^{-1} to $\sigma([\![\mathcal{L}_\vee]\!]_{\mathcal{S}'})$.*

Recall that we defined bisimulation between infLMP as cospans of morphisms. The image of this cospan in **infLMPa** becomes a span, as $\mathcal{F}$ is contravariant. This motivates the definition of bisimulation between abstract infLMPs as *spans* of morphisms instead of cospans.

Definition 12. *Two abstract infLMPs are bisimilar if they are related by a span of morphisms. In other words, a bisimulation between $\hat\mu_\mathcal{A}$ and $\hat\mu_\mathcal{B}$ is represented by the commutativity of the right diagram of Fig. 4.*

Bisimulation is easily shown to be an equivalence.

The following corollary shows that abstract infLMP objects are a generalization of ordinary infLMP objects. It is a direct consequence of the fact that $\mathcal{F}$ is a functor.

Corollary 3. *If infLMPs $\mathcal{S}$ and $\mathcal{S}'$ are bisimilar then $\mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{S}')$ also are.*

We will show later on that the converse is also true for LMPs.
We now define the functor that builds a concrete model from an abstract one.

Definition 13. *The contravariant functor* $\mathcal{G}$:**infLMPa**$\to$**infLMP** *is:*

- $\mathcal{G}(\hat{\mu} : \mathcal{L}_\vee \to \mathcal{A}) = (\hat{S}, \hat{\Sigma}, \hat{h})$ *where* $\hat{S}$ *is the set of atoms of* $\sigma(\hat{\mu}(\mathcal{L}_\vee))$*,* $\hat{\Sigma}$ *is defined by an isomorphism* $\mathsf{i} : \sigma(\hat{\mu}(\mathcal{L}_\vee)) \to \hat{\Sigma}$ *as in Corollary 2. Moreover, for* $s \in \hat{\mathcal{S}}$ *and* $A \in \hat{\Sigma}$*, writing* $h_{a,\phi}(s) := \sup\{q : s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_q \phi))\}$*, we define*
$$\hat{h}_a(s, A) := \sup_{\mathsf{i}(\hat{\mu}(\phi))\subseteq A} h_{a,\phi}(s) \ ;$$
- *for* $\rho : \mathcal{A}' \hookrightarrow \mathcal{A}$*,* $\mathcal{G}(\rho) : \hat{S} \twoheadrightarrow \hat{S}'$ *is the unique measurable map such that* $(\mathcal{G}(\rho))^{-1} = \mathsf{i}\,\rho\,(\mathsf{i}')^{-1}$*.*

Proof (that $\mathcal{G}$ *is a functor).* Since $\mathcal{A}$ is ω-distributive, the sub-σ-BA generated by $\hat{\mu}(\mathcal{L}_\vee)$ is isomorphic to a σ-algebra $\hat{\Sigma}$. We denote by i the isomorphism from $\sigma(\hat{\mu}(\mathcal{L}_\vee))$ to $\hat{\Sigma}$. We define $\hat{S}$ as the greatest element in $\hat{\Sigma}$.
— *Definition of* $\hat{h}$: The $h_{a,\phi}$'s are well-defined, and they are measurable. Indeed,
$$\begin{aligned} h^{-1}_{a,\phi}([q,1]) &= \{s \in S : h_{a,\phi}(s) \geq q\} \\ &= \{s : \sup\{r : s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_r \phi))\} \geq q\} \\ &= \cap_{r<q, r\in\mathbb{Q}}\ \mathsf{i}(\hat{\mu}(\langle a\rangle_r \phi)) \in \hat{\Sigma}. \end{aligned}$$
The only non trivial step is the third equality. Let s be such that $\sup\{r : s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_r \phi))\} \geq q$. Then $\forall r < q$, there is some $r' \geq r$ such that $s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_{r'} \phi))$. By Condition 2 of Theorem 4, we have $\hat{\mu}(\langle a\rangle_r \phi) \succcurlyeq \hat{\mu}(\langle a\rangle_{r'} \phi)$, and hence $s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_r \phi))$ for all $r < q$. Thus $s \in \cap_{r<q,r\in\mathbb{Q}} \{s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_r \phi))\}$. Conversely, if $s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_r \phi))$ for all $r < q$, then $\sup\{r : s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_r \phi))\} \geq q$, as wanted.

We have that the functions $\hat{h}_a$ are well-defined, and since they are defined as the supremum of countably many measurable functions, they are measurable. We also have that if $A = [\![\phi]\!]$, then $\hat{h}_a(s, A) = h_{a,\phi}(s)$, because of the super-additivity condition on $\hat{\mu}$ (Definition 9). This condition also implies that $\hat{h}$ is super additive and has value between 0 and 1. Indeed, let $A, B \in \hat{\Sigma}$ be disjoint sets. We have that $\hat{h}_a(s, A\cup B) \geq \hat{h}_a(s, A) + \hat{h}_a(s, B)$ because $\hat{h}_a(s, -)$ is defined as a supremum and because $h_{a,-}(s)$ is super-additive.

— *Definition of* $\mathcal{G}(\rho)$: since $\hat{\Sigma}'$ clearly separates points of $\hat{S}'$, Lemma 1 implies that $\mathcal{G}(\rho)$ is measurable, $\mathcal{G}(\mathsf{id}_\mathcal{A}) = \mathsf{id}_{(S,\Sigma)}$ and $\mathcal{G}(\rho \circ \rho') = \mathcal{G}(\rho') \circ \mathcal{G}(\rho)$. Moreover, since ρ is a monomorphism, $\mathcal{G}(\rho)$ is surjective because $(\mathcal{G}(\rho))^{-1}(X) \neq \varnothing$ for any $X \neq \varnothing$. To finish the proof, we want to show that, given any $s \in \hat{S}$, $a \in L$ and $B' \in \hat{\Sigma}'$, we have $\hat{h}_a(s, (\mathcal{G}(\rho))^{-1}(B')) = \hat{h}'_a(\mathcal{G}(\rho)(s), B')$ or, in other words, that
$$\sup_{\mathsf{i}(\hat{\mu}(\phi))\subseteq(\mathcal{G}(\rho))^{-1}(B')} \hat{h}_{a,\phi}(s) = \sup_{\mathsf{i}'(\hat{\mu}'(\phi))\subseteq B'} \hat{h}_{a,\phi}(\mathcal{G}(\rho)(s)).$$

This is a straightforward consequence of the fact that for each $\phi \in \mathcal{L}_\vee$, we have
1. $h_{a,\phi}(s) = h'_{a,\phi}(\mathcal{G}(\rho)(s))$
2. $\mathsf{i}(\hat{\mu}(\phi)) \subseteq (\mathcal{G}(\rho))^{-1}(B') \Leftrightarrow \mathsf{i}'(\hat{\mu}'(\phi)) \subseteq B'$

Proof of 1. Since $h_{a,\phi}(s) := \sup\{q | s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_q \phi))\}$, we only have to show that for any $q \in [0,1] \cap \mathbb{Q}$, we have $\mathcal{G}(\rho)(s) \in \mathsf{i}'(\hat{\mu}'(\langle a\rangle_q \phi)) \Leftrightarrow s \in \mathsf{i}(\hat{\mu}(\langle a\rangle_q \phi))$.
$$\mathcal{G}(\rho)(s) \in \mathsf{i}'(\hat{\mu}'(\langle a\rangle_q \phi)) \Leftrightarrow s \in (\mathcal{G}(\rho))^{-1}(\mathsf{i}'(\hat{\mu}'(\langle a\rangle_q \phi)))$$

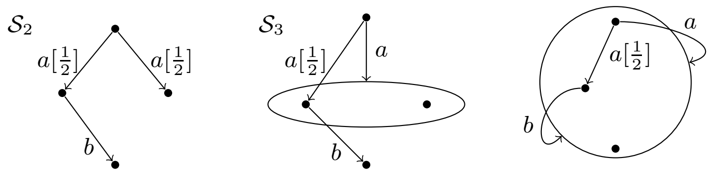


Fig. 5. Two infLMPs $\mathcal{S}_2$ and $\mathcal{S}_3$, and their image under $\mathcal{G} \circ \mathcal{F}$

$$
\begin{aligned}
&\Leftrightarrow s \in \mathsf{i} \circ \rho \circ (\mathsf{i}')^{-1} \circ \mathsf{i}' \circ \hat{\mu}'(\langle a \rangle_q \phi) \\
&\Leftrightarrow s \in \mathsf{i} \circ \rho \circ \hat{\mu}'(\langle a \rangle_q \phi) \\
&\Leftrightarrow s \in \mathsf{i} \circ \hat{\mu}(\langle a \rangle_q \phi) \qquad (1)
\end{aligned}
$$

where (1) follows from the fact that ρ being a morphism of **infLMPa**, we have $\rho \circ \hat{\mu}'(\psi) = \hat{\mu}(\psi)$ for any $\psi \in \mathcal{L}$.

Proof of 2.

$$
\begin{aligned}
\mathsf{i}'\hat{\mu}'(\phi) \subseteq B' &\Leftrightarrow \hat{\mu}'(\phi) \preccurlyeq (\mathsf{i}')^{-1}(B') \\
&\Leftrightarrow \rho \circ \hat{\mu}'(\phi) \preccurlyeq \rho((\mathsf{i}')^{-1}(B')) \qquad (2) \\
&\Leftrightarrow \hat{\mu}(\phi) \preccurlyeq \rho((\mathsf{i}')^{-1}(B')) \\
&\Leftrightarrow \mathsf{i} \circ \hat{\mu}(\phi) \subseteq \mathsf{i} \circ \rho \circ (\mathsf{i}')^{-1}(B') \qquad (3) \\
&\Leftrightarrow \mathsf{i} \circ \hat{\mu}(\phi) \subseteq (\mathcal{G}(\rho))^{-1}(B')
\end{aligned}
$$

in (2) (resp (3)), "$\Rightarrow$" follows from the fact that ρ (resp. i) is a morphism of σ-BA and "$\Leftarrow$" from the fact that it is a monomorphism (resp. isomorphism). ∎

Since $\mathcal{G}$ is a functor, we have a result similar to Corollary 3.

Corollary 4. *If $\hat{\mu}$ and $\hat{\mu}'$ are bisimilar, then $\mathcal{G}(\hat{\mu})$ and $\mathcal{G}(\hat{\mu}')$ also are.*

Example 3. Fig. 5 shows an LMP $\mathcal{S}_2$ and an infLMP $\mathcal{S}_3$ that have the same image under $\mathcal{G} \circ \mathcal{F}$. We can note a loss of information for $\mathcal{S}_2$. However, both processes simulate their image through a co-simulation morphism, (see Proposition 4).

It is easy to see that $\mathcal{F}$ and $\mathcal{G}$ are not inverse of one another. They are not quite adjunct either, but they are when restricted to LMPs, as we will show below. The following result shows that an abstract infLMP is isomorphic to its double dual. It is followed by a direct corollary.

Proposition 3. *$\mathcal{F} \circ \mathcal{G}(\hat{\mu}_{\mathcal{A}}) \cong \hat{\mu}_{\mathcal{A}}$ with isomorphism i given by Definition 13.*

Corollary 5. *$\mathcal{G} \circ \mathcal{F} \circ \mathcal{G}(\hat{\mu}_{\mathcal{A}}) \cong \mathcal{G}(\hat{\mu}_{\mathcal{A}})$ and $\mathcal{F} \circ \mathcal{G} \circ \mathcal{F}(\mathcal{S}) \cong \mathcal{F}(\mathcal{S})$.*

The following result shows that an infLMP always simulates its double dual.

Proposition 4. *The map $\eta_{\mathcal{S}} : \mathcal{S} \twoheadrightarrow \mathcal{G} \circ \mathcal{F}(\mathcal{S})$ that sends each state of $\mathcal{S}$ to its $\mathcal{L}_\vee$-equivalence class is a co-simulation morphism.*

Proof (sketch). Let $\eta_{\mathcal{S}}$ be the unique measurable map such that $\eta_{\mathcal{S}}^{-1} = \mathsf{i}^{-1}$ (where i is the isomorphism given in definition of $\mathcal{G}$) (it exists by Lemma 1). This map clearly sends any state $s \in \hat{\mathcal{S}} := \mathcal{G} \circ \mathcal{F}(\mathcal{S})$ to the atoms of $\sigma([\![\mathcal{L}_\vee]\!]_{\mathcal{S}})$ that correspond to the $\mathcal{L}_\vee$-equivalence class of s. Moreover, $\eta_{\mathcal{S}}$ is surjective because, for any atom $b \in \sigma([\![\mathcal{L}_\vee]\!]_{\mathcal{S}})$, $b \neq \mathbf{0}$, which implies that there exists an $s \in b$ and thus that there exists an $s \in \mathcal{S}$ such that $\eta_{\mathcal{S}}(s) = b$. To finish the proof, we have to show that $h_a(s, \eta_{\mathcal{S}}^{-1}(\hat{A})) \geq \hat{h}_a(\eta_{\mathcal{S}}(s), \hat{A})$, for $\hat{A} \in \hat{\Sigma}$. This is obtained from monotonicity of h_a, the fact that $s \in \eta_{\mathcal{S}}(s)$ and that $\forall s' \in \eta_{\mathcal{S}}(s), \forall B \in \sigma([\![\mathcal{L}_\vee]\!]_{\mathcal{S}})$, $s \in B \Leftrightarrow s' \in B$. ∎

Corollary 6. *$\mathcal{S}$ and $\mathcal{G} \circ \mathcal{F}(\mathcal{S})$ are $\mathcal{L}_\vee$-equivalent.*

Proof. By Corollary 5, we have $\mathcal{F} \circ \mathcal{G} \circ \mathcal{F} = \mathcal{F}$ which, because of the definition of $\mathcal{F}$, implies the results. ∎

Corollary 5 seems to indicate that $\mathcal{F}$ is the left adjoint of $\mathcal{S}$. Unfortunately, this will be true only if we replace $\mathcal{L}_\vee$ by $\mathcal{L}_\infty$ in the definition of **infLMPa**. Nevertheless, if we restrict ourselves to LMPs, (which is the case we are ultimately interested in) we do have adjunction.

In the following, **LMPa** is the full subcategory of **infLMPa** that is induced by $\{\mathcal{F}(\mathcal{S}) \mid \mathcal{S}$ is an LMP$\}$.

Theorem 5. *Let $\mathcal{F}_{LMP}$ (resp. $\mathcal{G}_{LMPa}$) be the restriction of $\mathcal{F}$ to the category **LMP** (resp. of $\mathcal{G}$ to the category **LMPa**) and let $\eta : I_{LMP} \xrightarrow{\bullet} (\mathcal{G}_{LMPa} \circ \mathcal{F}_{LMP})$ such that $\eta_{\mathcal{S}}$ is the co-simulation morphism defined in Proposition 4. Then $(\mathcal{F}_{LMP}, \mathcal{G}_{LMPa})$ is an adjunction pair whose natural transformation is η.*

We can now prove one of the main motivation for this work, that any set of positive formulas can be represented as an infLMP.

Theorem 6. *For any countable set of formulas of $\mathcal{L}_\vee$, there is a state of some infLMP that satisfies exactly these formulas (and all those that they imply).*

We therefore have the following result which, as stated before, shows that when restricted to LMPs, we have the reciprocal of Corollary 3.

Corollary 7. *Two LMPs $\mathcal{S}$ and $\mathcal{S}'$ are bisimilar iff $\mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{S}')$ also are.*

Proof. ($\Leftarrow$) By Corollaries 4 and 6, we have that $\mathcal{S}$ and $\mathcal{S}'$ are $\mathcal{L}_\vee$-equivalent, and since they are LMPs, then they are bisimilar.

6 Conclusion

We have introduced processes with a simple structure, infLMPs, that encode both non determinism and probabilistic behavior. We showed that they are a suitable abstraction for probabilistic processes properties like LMPs in the sense that there is an adjunction between them and their abstract representatives. This is a first step to a Stone-type duality theory that allows to manipulating properties only and stop considering states as a central concept. The two proposed functors allow to go back and forth between the concrete and abstract worlds.

It is easier to build an approximation theory when working only with properties (without states). For example one cannot have in general a concrete representation (with states) of a system that satisfies *only* the property $\phi \vee \phi'$; the concrete system will have a state that has property ϕ or a state that has property ϕ', or will even satisfy $\phi \wedge \phi'$. It will never satisfy $\phi \vee \phi'$ and not more. The fact that there always exists an infLMP that satisfies any countable set of properties of $\mathcal{L}_\vee$ (Theorem 6) indicates that infLMPs are a good level of generality if one wants to approximates stochastic processes. Approximating processes using properties has already been proposed for LMPs by Danos and Desharnais [12]. They showed that such approximations are not always LMPs. We propose an even weaker structure and hence endorse the use of an underspecified structure.

As future work, we plan to construct algorithms that will deal with infLMPs and their abstractions. Moreover, we want to generalize these ideas to the reinforcement learning (RL) framework. In RL, people are dealing with a kind of LMPs with rewards (called MDP). We think that a theory of approximations based on properties will be of interest there, especially because the main problem encountered in RL is the size of the model. Finally, we will study more deeply the use of infLMPs as encoders of a mix of probabilistic and non deterministic choice, in particular by analysing the associated demonic schedulers.

References

1. Larsen, K.G., Skou, A.: Bisimulation through probablistic testing. Information and Computation 94, 1–28 (1991)
2. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 481–496. Springer, Heidelberg (1994)
3. Blute, R., Desharnais, J., Edalat, A., Panangaden, P.: Bisimulation for labelled Markov processes. In: Proc. of LICS 1997, Warsaw, Poland (1997)
4. Danos, V., Desharnais, J., Laviolette, F., Panangaden, P.: Almost sure bisimulation in labelled Markov processes, 38 pages (2005), `http://www2.ift.ulaval.ca/~jodesharnais/Publications/almostSureDDLP05.pdf`
5. Kozen, D.: A probabilistic PDL. Journal of Computer and Systems Sciences 30(2), 162–178 (1985)
6. Johnstone, P.: Stone Spaces. Cambridge Studies in Advanced Mathematics, vol. 3. Cambridge University Press, Cambridge (1982)
7. Billingsley, P.: Probability and Measure. Wiley-Interscience, Hoboken (1995)
8. Rutten, J.J.M.M., de Vink, E.: Bisimulation for probabilistic transition systems: a coalgebraic approach. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 460–470. Springer, Heidelberg (1997)
9. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes I and II. Information and Computation 100, 1–41, 42–78 (1992)
10. Desharnais, J.: Labelled Markov Processes. PhD thesis. McGill University (2000)
11. Danos, V., Desharnais, J., Laviolette, F., Panangaden, P.: Bisimulation and cocongruence for probabilistic systems. Information and Computation, 22 pages (2005)

12. Danos, V., Desharnais, J.: Labeled Markov Processes: Stronger and faster approximations. In: Proc. of LICS 2003, Ottawa. IEEE, Los Alamitos (2003)
13. Choquet, G.: Theory of capacities. Ann. Inst. Fourier (Grenoble) 5, 131–295 (1953)
14. Goubault-Larrecq, J.: Continuous capacities on continuous state spaces. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 764–776. Springer, Heidelberg (2007)
15. Cattani, S., Segala, R., Kwiatkowska, M., Norman, G.: Stochastic transition systems for continuous state spaces and non-determinism. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 125–139. Springer, Heidelberg (2005)
16. Fecher, H., Leucker, M., Wolf, V.: Don't know in probabilistic systems. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 71–88. Springer, Heidelberg (2006)
17. Chadha, R., Viswanathan, M., Viswanathan, R.: Least upper bounds for probability measures and their applications to abstractions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 264–278. Springer, Heidelberg (2008)
18. Sikorski, R.: Boolean Algebras, 3rd edn. Springer, New York (1969)

HYPE: A Process Algebra for Compositional Flows and Emergent Behaviour

Vashti Galpin[1], Luca Bortolussi[2], and Jane Hillston[1]

[1] Laboratory for Foundations of Computer Science, University of Edinburgh
Vashti.Galpin@ed.ac.uk, Jane.Hillston@ed.ac.uk
[2] Department of Maths and Computer Science, University of Trieste
luca@dmi.units.it

Abstract. Several process algebras for modelling hybrid systems have appeared in the literature in recent years. These all assume that continuous variables in the system are modelled monolithically, often with the differential equations embedded explicitly in the syntax of the process algebra expression. In HYPE an alternative approach is taken which offers finer-grained modelling with each flow or influence affecting a variable modelled separately. The overall behaviour then emerges as the composition of these flows. This approach is supported by an operational semantics which distinguishes states as collections of flows and which is supported by an equivalence which satisfies the property that bisimilar HYPE models give rise to the same sets of continuous behaviours.

1 Introduction

HYPE is a novel process algebra for modelling hybrid systems. A hybrid system exhibits both discrete and continuous behaviour. It can be viewed as a system consisting of values which change continuously over time with respect to specific dynamics. Discrete events can cause discontinuous jumps in these values after which different dynamics may come into effect. These events can be triggered by conditions on the continuously changing values. The novelty of HYPE lies in how it captures the continuous dynamics of a system, and the separation of a discrete controller considered in parallel to the system under study. Unlike existing process algebras for hybrid systems, HYPE captures behaviour at a fine-grained level, composing distinct flows or influences. The dynamic behaviour then emerges, via the semantics of the language, from these compositional elements. We are inspired by the fluid flow semantics of PEPA models [15] which approximates the behaviour of large numbers of discrete components with a set of ordinary differential equations (ODEs).

Hybrid behaviour arises in a variety of systems, both engineered and natural. Consider a thermostatically controlled heater. The continuous variable is air temperature, and the discrete events are the switching on and off of the heater by the thermostat in response to the air temperature. Another example would be a genetic regulatory network, such as the Repressilator [9], in which genes can be switched on or off by interactions with their environment (more precisely, with transcription factor proteins). The behaviour of such systems can be regarded as a collection of sets of ODEs, the discrete events shifting the dynamic behaviour from the control of one set of ODEs to another. This is the approach taken with hybrid automata [13].

Process algebras have the advantage of being compositional hence models are built out of subcomponents and we aim to fully exploit this feature in HYPE. Existing process algebras for hybrid systems include ACP_{hs}^{srt} [3], hybrid χ [22], ϕ-calculus [20] and HyPA [6]. In [16], Khadim shows substantial differences in the approaches taken by these process algebras relating to syntax, semantics, discontinuous behaviour, flow-determinism, theoretical results and availability of tools. However, they are all similar in their approach in that the dynamic behaviour of each subcomponent must be fully described with the ODEs for the subcomponent given explicitly in the syntax of the process algebra, before the model can be constructed.

We aim for a finer-grained approach where each subcomponent is built up from a number of flows and hence the ODEs are only obtained once the model is constructed. By flow, we mean something that has an influence on a quantity of interest. For example, in a tank with two inlets and an outlet, both the inlets and the outlet influence the tank level, hence here we would identify three separate flows. The continuous part of the system is represented by the appropriate variables and the change over time of a given variable is determined by a number of active influences which represent flows and are additive in nature. Our approach also differs in that we explicitly require a controller that consists only of events.

We believe that the use of flows as the basic elements of model construction has advantages such as ease and simplification of modelling. This approach assists the modeller in allowing them to identify smaller or local descriptions of the model and then to combine these descriptions to obtain the larger system. The explicit controller also helps to separate modelling concerns.

The structure of the rest of the paper is as follows. In the next section we introduce our syntax for hybrid systems, explaining its components. In the following sections, we present the operational semantics and how we go from the notion of state to the ODEs which describe the system. We then discuss how this gives rise to a hybrid automaton. We consider a notion of bisimulation and our main result is that bisimilar systems have identical ODEs and finally we discuss related work.

2 HYPE Definition

This section will present HYPE by way of a running example of the temperature control system of an orbiting space vehicle. As the orbiter travels around the earth, it needs to regulate its temperature to remain within operational limits. It has insulation but needs to use a heater at low temperatures and at high temperatures it can erect a shade to reduce temperatures. The modelling spirit of HYPE focusses on flows. In our example, we identify four flows affecting the temperature. One is due to thermodynamic cooling, one is due to the heater, one is due to the heating effect of the sun and one is due to the cooling effect of the shade. The strength and form of a flow are modified by events. We first define the heater which can be on or off.

$$\mathit{Heat} \overset{\mathit{def}}{=} \underline{\mathrm{on}}{:}(h, r_h, \mathit{const}).\mathit{Heat} + \underline{\mathrm{off}}{:}(h, 0, \mathit{const}).\mathit{Heat} + \underline{\mathrm{init}}{:}(h, 0, \mathit{const}).\mathit{Heat}$$

This is a *summation* of prefixes. Each prefix consists of two actions. *Events* ($\underline{\mathrm{a}} \in \mathcal{E}$) are actions which happen instantaneously and trigger discrete changes. They can be caused by a controller or happen randomly and can depend on the global state of the system, specifically values of variables. In the example, the events are $\underline{\mathrm{on}}$, switching on; $\underline{\mathrm{off}}$, switching off and $\underline{\mathrm{init}}$, the initialisation event.

Activities ($\alpha \in \mathcal{A}$) are *influences* on the evolution of the continuous part of the system and define flows. An activity is defined as a triple and can be parameterised by a set of variables, $\alpha(\overrightarrow{X}) = (\iota, r, I(\overrightarrow{X}))$. This triple consists of an *influence name* ι, a rate of change (or *influence strength*) r and an *influence type name* $I(\overrightarrow{X})$ which describes how that rate is to be applied to the variables involved, or the actual form of the flow[1]. In Heat, there are two distinct activities, (h, r_h, const) and $(h, 0, \mathit{const})$. The first one captures the effect of the heater being off. It affects influence h which represents the influence from the heater on the orbiter's temperature, it has strength 0 and it is associated with the function called *const*. The second gives the effect of the heater being on: the influence name is again h, r_h is the strength of the heater, and the form it takes is *const*. The interpretation of influence types will be specified separately, so that experimentation with different functional forms of the heating flow can occur without modifying the subcomponent. Hence, in HYPE we separate the description of the logical structure of flows from their mathematical interpretation. We now describe the other flows for the example.

$$\begin{aligned} \mathit{Shade} &\overset{\mathit{def}}{=} \underline{\mathrm{up}}{:}(d, -r_d, \mathit{const}).\mathit{Shade} + \underline{\mathrm{down}}{:}(d, 0, \mathit{const}).\mathit{Shade} + \\ &\quad\ \underline{\mathrm{init}}{:}(d, 0, \mathit{const}).\mathit{Shade} \\ \mathit{Sun} &\overset{\mathit{def}}{=} \underline{\mathrm{light}}{:}(s, r_s, \mathit{const}).\mathit{Sun} + \underline{\mathrm{dark}}{:}(s, 0, \mathit{const}).\mathit{Sun} + \underline{\mathrm{init}}{:}(s, 0, \mathit{const}).\mathit{Sun} \\ \mathit{Cool}(X) &\overset{\mathit{def}}{=} \underline{\mathrm{init}}{:}(c, -1, \mathit{linear}(X)).\mathit{Cool}(X) \end{aligned}$$

The only event in the last definition is the initialisation event, as once cooling is in effect it does not change. The influence name is c and its strength is -1. The type $\mathit{linear}(X)$ will be interpreted as a linear function of its formal variable X. We also need to model the change in sunlight. We do this by keeping track of time with the following component (which is kept simple for reasons of space).

$$\mathit{Time} \overset{\mathit{def}}{=} \underline{\mathrm{light}}{:}(t, 1, \mathit{const}).\mathit{Time} + \underline{\mathrm{dark}}{:}(t, 1, \mathit{const}).\mathit{Time} + \underline{\mathrm{init}}{:}(t, 1, \mathit{const}).\mathit{Time}$$

These subcomponents can be combined and the formal variable X can be instantiated with the actual variable K to give the overall uncontrolled system.

$$\mathit{Sys} \overset{\mathit{def}}{=} (((\mathit{Heat} \underset{\{\underline{\mathrm{init}}\}}{\bowtie} \mathit{Shade}) \underset{\{\underline{\mathrm{init}}\}}{\bowtie} \mathit{Sun}) \underset{\{\underline{\mathrm{init}}\}}{\bowtie} \mathit{Cool}(K)) \underset{\{\underline{\mathrm{init}}, \underline{\mathrm{light}}, \underline{\mathrm{dark}}\}}{\bowtie} \mathit{Time}$$

Here $\underset{L}{\bowtie}$ represents parallel synchronisation. L is the set of events over which synchronisation must occur. Events not in L can occur independently. *Sys* is called the *uncontrolled system* because all events are possible and no causal or

[1] For convenience, we will use I for $I(\overrightarrow{X})$ when $\overrightarrow{X}$ can be inferred.

temporal constraints have been imposed yet. For instance, we need to specify that the heater can only be switched off after it has been switched on. We now give controllers/sequencers for the heater, the shade and the effect of the sun.

$$Con_h \overset{def}{=} \underline{\mathrm{on}}.\underline{\mathrm{off}}.Con_h \quad Con_d \overset{def}{=} \underline{\mathrm{up}}.\underline{\mathrm{down}}.Con_d \quad Con_s \overset{def}{=} \underline{\mathrm{light}}.\underline{\mathrm{dark}}.Con_s$$
$$Con \overset{def}{=} Con_h \underset{\emptyset}{\bowtie} Con_d \underset{\emptyset}{\bowtie} Con_s$$

Controllers only have event prefixes. Their behaviour is affected by the state of the system through event conditions which determine when events occur. The controlled system is constructed from synchronisation of the controller and the uncontrolled system and the controller must be prefixed by the initialisation event $\underline{\mathrm{init}}$. For the example, the controlled system is described by

$$\mathit{TempCtrl} \overset{def}{=} \mathit{Sys} \underset{M}{\bowtie} \underline{\mathrm{init}}.Con \quad \text{with} \quad M = \{\underline{\mathrm{init}}, \underline{\mathrm{on}}, \underline{\mathrm{off}}, \underline{\mathrm{up}}, \underline{\mathrm{down}}, \underline{\mathrm{light}}, \underline{\mathrm{dark}}\}.$$

This has defined the structure of our system but we require additional definitions to capture further details. We need to link each influence with an actual variable. This is done using the function iv. For the example, $iv(h) = iv(s) = iv(d) = iv(c) = K$ where K is the actual variable for the temperature of the orbiter, and $iv(t) = T$, the variable for time. Note that an influence can only be associated with one variable, in agreement with the interpretation of influences as flows. This does not mean that only one variable can be affected by an event. For another variable Y that was also affected by the heater being on (power consumption, say), we could define a subcomponent with a prefix $\underline{\mathrm{on}}{:}(p, r, I)$ and set $iv(p) = Y$.

We define the influence types as $[\![const]\!] = 1$ and $[\![linear(X)]\!] = X$. The influence types are used to describe influences are affected by variables in the system. The type *const* is used when there is no effect and $linear(X)$ is used when the value of the variable X modifies an influence. Mass action can also be defined through this mechanism.

Finally, we define what triggers an event, and how it affects variables, with the function ec. Each event condition consists of an activation condition which is a positive boolean formula containing equalities and inequalities on system variables or the symbol $\perp$, and a variable reset which is a conjunction of equality predicates on variables V and V' where V' denotes the new value that V will have after the reset, while V denotes the previous value. Resets of the form $V = V'$ can be left implicit. For the example, the function ec and associated event conditions are

$$ec(\underline{\mathrm{init}}) = (true, (K' = t_0 \wedge T' = 0))$$
$$ec(\underline{\mathrm{off}}) = (K \geq t_1, true) \qquad ec(\underline{\mathrm{on}}) = (K \leq t_2, true)$$
$$ec(\underline{\mathrm{up}}) = (K \geq t_3, true) \qquad ec(\underline{\mathrm{down}}) = (K \leq t_4, true)$$
$$ec(\underline{\mathrm{light}}) = (T = 12, true) \qquad ec(\underline{\mathrm{dark}}) = (T = 24, T' = 0)$$

where the $t_i (1 \leq i \leq 4)$ are fixed temperature values. Most events are urgent – the event must occur as soon as its event condition is satisfied. For events that can happen randomly such as breakdowns, we introduce a special event condition $\perp$ which means that the event can happen at some point in the future[2]. In the example, the $\underline{\mathrm{init}}$ event has an associated event condition of *true* and so this

[2] We have not done so here but probabilistic resets can be used. Tuffin *et al* [21] use a value drawn from a exponential distribution for the time until the next event.

must happen immediately and $\underline{\text{light}}$ happens when 12 hours have passed. The event $\underline{\text{init}}$ has a reset that defines the values of the variables and $\underline{\text{on}}$ has a reset of *true* meaning that no values are changed.

In the preceding informal discussion, we have introduced the main constituents of a HYPE model including the combination of flow components with a controller component, formal and actual variables, association between influences and variables, conditions that specify when events occur, and definitions for the influence type functions. To understand the dynamics of this system, we need to derive ODEs to describe how the variables change over time. To do this we present operational semantics that define the behaviour of our controlled system. Before that we present the formal definition of HYPE.

Definition 1. *A* controlled system *is constructed as follows.*

- Subcomponents *are defined by* $C_s(\overrightarrow{X}) = S$*, where* C_s *is the* subcomponent name *and* S *satisfies the grammar* $S' ::= \underline{a} : \alpha.C_s \mid S' + S'$ $(\underline{a} \in \mathcal{E},\ \alpha \in \mathcal{A})$*, with the free variables of* S *in* $\overrightarrow{X}$*.*
- Components *are defined by* $C(\overrightarrow{X}) = P$*, where* C *is the* component name *and* P *satisfies the grammar* $P' ::= C_s(\overrightarrow{X}) \mid C(\overrightarrow{X}) \mid P' \underset{L}{\bowtie} P'$*, with the free variables of* P *in* $\overrightarrow{X}$ *and* $L \subseteq \mathcal{E}$*.*
- *An* uncontrolled system Σ *is defined according to the grammar* $\Sigma' ::= C_s(\overrightarrow{V}) \mid C(\overrightarrow{V}) \mid \Sigma' \underset{L}{\bowtie} \Sigma'$*, where* $L \subseteq \mathcal{E}$ *and* $\overrightarrow{V}$ *is a set of system variables, instantiating the formal variables of* C *or* C_s*.*
- Controllers *only have events:* $M ::= \underline{a}.M \mid 0 \mid M + M$ *with* $\underline{a} \in \mathcal{E}$ *and* $L \subseteq \mathcal{E}$ *and* $Con ::= M \mid Con \underset{L}{\bowtie} Con$*.*
- *A* controlled system *is* $ConSys ::= \Sigma \underset{L}{\bowtie} \underline{init}.Con$ *where* $L \subseteq \mathcal{E}$*. The set of controlled systems is* $\mathcal{C}_{Sys}$*.*

A controlled system together with the appropriate sets and functions, gives a HYPE model.

Definition 2. *A* HYPE model *is a tuple* $(ConSys, \mathcal{V}, \mathcal{X}, IN, IT, \mathcal{E}, \mathcal{A}, ec, iv, EC, ID)$ *where*

- $ConSys$ *is a controlled system as defined above.*
- $\mathcal{V}$ *is a finite set of variables and* $\mathcal{X}$ *is a finite set of formal variables.*
- IN *is a set of influence names and* IT *is a set of influence type names.*
- $\mathcal{E}$ *is a set of events of the form* $\underline{a}$ *and* $\underline{a}_i$*.*
- $\mathcal{A}$ *is a set of activities of the form* $\alpha(\overrightarrow{X}) = (\iota, r, I(\overrightarrow{X})) \in (IN \times \mathbb{R} \times IT)$*.*
- $ec : \mathcal{E} \to EC$ *maps events to event conditions. Event conditions are pairs of formulas, the first with free variables in* $\mathcal{V}$ *and the second with free variables in* $\mathcal{V} \cup \mathcal{V}'$*.*
- $iv : IN \to \mathcal{V}$ *maps influence names to variable names.*
- EC *is a set of event conditions.*
- ID *is a collection of definitions consisting of a real-valued function for each influence type name* $[\![I(\overrightarrow{X})]\!] = f(\overrightarrow{X})$ *where the variables in* $\overrightarrow{X}$ *are from* $\mathcal{X}$*.*
- $\mathcal{E}$*,* $\mathcal{A}$*,* IN *and* IT *are pairwise disjoint.*

Prefix with influence:

$$\frac{}{\langle \underline{\mathrm{a}} : (\iota, r, I).E, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E, \sigma[\iota \mapsto (r, I)] \rangle}$$

Prefix without influence:

$$\frac{}{\langle \underline{\mathrm{a}}.E, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E, \sigma \rangle}$$

Choice:

$$\frac{\langle E, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E', \sigma' \rangle}{\langle E + F, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E', \sigma' \rangle} \qquad \frac{\langle F, \sigma \rangle \xrightarrow{\mathrm{a}} \langle F', \sigma' \rangle}{\langle E + F, \sigma \rangle \xrightarrow{\mathrm{a}} \langle F', \sigma' \rangle}$$

Parallel without synchronisation:

$$\frac{\langle E, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E', \sigma' \rangle}{\langle E \underset{M}{\bowtie} F, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E' \underset{M}{\bowtie} F, \sigma' \rangle}(\underline{\mathrm{a}} \notin M)$$

$$\frac{\langle F, \sigma \rangle \xrightarrow{\mathrm{a}} \langle F', \sigma' \rangle}{\langle E \underset{M}{\bowtie} F, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E \underset{M}{\bowtie} F', \sigma' \rangle}(\underline{\mathrm{a}} \notin M)$$

Parallel with synchronisation:

$$\frac{\langle E, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E', \tau \rangle \quad \langle F, \sigma \rangle \xrightarrow{\mathrm{a}} \langle F', \tau' \rangle}{\langle E \underset{M}{\bowtie} F, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E' \underset{M}{\bowtie} F', \Gamma(\sigma, \tau, \tau') \rangle}(\underline{\mathrm{a}} \in M, \Gamma \text{ defined})$$

Constant:

$$\frac{\langle E, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E', \sigma' \rangle}{\langle A, \sigma \rangle \xrightarrow{\mathrm{a}} \langle E', \sigma' \rangle}(A \overset{def}{=} E)$$

Fig. 1. Operational semantics for HYPE

When referring to a HYPE model, a term for the controlled system will be used, such as P, and the tuple will be implied. In the case of two HYPE models P and Q without reference to the tuple, we will assume two implied tuples with identical elements except for the first elements. The syntax of HYPE is moderately complex because hybrid systems are complex structures displaying continuous and discrete behaviour. The example shows how it is straightforward to construct a HYPE model once flows and the controller are identified.

3 Operational Semantics

To define the operational semantics, a notion of state is required.

Definition 3. *A* state *of the system is a function $\sigma : IN \rightarrow (\mathbb{R} \times IT)$. The set of all states is $\mathcal{S}$. A* configuration *consists of a controlled system together with a state $\langle ConSys, \sigma \rangle$ and the set of configurations is $\mathcal{F}$.*

For convenience, states may be written as a set of triples of the form $(\iota, r, I(\overrightarrow{X}))$. This is the same form as an activity to reflect the fact that the state captures the activities that are currently in effect. The notion of state here is not a valuation of system variables but rather a collection of flows that occur in the system.

The operational semantics give a labelled transition system over configurations $(\mathcal{F}, \mathcal{E}, \rightarrow \subseteq \mathcal{F} \times \mathcal{E} \times \mathcal{F})$. We write $F \xrightarrow{\underline{\mathrm{a}}} F'$ for $(F, \underline{\mathrm{a}}, F') \in \rightarrow$. In the following, $E, F \in \mathcal{C}_{Sys}$. The rules are given in Figure 1 and are fairly standard. In Choice, Prefix without influence, Parallel without synchronisation and Constant, states are not changed by the application of the rule. For Prefix with influence, the state needs to be updated, and for Parallel with synchronisation, the two new states in the premise of the rule need to be merged using the function Γ.

The updating function $\sigma[\iota \mapsto (r, I)]$ is defined by $\sigma[\iota \mapsto (r, I)](x) = (r, I)$ if $x = \iota$ and $\sigma[\iota \mapsto (r, I)](x) = \sigma(x)$ otherwise. The notation $\sigma[u]$ will also be used for an update, with $\sigma[u_1 \ldots u_n]$ denoting $\sigma[u_1] \ldots [u_n]$.

The partial function $\Gamma : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ is defined as follows.

$$(\Gamma(\sigma, \tau, \tau'))(\iota) = \begin{cases} \tau(\iota) & \text{if } \sigma(\iota) = \tau'(\iota), \\ \tau'(\iota) & \text{if } \sigma(\iota) = \tau(\iota), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

When synchronisation occurs, two states must be merged and the function uses the previous state and the new states to determine which values have changed and then puts these changed values into the new state. Γ will be undefined if both the second and third argument differ from the first argument, namely if the values in the new state both differ from the old state since this represents conflicting updates.

The next two definitions will be useful in referring to the states of the model.

Definition 4. *The* derivative set *of a controlled system P,* ds(P) *is defined as the smallest set satisfying*

- *if* $\langle P, \sigma\rangle \xrightarrow{\underline{init}} \langle P', \sigma'\rangle$ *then* $\langle P', \sigma'\rangle \in \mathrm{ds}(P)$
- *if* $\langle P', \sigma'\rangle \in \mathrm{ds}(P)$ *and* $\langle P', \sigma'\rangle \xrightarrow{\underline{a}} \langle P'', \sigma''\rangle$ *then* $\langle P'', \sigma''\rangle \in \mathrm{ds}(P)$.

Definition 5. *The set of* states *of the derivative set of a controlled system P is defined as* $\mathrm{st}(P) = \{\sigma \mid \langle Q, \sigma\rangle \in \mathrm{ds}(P)\}$.

In the *TempCntl* model, there are eight states of interest (we have omitted the state before the $\underline{\mathrm{init}}$ event) given in Figure 2. Each state captures the influences that are currently active. Since the influence strengths and types of c and t do not change, and each of h, d and s have two possible strengths, there are eight states. Here k abbreviates *const* and $l(X)$, *linear*(X). For example, σ_3 reflects that the heater (h) is off (and has no effect), the shade (d) is up, the sun (s) is shining, cooling (c) is happening, and time (t) is passing.

4 Hybrid Semantics

We extract a set of ODEs for each state which appears in a configuration in the labelled transition system. We will label this set as CS_σ where CS is the constant used for the controlled system and σ is the state.

$$\begin{aligned}
\sigma_0 &= \{h \mapsto (0,k),\ d \mapsto (0,k),\ s \mapsto (0,k),\ c \mapsto (-1, l(K)),\ t \mapsto (1,k)\}\\
\sigma_1 &= \{h \mapsto (0,k),\ d \mapsto (0,k),\ s \mapsto (r_s,k),\ c \mapsto (-1, l(K)),\ t \mapsto (1,k)\}\\
\sigma_2 &= \{h \mapsto (0,k),\ d \mapsto (-r_d,k),\ s \mapsto (0,k),\ c \mapsto (-1, l(K)),\ t \mapsto (1,k)\}\\
\sigma_3 &= \{h \mapsto (0,k),\ d \mapsto (-r_d,k),\ s \mapsto (r_s,k),\ c \mapsto (-1, l(K)),\ t \mapsto (1,k)\}\\
\sigma_4 &= \{h \mapsto (r_h,k),\ d \mapsto (0,k),\ s \mapsto (0,k),\ c \mapsto (-1, l(K)),\ t \mapsto (1,k)\}\\
\sigma_5 &= \{h \mapsto (r_h,k),\ d \mapsto (0,k),\ s \mapsto (r_s,k),\ c \mapsto (-1, l(K)),\ t \mapsto (1,k)\}\\
\sigma_6 &= \{h \mapsto (r_h,k),\ d \mapsto (-r_d,k),\ s \mapsto (0,k),\ c \mapsto (-1, l(K)),\ t \mapsto (1,k)\}\\
\sigma_7 &= \{h \mapsto (r_h,k),\ d \mapsto (-r_d,k),\ s \mapsto (r_s,k),\ c \mapsto (-1, l(K)),\ t \mapsto (1,k)\}
\end{aligned}$$

Fig. 2. The states of the orbiter temperature control system

Given a controlled system CS, and a derivative $\langle CS', \sigma\rangle \in \mathrm{ds}(CS)$, the ODEs associated with the state σ are defined as follows.

$$CS_\sigma = \left\{ \frac{\mathrm{d}V}{\mathrm{d}t} = \sum \{ r \times [\![I(\overrightarrow{W})]\!] \mid iv(\iota) = V \text{ and } \sigma(\iota) = (r, I(\overrightarrow{W}))\} \;\middle|\; V \in \mathcal{V} \right\}$$

So for each state, we have a collection of ODEs, one for each variable V. We obtain the following ODE from σ_3.

$$\mathit{TempCntl}_{\sigma_3} = \left\{ \frac{\mathrm{d}K}{\mathrm{d}t} = r_s - r_d - K,\ \frac{\mathrm{d}T}{\mathrm{d}t} = 1 \right\}$$

Therefore, this process enables us to obtain ODEs describing how the continuous part of the system evolves, and we have different sets of ODEs to describe the different dynamics that can be in operation. We wish to combine this information with the event conditions already defined and an obvious way to do this is to translate this information into a hybrid automaton. Therefore, this well-supported formalism provides a powerful back-end for HYPE.

Hybrid automata are dynamic systems presenting both discrete and continuous evolution. They consist of variables evolving continuously in time, subject to abrupt changes induced by discrete instantaneous *control* events. When discrete events happen the automaton enters its next *mode*, where the rules governing the flow of continuous variables change. See [13] for further details.

Definition 6. *A* hybrid automaton *is a tuple* $\mathcal{H} = (V, E, \mathbf{X}, \mathcal{E}, \mathit{flow}, \mathit{init}, \mathit{inv}, \mathit{event}, \mathit{jump}, \mathit{reset}, \mathit{urgent})$, *where:*

– $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a finite set of real-valued variables. The time derivative of* X_j *is* $\dot{X}_j$*, and the value of* X_j *after a change of* mode *is* X'_j*.*
– *the* control graph $G = (V, E)$ *is a finite labelled graph. Vertices* $v \in V$ *are the* (control) modes*, while edges* $e \in E$ *are called* (control) switches *and model the happening of a discrete event.*
– *Associated with each vertex* $v \in V$ *there is a set of ordinary differential equations* $\dot{\mathbf{X}} = \mathit{flow}(v)$ *referred to as the* flow conditions*. Moreover,* $\mathit{init}(v)$ *and* $\mathit{inv}(v)$ *are two formulae on* $\mathbf{X}$ *specifying the* admissible initial conditions *and*

some invariant conditions *that must be true during the continuous evolution of variables in v.*
- *Edges $e \in E$ of the control graph are labelled by an event $event(e) \in \mathcal{E}$ and by $jump(e)$, a formula on $\mathbf{X}$ stating for which values of variables each transition is active, and by $reset(e)$, a formula on $\mathbf{X} \cup \mathbf{X}'$ specifying the change of the variables' values after the transition has taken place. Moreover, each edge $e \in E$ can be declared urgent, by setting to true the boolean flag $urgent(e)$, meaning that the transition is taken at once when $jump(e)$ becomes true. Otherwise, the transition can be taken nondeterministically whenever $jump(e)$ is true.*

Consider a HYPE model $(P_0, \mathcal{V}, \mathcal{X}, IN, IT, \mathcal{E}, \mathcal{A}, ec, iv, EC, ID)$ and suppose its initial configuration is $\langle P_0, \sigma_0 \rangle \in \mathcal{F}$. For P_0 the only possible transition is the event $\underline{\text{init}}$. Let $\langle P, \sigma \rangle$ be the configuration reached after its occurrence, $\langle P_0, \sigma_0 \rangle \xrightarrow{\underline{\text{init}}} \langle P, \sigma \rangle$. Moreover, we denote by $act_{\underline{\text{a}}}$ and $res_{\underline{\text{a}}}$ the activation conditions and the resets associated with an event $\underline{\text{a}} \in \mathcal{E}$, so $ec(\underline{\text{a}}) = (act_{\underline{\text{a}}}, res_{\underline{\text{a}}})$.

Definition 7. *The hybrid automaton*
$\mathcal{H} = (V, E, \mathbf{X}, \mathcal{E}, flow, init, inv, event, jump, reset, urgent)$ can be obtained from the HYPE model $(P_0, \mathcal{V}, \mathcal{X}, IN, IT, \mathcal{E}, \mathcal{A}, ec, iv, EC, ID)$ as follows.

- *The set of modes V is the set of configurations reachable in 0 or more steps from $\langle P, \sigma \rangle$, namely* $\mathrm{ds}(P_0)$.
- *The edges E of the control graph connect two modes (v_1, v_2) iff $v_1 = \langle P_1, \sigma_1 \rangle$, $v_2 = \langle P_2, \sigma_2 \rangle$ and $\langle P_1, \sigma_1 \rangle \xrightarrow{\underline{a}} \langle P_2, \sigma_2 \rangle$ is a derivation for some $\underline{a}$.*
- *$\mathbf{X} = \mathcal{V}$ is the set of variables of the HYPE system.*
- *$\mathcal{E}$ is the set of events $\mathcal{E}$ of P_0.*
- *Let $v_j = \langle P_j, \sigma_j \rangle$, then*
$$flow(v_j)[X_i] = \sum\{r[\![I(\overrightarrow{W})]\!] \mid iv(\iota) = X_i \text{ and } \sigma_j(\iota) = (r, I(\overrightarrow{W}))\}$$
- $init(v) = \begin{cases} res_{\underline{init}}, \text{ if } v = \langle P, \sigma \rangle \\ false, \text{ otherwise} \end{cases}$ *with primes removed from variables*[3].
- *$inv(v) = true$.*
- *Let $e = (\langle P_1, \sigma_1 \rangle, \langle P_2, \sigma_2 \rangle)$ with $\langle P_1, \sigma_1 \rangle \xrightarrow{\underline{a}} \langle P_2, \sigma_2 \rangle$. Then $event(e) = \underline{a}$ and $reset(e) = res_{\underline{a}}$. Moreover, if $act_{\underline{a}} \neq \bot$, then $jump(e) = act_{\underline{a}}$ and $urgent(e) = true$, otherwise $jump(e) = true$ and $urgent(e) = false$.*

Figure 3 shows the HYPE model from the example as a hybrid automaton. Note that which states are visited is determined by the values of the $t_i (0 \leq i \leq 4)$.

5 Equivalence Semantics

We define equivalent behaviour with respect to the labelled transition system given in Section 3 thereby focussing on the configuration of the system rather than evaluations of continuous variables.

[3] $res_{\underline{\text{init}}}$ is a reset so uses primed variables to refer to the new values of variables whereas $init(v)$ is an initialization condition and refers to variables without primes.

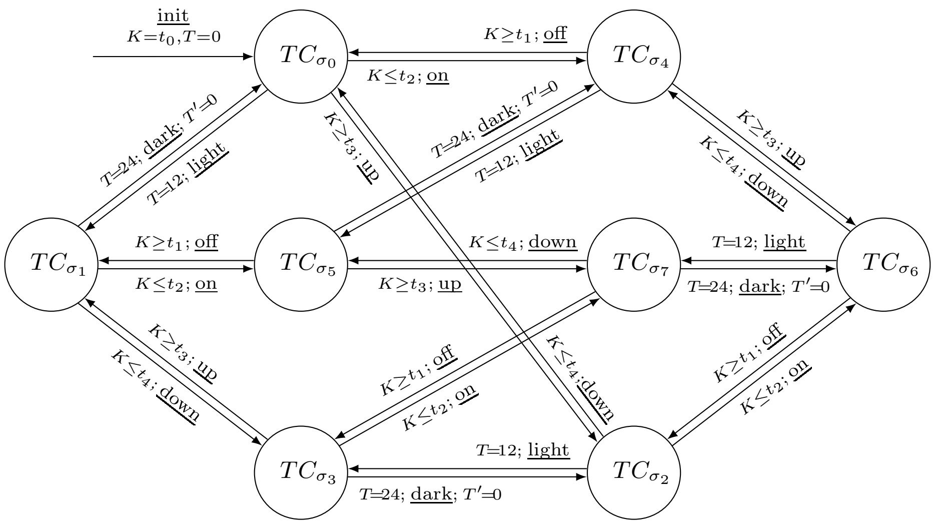


Fig. 3. Hybrid automaton of the orbiter temperature control system

Definition 8. *A relation* $B \subseteq \mathcal{C}_{Sys} \times \mathcal{C}_{Sys}$ *is a* system bisimulation *if for all* $(P, Q) \in B$ *whenever*

1. $\langle P, \sigma\rangle \xrightarrow{a} \langle P', \sigma'\rangle$, *there exists* $\langle Q', \sigma'\rangle$ *with* $\langle Q, \sigma\rangle \xrightarrow{a} \langle Q', \sigma'\rangle$, $(P', Q') \in B$.
2. $\langle Q, \sigma\rangle \xrightarrow{a} \langle Q', \sigma'\rangle$, *there exists* $\langle P', \sigma'\rangle$ *with* $\langle P, \sigma\rangle \xrightarrow{a} \langle P', \sigma'\rangle$, $(P', Q') \in B$.

P and Q are system bisimilar*,* $P \sim_s Q$ *if they are in a system bisimulation.*

System bisimulation is a congruence for our operators.

Theorem 1. $\sim_s$ *is a congruence for Prefix, Choice and Parallel.*

Proof sketch. Straightforward. Let $P_1 \sim_s P_2$*. Interesting cases are Prefix with influence and Parallel with synchronisation. For the former,* $\langle \underline{a} : (\iota, r, I).P_1, \sigma\rangle \xrightarrow{\underline{a}} \langle P_1, \sigma[\iota \mapsto (r, I)]\rangle$ *and likewise* $\langle \underline{a} : (\iota, r, I).P_2, \sigma\rangle \xrightarrow{\underline{a}} \langle P_2, \sigma[\iota \mapsto (r, I)]\rangle$ *as required. For the latter, we need to show that* $B = \{(P_1 \underset{L}{\bowtie} Q, P_2 \underset{L}{\bowtie} Q) | P_1 \sim_s P_2\}$ *is a system bisimulation. If* $\langle P_1 \underset{L}{\bowtie} Q, \sigma\rangle \xrightarrow{\underline{a}} \langle P_1' \underset{L}{\bowtie} Q', \Gamma(\sigma, \sigma', \sigma'')\rangle$ *and* $\underline{a} \in L$ *then* $\langle P_1, \sigma\rangle \xrightarrow{\underline{a}} \langle P_1', \sigma'\rangle$ *and* $\langle Q, \sigma\rangle \xrightarrow{\underline{a}} \langle Q', \sigma''\rangle$*. Since* $P_1 \sim_s P_2$*,* $\langle P_2, \sigma\rangle \xrightarrow{\underline{a}} \langle P_2', \sigma'\rangle$ *with* $P_1' \sim_s P_2'$*, and hence* $\langle P_2 \underset{L}{\bowtie} Q, \sigma\rangle \xrightarrow{\underline{a}} \langle P_2' \underset{L}{\bowtie} Q', \Gamma(\sigma, \sigma', \sigma'')\rangle$ *as required.* $\square$

We are interested in the link between the ODEs obtained from bisimilar systems. Before we can consider this, some definitions and lemmas are required.

As we saw in the example, the init event in the controlled system allows for the initialisation of variables. Typically subcomponents are defined as a number of simple loops, reflecting events that can occur and the associated changes in the continuous part of the system. These requirements can be formalised as follows.

Definition 9
A well-defined *controlled system has the following properties.*

1. *For each subcomponent $C_s(\overrightarrow{X}) \stackrel{\text{def}}{=} S$, the only subcomponent name that can appear in S is $C_s(\overrightarrow{X})$.*
2. *In each subcomponent $C_s(\overrightarrow{X})$, the event $\underline{a}$ can only appear once.*
3. *In each subcomponent $C_s(\overrightarrow{X})$, each ι that appears, must also appear in a prefix with $\underline{init}$.*
4. *Across all subcomponents, each pair $\underline{a}$ and ι must appear at most once together in a prefix.*
5. *In any component $C(\overrightarrow{X})$, any event that appears in more than one subcomponent must be synchronised on.*
6. *In the controlled system Σ and Con must have the same events and these must all appear in L.*

The example is a well-defined controlled system. For such systems, we can show that the current state in a configuration does not determine which future events happen (only the controller can influence this) and hence the state can be discounted in certain settings. Note that Condition 4 guarantees that Γ is always defined. Consider a prefix $\underline{a} : (\iota, r, I(\overrightarrow{X}))$. On the occurrence of $\underline{a}$, the value of ι in the state will be updated exactly once and hence Γ is always defined. If we did not have Condition 4 and event and influence names appeared multiple times, Γ could be undefined which is not a desired feature of modelling in HYPE. We now consider properties of well-defined HYPE systems.

Lemma 1. *In a well-defined controlled system, if $\langle P', \sigma[u_1 \ldots u_n]\rangle$ is a derivative of $\langle P, \sigma\rangle$, then $\langle P', \tau[u_1 \ldots u_n]\rangle$ is a derivative of $\langle P, \tau\rangle$.*

Proof sketch. It can be shown by induction on the derivation of transitions that if $\langle P', \sigma'\rangle$ is a n-step derivative of $\langle P, \sigma\rangle$, $\langle P', \tau'\rangle$ is a n-step derivative of $\langle P, \tau\rangle$. Moreover, $\sigma' = \sigma[u_1 \ldots u_n]$ and $\tau' = \tau[u_1 \ldots u_n]$ for appropriate $u_1, \ldots, u_n$. □

We can also consider what happens to a system after the first $\underline{init}$ event. The first transition must be an $\underline{init}$ event since *Con* is prefixed by $\underline{init}$ and all actions are synchronised. The following result shows that the starting state is irrelevant since the $\underline{init}$ action will set every value in the state.

Lemma 2. *Let P be a well-defined controlled system. If $\langle P, \sigma\rangle \xrightarrow{\underline{init}} \langle P', \sigma'\rangle$ and $\langle P, \tau\rangle \xrightarrow{\underline{init}} \langle P', \tau'\rangle$ then $\sigma' = \tau'$.*

Proof sketch. Since there is exactly one $\underline{init} : (\iota, r, I)$ prefix for every ι, the occurrence of an $\underline{init}$ event will update every ι value in the state and the previous value of ι is irrelevant. □

The following result shows that it is the prefixes which determine the behaviour of the controlled systems because of the restrictions imposed on well-defined controlled systems.

Definition 10. *The* set of prefixes *of an uncontrolled system Sys, pre(Sys) is defined structurally as follows.*

– $pre(\underline{a} : (\iota, r, I(\overrightarrow{X})).S) = \{\underline{a} : (\iota, r, I(\overrightarrow{X}))\}$
– $pre(S_1 + S_2) = pre(S_1) \cup pre(S_2)$
– $pre(P \underset{L}{\bowtie} Q) = pre(P) \cup pre(Q)$

Theorem 2. *Let $\Sigma_1 \underset{L}{\bowtie} \underline{init}.Con$ and $\Sigma_2 \underset{L}{\bowtie} \underline{init}.Con$ be two well-defined controlled systems. If $pre(\Sigma_1) = pre(\Sigma_2)$ then $\Sigma_1 \underset{L}{\bowtie} \underline{init}.Con \sim_s \Sigma_2 \underset{L}{\bowtie} \underline{init}.Con$.*

Proof sketch. We need to show that $\{(\Sigma_1, \Sigma_2)\}$ is a system bisimulation. Since the two systems are well-defined, any event can always occur and events appearing in more than one component are synchronised. Hence $\Sigma_1 \xrightarrow{a} \Sigma_1$ and $\Sigma_2 \xrightarrow{a} \Sigma_2$ for each event $\underline{a} \in pre(\Sigma_1)$. By congruence, we have the result. □

This result allows us to tell whether two HYPE models are bisimilar by inspecting the prefixes in the model description to see if they are the same. Hence bisimulation can be checked syntactically. The next two results show that bisimilar HYPE models have the same ODEs.

Lemma 3. *Let P and Q be well-defined controlled systems. If $P \sim_s Q$ then their states are equal, $\mathrm{st}(P) = \mathrm{st}(Q)$.*

Proof sketch. Consider $\langle P', \sigma' \rangle$ a derivative of $\langle P, \sigma \rangle$ then $\sigma' \in \mathrm{st}(P)$. Since $P \sim_s Q$, we can find $\langle Q', \sigma' \rangle$ with $\sigma' \in \mathrm{st}(Q)$. Hence $\mathrm{st}(P) \subseteq \mathrm{st}(Q)$ and vice versa. □

Theorem 3. *Let P and Q be well-defined controlled systems. If $P \sim_s Q$ then for every state $\sigma \in \mathrm{st}(P)$, $P_\sigma = Q_\sigma$.*

Proof sketch. The well-defined systems are $(P, \mathcal{V}, \mathcal{X}, IN, IT, \mathcal{E}, \mathcal{A}, ec, iv, EC, ID)$ and $(Q, \mathcal{V}, \mathcal{X}, IN, IT, \mathcal{E}, \mathcal{A}, ec, iv, EC, ID)$. By Lemma 3, $\sigma \in \mathrm{st}(P)$ implies $\sigma \in \mathrm{st}(Q)$. Hence

$$P_\sigma = \left\{ \frac{\mathrm{d}V}{\mathrm{d}t} = \sum \left\{ r[\![I(\overrightarrow{W})]\!] \;\middle|\; iv(\iota) = V \text{ and } \sigma(\iota) = (r, I(\overrightarrow{W})) \right\} \;\middle|\; V \in \mathcal{V} \right\} \text{ and}$$

$$Q_\sigma = \left\{ \frac{\mathrm{d}V}{\mathrm{d}t} = \sum \left\{ r[\![I(\overrightarrow{W})]\!] \;\middle|\; iv(\iota) = V \text{ and } \sigma(\iota) = (r, I(\overrightarrow{W})) \right\} \;\middle|\; V \in \mathcal{V} \right\}$$

which are clearly the same. □

The converse of Theorem 3 does not hold. It is possible for two states to be the same and hence give identical ODEs, but this does not mean that their associated derivatives are system bisimilar.

6 Related Work

As mentioned in the introduction, HYPE takes a finer grained, less monolithic approach than the other process algebras for hybrid systems [3, 6, 20, 22] because it enables the modelling of individual flows. In [16] the comparison of these other process algebras is based on a train gate controller example and in each case, the train, gate and controller components have to be fully, sequentially described and then composed in parallel. This would also be necessary for the modelling of our orbiter example. For each of these process algebras, somewhere in the syntactic description of the system, a term such as $\dot{K} = r_s - r_d - K$, as well as terms for each of the other seven ODEs for the variable K, would need to appear to describe the continuous behaviour that can occur. By comparison, a modeller using HYPE would only need to model the individual flows, and not construct the ODEs explicitly. This could allow non-experts to model hybrid systems more easily.

A classical formalism for expressing hybrid systems is hybrid automata [13]. They are usually specified by defining explicitly both the control graph and the dynamical conditions within each mode, in terms of differential equations or, more generally, differential inclusions. Two hybrid automata can be composed in parallel by synchronizing transitions on shared events in the control graph [13]. Flow conditions are combined by taking the logical conjunction of the predicates defining them. Where flows are defined by differential equations, the equation for each variable X must be defined only in one component, otherwise a logical inconsistency may arise. However, the variable may depend explicitly on variables governed by other components.

The modelling style of HYPE is quite different. Activities are identified with atomic flows acting on system variables. ODEs are then derived for an individual state by adding the different atomic flows acting on each variable. Activities can change in response to the happening of discrete events, which are controlled not by components but by an external controller and triggered by event conditions. This results in a separation of the description of the response of the system to events from the discrete control structure imposing causality on the happening of events. In contrast to hybrid automata, HYPE allows the separate description of flow conditions, event conditions, and the control graph, making easier the task of modifying the controller or the interactions with the environment. Furthermore, the fact that influence types are defined separately from the structure of the model also separates modelling concerns. In addition, compositionality of HYPE manifests on the set of activities (the state of the system) rather than on ODEs, hence we can allow different components of the system to influence the same continuous variable: the combined effect is obtained by superimposing flows, namely by addition on the right hand side of ODEs. Since a HYPE model can be expressed as a hybrid automaton, when using HYPE to model one gets the advantages of HYPE together with the formalism of hybrid automata. This is a distinct advantage over languages such as Charon [1], Shift [8], and HyCharts [11]. These are all compositional formalisms describing hybrid systems which do not map so readily to hybrid automata. They differ from HYPE in that they do not have the simple syntax and structured operational semantics of a process algebra.

Another formalism which has a mechanism to combine flows is hybrid action systems [19]. This language is based on Dijkstra's guarded command language and has predicate transformer semantics. Differential actions consist of guards before ODEs, and parallel composition of these actions is defined as a linear combination of functions with the addition of functions over any shared domains.

This differs from our approach where we associate influence names with a specific variable and then sum over all influences for a given variable to obtain the ODE.

Physical systems can also be modelled by constitutive equations and bond graphs [18]. This is a modelling technique in which a system's components are described by means of equations relating main physical quantities of interest. Components are then glued together by imposing suitable conservation laws. This results in a set of differential equations with algebraic constraints, which after an algebraic manipulation, can be simplified by removing variables and constraints. There are also hybrid extensions of the bond graph method which have been represented in a hybrid process algebra [5], to deal with discontinuities in physical systems (such as a bouncing ball).

In HYPE, instead, the modelling activity concentrates around the notion of flow or influence, which is not explicitly connected with physical quantities or with conservation laws, and components are described by specifying the way they react to external events through flows modifications. This may be less natural for certain physical systems, as one has to identify the different influences acting on each variable of interest (without relying on the implicit derivation mechanism provided by conservation laws). However, HYPE's modelling style is straightforwardly applicable to a wider class of systems, at different levels of abstraction.

6.1 Bisimulations on Hybrid Systems

Other process algebras for hybrid systems use a hybrid transition system with two types of transition: one type represents discrete events and the other continuous evolution of the system [16]. By comparison, our transition system only has transitions for events. This gives a smaller transition system on which it is possible to consider simpler notions of equivalence.

Bergstra and Middelburg [3] present two bisimulations for their process algebra for hybrid systems, defined over a hybrid transition system. One bisimulation fits with their axiomatic definition, and the other gives congruence with respect to the parallel operator. Recast for our transition system and for our language these two bisimulations equate the same controlled systems and are the same as our system bisimulation.

The standard notion of bisimulation of hybrid automata is defined for a transition system encoding the dynamical evolution. Both continuous transitions and discrete transitions are used. These two relations are combined (usually interleaving discrete and continuous transitions) into one relation which defines the behaviour of the system. The hybrid automata bisimulation is defined on this relation in the usual way [7, 12, 13]. Most of the research into these bisimulations focusses on finding restrictions on hybrid automata syntax implying the existence of a finite bisimulation quotient thus guaranteeing decidability of reachability and of model checking, such as [17].

The transition relation defined for HYPE is very different. It does not encode any notion of time-dynamics and hence it acts at the level of the system description, identifying equivalent modes. In this sense, it is closer to the notion of

U-bisimulation for hybrid automata which acts on the control graph [2]. If we consider a simplified form of U-bisimulation, we can show that any HYPE models that are system bisimilar, are also equated by this bisimulation. The converse does not hold since different states can lead to the same ODEs. Hence bisimulation for hybrid automata is coarser than that for HYPE. In a HYPE model we make explicit the source of each single flow of the system, while in a hybrid automaton flows are merged together in differential equations, and they cannot be separated out into single influences. Stated otherwise, in hybrid automata ODEs lose information about the logic of the system.

7 Conclusions and Further Work

We have presented a process algebra for hybrid systems with novel features that include a fine-grained approach to modelling flows and an explicit controller. Since a HYPE model can be expressed as a hybrid automaton, tools such as the model checker HyTech [14] can be used to explore these systems.

As well as the orbiter temperature control example, we have successfully modelled a dual-tank system (as described by [21]), a bottling line (as described by [3]) and the abstract view of the repressilator [10] (as described by [4]).

Considering further work, we wish to investigate bisimulation between models which differ in more than the description of the controlled system. This would involve the use of bijections or surjections between models. We also wish to consider a more general definition of bisimulation which allows an equivalence over states yet ensures that the same ODEs are produced. Related to the idea of equivalence is axiomatisation and this will also be investigated.

Our models currently give rise to hybrid automata with invariants which are always true (see Definition 7) and we will investigate relaxing this condition. We have focussed on the embedding of HYPE models in hybrid automata; in future we wish to identify the class of hybrid automata to which HYPE models correspond as a way to understand how to solve HYPE models.

Acknowledgements. Thanks to Stephen Gilmore and Pieter Cuijpers for their helpful comments. Vashti Galpin is supported by the EPSRC SIGNAL Project, Grant EP/E031439/1. Luca Bortolussi is supported by GNCS and FIRB LIBi. Jane Hillston is supported by the EPSRC under ARF EP/c543696/01.

References

1. Alur, R., Grosu, R., Lee, I., Sokolsky, O.: Compositional modeling and refinement for hierarchical hybrid systems. Journal of Logic and Algebraic Programming 68(1-2), 105–128 (2006)
2. Antoniotti, M., Mishra, B., Piazza, C., Policriti, A., Simeoni, M.: Modeling cellular behavior with hybrid automata: Bisimulation and collapsing. In: Priami, C. (ed.) CMSB 2003. LNCS, vol. 2602, pp. 57–74. Springer, Heidelberg (2003)
3. Bergstra, J.A., Middelburg, C.A.: Process algebra for hybrid systems. Theoretical Computer Science 335(2-3), 215–280 (2005)

4. Bortolusssi, L., Policriti, A.: Hybrid approximation of stochastic process algebras for systems biology. In: IFAC World Congress, Seoul, South Korea (July 2008)
5. Cuijpers, P., Broenink, J., Mosterman, P.: Constitutive hybrid processes: a process-algebraic semantics for hybrid bond graphs. Simulation 8, 339–358 (2008)
6. Cuijpers, P.J.L., Reniers, M.A.: Hybrid process algebra. Journal of Logic and Algebraic Programming 62(2), 191–245 (2005)
7. Davoren, J.M., Tabuada, P.: On simulations and bisimulations of general flow systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 145–158. Springer, Heidelberg (2007)
8. Deshpande, A., Göllü, A., Varaiya, P.: SHIFT: A formalism and a programming language for dynamic networks of hybrid automata. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1996. LNCS, vol. 1273, pp. 113–133. Springer, Heidelberg (1997)
9. Elowitz, M.B., Leibler, S.: A synthetic oscillatory network of transcriptional regulators. Nature 403, 335–338 (2000)
10. Galpin, V., Hillston, J., Bortolussi, L.: HYPE applied to the modelling of hybrid biological systems. Electronic Notes in Theoretical Computer Science 218, 33–51 (2008)
11. Grosu, R., Stauner, T.: Modular and visual specification of hybrid systems: An introduction to HyCharts. Formal Methods in System Design 21(1), 5–38 (2002)
12. Haghverdi, E., Tabuada, P., Pappas, G.J.: Bisimulation relations for dynamical, control, and hybrid systems. Theoretical Computer Science 342(2-3), 229–261 (2005)
13. Henzinger, T.A.: The theory of hybrid automata. In: LICS, pp. 278–292 (1996)
14. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HYTECH: A model checker for hybrid systems. International Journal on Software Tools for Technology Transfer 1(1-2), 110–122 (1997)
15. Hillston, J.: Fluid flow approximation of PEPA models. In: Second International Conference on the Quantitative Evaluation of Systems (QEST 2005), pp. 33–43. IEEE Computer Society, Los Alamitos (2005)
16. Khadim, U.: A comparative study of process algebras for hybrid systems. Computer Science Report CSR 06-23, Technische Universiteit Eindhoven (2006), `http://alexandria.tue.nl/extra1/wskrap/publichtml/200623.pdf`
17. Lafferriere, G., Pappas, G.J., Sastry, S.: O-minimal hybrid systems. Mathematics of Control, Signals, and Systems 13(1), 1–21 (2000)
18. Paynter, H.: Analysis and Design of Engineering Systems. MIT Press, Cambridge (1961)
19. Rönkkö, M., Ravn, A.P., Sere, K.: Hybrid action systems. Theoretical Computer Science 290(1), 937–973 (2003)
20. Rounds, W.C., Song, H.: The Φ-calculus: A language for distributed control of reconfigurable embedded systems. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 435–449. Springer, Heidelberg (2003)
21. Tuffin, B., Chen, D.S., Trivedi, K.S.: Comparison of hybrid systems and fluid stochastic Petri nets. Discrete Event Dynamic Systems: Theory and Applications 11, 77–95 (2001)
22. van Beek, D., Man, K., Reniers, M., Rooda, J., Schiffelers, R.: Syntax and consistent equation semantics of hybrid χ. Journal of Logic and Algebraic Programming 68(1-2), 129–210 (2006)

Variable Binding, Symmetric Monoidal Closed Theories, and Bigraphs

Richard Garner[1], Tom Hirschowitz[2], and Aurélien Pardon[3]

[1] Cambridge University
[2] CNRS, Université de Savoie
[3] ENS Lyon

Abstract. This paper investigates the use of symmetric monoidal closed (SMC) structure for representing syntax with variable binding, in particular for languages with linear aspects. In this setting, one first specifies an SMC *theory* $\mathcal{T}$, which may express binding operations, in a way reminiscent from higher-order abstract syntax (HOAS). This theory generates an SMC category $S(\mathcal{T})$ whose morphisms are, in a sense, terms in the desired syntax. We apply our approach to Jensen and Milner's (abstract binding) bigraphs, in which *processes* behave linearly, but *names* do not. This leads to an alternative category of bigraphs, which we compare to the original.

1 Introduction

How to rigorously handle variable binding? The recent amount of research on this issue attests its delicacy [10, 9, 15]. A main difficulty is perhaps to reconcile α-conversion with initial algebra semantics: α-conversion equates terms up to renaming of bound variables; initial algebra semantics requires that terms form the free, or initial, model specified by a given signature.

We here investigate an approach sketched by Coccia et al. [4], based on SMC theories, which they called GS·Λ theories. In this setting, one first specifies an SMC *theory* $\mathcal{T}$, which may express binding operations, in a way reminiscent from HOAS [27, 8, 17]. This theory freely generates an SMC category $S(\mathcal{T})$ whose morphisms are, in a sense, terms in the desired syntax. The known presentations of $S(\mathcal{T})$ mainly fall into two classes: syntactic or graphical. Our emphasis in this paper is on a graphical presentation of $S(\mathcal{T})$ and example applications.

We start in Section 2 with an expository account of SMC theories and our construction of $S(\mathcal{T})$. This construction yields a monadic adjunction, and hence provides an initial algebra semantics for variable binding. The morphisms of $S(\mathcal{T})$ look like abstract syntax, e.g., in the sense of Wadsworth's λ-graphs [33]. Technically, they are a variant of proof nets in intuitionistic multiplicative linear logic [13] (IMLL): they are equivalence classes of special graphs called *linkings*, which must satisfy a certain *correctness* condition. Linkings compose by "glueing" the graphs together, and correctness is stable under composition. Finally, a standard issue in variable binding is induction. We propose a general induction principle derived from Girard's sequentialisation theorem [13, 6].

We continue with a few examples in Section 3, to demonstrate the use of $S(\mathcal{T})$ as a representation for syntax with variable binding. In languages with variable binding, there is a standard notion of term with a hole, or *context*, and an associated operation of hole-filling, or substitution with capture. Morphisms of $S(\mathcal{T})$ and their composition are very close to contexts and hole-filling, except that hole-filling is generally total, while composition only concerns, well, composable morphisms. And moreover, contexts are here generalised to be multi-hole and higher-order (holes with holes, and so on). This kind of substitution would show up with any closed structure, e.g., cartesian closed categories, but is not directly available in more traditional approaches [9, 15, 10]. Conversely, non-linear capture-avoiding substitution requires a bit more work in our setting. We only sketch it here, and briefly discuss alternatives to the general induction principle of Section 2. Along the way, we prove a decomposition result showing the flexibility of our approach, and we observe that the use of SMC structure facilitates the cohabitation of linear and non-linear aspects in a common language.

To further support this latter claim, Section 4 studies Jensen and Milner's bigraphs [20] in our setting, in which *processes* behave linearly, but *names* do not. We translate each bigraphical signature $\mathcal{K}$ into an SMC theory $\mathcal{T}_{\mathcal{K}}$, and show that bigraphs over $\mathcal{K}$ essentially embed into $S(\mathcal{T}_{\mathcal{K}})$, the free SMC category generated by $\mathcal{T}_{\mathcal{K}}$. Furthermore, although $S(\mathcal{T}_{\mathcal{K}})$ is much richer than the original, the embedding is surjective on whole programs.

2 Symmetric Monoidal Closed Theories

In this section, we provide an overview of the construction of $S(\mathcal{T})$. A more technical presentation may be found in our work [11], which itself owes much to Trimble [32] and Hughes [19].

2.1 Signatures

Roughly, an SMC category is a category with a tensor product $\otimes$ on objects and morphisms, symmetric in the sense that $A \otimes B$ and $B \otimes A$ are isomorphic, and such that $(- \otimes A)$ has a right adjoint $(A \multimap -)$, for each object A. We do not give further details, since we are interested in describing the free such category, which is easier. Knowing that there is a category SMCCat of SMC categories and strictly structure-preserving functors should be enough to grasp the following.

An SMC *signature* Σ consists of a set X of *sorts*, equipped with a (directed) graph whose vertices are IMLL formulae over X, as defined by:

$$A, B, \ldots \in \mathcal{F}(X) ::= x \mid I \mid A \otimes B \mid A \multimap B \qquad\qquad x \in X,$$

where $\otimes$ is *tensor* and $\multimap$ is *(linear) implication.*

Example 1. Consider the π-calculus: we will see in Section 3 that the corresponding signature has one sort v for names and one sort t for processes, and among others two operations *send* and *get* of types:

$$(v \otimes v \otimes t) \xrightarrow{s} t \qquad\qquad (v \otimes (v \multimap t)) \xrightarrow{g} t.$$

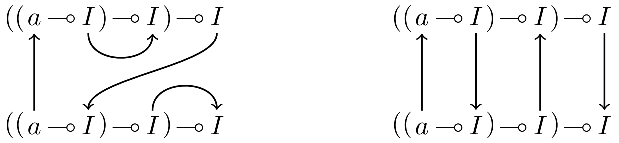
A morphism of signatures $(X, \Sigma) \longrightarrow (Y, \Sigma')$ is a function $X \xrightarrow{f} Y$, equipped with a morphism of graphs, whose vertex component is "$\mathcal{F}(f)$", i.e., the function sending any formula $A(x_1, \ldots, x_n)$ to $A(f(x_1), \ldots, f(x_n))$. This defines a category SMCSig of signatures.

There is a forgetful functor $\mathsf{SMCCat} \xrightarrow{U} \mathsf{SMCSig}$ sending each SMC category $\mathcal{C}$ to the graph with as vertices formulae in $\mathcal{F}(\mathrm{ob}(\mathcal{C}))$, and as edges $A \longrightarrow B$ the morphisms $[\![A]\!] \longrightarrow [\![B]\!]$ in $\mathcal{C}$, where $[\![A]\!]$ is defined inductively to send each syntactic connective to the corresponding function on $\mathrm{ob}(\mathcal{C})$.

We will now construct an SMC category $S(\Sigma)$ from any signature Σ, and extend this to a functor $\mathsf{SMCSig} \xrightarrow{S} \mathsf{SMCCat}$, left adjoint to U. How does $S(\Sigma)$ look like? Under the Curry-Howard-Lambek correspondence, an SMC signature amounts to a set of IMLL axioms, and the free SMC category $S(\Sigma)$ over a signature Σ has as morphisms IMLL proofs under the corresponding axioms, modulo cut elimination. Or, equivalently, morphisms are a variant of proof nets, which we introduce gradually in the next sections.

2.2 The Free Symmetric Monoidal Closed Category over a Set

In the absence of axioms, i.e., given only a set of sorts, or propositional variables, say X, Hughes [19] has devised a simple presentation of $S(X)$. Consider for a guiding example the two endomorphisms of $((a \multimap I) \multimap I) \multimap I$:



(domain on top for the whole paper, the right-hand morphism is the identity).

First, the *ports* of a formula, i.e., occurrences of sorts or of I, are given polarities: a port is *positive* when it lies to the left of an even number of $\multimap$'s in the abstract syntax tree, and *negative* otherwise[1]. For example, in the above formula, a and the middle I are negative, the other occurrences of I being positive. When constructing morphisms $A \longrightarrow B$, the ports in A and B will be assigned a *global* polarity, or a polarity *in* the morphism: the ports of B have their polarity in B, while those of A have the opposite polarity. For example, in the above examples, the occurrence of a in the domain is positive.

A *linking* is a partial function f from negative ports to positive ports, such that for each sort a, f maps negative a ports to positive a ports, bijectively. We observe that this allows to connect I ports to ports of any type. This last bit does not appear in the above example; it does in (1) below. Clearly from the example, linkings are kind of graphs, and we call their edges *wires*.

A linking is then *correct* when (i) it is a total function, and (ii) it satisfies the Danos-Regnier (DR) criterion [6]. The latter roughly goes as follows. An IMLL formula may be translated to a *classical* formula, as defined by the grammar:

[1] The sign of a port in A is directly apparent viewing A is a classical LL formula, see the next paragraph.

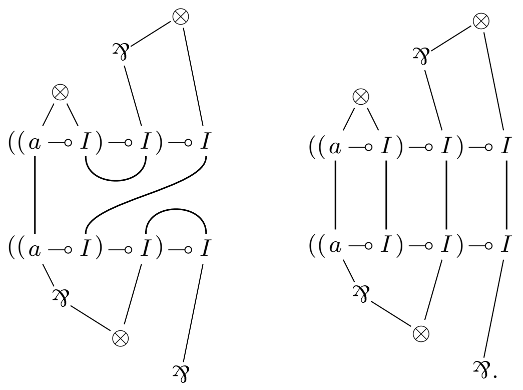


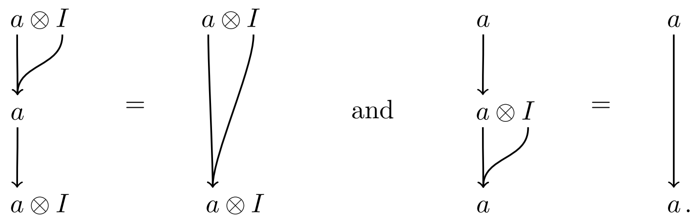
Fig. 1. Example switching

$$\begin{array}{rlclcl} A, B, \ldots ::= & x & \mid & I & \mid & A \otimes B \\ \mid & x^{\perp} & \mid & \perp & \mid & A \parr B. \end{array}$$

The de Morgan dual $A^{\perp}$ of A is defined as usual (by swapping connectives, vertically in the above grammar). We have removed $A \multimap B$, now encoded as $A^{\perp} \parr B$; some classical formulae are not expressible in IMLL, such as $\perp$, or $x \parr x$. The classical formulation of our above example is $((a^{\perp} \parr I) \otimes \perp) \parr I$.

Then, a *switching* of a classical formula is its abstract syntax tree, minus exactly one argument edge of each $\parr$. A *switching* of a linking $A \xrightarrow{f} B$ is a graph obtained by glueing (in the sense of pushouts in the category of undirected graphs) along ports the (undirected) wires of f with switchings of $A^{\perp}$ and B. The linking then satisfies DR iff all its switchings are acyclic and connected. On our above examples, sample switchings are depicted in Fig. 1.

Correct linkings compose by glueing along ports in the middle formula, which yields a category $S_0(X)$. For example, pre and postcomposing the structural isomorphism ρ with its obvious candidate inverse yields:



(1)

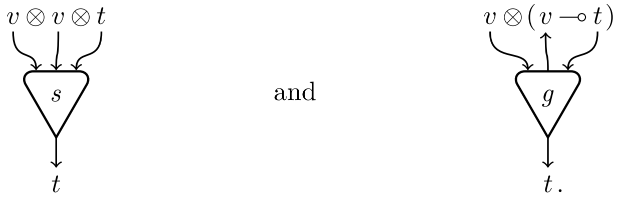
The former is not an identity. And indeed, correct linkings do not form an SMC category. Instead, they form the free *split* SMC category over X [19]. A split SMC category is like an SMC category, where λ and ρ are only required to have left inverses, as exemplified with ρ in (1).

Here is the final step: let a *rewiring* of some correct linking f be any linking obtained by changing the target of exactly one wire from some occurrence of I in f, without breaking correctness. A first example is that the left-hand morphism of (1) rewires to the identity; a hopefully more intuitive example is in Section 3.1. Then $S(X)$ is the result of quotienting $S_0(X)$ by the equivalence relation generated by rewiring.

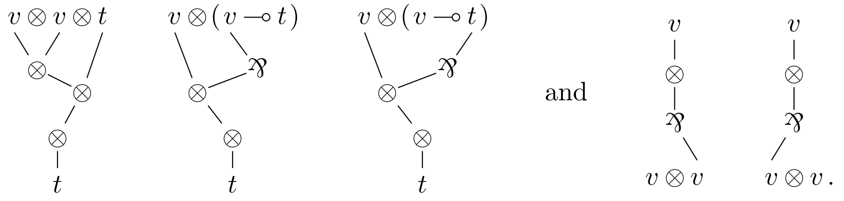
2.3 The Free Symmetric Monoidal Closed Category over a Signature

We now extend S to SMC signatures Σ: we have a set of sorts X, plus a set of operations. We enrich linkings with, for each operation $A \xrightarrow{c} B$, a formal morphism, pictured by a *cell*, in the style of *interaction nets* [21].

Example 2. The π-calculus send and get operations yield cells



We then extend linkings $A \longrightarrow B$ to include cells in a suitable way — a glance at Fig. 2 might help. We consider linking equivalent modulo the choice of support, i.e., the choice of cells. Linkings compose as before. The question is then: what is a switching in the extended setting? The answer is that taking a switching of a cell $A \xrightarrow{c} B$ is replacing the cell with a switching of $A \otimes B^{\perp}$. For example, consider the send and get operations, and a *contraction* operation $v \xrightarrow{c} v \otimes v$. Their respective switchings are:



To understand why this is right, observe that SMC categories have a *functional completeness* property, in the sense of Lambek and Scott [22]. Roughly, this means that any morphism $C \longrightarrow D$ using a cell $A \xrightarrow{c} B$ may be parameterised over it, i.e., be decomposed as

$$C \xrightarrow{\cong} I \otimes C \xrightarrow{\ulcorner c \urcorner \otimes C} (A \multimap B) \otimes C \xrightarrow{f} D, \qquad (2)$$

where $\ulcorner c \urcorner$ is the currying of c. An example such decomposition is pictured (left) in Fig. 4 below. This rightly suggests that an operation $A \xrightarrow{c} B$ should have the same switchings as $A \multimap B$ in the domain, i.e., $A \otimes B^{\perp}$. Quotienting under rewiring as before yields the expected functor S:

Theorem 1. *The functors S and U yield a monadic adjunction*

$$\begin{tikzcd} \mathsf{SMCSig} \arrow[r, shift left, "S"] \arrow[r, phantom, "\perp"] & \mathsf{SMCCat}. \arrow[l, shift left, "U"] \end{tikzcd}$$

2.4 The Free Symmetric Monoidal Closed Category over a Theory

That gives the construction for signatures. We now extend it to SMC theories: define a theory $\mathcal{T}$ to be given by a signature Σ, together with a set $E_{A,B}$ of equations between morphisms in $S(\Sigma)(A,B)$, for all A, B. The free SMC category $S(\mathcal{T})$ generated by such a theory is then the quotient of $S(\Sigma)$ by the equations. Constructing $S(\mathcal{T})$ graphically is more direct than could have been feared: we first define the binary predicate $f_1 \sim f_2$ relating two morphisms $C \overset{f_1,f_2}{\rightrightarrows} D$ in $S(\Sigma)$ as soon as each f_i decomposes (remember (2) and the left-hand part of Fig. 4) as

$$C \xrightarrow{\cong} I \otimes C \xrightarrow{\ulcorner g_i \urcorner \otimes C} (A \multimap B) \otimes C \xrightarrow{f} D$$

with a common f, with $(g_1, g_2) \in E_{A,B}$. The smallest equivalence relation by this relation $\sim$ is stable under composition, and we define $S(\mathcal{T})$ by quotienting $S(\Sigma)$ under it.

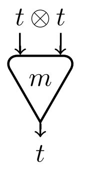
Finally, $S(\mathcal{T})$ is initial in the following sense. Let the category of *representations* of $\mathcal{T}$ be the full subcategory of the comma category $\Sigma{\downarrow}U$ whose objects are the morphisms $\Sigma \longrightarrow U(\mathcal{C})$, for which $\mathcal{C}$ is an SMC category satisfying the equations in E. Now consider the morphism $\Sigma \xrightarrow{\eta} US(\Sigma) \xrightarrow{q} US(\mathcal{T})$, where q is the quotient by the equations in E.

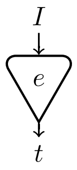
Theorem 2. *This morphism is initial in the category of representations of $\mathcal{T}$.*

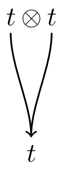
2.5 Commutative Monoid Objects

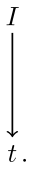
We finally slightly tune the above construction to better handle the special case of commutative monoids. In a given theory $\mathcal{T} = (\Sigma, E)$, assume that a sort t is equipped with two operations $t \otimes t \xrightarrow{m} t$ and $I \xrightarrow{e} t$, with equations making it into a commutative monoid (m is associative and commutative, e is its unit). Further assume that m and e do not occur in other equations. In this case, we sketch (for lack of space) an alternative, more economic description of morphisms in $S(\mathcal{T})$.

Start from the original definition, relax the bijection condition on linkings, i.e., allow them to map negative t ports to positive t ports non-bijectively, and then replace m and e as follows:



and


become


and


For a commutative comonoid (c, w), the dual trick does not work so easily, because of problems with weakening (weakening has an output I port, which cannot be left unattached, as opposed to the input I port of e). But still, a non-empty tree of c's may be represented by several arrows leaving its root. Observe that while m has as only switching the complete graph, c has two switchings (the formula is $v \otimes (v^{\perp} \parr v^{\perp})$).

2.6 Modularity and Sequentialisation as Induction

Melliès [24] convincingly explains the need for *modular* models of programming languages and calculi. In a slightly different sense, we argue that SMC categories provide a modular model of syntax. Namely, we obtain, for any theory $\mathcal{T}$:

Proposition 1. *For any (representative of a) proof net $A \xrightarrow{f} B$ in $S(\mathcal{T})$ with a set C of cells, and any partition of C into C_1 and C_2, f decomposes as $f_2 \circ f_1$, where each f_i contains exactly the cells in C_i.*

The proof is by inductively applying the decomposition (2). Intuitively, Proposition 1 says that, thinking of operations in Σ as atomic building blocks, each morphism may be obtained by plugging such blocks together by composition (see the left-hand part of Fig. 4). In a sense, this is an induction principle (a morphism only has finitely many cells). But it does not prevent wires to have a complex structure in the obtained components. We thus need a more powerful induction principle.

Let us fix a theory $\mathcal{T} = (\Sigma, E)$ for the rest of this section. A first, general induction principle on $S(\mathcal{T})$ is given by sequentialisation, in the sense of Girard [13, 6], as follows. Pick your preferred syntactic presentation of IMLL, e.g., the original sequent calculus [13] for concreteness. There is a well-known mapping from axiom-free proofs to proof nets, say $p \mapsto [p]$. Observe that by understanding the operations $A \xrightarrow{c} B$ in Σ as axioms $c \colon A \vdash B$, this mapping extends to proofs with axioms in Σ, by mapping any such c to a corresponding cell. We then have:

Theorem 3. *Any morphism $A \xrightarrow{f} B$ in $S(\mathcal{T})$ has an antecedent proof p in* IMLL *plus axioms in Σ, such that $[p] = f$.*

This provides an easy induction scheme over the morphisms of $S(\mathcal{T})$. However, this scheme has deficiencies: for example, the antecedent proofs it provides do not have to be cut-free; moreover, morphisms may not be decomposed downwards as with standard induction schemes. For an example of the latter, assume you have a λ-term starting with a λ-abstraction; the induction scheme might reveal this only after a few decompositions.

3 First Examples

In this section, we explain how to build the λ-calculus in stages, starting from the linear λ-calculus, and passing through a kind of λ-calculus with sharing of terms.

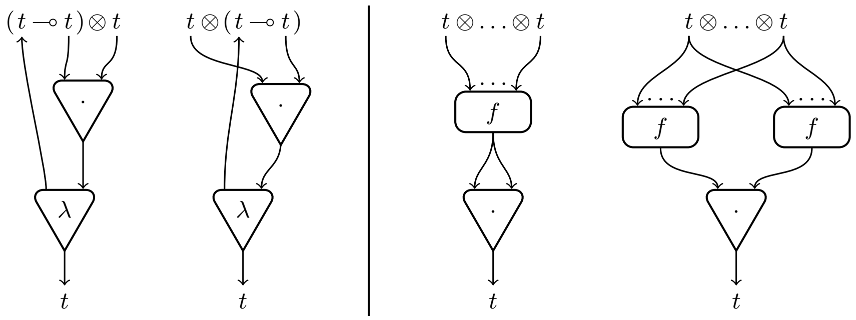

$(t \multimap t) \otimes t$
$t \otimes (t \multimap t)$
$t \otimes \ldots \otimes t$
$t \otimes \ldots \otimes t$
$\cdot$
λ
f
t

Fig. 2. Examples: linearity and sharing

We briefly discuss induction principles in this particular case. We then proceed with a π-calculus example, which we will use as our main example in Section 4. We end by a few example uses of higher order and modularity (Proposition 1), notably related to reduction and labelled transition rules.

3.1 Lambda-Calculus, Linearity, Induction

We start with the easiest application: the untyped λ-calculus. If we naively mimick HOAS to guess a signature for the λ-calculus, we obtain one sort t and operations $t \otimes t \xrightarrow{\cdot} t$ and $(t \multimap t) \xrightarrow{\lambda} t$. However, the free SMC category on this signature is the *linear* λ-calculus, as shown by the following standard result:

Proposition 2. *Morphisms $I \longrightarrow t$ are in bijection with closed linear λ-terms.*

Composition in our category is like *context* application in λ-calculus. A context is a term with (possibly several, numbered) holes, and context application is replacement of the hole with a term (or another context), possibly capturing some variables. The correspondence is tedious to formalise though, because contexts do not have enough information. For example, consider the context $\lambda x.(\Box_0 \cdot \Box_1)$ with two holes $\Box_0$ and $\Box_1$. Exactly one of $\Box_0$ and $\Box_1$ may use x, but this information is not contained in the context, which makes context application partial. In our setting, each possibility corresponds to one of the two morphisms on the left of Fig. 2.

A first attempt to recover the full λ-calculus is to add a contraction and a weakening $t \xrightarrow{c} t \otimes t$ and $t \xrightarrow{w} I$ to our signature, with the equations making (c, w) into a commutative comonoid. The free SMC category on this theory is close to Wadsworth's λ-graphs [33], which are a kind of λ-terms with a fine representation of sharing. For example, the two morphisms on the right of Fig. 2 are different, because contraction is not natural.

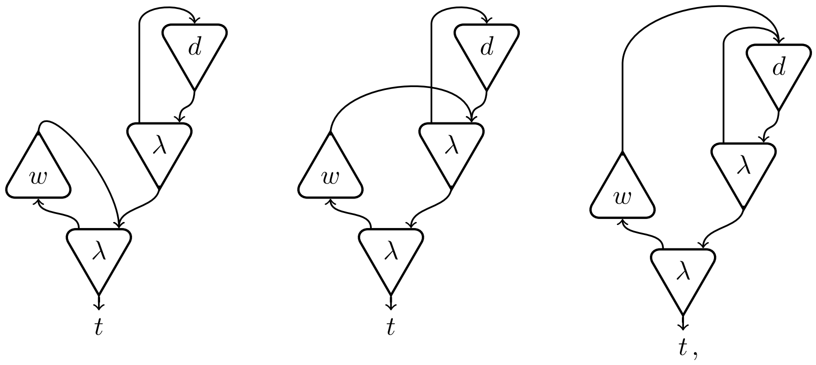
To obtain the standard λ-calculus without sharing, we now consider two sorts: a sort t for terms, and a sort v for variables, an idea that has been explored independently in *weak* HOAS and tile logic [8, 17, 18, 2]. The theory then contains:

$$t \otimes t \xrightarrow{\cdot} t \qquad (v \multimap t) \xrightarrow{\lambda} t \qquad v \xrightarrow{c} v \otimes v \qquad v \xrightarrow{w} I \qquad v \xrightarrow{d} t,$$

where the latter is instantiation of a variable as a term, plus the equations making (c, w) into a commutative comonoid. We obtain:

Proposition 3. *Morphisms $I \longrightarrow t$ are in bijection with closed λ-terms. Morphisms not using c nor w are in bijection with closed linear λ-terms.*

The commutative comonoid structure on v, and Trimble rewiring are crucial to this result. Intuitively, the latter allows a weakened variable to be indifferently linked anywhere under its scope. For example, the term $\lambda x.\lambda y.y$ has linkings



which all equivalent under Trimble rewiring.

Beyond Proposition 3, we also may recover open terms as follows. Mimicking the standard construction of a monad from an operad using coends [23], for any set X, let $T(X)$ contain triples of a natural number n, a function from the ordinal n to X, and a morphism $t^{\otimes n} \longrightarrow t$, where $t^{\otimes 0} = I$ and $t^{\otimes n+1} = t^{\otimes n} \otimes t$. Two such triples (n, u, f) and (n, v, g) are considered equivalent when there is a permutation $\sigma\colon n \longrightarrow n$ such that $g = f \circ \sigma$ and $v = u \circ \sigma$ (finite ordinals and permutations form a subcategory of $S(\mathcal{T})$ through the embedding $n \mapsto t^{\otimes n}$).

Theorem 4. *The function T extends to a monad on* Set*, isomorphic to the monad sending each set X to the set of λ-terms with variables in X modulo α-conversion.*

Multiplication for this monad, i.e., substitution, works as follows: for any triple of a number n, a morphism $t^{\otimes n} \xrightarrow{f} t$, and a function $n \xrightarrow{u} T(X)$, let each $u_i = (n_i, v_i, f_i)$, and $m = \Sigma n_i$. Multiplication maps (n, u, f) to m, the morphism

$$t^{\otimes \Sigma n_i} \xrightarrow{\bigotimes f_i} t^{\otimes n} \xrightarrow{f} t$$

and the coproduct function $\Sigma n_i \xrightarrow{[v_0, \ldots, v_{n-1}]} X$.

We may derive from Theorem 4 an analogue of the standard induction scheme on λ-terms. We also expect to derive it directly, but for now defer a full treatment for further work. Also, it seems worth investigating sufficient conditions on the signature for such a general induction scheme to be derived. For instance, it is not at all obvious which induction scheme should be derived from the signature we choose in the next section for the π-calculus.

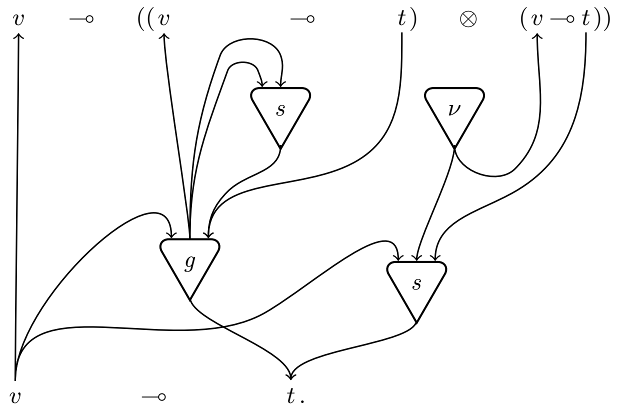


Fig. 3. A π-calculus example

3.2 Pi-Calculus Example

A reasonable theory $\mathcal{T}$ for the π-calculus could have at least the operations s and g specified above, plus commutative comonoid structure (c, w) on v, plus commutative monoid structure $(|, \mathbf{0})$ on t. Consider furthermore a name restriction operation $I \xrightarrow{\nu} v$, with the equation $w \circ \nu = \mathrm{id}_I$. We do not claim that this theory $\mathcal{T}$ is the right one for the π-calculus, but it is relevant for bigraphs. (An alternative type for ν is $(v \multimap t) \longrightarrow t$ [2, 18].)

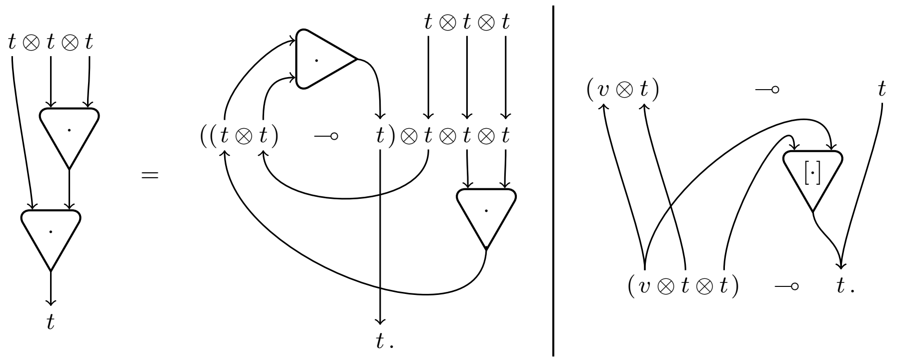
Consider the π-calculus term with holes $(a(x).(\square_0 \mid \bar{x}\langle x\rangle)) \mid \nu b.(\bar{a}\langle b\rangle.\square_1)$. This term may have many different interpretations as a morphism in $S(\mathcal{T})$. A first possibility is depicted in Fig. 3. Recall: several arrows leaving a v port mean a tree of contractions; several arrows entering a t port mean a tree of parallel compositions; a positive t port with no input arrow means a 0.

The holes $\square_0$ and $\square_1$ are represented by the occurrences of t in the domain formula, in order. The free variable a of the term is represented by the occurrence of v in the codomain. It is used three times: twice following the term, and once more for transmitting it to $\square_0$ and $\square_1$.

But the language of SMC categories allows additional flexibility w.r.t. syntax. For example, we could choose to impose that $\square_0$ and $\square_1$ may not use a. That would mean changing the domain for $(v \multimap t) \otimes (v \multimap t)$, and removing the leftmost wire. Or, we could, e.g., only allow $\square_0$ to use a, and not $\square_1$. That would only mean change the domain to $((v \otimes v) \multimap t) \otimes (v \multimap t)$ (the leaves do not change, so the wires may remain the same).

3.3 Higher Order and Modularity

We now give an example decomposition as in Proposition 1, in the λ-calculus. Consider the context with numbered holes $(\square_0 \cdot \square_1) \cdot \square_2$. The decomposition obtained by Proposition 1 for C_1 containing exactly the outermost application is depicted left in Fig. 4. Such a decomposition is not possible in bigraphs, mainly because it makes use of a higher-order formula, namely $(t \otimes t) \multimap t$.



$t \otimes t \otimes t$
$\cdot$
$\cdot$
t
$=$
$t \otimes t \otimes t$
$\cdot$
$((t \otimes t) \multimap t) \otimes t \otimes t \otimes t$
$\cdot$
$t\,.$
$(v \otimes t) \multimap t$
$[\cdot]$
$(v \otimes t \otimes t) \multimap t\,.$

Fig. 4. Examples: modularity and higher order

A possible use of such decompositions and higher order is in specifying reduction rules parametrically, as opposed to ground reduction rules. For example, the π-calculus rule $a(x).\,\square_0\,|\bar{a}\langle x\rangle.\,\square_1 \longrightarrow \square_0\,|\square_1$ may be represented as a rule between morphisms looking like Fig. 3, which we omit for lack of space. Another possible use is in specifying transitions using second-order contexts, in the style of [28, 12]. An example, using Cardelli and Gordon's Mobile Ambients [3], is the transition rule $\mathtt{in}\,a.P \xrightarrow{\lambda Q.(a[Q]\mid\square)} a[P|Q]$, whereby the process $\mathtt{in}\,a.P$, in the presence of a process of the shape $a[Q]$, migrates inside the location a, to yield $a[P|Q]$. A possible representation using SMC theories would take as states of the labelled transition system morphisms $I \xrightarrow{f} A$ in the free SMC category generated by the obvious SMC theory for Mobile Ambients, with as transitions $f \xrightarrow{\ell} g$ certain morphisms $A \xrightarrow{\ell} B$ such that $\ell \circ f = g$. In the above example, $\mathtt{in}\,a.P$ is represented a morphism $I \longrightarrow (v \otimes t) \multimap t$ obtained by currying the operation $v \otimes t \xrightarrow{\mathtt{in}} t$ from the signature, and the label is the morphism depicted right in Fig. 4.

4 Binding Bigraphs

In this section, we consider (abstract binding) bigraphs [20]. They are a framework for reasoning about distributed and concurrent programming languages, designed to encompass both the π-calculus [26] and the Mobile Ambients calculus [3]. We are here only concerned with bigraphical syntax: any so-called *bigraphical signature* $\mathcal{K}$ generates a *pre-category*, and then a category $M(\mathcal{K})$, whose objects are *bigraphical interfaces*, and whose morphisms are bigraphs.

Its main features are (i) the presence of *relative pushouts* (RPOs) in the pre-category, which makes it well-behaved w.r.t. bisimulations, and that (ii) in both the pre-category and the category, the so-called *structural* equations become equalities. Also, bigraphs follow a scoping discipline ensuring that, roughly, bound variables are only used below their binder.

We now recall bigraphs and sketch our interpretation in terms of SMC theories, which we compare to the original (see our preprint [16] for a more technical account).

4.1 Bigraphs

We work with a slightly twisted definition of bigraphs, in two respects. First, we restrict Jensen and Milner's *scope* rule by adding a *binding* rule to be respected by bigraphs. This rule rectifies a deficiency of the scope rule, which prevented bigraphs to be stable under composition in the original paper [20][2]. It was added in later work [25]. Our second twist is to take names in a fixed, infinite, and totally ordered set, say $\mathcal{X}$. This helps relating our approach with the original.

A *bigraphical signature* is a set of operations, or *controls* $k \in \mathcal{K}$, with arity given by a pair of natural numbers $a_k = (B_k, F_k) = (n, m)$, where $B_k = n$ is the number of *binding* ports of k, $F_k = m$ being its number of *free* ports. Additionally, a signature specifies a set $\mathcal{A} \subseteq \mathcal{K}$ of *atomic* controls, whose binding arity has to be 0.

Typically, send and get have arities: $a_s = (0, 2)$ and $a_g = (1, 1)$. They are not atomic (send would be atomic in the asynchronous π-calculus). The other operations of the π-calculus are all kind of built into bigraphical structure, as we will see shortly.

Bigraphs form a category, whose objects are *interfaces*. An interface is a triple $U = (n, X, \ell)$, where n is a natural number, $X \subseteq \mathcal{X}$ is a finite set of names, and $X \xrightarrow{\ell} n + \{\perp\}$ is a *locality* map (n is identified with the set $\{0, \ldots, n-1\}$, i.e., the ordinal n). Names x with $\ell(x) = i \in n$ are *located* at i; others are *global*.

Introducing the morphisms, i.e., bigraphs, themselves seems easier by example. We thus continue with an example bigraph in Fig. 5, which will correspond to the proof net in Fig. 3. The codomain of this bigraph, which is graphically its upper, outer face, is $W = (1, \{a\}, \{a \mapsto \perp\})$: the element $0 \in 1$ represents the (only) outer box, which we accordingly marked 0. The global name a is the common end of the group of three wires reaching the top side of the box.

The domain of our example bigraph, which is graphically its inner face when the grey parts are thought of as holes (plus a'), is $U = (2, \{a', x, b\}, \{x \mapsto 0, b \mapsto 1, a' \mapsto \perp\})$. Comparing this to the domain of our morphism in Fig. 3, we observe that the elements 0 and 1 of 2 correspond to $\square_0$ and $\square_1$. Furthermore, the name a' being global corresponds to the domain $v \multimap ((v \multimap t) \otimes (v \multimap t))$ of Fig. 3 having both t's *under the scope* of the first v (i.e., there is an implication with the t's on its right, v on its left, and no other implication on the paths from it to them). Finally, the locality map sending x to 0 corresponds to the second v having only the first t under its scope, and similarly for b being sent to 1.

The morphism itself is a compound of two graphical structures. The first structure, the *place* graph, is a forest (here a tree), whose leaves are the inner

[2] Being peers only involves inner names or ports by definition, not outer names. Thus, binding ports may be linked to them. It is then easy to show that the scope rule is not stable under composition. The binding rule [25] is the straightforward fix.

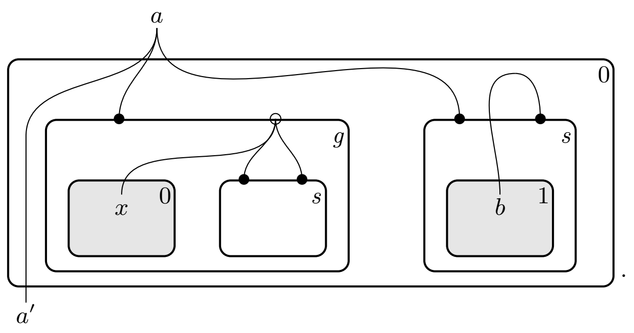


Fig. 5. An example bigraph

0 and 1, the *sites*, and whose root is the outer 0. (Atomic controls would have to be leaves.) Following Milner and Jensen, we represent nodes by regions in the plane, the parent of a region being the immediately enclosing region. The second structure, the *link* graph, is a bit more complicated to formalise. First, each internal (i.e., non leaf, non root) node v is labelled with an operation $k_v \in \mathcal{K}$. We then compute the set of *ports* P: it is the set of pairs (v, i), where v is a node, and $i \in B_{k_v} + F_{k_v}$ is in either component of the arity of v. The link graph is then a function $P + X \xrightarrow{link} E + Y$, where $X = \{a', x, b\}$ is the set of *inner* names, $Y = \{a\}$ is the set of *outer* names, and E is the set of *edges*. In our morphism, a' and both occurrences of a (i.e., the black dots connected to a in the picture) are mapped to the outer name a by the *link* map. Furthermore, E is a two-element set, say $\{x', b'\}$. The edge x' acts as a link from the name x received by the get node g to its three occurrences (the three dots connected to it in the picture). Formally, the three involved ports and the name x are all sent to x' by the *link* map. The edge b' represents the νb in the term; formally, both b and the involved port of the right-hand s node are sent to b' by the *link* map.

Until now, there is not much difference between the edge representing the bound name x received on a and the bound name b created by νb. The difference comes in when we check the *scope* and *binding* rules. The binding rule requires that each binding port (such as the one marked with a circle in Fig. 5) be sent to an edge, as opposed to a name in the codomain. The scope rule further requires that its *peers*, i.e., the ports and names connected to the same edge, lie strictly below it in the place graph. For ports, this should be clear. For inner names, this means that they should be located at some site below it. In our example, the inner 0 node indeed lies below the get node, for instance. This all ensures that bound names are only used below their binder.

Remark 1. An edge is connected to at most one binding port, by acyclicity of the place graph. An edge connected to one binding port is called *bound.*

Composition $g \circ f$ in the category of bigraphs $M(\mathcal{K})$ is by plugging the outer boxes of f into the inner boxes of g, in order, and connecting names straightforwardly. This only works if we quotient out bigraphs by the natural notion of isomorphism, i.e., modulo choice of nodes and edges. We actually consider a further quotient:

removing an edge from E which was outside the image of *link*. The whole is called *lean support* equivalence by Jensen and Milner.

4.2 Bigraphs as Symmetric Monoidal Closed Theories

We now describe our intepretation for bigraphs, starting with signatures. Consider any signature $(\mathcal{K}, B, F, \mathcal{A})$. We translate it into the following SMC signature $\mathcal{T}_{\mathcal{K}}$, which has two sorts $\{t, v\}$, standing for terms and variables (or names), and whose operations consist of *structural* operations and equations, plus *logical* operations. The *structural* part, accounting for the built-in structure of bigraphs, is as in Section 3.2, i.e., it consists of: a commutative monoid structure $(|, \mathbf{0})$ on t, a commutative comonoid structure (c, w) on v, and a name restriction $I \xrightarrow{\nu} v$, such that $w \circ \nu = \mathrm{id}_I$.

The *logical* part consists, for each control $k \in \mathcal{K}$ with $a_k = (n, m)$, of an operation $v^{\otimes m} \xrightarrow{k} t$ if k is atomic (and $n = 0$), and $(v^{\otimes n} \multimap t) \otimes v^{\otimes m} \xrightarrow{k} t$ otherwise. For example, recall send and get, defined above to have arities $(1, 1)$ and $(0, 2)$, this gives (up to isomorphism) the operations $(v \otimes v \otimes t) \xrightarrow{s} t$ and $(v \otimes (v \multimap t)) \xrightarrow{g} t$ from Section 2.3. An asynchronous send operation in the style of the asynchronous π-calculus, would have bigraphical arity $(0, 2)$, which would be translated into $v \otimes v \xrightarrow{s'} t$ because of atomicity.

Now, on objects, we define our functor T by:

$$\mathsf{T}(n, X, \ell) = v^{\otimes n_g} \multimap \bigotimes_{i \in n} (v^{\otimes n_i} \multimap t), \tag{3}$$

where $n_g = |\ell^{-1}(\perp)|$ and for all $i \in n$, $n_i = |\ell^{-1}(i)|$. The ordering on $\mathcal{X}$ induces a bijection between X and v leaves in the formula, which the translation of morphisms exploits. On our main example, this indeed maps the domain and codomain of Fig. 5 to those of Fig. 3.

We will here only describe the translation of morphisms on Fig. 5, for readability. The full translation is available in the companion preprint [16]. Starting from Fig. 5, a first step is to represent the place graph more traditionally, i.e., as usual with trees. But in order to avoid confusion between the place and link graphs, we represent each node as a cell, and adopt the convention that edges from the place graph relate a principal port (i.e., the vertex of a cell) to a rightmost auxiliary port (i.e., a rightmost point in the opposed segment). Wires from the link graph thus leave from other auxiliary ports.

Finally, edges in E in the bigraph are pointed to by ports and inner names. We now represent them as (nullary) ν cells with pointers to their principal port. We obtain the hybrid picture in Fig. 6, where we have drawn the connectives to emphasise the relationship with Fig. 3. And indeed we have almost obtained the desired proof net. A first small problem is the direction of wires in the link graph which, intuitively, go from occurrences of names to their creator (be it a ν or an outer name). So we start by reversing the flow of the link graph (implicitly introducing trees of contractions).

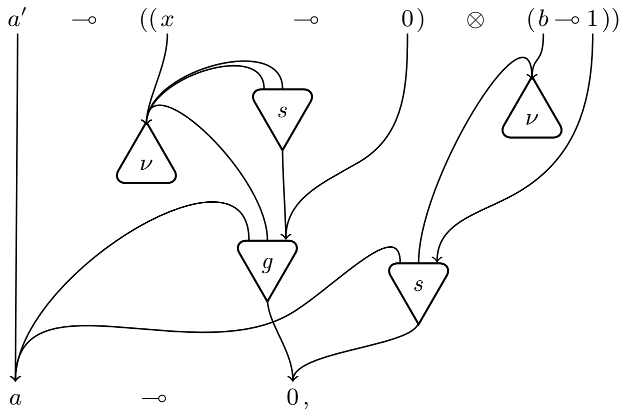


Fig. 6. A hybrid picture between bigraphs and proof nets

This does not completely correct the mismatch, however, because in the case of bound edges like x' in our example bigraph, the ν cell is absent from Fig. 3. But by Remark 1, the name in question has a unique binding occurrence, and the ν cell may be understood as an indirection between this binding occurrence and the others. Contracting this indirection (and fixing the orientation accordingly) yields exactly the desired proof net in Fig. 3.

The procedure sketched on our example generalises, up to some subtleties with unused names (where should the weakenings point to?), and we have

Theorem 5. *The function* T *extends to a functor* $M(\mathcal{K}) \xrightarrow{\mathsf{T}} S(\mathcal{T}_{\mathcal{K}})$*, faithful, essentially injective on objects, and neither full nor surjective on objects.*

The functor is not strictly injective on objects, because any two interfaces equal up to their (ordered) choice of names have the same image. A counterexample to fullness is the canonical morphism $(v \multimap I \multimap t) \longrightarrow (I \multimap v \multimap t)$: it amounts to making a global variable local, which is forbidden in bigraphs.

Despite non-fullness, the overall scoping discipline of bigraphs is maintained, in the sense that T is full on whole programs, i.e., bigraphs with neither sites nor names in their interfaces. More generally, it is full on *ground* bigraphs, i.e., bigraphs in $M(\mathcal{K})((\emptyset, 0, \emptyset), U)$, for some interface U:

Theorem 6. *For any such* U*, we have* $S(\mathcal{T}_{\mathcal{K}})(I, \mathsf{T}(U)) \cong M(\mathcal{K})((\emptyset, 0, \emptyset), U)$.

So, $S(\mathcal{T}_{\mathcal{K}})$ has as many whole programs as $M(\mathcal{K})$, but more program fragments.

5 Conclusions

Related work. Various flavours of closed categories have long been known to be closely related to particular calculi with variable binding [22, 1]. As mentioned in the introduction, our approach may be considered as an update and further investigation of Coccia et al. [4]. Also, the relation between our approach and Tanaka's work on variable binding in a linear setting [31] remains unclear to us.

A number of papers have been devoted to better understanding (various kinds of) bigraphs, be it as sortings [7], as cospans over graphs [30], through directed bigraphs [14], or as a language with variable binding [5]. We appear to be the first to reconcile a full treatment of scope (Theorem 6) with initial algebra semantics.

Future work. We should further investigate induction principles in our setting (see Section 3.1). We should also try to use our approach in an actual implementation.

On the bigraphical side, it might be useful to understand the scope rule induced by our functor T in bigraphical terms. Also, we should study RPOs in our approach, possibly by investigating (any form of) *concrete* bigraphs [20, 29].

Another natural research direction from this paper concerns the dynamics of bigraphs. Our hope is that Bruni et al.'s [2] very modular approach to dynamics may be revived, and work better with SMC structure than with cartesian closed structure. Specifically, with SMC structure, there is no duplication at the static level, which might simplify matters.

References

[1] Barber, A., Gardner, P., Hasegawa, M., Plotkin, G.: From action calculi to linear logic. In: Nielsen, M. (ed.) CSL 1997. LNCS, vol. 1414. Springer, Heidelberg (1998)

[2] Bruni, R., Montanari, U.: Cartesian closed double categories, their lambda-notation, and the pi-calculus. In: LICS 1999. IEEE Computer Society, Los Alamitos (1999)

[3] Cardelli, L., Gordon, A.: Mobile ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, p. 140. Springer, Heidelberg (1998)

[4] Coccia, M., Gadducci, F., Montanari, U.: GS·Λ theories: A syntax for higher-order graphs. In: CTCS 2002. ENTCS, vol. 69. Elsevier, Amsterdam (2003)

[5] Damgaard, T., Birkedal, L.: Axiomatizing binding bigraphs. Nordic Journal of Computing 13(1-2) (2006)

[6] Danos, V., Regnier, L.: The structure of multiplicatives. Archive for Mathematical Logic 28 (1989)

[7] Debois, S.: Sortings & bigraphs. PhD thesis, IT University of Copenhagen (2008)

[8] Despeyroux, J., Felty, A., Hirschowitz, A.: Higher-order abstract syntax in Coq. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 124–138. Springer, Heidelberg (1995)

[9] Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: LICS 1999. IEEE Computer Society, Los Alamitos (1999)

[10] Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax involving binders. In: LICS 1999. IEEE Computer Society, Los Alamitos (1999)

[11] Garner, R.H.G., Hirschowitz, T., Pardon, A.: Graphical presentations of symmetric monoidal closed theories. CoRR, abs/0810.4420 (2008)

[12] Di Gianantonio, P., Honsell, F., Lenisa, M.: RPO, second-order contexts, and λ-calculus. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 334–349. Springer, Heidelberg (2008)

[13] Girard, J.-Y.: Linear logic. Theoretical Comput. Sci. 50 (1987)

[14] Grohmann, D., Miculan, M.: Directed bigraphs. ENTCS 173 (2007)

[15] Hirschowitz, A., Maggesi, M.: Modules over monads and linearity. In: Leivant, D., de Queiroz, R. (eds.) WoLLIC 2007. LNCS, vol. 4576, pp. 218–237. Springer, Heidelberg (2007)

[16] Hirschowitz, T., Pardon, A.: Binding bigraphs as symmetric monoidal closed theories. CoRR, abs/0810.4419 (2008)

[17] Hofmann, M.: Semantical analysis of higher-order abstract syntax. In: LICS 1999. IEEE Computer Society, Los Alamitos (1999)

[18] Honsell, F., Miculan, M., Scagnetto, I.: Pi-calculus in (co)inductive-type theory. Theor. Comput. Sci. 253(2) (2001)

[19] Hughes, D.J.D.: Simple free star-autonomous categories and full coherence. ArXiv Mathematics e-prints, math/0506521 (June 2005)

[20] Jensen, O.H., Milner, R.: Bigraphs and mobile processes (revised). Technical Report TR580, University of Cambridge (2004)

[21] Lafont, Y.: Interaction nets. In: POPL. ACM, New York (1990)

[22] Lambek, J., Scott, P.: Introduction to Higher-Order Categorical Logic. Cambridge University Press, Cambridge (1986)

[23] Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Graduate Texts in Mathematics, vol. 5. Springer, Heidelberg (1998)

[24] Melliès, P.-A.: Double categories: a modular model of multiplicative linear logic. Mathematical Structures in Computer Science 12 (2002)

[25] Milner, R.: Bigraphs whose names have multiple locality. Technical Report TR603, University of Cambridge (2004)

[26] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Information and Computation 100(1) (1992)

[27] Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: PLDI 1988. ACM, New York (1988)

[28] Rathke, J., Sobocinski, P.: Deconstructing behavioural theories of mobility. In: TCS 2008. Springer, Heidelberg (2008)

[29] Sassone, V., Sobociński, P.: Deriving bisimulation congruences using 2-categories. Nordic Journal of Computing 10(2) (2003)

[30] Sassone, V., Sobociński, P.: Reactive systems over cospans. In: LICS 2005. IEEE Press, Los Alamitos (2005)

[31] Tanaka, M.: Abstract syntax and variable binding for linear binders. In: Nielsen, M., Rovan, B. (eds.) MFCS 2000. LNCS, vol. 1893, p. 670. Springer, Heidelberg (2000)

[32] Trimble, T.: Linear logic, bimodules, and full coherence for autonomous categories. PhD thesis, Rutgers University (1994)

[33] Wadsworth, C.: Semantics and pragmatics of the lambda calculus. PhD thesis, University of Oxford (1971)

Partial Order Reduction for Probabilistic Systems: A Revision for Distributed Schedulers*

Sergio Giro, Pedro R. D'Argenio, and Luis María Ferrer Fioriti

FaMAF, Universidad Nacional de Córdoba - CONICET
Ciudad Universitaria - 5000 Córdoba - Argentina
{sgiro,dargenio,ferrer}@famaf.unc.edu.ar

Abstract. The technique of partial order reduction (POR) for probabilistic model checking prunes the state space of the model so that a maximizing scheduler and a minimizing one persist in the reduced system. This technique extends Peled's original restrictions with a new one specially tailored to deal with probabilities. It has been argued that not all schedulers provide appropriate resolutions of nondeterminism and they yield overly safe answers on systems of distributed nature or that partially hide information. In this setting, maximum and minimum probabilities are obtained considering only the subset of so-called *distributed* or *partial information* schedulers. In this article we revise the technique of partial order reduction (POR) for LTL properties applied to probabilistic model checking. Our reduction ensures that *distributed* schedulers are preserved. We focus on two classes of distributed schedulers and show that Peled's restrictions are valid whenever schedulers use only local information. We show experimental results in which the elimination of the extra restriction leads to significant improvements.

1 Introduction

Markov decision processes (MDPs) are widely used in diverse fields ranging from ecology to computer science. They are useful to model and analyse systems in which both probabilistic and nondeterministic choices interact. Particularly, composition oriented versions of MDPs like probabilistic automata [21] or probabilistic modules [12] are aimed to model concurrent and distributed systems.

In the area of system verification, model checking stands out as a model analysis technique for MDPs [22,3]. Moreover, probabilistic model checkers have been developed, notably PRISM [20] and LiQuor [7]. Probabilistic model checking is a push-button technique to calculate maximum and minimum probability values of the satisfaction of a temporal formula in a given model. To obtain these values, the technique requires to universally quantify on all possible resolutions of the inherent nondeterminism of the MDP. The resolution of such nondeterminism is carried out by the so-called *schedulers* (called also adversaries or policies, see e.g. [22,3,21]). Schedulers transform MDPs into Markov chains by selecting one

* Supported by ANPCyT project PICT 26135 and CONICET project PIP 6391.

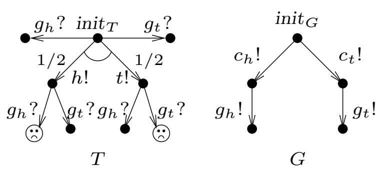


Fig. 1. T tosses a coin, G guesses heads or tails

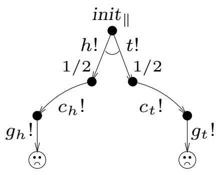


Fig. 2. A dubious scheduling

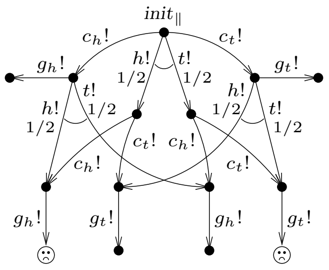


Fig. 3. The product of T and G

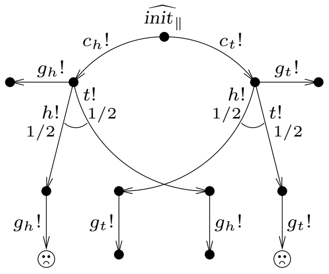


Fig. 4. A POR based reduction

of the enabled transitions at every step in the execution of the system. Therefore, the goal of probabilistic model checking is to find the maximum (or minimum) probability value of a formula over all possible schedulers. Since nondeterminism is an abstraction of deterministic choices in the implementation, concurrent behaviour, or even unknown probabilistic behaviour of the system, such maximum and minimum values are only safe bounds on the actual probability that the formula holds.

In several previous works, it has been argued that quantifying over all possible schedulers yields overly pessimistic bounds in case the components of the system do not share all information [11,12,6,5,13,15,14]. This can be seen in the following example. Consider two persons: one of them tosses a coin and the other one guesses heads or tails. This can be modelled as two processes, respectively depicted as T and G in Fig. 1. (Actions are written in an I/O fashion, with suffixes ? and ! representing input and output respectively.) Process T first tosses a coin that may land heads ($h!$) or tails ($t!$) with probability $\frac{1}{2}$ each. If, after tossing heads, T is notified of a guessed heads ($g_h?$), then T looses the game (represented with ☹); if instead it is notified of a guessed tails ($g_t?$), then T wins the game. The case in which T tossed tails is dual. In addition, if T is notified too early (before tossing the coin), the game is aborted. (Since we deal with input enabled systems, this modelling choice turns to be a convenient simplification.) Process G is simpler: it proceeds to guess by choosing heads ($c_h!$) or tails ($c_t!$) and then it notifies its guess by producing outputs $g_h!$ and $g_t!$, respectively. We take g_h and g_t to be the only two actions shared between T and G. In this way, G does not have any means to know the outcome of the coin toss, nor T to know the guess of G, until they synchronise on g_h or g_t. The composed system is depicted by the product automaton in Fig. 3. Notice, however, that an *almighty* scheduler may let G guess the correct answer with probability 1, as follows. It first lets T toss the coin and then it lets G choose transition $c_h!$ or $c_t!$ depending on whether T tossed heads (i.e. T output $h!$) or it tossed tails (i.e. T output $t!$), respectively. Such scheduler is depicted in Fig. 2. Therefore, by quantifying over all possible schedulers, the maximum probability of G guessing heads or tails (i.e. of reaching ☹) is 1, disregarding the fact that G has no information about the coin. This "total information" assumption underlies all probabilistic model checkers existing today. In conclusion, probabilistic model checkers produce safe upper and lower bounds for the actual probabilities, but they may result too conservative. Needless to say that, in this example, in which T and G do not share all information, we expect the maximum probability of guessing to be $\frac{1}{2}$.

The above observation is fundamental in distributed systems in which components share little information with each other, as well as in security protocols, where the possibility of information hiding is a fundamental assumption (a well-known example of these systems being the *dining cryptographers problem* [4]). The phenomenon we illustrated has been first observed in [21] from the point of view of compositionality, and has been studied on partial information settings in [11,12,5,13,15,14].

In order to limit the variety of behaviours introduced by arbitrary schedulers, classes of schedulers that only consider partial information were proposed in the literature. In particular, we are interested in the class of so-called *distributed schedulers*. Such schedulers were studied in [12] in a synchronous setting and in [5,14] in an asynchronous setting. A distributed scheduler is constructed from *local* schedulers, which are schedulers for single components of the system defined in the usual way, and a mechanism to combine them. Such mechanism could be related to a projection function [12], a passing token [6,5], execution rates of the components [15], or schedulers that define the way in which components interleave [14]. These approaches result in different types of distributed schedulers. We remark that the scheduler of Fig. 2 would not be a valid scheduler in this new setting since the choice for G depends only on information which is external to (and not observed by) G. In fact, a local scheduler for G yielding such a behaviour would not be definable, since the local scheduler depends only on the local history of G, which is certainly the same as long as G does not execute any transition.

As a consequence, a question arises of whether it is possible to do probabilistic model checking under a class of distributed schedulers, i.e., by universally quantifying on such subset of schedulers. Unfortunately, in [13] we showed that the bounds for the probability of properties cannot be calculated nor even approximated if the schedulers are restricted to be distributed. Actually, the framework in [13] is the same synchronous setting of [12]. We later extended these undecidability results to other classes of schedulers [16].

In this paper, we take advantage of the fact that not all schedulers are needed and revise the partial order technique for probabilistic model checking. Partial order reduction (POR) [18,9] is a well-known technique to cope with the state explosion problem. This technique exploits the structure of the product model naturally introduced when interleaving the components. The idea is to eliminate redundant states and transitions but keeping some *representative* ones. Such states and transitions are representative in the sense that, if a given path of the original system is relevant to the property being checked, then an equivalent path should be found in the reduced system. To ensure that representative states and transitions remain in the reduced model, the reduced model must comply with certain conditions [9].

POR for probabilistic model checking was simultaneously introduced in [2] and [10]. These works were oriented to model check LTL properties on MDPs and showed that the reduced model should meet one extra condition apart from those of the non-probabilistic case. This extra condition (call it **A5**) can be stated as follows: for all states in which a probabilistic transition can be reached without executing a transition from the ample set, the set of transitions enabled in the reduced model (called the ample set) is required to either have only one transition or coincide with the original enabled transitions. This technical and non-intuitive condition has been introduced precisely to *not* eliminate the behaviour introduced by schedulers like the one of Fig. 2.

In this paper we study POR techniques for two classes of distributed schedulers and show that condition **A5** can be relaxed for one class and eliminated for the other. Therefore, while the composed system of Fig. 3 is irreducible under condition **A5**, it can be reduced to the one of Fig. 4 in any of our settings.

The way in which we have conceived our technique further exploits the structure of product models and, in particular, the fact that not all information is shared. Moreover, it paves the way for the application of POR to probabilistic *symbolic* model checking. In fact, we are currently busy implementing the technique into PRISM. In this article, we report preliminary promising results of such implementation.

2 Interleaved Probabilistic Input/Output Automata

First, we briefly recall the standard framework of Markov Decision Processes (MDP), just to establish the terminology used in the paper. Later on, we present the framework of Interleaved Probabilistic Input/Output Automata (IPIOA) and show that they are a particular case of MDPs.

A Markov decision process consists of a set of states S, a set of transitions *Trans* and a function $\mathit{enabled} : \mathsf{S} \to \mathcal{P}(\mathit{Trans})$. Each $\alpha \in \mathit{Trans}$ is a function $\alpha : \mathsf{S} \times \mathsf{S} \to [0,1]$ such that $\sum_{s'} \alpha(s, s') = 1$ for all $s \in \mathsf{S}$, $\alpha \in \mathit{enabled}(s)$. A path is a sequence $\sigma = s_1.\alpha_1. \cdots .\alpha_{n-1}.s_n$ such that $\alpha_i \in \mathit{enabled}(s_i)$ and $\alpha_i(s_i, s_{i+1}) > 0$ for all $1 \leq i < n$. Paths can be finite or infinite. Let $\mathit{Paths}^{\mathit{fin}}$ denote the set of all finite paths. Given a finite path σ, the last state of σ is denoted by $\mathit{last}(\sigma)$, and the set of all infinite paths having σ as a prefix is denoted by $\sigma^{\uparrow}$.

The probability of a set of (infinite) paths depends on how the nondeterminism is resolved. So, we define the probability of a set of paths under a given *scheduler*. At every step in the execution of the system, the scheduler chooses one of the enabled transitions. Given a system and a scheduler, the probability of a set of paths is thus completely determined.

In the standard approach, the choice of the next transition is based on the complete history of the system. So, *arbitrary* schedulers are defined as functions mapping finite paths to transitions. A scheduler is then a function $\eta : \mathit{Paths}^{\mathit{fin}} \to \mathit{Trans}$ such that $\eta(\sigma) \in \mathit{enabled}(\mathit{last}(\sigma))$ for all $\sigma \in \mathit{Paths}^{\mathit{fin}}$. The set of all schedulers for an MDP P is denoted by $\mathit{Sched}(P)$.

For all finite path σ, the probability of the set $\sigma^{\uparrow}$ under scheduler η is $\prod_{i=1}^{n-1} \alpha_i(s_i, s_{i+1})$, where $\alpha_i = \eta(s_1.\alpha_1.\cdots.\alpha_{i-1}.s_i)$ for all i, $1 \leq i < n$. The probability of every set of infinite paths considered in this paper is uniquely defined by such probabilities since, in the standard way (namely, by resorting to the Carathéodory extension theorem), the probability of the cylinders $\sigma^{\uparrow}$ can be extended to the least σ-field containing all cylinders.

Next, we present a framework based on the Switched PIOA introduced by Cheung *et al.* [6]. It uses reactive and generative structures (see [17]). Generative transitions model both communication and state change. The component executing a generative transition chooses both a label a to output (e.g. in Fig. 1, h, t) and a new state s according to a given distribution. Reactive transitions specify how a component reacts to a given input. Since the input is not chosen, reactive transitions are simply distributions on states. For a finite set S, we denote by $\mathit{Prob}(S)$ the set of all discrete probability distributions over the set S. Given a set ActLab of action labels and a set S of states, the set of generative transitions $\mathsf{T_G}$ on $(\mathsf{S}, \mathsf{ActLab})$ is $\mathit{Prob}(\mathsf{S} \times \mathsf{ActLab})$, and the set $\mathsf{T_R}$ of reactive transitions is $\mathit{Prob}(\mathsf{S})$. A generative structure on $(\mathsf{S}, \mathsf{ActLab})$ is a function $\mathsf{G} : \mathsf{S} \to \mathcal{P}(\mathsf{T_G})$ and a reactive structure on $(\mathsf{S}, \mathsf{ActLab})$ is a function $\mathsf{R} : \mathsf{S} \times \mathsf{ActLab} \to \mathcal{P}(\mathsf{T_R})$. For $g \in \mathsf{T_G}$, the set of labels a such that $g(s, a) > 0$ for some s is denoted by $\mathsf{ActLab}(g)$. Given a state s, $\mathit{Dirac}(s)$ is the reactive transition r such that $r(s) = 1$.

In our framework, a system is obtained by composing several *probabilistic I/O atoms.*

Definition 1. *A probabilistic I/O atom is a 5-tuple* $(\mathsf{S}, \mathsf{ActLab}, \mathsf{G}, \mathsf{R}, \mathsf{init})$*, where* S *is a finite set of states,* ActLab *is a finite alphabet of actions labels, and* G *(*R*, resp.) is a generative (reactive, resp.) structure in* $(\mathsf{S}, \mathsf{ActLab})$. $\mathsf{init} \in \mathsf{S}$ *is the initial state. We require the atoms to be input-enabled, so* $\mathsf{R}(s, a) \neq \emptyset$ *for every* $s \in \mathsf{S}$*,* $a \in \mathsf{ActLab}$.

We often write S_i to denote the set of states of an atom A_i and similarly for the other elements of the 5-tuple. In addition, we write $\mathsf{T_G}_i$ ($\mathsf{T_R}_i$, resp.) for the set of generative (reactive, resp.) transitions on $(\mathsf{S}_i, \mathsf{ActLab}_i)$.

An interleaved probabilistic I/O system P is a set $\mathit{Atoms}(P)$ of probabilistic I/O atoms $A_1, \cdots, A_N$. The set of states of the system is $\prod_i \mathsf{S}_i$, and the initial state of the system is $\mathsf{init} = (\mathsf{init}_1, \cdots, \mathsf{init}_N)$.

In order to define how the system evolves, we define *compound transitions*, which are the transitions performed by the system as a whole. In such compound transitions, one of the atoms executes a generative transition, and the other atoms execute reactive transitions in case the label generated is in their alphabet. The generative transition g_i involved in this compound transition may output different labels (i.e. the set $\mathsf{ActLab}(g_i)$ may contain more than one action label).

Moreover, the same label may lead to different states, i.e. it is possible that $g_i(s_i, a) > 0$ and $g_i(s'_i, a) > 0$ for two different (local) states s_i and s'_i. Therefore, we have to ensure that each of this equally labelled outputs is matched by the same reactive transition in every atom capable of observing such a label. To this end, we introduce *input choice functions* f_j. An input choice function f_j for g_i maps each label in $\mathsf{ActLab}_j \cap \mathsf{ActLab}(g_i)$ to a reactive transition. The intended meaning is that $f_j(a)$ is executed by atom A_j in case label a is output by g_i. Formally, a compound transition is a tuple $\alpha = (g_i, f_1, \cdots, f_{i-1}, f_{i+1}, \cdots, f_N)$ where g_i is a generative transition in the atom A_i (the *active* atom) and $f_j : \mathsf{ActLab}_j \cap \mathsf{ActLab}(g_i) \to \mathsf{T}_{\mathsf{R}j}$ for all $j \neq i$. The atom A_i that executes the generative transition is denoted by $\mathit{GenAtom}(\alpha)$. The set of all compound transitions is denoted by $\mathit{Trans}(P)$. A compound transition α is enabled in a given state $(s_1, \cdots, s_N)$ if $g_i \in \mathsf{G}_i(s_i)$ and $f_j(a) \in \mathsf{R}_j(s_j, a)$ for all $j \neq i$ and $a \in \mathsf{ActLab}_j \cap \mathsf{ActLab}(g_i)$. We denote by $\mathit{enabled}(s)$ the set of all compound transitions enabled in state s. In case $a \notin \mathsf{ActLab}_j$ and a is output by a generative transition, the atom A_j can only remain in its actual state. In order to reflect this fact, given input choice functions f_j for g_i, we define the functions $f^*_{j,s_j} : \bigcup_i \mathsf{ActLab}_i \to \mathsf{T}_{\mathsf{R}i}$ such that $f^*_{j,s_j}(a) = f_j(a)$ if $a \in \mathsf{ActLab}_j \cap \mathsf{ActLab}(g_i)$ and $f^*_{j,s_j}(a) = \mathit{Dirac}(s_j)$, otherwise.

Once a compound transition α is fixed, the probability $\alpha(s, s')$ of reaching a state $s' = (s'_1, \cdots, s'_N)$ from a state $s = (s_1, \cdots, s_N)$ is $\sum_a g_i(s'_i, a) \cdot \prod_{j=1}^{N} f^*_{j,s'_j}(a)(s'_j)$ for all s in which α is enabled. Using the definition of generative and reactive transitions and simple arithmetic, it can be proven that $\sum_{s'} \alpha(s, s') = 1$ for all s and α enabled in s. In consequence, IPIOA are MDPs in which the set of MDP transitions corresponds to IPIOA compound transitions.

In order to ease some definitions, we introduce a fictitious "stutter" compound transition ς. Intuitively, this transition is executed iff the system has reached a state in which no atom is able to generate a transition. The probability $\varsigma(s, s')$ of reaching s' from s using ς is 1, if $s = s'$, or 0, otherwise.

The atoms *involved* in a compound transition α are the active atom and every atom capable of engaging in a communication with labels output by the active atom when α is executed. Let g_i be the generative transition in α, then

$$\mathit{Inv}(\alpha) = \{A_j \mid \exists a \in \mathsf{ActLab}_j, s_i \in \mathsf{S}_i \bullet g(a, s_i) > 0\} .$$

No atom is involved in ς.

In the following, we suppose that input-enabled atoms $A_1, \ldots, A_N$ are given, and we consider the system P comprising all the atoms A_i. We call this system "the compound system". The states (paths, resp.) of the compound system are called global states (global paths, resp.) and the states (paths, resp.) of each atom are called local states (local paths, resp.).

As we have seen, it may be unrealistic to assume that the schedulers are able to see the full history of all the components in the system. In the following, we define restricted classes of schedulers in order to exclude unrealistic behaviours.

For the sake of simplicity, the framework in this paper considers only non-randomized schedulers. However, all of our results also hold for randomized schedulers (see [16]).

Distributed schedulers. Distributed schedulers were introduced in [6] but we use the revised definition of [14]. In a distributed setting as the one we have introduced, different kinds of nondeterministic choices need to be resolved. An atom needs an *output* scheduler to choose the next generative transition. In addition, it may be the case that many reactive transitions are enabled for a single label in the same atom. So, for each atom we need an *input* scheduler in order to choose a reactive transition for each previous history and for each label. Similar types of scheduler have been defined by Cheung *et al.* [6]. Such schedulers are able to make decisions based only on the local history of the atom. Therefore we need to extract the local history out of the global execution for which we define the notion of *projection*.

Given a path σ, the projection $[\sigma]_i$ of the path σ over an atom A_i is defined inductively as follows: **(1)** $[(\mathsf{init}_1, \cdots, \mathsf{init}_N)]_i = \mathsf{init}_i$, **(2)** $[\sigma.\alpha.s]_i = [\sigma]_i$ if $A_i \notin \mathit{Inv}(\alpha)$, **(3)** $[\sigma.(g_j, \cdots).s]_i = [\sigma]_i.g_i.\pi_i(s)$ if $j = i$ (where π_i is the usual projection on tuples) and **(4)** $[\sigma.(g_j, \cdots, f_i, \cdots).s]_i = [\sigma]_i.f_i.\pi_i(s)$ if $i \in \mathit{Inv}(\alpha)$ and $j \neq i$. The set of all the projections over an atom A_i is denoted by $\mathsf{Proj}_i(P)$.

An output scheduler for the atom A_i is a function $\Theta_i : \mathsf{Proj}_i(P) \to \mathsf{T}_{\mathsf{G}i}$ such that, if $\mathsf{G}_i(\mathit{last}([\sigma]_i)) \neq \emptyset$ then $\Theta_i([\sigma]_i) = g \implies g \in \mathsf{G}_i(\mathit{last}([\sigma]_i))$. An input scheduler for an atom A_i is a function $\Upsilon_i : \mathsf{Proj}_i(P) \times \mathsf{ActLab}_i \to \mathsf{T}_{\mathsf{R}i}$ s.t. $\Upsilon_i([\sigma]_i, a) = r \implies r \in \mathsf{R}_i(\mathit{last}([\sigma]_i), a)$. Note that, if the output scheduler Θ_i fixes a generative transition for a given local path $[\sigma]_i$, then the actions in the generative transition can be executed in every global path σ' s.t. $[\sigma']_i = [\sigma]_i$, since we require the atoms to be input-enabled.

An important modification with respect to the framework in [6,5] is the addition of an *interleaving scheduler* that selects the active component to perform the next output. We first consider arbitrary interleaving schedulers that take decisions based on the complete history of the whole system.

An interleaving scheduler is a map that, for a given (global) history, chooses an active atom that will be the next to execute an output transition (according to its output scheduler). Formally, an *interleaving scheduler* is a function $\mathcal{I} : \mathit{Paths}(P) \to \{A_1, \cdots, A_N\}$ such that, if there exists i with $\mathsf{G}_i(\mathit{last}([\sigma]_i)) > 0$ (that is, if there is some atom being able to generate a transition) then $\mathcal{I}(\sigma) = A_i \implies \mathsf{G}_i(\mathit{last}([\sigma]_i)) \neq \emptyset$.

A distributed scheduler for the compound system results by the appropriate composition of the interleaving scheduler and the output and input schedulers of each atom.

Definition 2. *Given an interleaving scheduler $\mathcal{I}$, input schedulers Υ_i and output schedulers Θ_i for each atom i, the* distributed scheduler *η obtained by composing $\mathcal{I}$, Θ_i and Υ_i is defined as $\eta(\sigma) = (g_i, f_1, \cdots, f_{i-1}, f_{i+1}, \cdots, f_N)$ where $\mathcal{I}(\sigma) = A_i$, $g_i = \Theta_i([\sigma]_i)$, and $f_j(a) = \Upsilon_j([\sigma]_j, a)$ for all $j \neq i$ and $a \in \mathsf{ActLab}_j$. In case there is no generative transition enabled, we require $\eta(\sigma) = \varsigma$. The set of distributed schedulers of P is denoted by $\mathit{Dist}(P)$.*

Note that, even when interleaving schedulers are unrestricted, compound schedulers for the compound system are still restricted, since the local schedulers can only see the portion of the history corresponding to the component.

Strongly distributed schedulers. Strongly distributed schedulers were introduced in [14] as a smaller but yet meaningful class of distributed schedulers.

Distributed schedulers provide an accurate model in case the interleaving scheduler has access to all information. As an example, suppose that the atoms represent processes running on the same computer, and the interleaving scheduler plays the role of the operating system scheduler. In case such a scheduler assigns priorities to the processes by gathering information from all processes states, a total information interleaving scheduler is a natural model.

On the contrary, if we are analysing an agreement protocol and each atom models an independent node in a network, then the order in which two nodes execute cannot change according to information that is not available to them. A simple toy example suffices to show how the worst-case probability is affected by the information available to the interleaving scheduler.

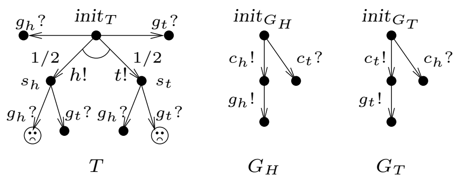


Fig. 5. Motivating strongly distributed schedulers

The atoms in Fig. 5 represent a game in which T plays against G_H and G_T. Atom T loses if it receives a $g_h!$ and the coin has landed heads, and similarly for $g_t!$ and tails. Atoms G_H and G_T take the guess in the following fashion: in case G_H believes that the coin landed heads, it outputs $c_h!$ and then $g_h!$. Conversely, in case G_T believes that the coin landed tails, it outputs $c_t!$ and then $g_t!$. Note that, as soon as one of the players chooses an option, the other one accepts it and stays quiet. Following the intuition given by the example in the introduction, there is no way that G_H and G_T can make an arrangement in such a way that they win all the times. However, the interleaving scheduler that selects T in init, and then G_H for the path $\sigma_{heads} = \mathsf{init}\,.h!.(s_h, \mathsf{init}_{G_H}, \mathsf{init}_{G_T})$ and G_T for the path $\sigma_{tails} = \mathsf{init}\,.t!.(s_t, \mathsf{init}_{G_H}, \mathsf{init}_{G_T})$ leads T to ☹ with probability 1.

This scheduler is distributed according to Def. 2, since distributed schedulers restrict the resolution of internal nondeterministic choices, and these atoms have no such choices. In particular, the interleaving scheduler can arrange the execution of G_H and G_T according to the hidden information in T.

The compound model for T, G_H and G_T is very similar to the one in Fig. 3. In fact, since in the graphical representation we omit information concerning the structure of the states, the graphical representation is the same. The unrealistic scheduler in which T losses all the time is also very similar to the unrealistic scheduler in Fig. 2.

The information available to atoms A and B can be defined as $[\sigma]_{A,B} = ([\sigma]_A, [\sigma]_B)$. Note that $[\sigma_{heads}]_{G_H,G_T} = [\sigma_{tails}]_{G_T,G_T} = (\mathsf{init}_{G_H}, \mathsf{init}_{G_T})$. In addition, in the unrealistic scheduler we have $\mathcal{I}(\sigma_{heads}) = G_H$ and $\mathcal{I}(\sigma_{tails}) = G_T$. Generalizing G_H and G_T to be two atoms A, B, and σ_{heads}, σ_{tails} to be two paths σ, σ', we obtain the following condition: for all atoms $A \neq B$ there cannot be two paths σ, σ' such that: **(1)** $[\sigma]_{A,B} = [\sigma']_{A,B}$ and **(2)** atom A is scheduled in σ and **(3)** atom B is scheduled in σ'. Formally:

$$\forall A, B \bullet A \neq B \implies \forall \sigma \bullet \ \nexists \sigma' \bullet [\sigma]_{A,B} = [\sigma']_{A,B} \wedge \mathcal{I}(\sigma) = A \wedge \mathcal{I}(\sigma') = B \ . \quad (1)$$

Definition 3. *A scheduler η is* strongly distributed *iff η is distributed and (1) holds on the interleaving scheduler $\mathcal{I}$ that defines η. The set of strongly distributed schedulers of P is denoted by SDist(P).*

In [14] (where strongly distributed schedulers are introduced for the first time) we prove some properties to further support the fact that (1) is a natural restriction whenever the interleaving nondeterminism is resolved a distributed fashion. In particular, we prove that (1) implies a more general condition in which A and B are replaced with two disjoint sets of atoms $\mathcal{A}$ and $\mathcal{B}$. Strongly distributed schedulers also generalize the *rate schedulers* of [15].

3 Partial Order Reduction under Distributed Schedulers

In this section, we develop two variants of POR for probabilistic systems (each variant corresponding to a class of schedulers) using the *ample sets* construction of [9]. Such variants exploit the distributedness assumptions on schedulers in order to improve the reduction.

Partial order reduction for $\mathsf{LTL}_{\setminus\{\text{next}\}}$. Given a system and a property, the technique of partial order reduction yields another system with less transitions. The reduced system is constructed by traversing the state space. When expanding a given state, not all the transitions enabled are considered. An *ample set* *ample*(s) must be calculated for each state s, and only transitions in the ample set are considered during the search. POR techniques impose restrictions on the ample sets to ensure that, for each property, the reduced system complies with the property iff the original system does.

We focus on the case where LTL properties do not contain the next operator. Given a set AP of atomic propositions and a labelling function $\mathsf{L} : \mathsf{S} \to \mathcal{P}(\mathsf{AP})$, the set of $\mathsf{LTL}_{\setminus\{\text{next}\}}$ formulae are generated by the following grammar.

$$\phi ::= \mathit{True} \mid l \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathsf{U} \phi_2 \ ,$$

where *True* is a constant and $l \in \mathsf{AP}$. Intuitively, an infinite path ρ satisfies $\phi_1 \mathsf{U} \phi_2$ (denoted by $\rho \models \phi_1 \mathsf{U} \phi_2$) iff there is position in ρ in which ϕ_2 holds, and ϕ_1 holds in all intermediate positions of ρ from the beginning until the position in which ϕ_2 holds. As usual, we write $\mathsf{F}\phi$ for $\mathit{True}\,\mathsf{U}\,\phi$, and $\Pr^\eta(\phi)$ for $\Pr^\eta(\{\rho \mid \rho \models \phi\})$.

Restrictions to the ample sets are based on the notion of *independence*. We say that two transitions α, β are *independent* iff $(\exists s : \{\alpha, \beta\} \subseteq \mathit{enabled}(s)) \implies \mathit{Inv}(\alpha) \cap \mathit{Inv}(\beta) = \emptyset$. So, two transitions are independent only if the execution of one of them does not interfere with the execution of the other one. Note that the order of execution is irrelevant and that neither of them can disable the other. Notice also that this definition is of a more structural nature than the one in [2]. This is no surprise, since our improvements profit from the structure of the model.

We need some additional definitions before presenting the restrictions for POR. A compound transition α is *stutter* iff $\alpha(s,s') = 0$ for all s such that $\alpha \in \mathit{enabled}(s)$ and s' such that $\mathsf{L}(s) \neq \mathsf{L}(s')$. An *end component* (EC) is a pair (T, A) where $A : T \to \mathcal{P}(\mathit{Trans})$ and T is a set of states such that: **(1)** $\emptyset \neq A(s) \subseteq \mathit{enabled}(s)$ for all $s \in T$, **(2)** $\alpha(s,t) > 0$ implies $t \in T$, for all $s \in T$, $\alpha \in A(s)$ **(3)** for every $s, t \in T$ there exists a path from s to t.

The restrictions for the ample sets of [2] to preserve $\mathsf{LTL}_{\backslash\{\text{next}\}}$ properties under unrestricted full-history dependent schedulers are listed below. $\widehat{\mathsf{S}}$ denotes the set of reachable states in the reduced system $\widehat{P}$, which is constructed by taking $\mathit{ample}(s)$ to be the set of enabled transitions in $s \in \widehat{\mathsf{S}}$.

(**A1**) *For all states* $s \in \mathsf{S}$*,* $\emptyset \neq \mathit{ample}(s) \subseteq \mathit{enabled}(s)$*,*
(**A2**) *If* $s \in \widehat{\mathsf{S}}$ *and* $\mathit{ample}(s) \neq \mathit{enabled}(s)$*, then each transition* $\alpha \in \mathit{ample}(s)$ *is stutter,*
(**A3**) *For each path* $\sigma = s.\alpha_1.s_1.\alpha_2.\cdots.\alpha_n.s_n.\gamma\cdots$ *in* P *where* $s \in \widehat{\mathsf{S}}$ *and* γ *is dependent on* $\mathit{ample}(s)$ *there exists an index* $1 \leq i \leq n$ *such that* $\alpha_i \in \mathit{ample}(s)$*,*
(**A4**) *If* (T, A) *is an EC in* $\widehat{P}$ *and* $\alpha \in \bigcap_{t \in T} \mathit{enabled}(t)$*, then* $\alpha \in \bigcup_{t \in T} \mathit{ample}(t)$
(**A5**) *If* $s.\alpha_1.s_1.\alpha_2.s_2.\cdots.\alpha_n.s_n.\gamma.s_{n+1}$ *is a path in* P *where* $s \in \widehat{\mathsf{S}}$*,* $\alpha_1, \cdots, \alpha_n$*,* $\gamma \notin \mathit{ample}(s)$ *and* γ *is probabilistic (i.e.* $0 < \gamma(s', t') < 1$ *for some* s', t'*) then* $|\mathit{ample}(s)| = 1$*.*

Conditions **A1**–**A3** are original for POR on non-probabilistic systems [9]. **A1** ensures that the reduced model is a submodel of the original one, and that it does not have terminal states (since the original model does not have either). **A3** enforces that any finite sequence of transitions leaving a state s that does not contain a transition in *ample*(s) can be extended with such transition. Together with **A2**, they ensure that any execution in the original system can be mimicked by an observational equivalent trace in the reduced system. Besides, notice that **A3** is the only condition that is concretely related to the notion of (in)dependence. Condition **A4** is a probabilistic variant of Peled's cycle condition on non-probabilistic models. In such models this condition enforces that, if a transition is enabled indefinitely along a path in the original system, then the transition is enabled in the reduced system in at least one state in such a path. Therefore, condition **A4** ensures that all fair paths are also represented in the reduced system. This variant is somewhat weaker than the original one: since the restriction does not concern cycles, but end components, some cycles are not required to comply with the restriction. Condition **A5** is particular for probabilistic models. Contrarily to the other conditions, **A5** is technical and non-intuitive and has been introduced precisely to *not* eliminate the behaviour introduced by (non-distributed!) schedulers like the one of the example in Fig. 2. We remark that if the model P is non-probabilistic, condition **A5** has no effect and condition **A4** reduces to Peled's original cycle condition. As a consequence, conditions **A1**–**A5** behave exactly in the same way as Peled's original conditions for POR on non-probabilistic models.

In case we assume that the schedulers are distributed, we can replace **A5** by (**A5′**) *If* $s.\alpha_1.s_1.\alpha_2.s_2 \cdots \alpha_n.s_n.\gamma.s_{n+1}$ *is a path in* P *where* $s \in \widehat{S}$, $\alpha_1,\cdots,\alpha_n,\gamma \notin$ *ample*(s) *and* γ *is a probabilistic transition, then either ample*$(s) =$ *enabled*(s) *or GenAtom*$(\beta) =$ *GenAtom*(β')*, for all* $\beta, \beta' \in$ *ample*(s)*.*

Condition **A5′** relaxes condition **A5** in the sense that if a probabilistic transition can be reached from state s without executing a transition from the ample set, all transitions enabled in the reduced model (i.e. in *ample*(s)) are generated by the same atom. Contrarily to **A5**, **A5′** does not requires *ample*(s) to be a singleton; *ample*(s) may contain several transitions as long as they are generated by a single atom.

The result is formalized in the following theorem.

Theorem 1. *Let* ϕ *be an* $\mathsf{LTL}_{\setminus\{\text{next}\}}$ *formula and* P *be an IPIOA. Let* $\widehat{P}$ *be a reduction of* P *complying with conditions* **A1**–**A4**, **A5′**. *Then,*

$$\sup_{\eta \in Dist(P)} \Pr^{\eta}(\phi) \le \sup_{\eta \in Sched(\widehat{P})} \Pr^{\eta}(\phi) .$$

In case we assume *strongly distributed schedulers*, **A5** can be disregarded.

Theorem 2. *Let* ϕ, P *be as in Theorem 1. Let* $\widehat{P}$ *be a reduction of* P *complying with conditions* **A1**–**A4**. *Then,* $\sup_{\eta \in SDist(P)} \Pr^{\eta}(\phi) \le \sup_{\eta \in Sched(\widehat{P})} \Pr^{\eta}(\phi)$.

As an example, recall atoms T and G in Fig. 1 and the non-distributed scheduler η^w in Fig. 2. According to Theorem 1 the reduction in Fig. 4 is correct in case distributed schedulers are assumed. However, in the original system P we have $\Pr^{\eta^w}(\mathsf{F}☹) = 1$, while in $\widehat{P}$ we have $\Pr^{\eta}(\mathsf{F}☹) \le \frac{1}{2}$ for all η. This is due to the fact that η^w is not distributed. In fact, the supremum over all distributed schedulers in P is $\frac{1}{2}$, which coincides with $\sup_{\eta \in Sched(\widehat{P})} \Pr^{\eta}(\mathsf{F}☹)$. Recall now the example in Fig. 5 with atoms T, G_H and G_T. We mentioned that the scheduler of Fig. 2 *is* distributed in this setting. Call this scheduler η^d. If we assume strongly distributed schedulers, the reduction in Fig. 4 is allowed, and there is no scheduler yielding probability 1 in the reduced system. This is correct, since the scheduler η^d is *not* strongly distributed. However, if we want to preserve all distributed schedulers (even those that are *not* strongly distributed) then condition **A5′** prevents the reduction in Fig. 4, since $c_h!$ and $c_t!$ are generated by atoms G_H and G_T, resp. This is exactly what we want, since the scheduler η^d is a valid distributed scheduler for T, G_H and G_T, and so a corresponding scheduler yielding probability 1 must exist in the reduced system.

Correctness of our techniques. The proofs of Theorems 1 and 2 are quite technical and several details are involved. However, these proofs rely on the same principle as in the non-probabilistic case. Our aim is to give an explanation so that the reader can have proper insight on the validity of our techniques. For fully detailed proofs, see [16].

In the non-probabilistic case, the standard argument is as follows. For every property ϕ, we need to prove that ϕ is satisfied in all paths in P if and only if ϕ is satisfied in all paths in $\widehat{P}$. Since $\widehat{P}$ is a subgraph of P, one implication is trivial. For the other implication, the conditions on the reduction are used to prove that, if some path ρ in P does not satisfy ϕ, then $\widehat{\rho} \not\models_{\widehat{P}} \phi$ for some $\widehat{\rho}$.

Similarly, in our case it is sufficient to prove that, for each scheduler η in the original system, there exists a corresponding $\widehat{\eta}$ in the reduced system. The probability values for η and $\widehat{\eta}$ must coincide for all paths relevant to ϕ. We prove that, for each *distributed* (*strongly distributed*, resp.) scheduler, there is a corresponding scheduler in the reduced system that yields the same probability value. As a consequence, it may be the case that, for some non-distributed schedulers, there are no corresponding schedulers in the reduced system. However, this causes no harm since schedulers are assumed to be distributed.

Given a non-probabilistic system P, let $\rho = s_1.\alpha_1.s_2.\alpha_2.\cdots$ and ϕ such that $\rho \not\models_{\widehat{P}} \phi$. We sketch how the corresponding path $\widehat{\rho}$ is constructed in the standard approach. If $\alpha_1 \in \mathit{ample}(s_1)$, then $\widehat{\rho}$ starts with $s_1.\alpha_1$ and the construction continues from $s_2.\alpha_2.\cdots$. On the contrary side, if $\alpha_1 \notin \mathit{ample}(s_1)$, then $\widehat{\rho}$ cannot start with $s_1.\alpha_1$, since α_1 is not enabled for s_1 in $\widehat{P}$. However, condition **A1** ensures that $\mathit{ample}(s_1) \neq \emptyset$. For simplicity, let's consider the case in which some $\beta \in \mathit{ample}(s_1)$ is eventually executed in ρ. W.l.o.g., we can take such a β to be the first transition α_n in ρ such that $\alpha_n \in \mathit{ample}(s_1)$. Then, by condition **A3** and definition of independence, we have that $\rho' = s_1.\alpha_n.s'_2.\alpha_1.\cdots.s'_{n-1}.\alpha_{n-1}.s_n.\alpha_{n+1}.\cdots$ is a path in P. (Here, s'_i denotes the state such that $\alpha_{i-1}(s'_i, s'_{i+1}) = 1$, since the system is non-probabilistic.) Let $\ell_i = \mathsf{L}(s_i)$ for all i. Then, since **A2** requires the transitions in $\mathit{ample}(s)$ to be stuttering, the sequence $\mathsf{L}(\rho)$ of labels in ρ has the form $\ell_1 \cdots \ell_n \ell_n \ell_{n+2} \cdots$. Condition **A2** can be used to prove that $\mathsf{L}(\rho') = \ell_1 \ell_1 \ell_2 \cdots$. So, since $\mathsf{L}(\rho)$ and $\mathsf{L}(\rho')$ differ only in the amount of times that each ℓ_i appears, and $\mathsf{LTL}_{\setminus\{\text{next}\}}$ formulae are stuttering-invariant, $\phi \not\models_P \rho'$. Having found ρ', we let $\widehat{\rho}$ start with $s_1.\alpha_n$ and continue the construction using $s'_2.\alpha_1.\cdots.s'_{n-1}.\alpha_{n-1}.s_n.\alpha_{n+1}.\cdots$. The case in which no transition β is executed in ρ is similar (see [9]).

In summary, the key step of the construction is to "move" β across the α_i's so that it executes immediately after s_1. In the probabilistic case, we must deal with schedulers (which have a tree-like structure) instead of mere paths, and so it is not clear how a transition can be moved. Consider the scheduler η in Fig. 6 (a) and the reduction in Fig. 4. The corresponding scheduler in $\widehat{T \parallel G}$ cannot start with the probabilistic transition $\frac{1}{2}h! + \frac{1}{2}t!$, since it is not enabled in $\widehat{\mathsf{init}_\parallel}$. However, the same probabilistic effect is obtained by the scheduler $\widehat{\eta}$ that executes $c_h!$ in the first place, as illustrated in Fig. 6 (b). In this figure, $c_h!$ is moved across both $h!$ and $t!$. In the general case, the transition in the ample set is moved across the transitions in all branches. Note that, in order to move $c_h!$ after init, we rely on the fact that $c_h!$ is executed after both $h!$ and $t!$. In fact, there is no way to transform the non-distributed scheduler in Fig. 2 into a scheduler for the reduced system in Fig. 4: although $c_h!, c_t! \in \mathit{ample}(\mathsf{init}_\parallel)$, we have that $c_h!$ is chosen in one of the branches, while $c_t!$ is chosen in the other.

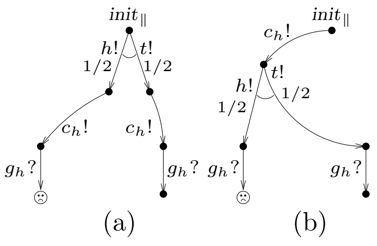


Fig. 6. Transforming a scheduler in the coin example

Groesser *et al.* [2] showed how schedulers for the original system can be mapped to schedulers in the reduced system. They require condition **A5** because the transformation is not possible for some schedulers and some reductions, even if such reductions comply with **A1**–**A4**. However, we proved in [16] that a similar transformation can be carried out for all schedulers η complying with the following condition:

$$\eta(\sigma) \in \mathit{ample}(s_1) \ \wedge\ \eta(\sigma') \in \mathit{ample}(s_1) \implies \eta(\sigma) = \eta(\sigma') \qquad (2)$$

for all $\sigma = s_1.\alpha_1.\cdots.\alpha_{n-1}.s_n$, $\sigma' = s_1.\alpha'_1.\cdots.\alpha'_{n'-1}.s'_{n'}$ such that the α_k's and the α'_k's are independent from $\mathit{ample}(s)$. Roughly speaking, the ample transition must be the same in all branches in which an ample transition appears.

We show that (2) holds if: **(1)** η is distributed and **A5′** holds or **(2)** η is strongly distributed. If η is distributed, let I be $\cup_{\beta \in \mathit{ample}(s_1)} \mathit{Inv}(\beta)$ and let σ, σ' be as in (2). Since the α_k's and the α'_k's are independent from all the transitions in $\mathit{ample}(s_1)$, we have $I \cap \mathit{Inv}(\alpha_k) = I \cap \mathit{Inv}(\alpha_{k'}) = \emptyset$ for all k. Then, $[\sigma]_i = [\sigma']_i = s_1$ for all $A_i \in I$. By **A5′**, we have $\mathit{GenAtom}(\eta(\sigma)) = \mathit{GenAtom}(\eta(\sigma'))$. Let $A_i = \mathit{GenAtom}(\eta(\sigma))$ and let Θ_i be the output scheduler that defines η. Then, $\Theta_i([\sigma]_i) = \Theta_i([\sigma']_i)$, and so the generative transition is the same in both $\eta(\sigma)$ and $\eta(\sigma')$. The same argument can be used to show that the input choice functions are the same in both $\eta(\sigma)$ and $\eta(\sigma')$, and so $\eta(\sigma) = \eta(\sigma')$.

In case η is strongly distributed, we define $A_i = \mathit{GenAtom}(\eta(\sigma))$ and $A_{i'} = \mathit{GenAtom}(\eta(\sigma'))$. Then, $[\sigma]_i = [\sigma']_i\ (= s_1)$ and $[\sigma]_{i'} = [\sigma']_{i'}\ (= s_1)$. Let $\mathcal{I}$ be the interleaving scheduler that defines η. By Eqn. (1), we have $A_i = \mathcal{I}(\sigma) = \mathcal{I}(\sigma') = A_{i'}$, and so $\mathit{GenAtom}(\eta(\sigma)) = \mathit{GenAtom}(\eta(\sigma'))$. Following the same reasoning as in the case of distributed schedulers, we conclude that $\eta(\sigma) = \eta(\sigma')$.

The bottom line is that the restrictions imposed to schedulers (together with **A5′**, in case distributed schedulers are assumed) allow to transform every scheduler in P into a scheduler in $\widehat{P}$ without requiring **A5**.

Using our technique with existing model checking algorithms. We emphasize that, although the correctness of the reduction relies on the assumption that the schedulers are distributed (strongly distributed, resp.), the reduced system is analysed assuming *total information* (because of the undecidability result in [13], the verification under partial information cannot be carried out in a fully automated fashion). The result of the verification thus corresponds to a *pessimistic* analysis of the reduced system. As a consequence, the bounds obtained are still safe, but they are not so tight as for distributed (strongly distributed, resp.) schedulers.

As an example, suppose that we are interested in finding the supremum probability that a system P fails under distributed schedulers. Suppose that 0.1 is the highest probability of failure allowed by the specification. Moreover, suppose that, by using the standard model checking algorithm for MDPs (e.g. [3]), we calculate that the supremum probability of a failure quantifying over all schedulers is 0.15. According to this analysis, the system would not meet the specification. However, the schedulers yielding probabilities greater than 0.1

might be "unrealistic" schedulers as the one in Fig. 2. Suppose that we construct $\widehat{P}$ as described above. Then, we can use the algorithm in [3] to calculate $S = \sup_{\eta \in Sched(\widehat{P})} \Pr^{\eta}(\mathsf{F}\ \mathit{Fail})$. If $S = 0.05$, then Theorem 1 above ensures that $\sup_{\eta \in Dist(P)} \Pr^{\eta}(\mathsf{F}\ \mathit{Fail}) \leq 0.05$, and so the system meets the specification. In this sense, the bounds are *safe* with respect to $Dist(P)$. Note that, in this case, the reduction has prevented some schedulers that are not distributed and so the verification on $\widehat{P}$ is more accurate than the verification on P.

4 Concluding Remarks

We have presented a theoretical framework to perform partial order reduction for probabilistic model checking. Our technique is a revision of previous works [2,10]. We showed that, in the context of distributed systems, the bounds for the probability values calculated by the technique on those works may result overly safe. We then showed that the new condition of [2,10] to construct the ample set may be relaxed or even dropped. This simplifies the algorithm and results in smaller reductions.

The POR technique for symbolic representation introduced in [1] constructs ample sets with transitions from several atoms. So, Theorem 2 allows us to apply a similar technique for probabilistic systems. We are currently busy implementing the technique into PRISM using these ideas. Preliminary results are shown in Table 1.

We have selected two notable case studies. Table 1(a) reports results checking anonymity on the dining cryptographers problem [4]. Column "%" indicates in percentage how small is the reduced model with respect of the full system. Thus, for instance, the size of the state space of the reduced model is 23.58% of the size of the state space of the full model for 11 cryptographers (i.e., more than 4 times smaller). Note that, in general, the construction time of the system is significantly more expensive for POR when compared to the construction time of the full system. Nonetheless, the calculation time of the probability values is significantly larger in the full model. Thus, the total processing time on large systems is better under POR (see the 11 cryptographers). We remark that the old POR reduction (including **A5**) achieves the same results in this case study. However, our results using POR for the symbolic representation and explicit vectors for calculation (the so-called "hybrid" approach [20]) significantly improve the explicit approach of LiQuor [8].

Still more interesting is our second case study. It reports on the verification of the Binary Exponential Backoff protocol of the IEEE 802.3. (The model is the same used in [19] adapted to PRISM notation.) We calculated the maximum and minimum probability that a colliding host aborts transmission after multiple collisions. The numbers n, N, and K are respectively the number of colliding hosts, the maximum number of attempts to seize the channel, and the maximum time slots. Table 1(b) shows reductions yielding sizes up to 5% of the full state space. More interesting is that reduction using only **A1**–**A4** is significantly more efficient than the old reduction with **A1**–**A5** (up to 58.88% in case 6/3/4). We

Table 1. Summary of Experimental Results

(a) Dining Cryptographers

	Full			**A1**–**A4** reduct.			
n	size	constr.	total	size	%	constr.	total
7*	287666	0m00.19	0m03.53	115578	40.18	0m13.01	0m16.59
8*	1499657	0m00.30	0m16.18	526329	35.10	0m36.69	0m52.96
9*	7695856	0m00.44	1m24.84	2363896	30.72	1m46.16	2m29.15
10	39005612	0m00.70	4m41.10	10495991	26.91	4m48.19	6m40.37
11	195718768	0m01.11	29m43.34	46159864	23.58	13m12.84	21m02.46

(b) Binary exponential backoff (size comparison)

Model	Full	**A1**–**A5** reduct.		**A1**–**A4** reduct.		
n / N / 2^K	size	size	% full	size	% full	% **A5**
4 / 3 / 4	532326	191987	36.07	126629	23.79	65.96
5 / 3 / 4	13866186	2752750	19.85	1690227	12.19	61.40
6 / 3 / 4	357387872	36974560	10.35	21771724	6.09	58.88
4 / 3 / 8	3020342	913379	30.24	604457	20.01	66.18
5 / 3 / 8	115442928	18569442	16.09	11585347	10.04	62.39
6 / 3 / 8	4318481408	353075296	8.18	212917856	4.93	60.30

(c) Binary exponential backoff (time comparison)

Model	Full		**A1**–**A5** reduct.		**A1**–**A4** reduct.	
n / N / 2^K	constr.	total	constr.	total	constr.	total
4 / 3 / 4	0m01.39	1m04.27	0m18.96	1m22.98	0m18.02	1m13.36
5 / 3 / 4	0m03.49	11m32.99	1m16.82	8m15.60	1m14.53	6m50.12
6 / 3 / 4	0m07.55	5h03m39.81	4m00.95	1h13m11.39	5m15.06	53m35.43
4 / 3 / 8	0m02.05	3m33.62	0m23.85	3m01.88	0m22.78	2m28.50
5 / 3 / 8	0m05.41	1h21m13.54	1m36.41	30m42.82	1m41.18	22m09.23
6 / 3 / 8	0m13.30	—	5m14.95	12h31m57.39	6m44.82	7h45m46.75

Entries marked with * run on a Pentium 4 630, 3.0Ghz with 2Gb memory, while all the others run on an Opteron 8212 (dual core) with 32Gb memory.

obtained similar satisfactory results on time comparison, notably (again) in case 6/3/4. We note that the 6/3/8 full model could not be analysed because the state space was too large to fit in the hybrid engine of PRISM.

Of course, not all examples we ran yielded such impressive results. We have experienced very little reduction in cases in which components depend very much from each other. This is nonetheless reasonable as our technique is precisely devised for distributed system with little sharing. In particular, both case studies have few communication points and significant local processing.

It is in our plans to report soon on the details of the implementation of the tool under development.

References

1. Alur, R., Brayton, R.A., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state-space exploration. Formal Methods in System Design 18(2), 97–116 (2001)
2. Baier, C., Größer, M., Ciesinski, F.: Partial order reduction for probabilistic systems. In: QEST 2004, Washington, DC, USA, pp. 230–239. IEEE CS, Los Alamitos (2004)

3. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 288–299. Springer, Heidelberg (1995)
4. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. J. Cryptology 1(1), 65–75 (1988)
5. Cheung, L.: Reconciling Nondeterministic and Probabilistic Choices. PhD thesis, Radboud Universiteit Nijmegen (2006)
6. Cheung, L., Lynch, N., Segala, R., Vaandrager, F.: Switched PIOA: Parallel composition via distributed scheduling. Theor. Comput. Sci. 365(1-2), 83–108 (2006)
7. Ciesinski, F., Baier, C.: LiQuor: A tool for qualitative and quantitative linear time analysis of reactive systems. In: QEST 2006, pp. 131–132. IEEE CS, Los Alamitos (2006)
8. Ciesinski, F., Baier, C., Größer, M., Klein, J.: Reduction techniques for model checking markov decision processes. In: QEST 2008, pp. 45–54 (2008)
9. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
10. D'Argenio, P., Niebert, P.: Partial order reduction on concurrent probabilistic programs. In: QEST 2004, Washington, DC, USA, pp. 240–249. IEEE CS, Los Alamitos (2004)
11. de Alfaro, L.: The verification of probabilistic systems under memoryless partial-information policies is hard. In: PROBMIV 1999. TR CSR-99-8, University of Birmingham, pp. 19–32 (1999)
12. de Alfaro, L., Henzinger, T.A., Jhala, R.: Compositional methods for probabilistic systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 351–365. Springer, Heidelberg (2001)
13. Giro, S., D'Argenio, P.: Quantitative model checking revisited: neither decidable nor approximable. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 179–194. Springer, Heidelberg (2007)
14. Giro, S., D'Argenio, P.: On the expressive power of schedulers in distributed probabilistic systems. In: Proc. of QAPL 2009, York, UK, March 28-29 (2009), Extended version, `cs.famaf.unc.edu.ar/~sgiro/QAPL09-ext.pdf`
15. Giro, S., D'Argenio, P.: On the verification of probabilistic I/O automata with unspecified rates. In: Proc. of 24th SAC, pp. 582–586. ACM Press, New York (2009)
16. Giro, S., D'Argenio, P.: Partial order reduction for probabilistic systems assuming distributed schedulers. Technical Report Serie A, Inf. 2009/02, FaMAF, UNC (2009), `http://cs.famaf.unc.edu.ar/~sgiro/TR-A-INF-09-2.pdf`
17. Glabbeek, R.v., Smolka, S., Steffen, B.: Reactive, generative, and stratified models of probabilistic processes. Information and Computation 121, 59–80 (1995)
18. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
19. Jeannet, B., D'Argenio, P., Larsen, K.: Rapture: A tool for verifying Markov Decision Processes. In: Cerna, I. (ed.) Tools Day 2002, Brno, Czech Republic, Technical Report, Faculty of Informatics, Masaryk University Brno (2002)
20. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. International Journal on Software Tools for Technology Transfer (STTT) 6(2), 128–142 (2004)
21. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. PhD thesis, Laboratory for Computer Science, MIT (1995)
22. Vardi, M.: Automatic verification of probabilistic concurrent finite state programs. In: Procs. of 26th FOCS, pp. 327–338. IEEE Press, Los Alamitos (1985)

Model-Checking Games for Fixpoint Logics with Partial Order Models

Julian Gutierrez and Julian Bradfield

LFCS. School of Informatics. University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, UK

Abstract. We introduce model-checking games that allow *local* second-order power on *sets* of independent transitions in the underlying partial order models where the games are played. Since the one-step interleaving semantics of such models is not considered, some problems that may arise when using interleaving semantics are avoided and new decidability results for partial orders are achieved. The games are shown to be *sound* and *complete*, and therefore determined. While in the interleaving case they coincide with the local model-checking games for the μ-calculus ($L\mu$), in a noninterleaving setting they verify properties of Separation Fixpoint Logic (SFL), a logic that can specify in partial orders properties not expressible with $L\mu$. The games underpin a novel decision procedure for model-checking all temporal properties of a class of infinite and regular event structures, thus improving previous results in the literature.

Keywords: Modal and temporal logics; Model-checking games; Hintikka game semantics; Partial order models of concurrency; Process algebras.

1 Introduction

Model-checking games, also called Hintikka evaluation games, are played by two players, a "Verifier" Eve ($\exists$) and a "Falsifier" Adam ($\forall$). These *logic games* are played in a formula ϕ and a mathematical model $\mathfrak{M}$. In a game $\mathcal{G}(\mathfrak{M}, \phi)$ the goal of Eve is to show that $\mathfrak{M} \models \phi$, while the goal of Adam is to refute such an assertion. Solving these games amounts to answering the question of whether or not Eve has a strategy to win all plays in the game $\mathcal{G}(\mathfrak{M}, \phi)$. These games have a long history in mathematical logic and in the last two decades have become an active area of research in computer science, both from theoretical and practical view points. Good introductions to the subject can be found in [2,10].

In concurrency and program verification, most usually ϕ is a modal or a temporal formula and $\mathfrak{M}$ is a Kripke structure or a labelled transition system (LTS), i.e., a graph structure, and the two players play the game $\mathcal{G}(\mathfrak{M}, \phi)$ *globally* by picking single elements of $\mathfrak{M}$, according to the game rules defined by ϕ. This setting works well for concurrent systems with *interleaving* semantics since one always has a notion of global state enforced by the (nondeterministic) sequential computation of atomic actions, which in turn allows the players to choose only single elements of the structure $\mathfrak{M}$. However, when considering concurrent systems with partial order models explicit notions of *locality* and *concurrency* have to be taken into account. A possible solution to this problem – the traditional approach – is to use the one-step interleaving semantics of such models in order to recover the *globality* and *sequentiality* of the semantics of formulae.

This solution is, however, problematic for at least five reasons. Firstly, interleaving models usually suffer from the state space explosion problem. Secondly, interleaving interpretations cannot be used to give completely satisfactory game semantics to logics with partial order models as all information on independence in the models is lost in the interleaving simplification. Thirdly, although temporal properties can still be verified with the interleaving simplification, properties involving concurrency, causality and conflict, natural to partial order models of concurrency, can no longer be verified. From a more practical standpoint, partial order reduction methods cannot be applied directly to interleaving models in order to build less complex model checkers based on these techniques. Finally, the usual techniques for verifying interleaving models cannot always be used to verify partial order ones since such problems may become undecidable.

For these reasons, we believe that the study of verification techniques for partial order models continues to deserve much attention since they can help alleviate some of the limitations related with the use of interleaving models. We, therefore, abandon the traditional approach to defining model-checking games for logics with partial order models and introduce a new class of games called 'trace local monadic second-order (LMSO) model-checking games', where *sets* of independent elements of the structure at hand can be *locally* recognised. These games avoid the need of using the one-step interleaving semantics of partial order models, and thus define a more natural framework for analysing fixpoint modal logics with noninterleaving semantics. As a matter of fact, their use in the temporal verification of a class of regular event structures [11] improves previous results in the literature [5,7]. We do so by allowing a free interplay of fixpoint operators and local second-order power on *conflict-free* sets of transitions.

The logic we consider is Separation Fixpoint Logic (SFL) [3], a μ-calculus ($L\mu$) extension that can express causal properties in partial order models (e.g., transition systems with independence, Petri nets or event structures), and allows for doing *dynamic* local reasoning. The notion of locality in SFL, namely separation or disjointness of independent sets of resources, was inspired by the one defined *statically* for Separation Logic [8]. Since SFL is as expressive as $L\mu$ in an interleaving context, nothing is lost with respect to the main approaches to logics for concurrency with interleaving semantics. Instead, logics and techniques for interleaving concurrency are extended to a partial order setting with SFL.

Section 2 contains some background concepts and definitions. In Section 3, trace LMSO model-checking games are defined, and in Section 4 their soundness and completeness is proved. In Section 5, we show that they are decidable and their coincidence with the local model-checking games for $L\mu$ in the interleaving case. In Section 6 the game is used to effectively model-check a class of regular and infinite event structures. The paper concludes with Section 7.

2 Preliminaries

2.1 A Partial Order Model of Concurrency

A *transition system with independence* (TSI) [6] is an LTS where independent transitions can be recognised. Formally, a TSI $\mathfrak{T}$ is a structure (S, s_0, T, Σ, I), where S is a set of states with initial state s_0, $T \subseteq S \times \Sigma \times S$ is a transition relation, Σ is a set of labels, and $I \subseteq T \times T$ is an irreflexive and symmetric relation on independent transitions. The binary relation $\prec$ on transitions defined by

$$(s, a, s_1) \prec (s_2, a, q) \Leftrightarrow$$
$$\exists b.(s, a, s_1) I (s, b, s_2) \wedge (s, a, s_1) I (s_1, b, q) \wedge (s, b, s_2) I (s_2, a, q)$$

expresses that two transitions are instances of the same *action*, but in two different interleavings. We let $\sim$ be the least equivalence relation that includes $\prec$, i.e., the reflexive, symmetric and transitive closure of $\prec$. The equivalence relation $\sim$ is used to group all transitions that are instances of the same action in all its possible interleavings. Additionally, I is subject to the following axioms:

- **A1**. $(s, a, s_1) \sim (s, a, s_2) \Rightarrow s_1 = s_2$
- **A2**. $(s, a, s_1) I (s, b, s_2) \Rightarrow \exists q.(s, a, s_1) I (s_1, b, q) \wedge (s, b, s_2) I (s_2, a, q)$
- **A3**. $(s, a, s_1) I (s_1, b, q) \Rightarrow \exists s_2.(s, a, s_1) I (s, b, s_2) \wedge (s, b, s_2) I (s_2, a, q)$
- **A4**. $(s, a, s_1) \prec \cup \succ (s_2, a, q) I (w, b, w') \Rightarrow (s, a, s_1) I (w, b, w')$

Axiom **A1** states that from any state, the execution of an action leads always to a unique state. This is a determinacy condition. Axioms **A2** and **A3** ensure that independent transitions can be executed in either order. Finally, **A4** ensures that the relation I is well defined. More precisely, **A4** says that if two transitions t and t' are independent, then all other transitions in the equivalence class $[t]_\sim$ (i.e., all other transitions that are instances of the same action but in different interleavings) are independent of t' as well, and vice versa.

Local Dualities. Based on the previous axiomatization one can define two ways of observing *concurrency* such that in each case they are dual to *causality* and *conflict*, respectively [3]. The semantics of SFL is based on these dualities. Given a state s and a transition $t = (s, a, s')$, s is called the source node, $src(t) = s$, and s' the target node, $trg(t) = s'$. Thus, define the following relations on transitions:

$$\otimes \stackrel{\text{def}}{=} \{(t_1, t_2) \in T \times T \mid src(t_1) = src(t_2) \wedge t_1 \; I \; t_2\}$$
$$\# \stackrel{\text{def}}{=} \{(t_1, t_2) \in T \times T \mid src(t_1) = src(t_2) \wedge \neg(t_1 \; I \; t_2)\}$$
$$\ominus \stackrel{\text{def}}{=} \{(t_1, t_2) \in T \times T \mid trg(t_1) = src(t_2) \wedge t_1 \; I \; t_2\}$$
$$\leq \stackrel{\text{def}}{=} \{(t_1, t_2) \in T \times T \mid trg(t_1) = src(t_2) \wedge \neg(t_1 \; I \; t_2)\}$$

2.2 Sets in a Local Context

The relation $\otimes$ defined on pairs of transitions, can be used to recognize *sets* where every transition is independent of each other and hence can all be executed concurrently. Such sets are said to be *conflict-free* and belong to the same *trace*.

Definition 1. A *conflict-free set of transitions* P is a set of transitions with the same source node, where $t \otimes t'$ for each two elements t and t' in P.

Given a TSI $\mathfrak{T} = (S, s_0, T, \Sigma, I)$, all conflict-free sets of transitions at a state $s \in S$ can be defined locally from the *maximal set* of transitions $R_{\max}(s)$, where $R_{\max}(s)$ is the set of all transitions $t \in T$ such that $src(t) = s$. Hence all maximal sets and conflict-free sets of transitions are fixed given a particular TSI. Now we define the notion of locality used to give the semantics of SFL.

Definition 2. Given a TSI $\mathfrak{T}$, a *support set* R in $\mathfrak{T}$ is either a maximal set of transitions in $\mathfrak{T}$ or a non-empty conflict-free set of transitions in $\mathfrak{T}$.

Given a TSI, the set of all its support sets is denoted by $\mathfrak{P}$. The next definition is useful when defining the semantics of SFL: $R_1 \uplus R_2 \sqsubseteq R \overset{\text{def}}{=} R_1 \cup R_2 \subseteq R$, and R_1 and R_2 are two disjoint non-empty conflict-free support sets, and $\neg\exists t \in R \setminus (R_1 \cup R_2).\ \forall t' \in R_1 \cup R_2.\ t \otimes t'$.

Notice that the last definition characterizes support sets R_1 and R_2 that contain only concurrent transitions and cannot be made any bigger with respect to the support set R. Therefore, the sets $R_1 \uplus R_2$ with the properties above are the biggest *traces* that can be recognized from a support set R.

2.3 Separation Fixpoint Logic

Definition 3. *Separation Fixpoint Logic* (SFL) [3] has formulae ϕ built from a set Var of variables $Y, Z, \ldots$ and a set $\mathcal{L}$ of labels $a, b, \ldots$ by the following grammar:

$$\phi ::= Z \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \langle K \rangle_c \phi_1 \mid \langle K \rangle_{nc} \phi_1 \mid \phi_1 * \phi_2 \mid \mu Z.\phi_1$$

where $Z \in \text{Var}$, $K \subseteq \mathcal{L}$, and $\mu Z.\phi_1$ has the restriction that any free occurrence of Z in ϕ_1 must be within the scope of an even number of negations. Dual operators are defined as expected: $\phi_1 \vee \phi_2 \overset{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi_1 \bowtie \phi_2 \overset{\text{def}}{=} \neg(\neg\phi_1 * \neg\phi_2)$, $[K]_c\,\phi_1 \overset{\text{def}}{=} \neg\langle K \rangle_c \neg\phi_1$, $[K]_{nc}\,\phi_1 \overset{\text{def}}{=} \neg\langle K \rangle_{nc} \neg\phi_1$, $\nu Z.\phi_1 \overset{\text{def}}{=} \neg\mu Z.\neg\phi_1\,[\neg Z/Z]$.

Definition 4. A *TSI-based SFL model* $\mathfrak{M}$ is a TSI $\mathfrak{T} = (S, s_0, T, \Sigma, I)$ together with a valuation $\mathcal{V} : \text{Var} \to 2^{\mathfrak{S}}$, where $\mathfrak{S} = S \times \mathfrak{P} \times \mathfrak{A}$ is the set of tuples (s, R, t) of states $s \in S$, support sets $R \in \mathfrak{P}$ in the TSI $\mathfrak{T}$, and transitions $t \in \mathfrak{A} = T \cup \{t_\epsilon\}$. The denotation $\|\phi\|^{\mathfrak{T}}_{\mathcal{V}}$ of an SFL formula ϕ in the model $\mathfrak{M} = (\mathfrak{T}, \mathcal{V})$ is a subset of $\mathfrak{S}$, given by the following rules (omitting the superscript $\mathfrak{T}$):

$$\begin{aligned}
\|Z\|_{\mathcal{V}} &= \mathcal{V}(Z) \\
\|\neg\phi\|_{\mathcal{V}} &= \mathfrak{S} - \|\phi\|_{\mathcal{V}} \\
\|\phi_1 \wedge \phi_2\|_{\mathcal{V}} &= \|\phi_1\|_{\mathcal{V}} \cap \|\phi_2\|_{\mathcal{V}} \\
\|\langle K \rangle_c \phi\|_{\mathcal{V}} &= \{(s, R, t) \mid \exists b \in K.\ \exists s' \in S. \\
&\qquad t \leq (t' = s \xrightarrow{b} s') \wedge t' \in R \wedge (s', R'_{\max}(s'), t') \in \|\phi\|_{\mathcal{V}}\} \\
\|\langle K \rangle_{nc} \phi\|_{\mathcal{V}} &= \{(s, R, t) \mid \exists b \in K.\ \exists s' \in S. \\
&\qquad t \ominus (t' = s \xrightarrow{b} s') \wedge t' \in R \wedge (s', R'_{\max}(s'), t') \in \|\phi\|_{\mathcal{V}}\} \\
\|\phi_1 * \phi_2\|_{\mathcal{V}} &= \{(s, R, t) \mid \exists R_1, R_2. \\
&\qquad R_1 \uplus R_2 \sqsubseteq R \wedge (s, R_1, t) \in \|\phi_1\|_{\mathcal{V}} \wedge (s, R_2, t) \in \|\phi_2\|_{\mathcal{V}}\} \\
\|\mu Z.\phi(Z)\|_{\mathcal{V}} &= \bigcap\{Q \subseteq \mathfrak{S} \mid \|\phi\|_{\mathcal{V}[Z:=Q]} \subseteq Q\}
\end{aligned}$$

where $\mathcal{V}[Z := Q]$ is the valuation $\mathcal{V}'$ which agrees with $\mathcal{V}$ save that $\mathcal{V}'(Z) = Q$. A tuple (s, R, t) of a model $\mathfrak{M}$ is called a *process*. The initial process is the tuple $(s_0, R_{\max}(s_0), t_\epsilon)$, where s_0 is the initial state of $\mathfrak{T}$ and t_ϵ is the empty transition, such that for all $t \in T$ if $s_0 = src(t)$, then $t_\epsilon \leq t$.

Remark 1. *The models we consider here are infinite state systems of finite branching, i.e., image-finite. Also, henceforth, w.l.o.g., we consider only formulae in positive normal form for the definition of the games in the next section.*

3 Trace LMSO Model-Checking Games

Trace LMSO model-checking games $\mathcal{G}(\mathfrak{M}, \phi)$ are played on a model $\mathfrak{M} = (\mathfrak{T}, \mathcal{V})$, where $\mathfrak{T} = (S, s_0, T, \Sigma, I)$ is a TSI, and on an SFL formula ϕ. The game can also be presented as $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$, or even as $\mathcal{G}_{\mathfrak{M}}(s_0, \phi)$, where $H_0 = (s_0, R_{\max}(s_0), t_\epsilon)$ is the *initial process* of $\mathfrak{S}$. The board in which the game is played has the form $\mathfrak{B} = \mathfrak{S} \times Sub(\phi)$, for a process space $\mathfrak{S} = S \times \mathfrak{P} \times \mathfrak{A}$ of states s in S, support sets R in $\mathfrak{P}$ and transitions t in $\mathfrak{A}$ in the TSI $\mathfrak{T}$. The subformula set $Sub(\phi)$ of an SFL formula ϕ is defined by the Fischer–Ladner closure of SFL formulae in the standard way.

A play is a possibly infinite sequence of configurations $C_0, C_1, ...$, written as $(s, R, t) \vdash \phi$ or $H \vdash \phi$ whenever possible; each C_i is an element of the board $\mathfrak{B}$. Every play starts in the configuration $C_0 = H_0 \vdash \phi$, and proceeds according to the rules of the game given in Fig. 1. As usual for model-checking games, player $\exists$ tries to prove that $H_0 \models \phi$ whereas player $\forall$ tries to show that $H_0 \not\models \phi$.

The rules (FP) and (VAR) control the unfolding of fixpoint operators. Their correctness is based on the fact that $\sigma Z.\phi \equiv \phi[\sigma Z.\phi/Z]$ according to the semantics of the logic. Rules ($\vee$) and ($\wedge$) have the same meaning as the disjunction and conjunction rules, respectively, in a Hintikka game for propositional logic. Rules ($\langle\ \rangle_c$), ($\langle\ \rangle_{nc}$), ($[\ ]_c$) and ($[\ ]_{nc}$) are like the rules for quantifiers in a standard Hintikka game semantics for first-order (FO) logic, provided that the box and diamond operators behave, respectively, as restricted universal and existential quantifiers sensitive to the causal information in the partial order model.

Finally, the most interesting rules are ($*$) and ($\bowtie$). Local monadic second-order moves are used to recognize conflict-free sets of transitions in $\mathfrak{M}$, i.e., those in the same *trace*. Such moves, which restrict the second-order power (locally) to traces, give the name to this game. The use of ($*$) and ($\bowtie$) requires both players to make a choice, but at different levels and with different amount of knowledge. The first player must look for two non-empty conflict-free sets of transitions, with no information on which formula ϕ_i the other player will choose afterwards.

Guided by the semantics of $*$ (resp. $\bowtie$), it is defined that player $\exists$ (resp. $\forall$) must look for a pair of non-empty conflict-free sets of transitions R_0 and R_1 to be assigned to each formula ϕ_i as their support sets. This situation is equivalent to playing a trace for each subformula in the configuration. Then player $\forall$ (resp. $\exists$) must choose one of the two subformulae, with full knowledge of the sets that have been given by player $\exists$ (resp. $\forall$). It is easy to see that $*$ should be regarded as a special kind of conjunction and $\bowtie$ of disjunction. Indeed, they are a structural conjunction and disjunction, respectively.

(FP)	$\dfrac{H \vdash \sigma Z.\phi}{H \vdash Z}$	$\sigma \in \{\mu, \nu\}$	(VAR)	$\dfrac{H \vdash Z}{H \vdash \phi}$	$fp(Z) = \sigma Z.\phi$
$(\vee)$	$\dfrac{H \vdash \phi_0 \vee \phi_1}{H \vdash \phi_i}$	$[\exists]\, i: \quad i \in \{0,1\}$	$(\wedge)$	$\dfrac{H \vdash \phi_0 \wedge \phi_1}{H \vdash \phi_i}$	$[\forall]\, i: \quad i \in \{0,1\}$
$(\langle\ \rangle_c)$	$\dfrac{(s,R,t) \vdash \langle K\rangle_c \phi}{(s', R'_{\max}(s'), t') \vdash \phi}$	$[\exists]\, b: \quad b \in K, s \xrightarrow{b} s' = t' \in R, t \leq t'$			
$(\langle\ \rangle_{nc})$	$\dfrac{(s,R,t) \vdash \langle K\rangle_{nc} \phi}{(s', R'_{\max}(s'), t') \vdash \phi}$	$[\exists]\, b: \quad b \in K, s \xrightarrow{b} s' = t' \in R, t \ominus t'$			
$([\ ]_c)$	$\dfrac{(s,R,t) \vdash [K]_c\, \phi}{(s', R'_{\max}(s'), t') \vdash \phi}$	$[\forall]\, b: \quad b \in K, s \xrightarrow{b} s' = t' \in R, t \leq t'$			
$([\ ]_{nc})$	$\dfrac{(s,R,t) \vdash [K]_{nc}\, \phi}{(s', R'_{\max}(s'), t') \vdash \phi}$	$[\forall]\, b: \quad b \in K, s \xrightarrow{b} s' = t' \in R, t \ominus t'$			
$(*)$	$\dfrac{(s,R,t) \vdash \phi_0 * \phi_1}{(s,R_i,t) \vdash \phi_i}$	$[\exists]\, R_0, R_1; [\forall]\, i: \quad R_0 \uplus R_1 \sqsubseteq R, i \in \{0,1\}$			
$(\bowtie)$	$\dfrac{(s,R,t) \vdash \phi_0 \bowtie \phi_1}{(s,R_i,t) \vdash \phi_i}$	$[\forall]\, R_0, R_1; [\exists]\, i: \quad R_0 \uplus R_1 \sqsubseteq R, i \in \{0,1\}$			

Fig. 1. Trace LMSO Model-Checking Game Rules of SFL. Whereas the notation $[\forall]$ denotes a choice made by Player $\forall$, the notation $[\exists]$ denotes a choice by Player $\exists$.

Definition 5. The following rules are the *winning conditions* that determine a unique winner for every finite or infinite play $C_0, C_1, ...$ in a game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$.

Player $\forall$ wins a finite play $C_0, C_1, ..., C_n$ or an infinite play $C_0, C_1, ...$ iff:

1. $C_n = H \vdash Z$ and $H \notin \mathcal{V}(Z)$.
2. $C_n = (s,R,t) \vdash \langle K\rangle_c \psi$ and $\{(s', R', t') : b \in K \wedge t \leq t' = s \xrightarrow{b} s' \in R\} = \emptyset$.
3. $C_n = (s,R,t) \vdash \langle K\rangle_{nc} \psi$ and $\{(s', R', t) : b \in K \wedge t \ominus t' = s \xrightarrow{b} s' \in R\} = \emptyset$.
4. $C_n = (s,R,t) \vdash \phi_0 * \phi_1$ and $\{R_0 \cup R_1 : R_0 \uplus R_1 \sqsubseteq R\} = \emptyset$.
5. The play is infinite and there are infinitely many configurations where Z appears, such that $fp(Z) = \mu Z.\psi$ for some formula ψ and Z is the syntactically outermost variable in ϕ that occurs infinitely often.

Player $\exists$ wins a finite play $C_0, C_1, ..., C_n$ or an infinite play $C_0, C_1, ...$ iff:

1. $C_n = H \vdash Z$ and $H \in \mathcal{V}(Z)$.
2. $C_n = (s,R,t) \vdash [K]_c\, \psi$ and $\{(s', R', t') : b \in K \wedge t \leq t' = s \xrightarrow{b} s' \in R\} = \emptyset$.
3. $C_n = (s,R,t) \vdash [K]_{nc}\, \psi$ and $\{(s', R', t) : b \in K \wedge t \ominus t' = s \xrightarrow{b} s' \in R\} = \emptyset$.
4. $C_n = (s,R,t) \vdash \phi_0 \bowtie \phi_1$ and $\{R_0 \cup R_1 : R_0 \uplus R_1 \sqsubseteq R\} = \emptyset$.
5. The play is infinite and there are infinitely many configurations where Z appears, such that $fp(Z) = \nu Z.\psi$ for some formula ψ and Z is the syntactically outermost variable in ϕ that occurs infinitely often.

4 Soundness and Completeness

Let us first give some intermediate results. Due to lack of space some proofs are omitted or sketched. Let $\mathfrak{T}$ be a TSI and $C = (s, R, t) \vdash \psi$ a configuration in the game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$, as defined before. As usual, the denotation $\|\phi\|^{\mathfrak{T}}_{\mathcal{V}}$ of an SFL formula ϕ in the model $\mathfrak{M} = (\mathfrak{T}, \mathcal{V})$ is a subset of $\mathfrak{S}$. We say that a configuration C of $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$ is *true* iff $(s, R, t) \in \|\psi\|^{\mathfrak{T}}_{\mathcal{V}}$ and *false* otherwise.

Fact 1. *SFL is closed under negation.*

Lemma 1. *A game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$, where player $\exists$ has a winning strategy, has a dual game $\mathcal{G}_{\mathfrak{M}}(H_0, \neg\phi)$ where player $\forall$ has a winning strategy, and conversely.*

Proof. We use Fact 1 and duality and completeness of winning conditions. $\square$

Lemma 2. *Player $\exists$ preserves falsity and can preserve truth with her choices. Player $\forall$ preserves truth and can preserve falsity with his choices.*

Proof. The cases for the rules $(\wedge)$ and $(\vee)$ are just as for the Hintikka evaluation games for FO logic. Thus, let us go on to check the rules for the other operators. Firstly, consider the rule $(\langle\ \rangle_c)$ and a configuration $C = (s, R, t) \vdash \langle K \rangle_c \psi$, and suppose that C is false. In this case there is no $b \in K$ such that $t \leq t' = s \xrightarrow{b} s' \in R$ and $(s', R'_{\max}(s'), t') \in \|\psi\|^{\mathfrak{T}}_{\mathcal{V}}$. Hence, the following configurations will be false as well. Contrarily, if C is true, then player $\exists$ can make the next configuration $(s', R'_{\max}(s'), t') \vdash \psi$ true by choosing a transition $t' = s \xrightarrow{b} s' \in R$ such that $t \leq t'$. The case for $(\langle\ \rangle_{nc})$ is similar (simply change $\leq$ for $\ominus$), and the cases for $([\ ]_c)$ and $([\ ]_{nc})$ are dual. Now, consider the rule $(*)$ and a configuration $C = (s, R, t) \vdash \psi_0 * \psi_1$, and suppose that C is false. In this case there is no pair of sets R_0 and R_1 such that $R_0 \uplus R_1 \sqsubseteq R$ and both $(s, R_0, t) \in \|\psi_0\|^{\mathfrak{T}}_{\mathcal{V}}$ and $(s, R_1, t) \in \|\psi_1\|^{\mathfrak{T}}_{\mathcal{V}}$ to be chosen by player $\exists$. Hence, player $\forall$ can preserve falsity by choosing the $i \in \{0, 1\}$ where $(s, R_i, t) \notin \|\psi_i\|^{\mathfrak{T}}_{\mathcal{V}}$, and the next configuration $(s, R_i, t) \vdash \psi_i$ will be false as well. On the other hand, suppose that C is true. In this case, regardless of which i player $\forall$ chooses, player $\exists$ has previously fixed two support sets R_0 and R_1 such that for every $i \in \{0, 1\}$, $(s, R_i, t) \in \|\psi_i\|^{\mathfrak{T}}_{\mathcal{V}}$. Therefore, the next configuration $(s, R_i, t) \vdash \psi_i$ will be true as well. Finally, the deterministic rules (FP) and (VAR) preserve both truth and falsity because of the semantics of fixpoint operators. Recall that for any process H, if $H \in \|\sigma Z.\psi\|$ then $H \in \|\psi\|_{Z:=\|\sigma Z.\psi\|}$ for all free variables Z in ψ. $\square$

Lemma 3. *In any infinite play of a game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$ there is a unique syntactically outermost variable that occurs infinitely often.*

Proof. By a contradiction argument and following an analysis of the structure of those formulae that appear in the configurations of infinite plays. $\square$

Fact 2. *Only rule* (VAR) *can increase the size of a formula in a configuration. All other rules decrease the size of formulae in configurations.*

Lemma 4. *Every play of a game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$ has a uniquely determined winner.*

Proof. For finite plays follows from winning conditions one to four (Definition 5). For plays of infinite length, by analysing the unfolding of fixpoints and winning conditions five of both players. We use Fact 2 and Lemma 3 in this case. □

Definition 6. ***(Approximants).*** *Let $fp(Z) = \mu Z.\phi$ for some formula ϕ and let $\alpha, \lambda \in \mathbb{O}\mathrm{rd}$ be two ordinals, where λ is a limit ordinal. Then:*

$$Z^0 := \mathrm{ff}, \qquad Z^{\alpha+1} = \phi\,[Z^\alpha/Z], \qquad Z^\lambda = \bigvee_{\alpha<\lambda} Z^\alpha$$

For greatest fixpoints the approximants are defined dually. We can now show that the analysis for fixpoint modal logics [1] can be extended to this scenario.

Theorem 1. ***(Soundness).*** *Let $\mathfrak{T}$ be the TSI in the model $\mathfrak{M} = (\mathfrak{T}, \mathcal{V})$ of a formula ϕ in the game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$. If $H_0 \notin \|\phi\|^{\mathfrak{T}}_{\mathcal{V}}$ then player $\forall$ wins $H_0 \vdash \phi$.*

Proof. Suppose $H_0 \notin \|\phi\|^{\mathfrak{T}}_{\mathcal{V}}$. We construct a possibly infinite game tree that starts in $H_0 \vdash \phi$, for player $\forall$. We do so by preserving falsity according to Lemma 2, i.e., whenever a rule requires player $\forall$ to make a choice then the tree will contain the successor configuration that preserves falsity. All other choices that are available for player $\exists$ are included in the game tree.

First, consider only finite plays. Since player $\exists$ only wins finite plays that end in true configurations, then she cannot win any finite play by using her winning conditions one to four. Hence, player $\forall$ wins each finite play in this game tree.

Now, consider infinite plays. The only chance for player $\exists$ to win is to use her winning condition five. So, let the configuration $H \vdash \nu Z.\phi$ be reached such that Z is the syntactically outermost variable that appears infinitely often in the play according to Lemma 3. In the next configuration $H \vdash Z$, variable Z is interpreted as the least approximant Z^α such that $H \notin \|Z^\alpha\|^{\mathfrak{T}}_{\mathcal{V}}$ and $H \in \|Z^{\alpha-1}\|^{\mathfrak{T}}_{\mathcal{V}}$, by the principle of fixpoint induction. As a matter of fact, by monotonicity and due to the definition of fixpoint approximants it must also be true that $H \in \|Z^\beta\|^{\mathfrak{T}}_{\mathcal{V}}$ for all ordinals β such that $\beta < \alpha$. Note that, also due to the definition of fixpoint approximants, α cannot be a limit ordinal λ because this would mean that $H \notin \|Z^\lambda = \bigwedge_{\beta<\lambda} Z^\beta\|^{\mathfrak{T}}_{\mathcal{V}}$ and $H \in \|Z^\beta\|^{\mathfrak{T}}_{\mathcal{V}}$ for all $\beta < \lambda$, which is impossible.

Since Z is the outermost variable that occurs infinitely often and the game rules follow the syntactic structure of formulae, the next time that a configuration $C' = H' \vdash Z$ is reached, Z can be interpreted as $Z^{\alpha-1}$ in order to make C' false as well. And again, if $\alpha - 1$ is a limit ordinal λ, there must be a $\gamma < \lambda$ such that $H' \notin \|Z^\gamma\|^{\mathfrak{T}}_{\mathcal{V}}$ and $H' \in \|Z^{\gamma-1}\|^{\mathfrak{T}}_{\mathcal{V}}$. One can repeat this process even until $\lambda = \omega$.

But, since ordinals are well-founded the play must eventually reach a false configuration $C'' = H'' \vdash Z$ where Z is interpreted as Z^0. And, according to Definition 6, $Z^0 := \mathrm{tt}$, which leads to a contradiction since the configuration $C'' = H'' \vdash \mathrm{tt}$ should be false, i.e., $H'' \in \|\mathrm{tt}\|^{\mathfrak{T}}_{\mathcal{V}}$ should be false, which is impossible. In other words, if H had failed a maximal fixpoint, then there must have been a descending chain of failures, but, as can be seen, there is not.

As a consequence, there is no such least α that makes the configuration $H \vdash Z^\alpha$ false, and hence, the configuration $H \vdash \nu Z.\phi$ could not have been false either.

Therefore, player $\exists$ cannot win any infinite play with her winning condition 5 either. Since player $\exists$ can win neither finite plays nor infinite ones whenever $H_0 \notin \|\phi\|^{\mathfrak{T}}_{\mathcal{V}}$, then player $\forall$ must win all plays of $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$. $\square$

Remark 2. *If only finite state systems are considered* $\mathbb{O}$rd, *the set of ordinals, can be replaced by* $\mathbb{N}$, *the set of natural numbers.*

Theorem 2. ***(Completeness).*** *Let $\mathfrak{T}$ be the TSI in the model $\mathfrak{M} = (\mathfrak{T}, \mathcal{V})$ of a formula ϕ in the game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$. If $H_0 \in \|\phi\|^{\mathfrak{T}}_{\mathcal{V}}$ then player $\exists$ wins $H_0 \vdash \phi$.*

Proof. Suppose that $H_0 \in \|\phi\|^{\mathfrak{T}}_{\mathcal{V}}$. Due to Fact 1 it is also true that $H_0 \notin \|\neg\phi\|^{\mathfrak{T}}_{\mathcal{V}}$. According to Theorem 1, player $\forall$ wins $H_0 \vdash \neg\phi$, i.e., has a winning strategy in the game $\mathcal{G}_{\mathfrak{M}}(H_0, \neg\phi)$. And, due to Lemma 1, player $\exists$ has a winning strategy in the dual game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$. Therefore, player $\exists$ wins $H_0 \vdash \phi$ if $H_0 \in \|\phi\|^{\mathfrak{T}}_{\mathcal{V}}$. $\square$

Theorems 1 and 2 imply that the game is determined. Determinacy and perfect information make the notion of truth defined by this Hintikka game semantics coincide with its Tarskian counterpart.

Corollary 1. ***(Determinacy).*** *Player $\forall$ wins the game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$ iff player $\exists$ does not win the game $\mathcal{G}_{\mathfrak{M}}(H_0, \phi)$.*

5 Local Properties and Decidability

We have shown that trace LMSO model-checking games are still sound and complete even when players are allowed to manipulate sets of independent transitions. Importantly, the power of these games, and also of SFL, is that such a second-order quantification is kept both *local* and restricted to transitions in the same *trace*. We now show that trace LMSO model-checking games enjoy several local properties that in turn make them *decidable* in the finite case. Such a decidable result is used in the forthcoming sections to extend the decidability border of model-checking a category of partial order models of concurrency.

Proposition 1. ***(Winning strategies).*** *The winning strategies for the trace LMSO model-checking games of Separation Fixpoint Logic are history-free.*

Proof. Consider a winning strategy π for player $\exists$. According to Lemma 2 and Theorem 2 such a strategy consists of preserving truth with her choices and annotating variables with their approximant indices. But neither of these two tasks depends on the history of a play. Instead they only depend on the current configuration of the game. In particular notice that, of course, this is also the case for the structural operators since the second-order quantification has only a local scope. Similar arguments apply for the winning strategies of player $\forall$. $\square$

This result is key to achieve *decidability* of these games in the presence of the local second-order quantification on the traces of the partial order models we consider. Also, from a more practical standpoint, memoryless strategies are desirable as they are easier to synthesize. However, synthesis is not studied here.

Theorem 3. *The model-checking game for finite Transition systems with independence against Separation Fixpoint Logic specifications is decidable.*

Proof. Since the game is determined, finite plays are decided by winning conditions one to four of either player. Now consider the case of plays of infinite length; since the winning strategies of both players are history-free, we only need to look at the set of different configurations in the game, which is finite even for plays of infinite length. Now, in a finite system an infinite play can only be possible if the model is cyclic. But, since the model has a finite number of states, there is an upper bound on the number of fixpoint approximants that must be calculated (as well as on the number of configurations of the game board that must be checked) in order to ensure that either a greatest fixpoint is satisfied or a least fixpoint has failed. As a consequence, all possible history-free winning strategies for a play of infinite length can be computed, so that the game can be decided using winning condition five of one of the players. □

Remark 3. *A naive local tableau algorithm is at least doubly exponential in the system size, but applying global model-checking techniques, a formula of length k and alternation depth d on a system of size n can be decided in time $k.2^{\mathcal{O}(nd)}$.*

The Interleaving Case. Local properties of trace LMSO model-checking games can also be found in the interleaving case, namely, they coincide with the local model-checking games for the modal μ-calculus as defined by Stirling [9]. As shown in [3] interleaving systems can be cast using SFL by both syntactic and semantic means. The importance of this feature of SFL is that even having constructs for independence and a partial order model, nothing is lost with respect to the main approaches to interleaving concurrency. Recall that $L\mu$ can be obtained from SFL by considering the $*$-free language and using only the following derived operators: $\langle K\rangle\phi = \langle K\rangle_c\phi \vee \langle K\rangle_{nc}\phi$ and $[K]\,\phi = [K]_c\,\phi \wedge [K]_{nc}\,\phi$.

Proposition 2. *If either a model with an empty independence relation or the syntactic $L\mu$ fragment of SFL is considered, then the trace LMSO model-checking games for SFL degenerate to the local model-checking games for the μ-calculus.*

6 Model-Checking Partial Order Models of Concurrency

In this section we use trace LMSO model-checking games to push forward the decidability border of the model-checking problem of a particular class of partial order models, namely, of a class of event structures [6]. More precisely, we improve previous results [5,7] in terms of logical expressive power.

6.1 SFL on Event Structures

Definition 7. A *labelled event structure* $\mathfrak{E}$ is a tuple $(E, \preccurlyeq, \sharp, \eta, \Sigma)$, where E is a set of events that are partially ordered by $\preccurlyeq$, the causal dependency relation on events, $\sharp \subseteq E \times E$ is an irreflexive and symmetric conflict relation, and $\eta : E \rightarrow \Sigma$ is a labelling function such that the following holds:

If $e_1, e_2, e_3 \in E$ and $e_1 \sharp e_2 \preccurlyeq e_3$, then $e_1 \sharp e_3$.
$\forall e \in E$ the set $\{e' \in E \mid e' \preccurlyeq e\}$ is finite.

The independence relation on events is defined with respect to the causal and conflict relations. Two events e_1 and e_2 are *concurrent*, denoted by $e_1 \ \mathsf{co}\ e_2$, iff $e_1 \not\preccurlyeq e_2$ and $e_2 \not\preccurlyeq e_1$ and $\neg(e_1 \sharp e_2)$. The notion of computation state for event structures is that of a *configuration*. A configuration C is a conflict-free set of events (i.e., if $e_1, e_2 \in C$, then $\neg(e_1 \sharp e_2)$) such that if $e \in C$ and $e' \preccurlyeq e$, then $e' \in C$. The restriction to image-finite models implies that the partial order $\preccurlyeq$ of $\mathfrak{E}$ is of *finite branching*, and hence for all C, the set of immediately next configurations is bounded. If one further requires that for all $e \in C$, the set of future non-isomorphic configurations rooted at e defines an equivalence relation of *finite index*, then $\mathfrak{E}$ is also *regular* [11].

An event structure $\mathfrak{E} = (E, \preccurlyeq, \sharp, \eta, \Sigma)$ determines a TSI $\mathfrak{T} = (S, T, \Sigma, I)$ by means of an inclusion functor from the category $\mathcal{ES}$ of event structures to the category $\mathcal{TSI}$ of TSI. Here we give such a mapping in a set-theoretic way since this presentation is more convenient for us. A categorical presentation can be found in [4]. The construction $\lambda : \mathcal{ES} \rightarrow \mathcal{TSI}$ is as follows:

$$
\begin{aligned}
S &= \{C \subseteq E \mid \forall e_1, e_2 \in C.\ \neg(e_1 \sharp e_2), (e \in C \wedge e' \preccurlyeq e \Rightarrow e' \in C)\}\ . \\
T &= \{(C, a, C') \in S \times \Sigma \times S \mid \exists e \in E.\ \eta(e) = a, e \notin C, C' = C \cup \{e\}\} \\
I &= \{((C_1, a, C_1'), (C_2, b, C_2')) \in T \times T \mid \exists (e_1, e_2) \in \mathsf{co}. \\
&\qquad \eta(e_1) = a, \eta(e_2) = b, C_1' = C_1 \cup \{e_1\}, C_2' = C_2 \cup \{e_2\}\}
\end{aligned}
$$

where the set of states S of the TSI $\mathfrak{T}$ is isomorphic to the set *Conf* of *configurations* $C \subseteq E$ of the event structure $\mathfrak{E}$, and the set of labels Σ remains the same. Since the semantics of SFL is given only by defining the relations $\leq$, $\ominus$, $\#$ and $\otimes$ on pairs of transitions at every state, i.e., on pairs of events at every configuration, such a semantics can also be given directly from the elements of an event structure using the construction above. Moreover, support sets and all elements needed to build a lattice $\mathfrak{S}$ and hence a model for SFL in the category of event structures are defined using the same definitions as for the TSI case.

6.2 A Computable Folding Functor from Event Structures to TSI

Although we have defined satisfiability of SFL formulae in event structure models, model-checking these structures is rather difficult since very simple concurrent systems can have infinite event structures as models, in particular, all those with recursive behaviour. In order to overcome this problem we define a morphism (a functor) that folds a possibly infinite event structures into a TSI. Such a morphism and the procedure to effectively compute it is described below.

The Quotient Set Method. Let $Q = (\mathit{Conf}/\sim)$ be the *quotient set representation* of *Conf* by $\sim$ in a finite or infinite event structure $\mathfrak{E}$, where *Conf* is the set of configurations in $\mathfrak{E}$ and $\sim$ is an equivalence relation on such configurations. The equivalence class $[X]_\sim$ of a configuration $X \in \mathit{Conf}$ is the set $\{C \in \mathit{Conf} \mid C \sim X\}$. A quotient set Q where $\sim$ is decidable is said to have a decidable characteristic function, and will be called a *computable quotient set*.

Definition 8. A *regular quotient set* $(Conf/\sim)$ of an event structure $\mathfrak{E}$ is a computable quotient set representation of $\mathfrak{E}$ with a finite number of equivalence classes.

Having defined a regular quotient set representation of $\mathfrak{E}$, the morphism $\lambda : \mathcal{ES} \to \mathcal{TSI}$ above can be modified to defined a new map $\lambda_f : \mathcal{ES} \to \mathcal{TSI}$ which folds a (possibly infinite) event structure into a TSI:

$$\begin{aligned} S &= \{[C]_\sim \subseteq Conf \mid \exists [X]_\sim \in Q = (Conf/\sim).\ C \sim X\} \\ T &= \{([C]_\sim, a, [C']_\sim) \in S \times \Sigma \times S \mid \exists e \in E.\ \eta(e) = a, e \notin C, C' = C \cup \{e\}\} \\ I &= \{(([C_1]_\sim, a, [C_1']_\sim), ([C_2]_\sim, b, [C_2']_\sim)) \in T \times T \mid \exists (e_1, e_2) \in \mathsf{co}. \\ &\quad\ \eta(e_1) = a, \eta(e_2) = b, C_1' = C_1 \cup \{e_1\}, C_2' = C_2 \cup \{e_2\}\} \end{aligned}$$

Lemma 5. *Let $\mathfrak{T}$ be a TSI and $\mathfrak{E}$ an event structure. If $\mathfrak{T} = \lambda_f(\mathfrak{E})$, then the models $(\mathfrak{T}, \mathcal{V})$ and $(\mathfrak{E}, \mathcal{V})$ satisfy the same set of SFL formulae.*

Proof. The morphism $\lambda_f : \mathcal{ES} \to \mathcal{TSI}$ from the category of event structures to the category of TSI has a unique right adjoint $\varepsilon : \mathcal{TSI} \to \mathcal{ES}$, the unfolding functor that preserves labelling and the independence relation between events, such that for any $\mathfrak{E}$ we have that $\mathfrak{E}' = (\varepsilon \circ \lambda_f)\ (\mathfrak{E})$, where $\mathfrak{E}'$ is isomorphic to $\mathfrak{E}$. But SFL formulae do not distinguish between models and their unfoldings, and hence cannot distinguish between $(\mathfrak{T}, \mathcal{V})$ and $(\mathfrak{E}', \mathcal{V})$. Moreover, SFL formulae do not distinguish between isomorphic models equally labelled, and therefore cannot distinguish between $(\mathfrak{E}', \mathcal{V})$ and $(\mathfrak{E}, \mathcal{V})$ either. $\square$

Having defined a morphism λ_f that preserves SFL properties, one can now define a procedure that constructs a TSI model from a given event structure.

Definition 9. Let $\mathfrak{E} = (E, \preccurlyeq, \sharp, \eta, \Sigma)$ be an event structure and $(Conf/\sim)$ a regular quotient set representation of $\mathfrak{E}$. A *representative set E_r* of $\mathfrak{E}$ is a subset of E such that $\forall C \in Conf.\ \exists X \subseteq E_r.\ C \sim X$.

Lemma 6. *Let $\mathfrak{E}$ be an event structure. If $\mathfrak{E}$ is represented as a regular quotient set $(Conf/\sim)$, then a finite representative set E_r of $\mathfrak{E}$ is effectively computable.*

Proof. Construct a finite representative set E_r as follows. Start with $E_r = \emptyset$ and $C_j = C_0 = \emptyset$, the initial configuration or root of the event structure. Check $C_j \sim X_i$ for every equivalence class $[X_i]_\sim$ in $Q = (Conf/\sim)$ and whenever $C_j \sim X_i$ holds define both a new quotient set $Q' = Q \setminus [X_i]_\sim$ and a new $E_r = E_r \cup C_j$. This subprocedure terminates because there are only finitely many equivalence classes to check and the characteristic function of the quotient set is decidable. Now, do this recursively in a breadth-first search fashion in the partial order defined on E by $\preccurlyeq$, and stop when the quotient set is empty. Since $\preccurlyeq$ is of finite branching and all equivalence classes must have finite configurations, the procedure is bounded both in depth and breath and the quotient set will always eventually get smaller. Hence, such a procedure always terminates. It is easy to see that this procedure only terminates when E_r is a representative set of $\mathfrak{E}$. $\square$

A finite representative set E_r is big enough to define all states in the TSI representation of $\mathfrak{E}$ when using λ_f. However, such a set may not be enough to recognize all transitions in the TSI. In particular, cycles cannot be recognized using E_r. Therefore, it is necessary to compute a set E_f where cycles in the TSI can be recognized. We call E_f a *complete representative set* of $\mathfrak{E}$. The procedure to construct E_f is similar to the previous one.

Lemma 7. *Let $\mathfrak{E} = (E, \preccurlyeq, \sharp, \eta, \Sigma)$ be an event structure and E_r a finite representative set of $\mathfrak{E}$. If $\mathfrak{E}$ is represented as a regular quotient set $(Conf / \sim)$, then a finite complete representative set E_f of $\mathfrak{E}$ is effectively computable.*

Proof. Start with $E_f = E_r$, and set $\mathfrak{C} = Conf(E_r)$, the set of configurations generated by E_r. For each C_j in E_r check in $\preccurlyeq$ the set $Next(C_j)$ of next configurations to C_j, i.e., those configurations C'_j such that $C'_j = C_j \cup \{e\}$ for some event e in $E \setminus C_j$. Having computed $Next(C_j)$, set $E_f = E_f \cup (\bigcup Next(C_j))$ and $\mathfrak{C} = \mathfrak{C} \setminus \{C_j\}$, and stop when $\mathfrak{C}$ is empty. This procedure behaves as the one described previously. Notice that at the end of this procedure E_f is complete since it contains the next configurations of all elements in E_r. □

Proposition 3. *The TSI $\mathfrak{T}$ generated from an event structure $\mathfrak{E}$ using λ_f and a finite complete representative E_f of $\mathfrak{E}$ is the smallest TSI that represents $\mathfrak{E}$.*

Proof. From Lemmas 6 and 7. There is only one state in $\mathfrak{T}$ for each equivalence class in the quotient set representation of $\mathfrak{E}$. Similarly there can be only one transition in $\mathfrak{T}$ for each relation on the equivalence classes of configurations in $\mathfrak{E}$ since, due to **A1** of TSI (determinacy), λ_f forgets repeated transitions in T. □

6.3 Temporal Verification of Regular Infinite Event Structures

Based on Lemmas 5 and 7 and on Theorem 3, we can give a decidability result for the class of event structures studied in [5,11] against SFL specifications. Such a result, which is obtained by representing a regular event structure as a regular quotient set, is a corollary of the following theorem:

Theorem 4. *The model-checking problem for an event structure $\mathfrak{E}$ represented as a regular quotient set $(Conf / \sim)$ against SFL specifications is decidable.*

Regular Event Structures as Finite CCS Processes. A regular event structure [5,11] can be generated by a finite concurrent system represented by a finite number of (possibly recursive) CCS processes [12]. Syntactic restrictions on CCS that generate only finite systems have been studied. Finiteness of CCS processes and restriction to image-finite models give both requirements for regularity on the event structures that are generated. Now, w.l.o.g., consider only deterministic CCS processes without auto-concurrency. A CCS process is deterministic if whenever $a.M + b.N$, then $a \neq b$, and similarly has no auto-concurrency if whenever $a.M \parallel b.N$, then $a \neq b$. Notice that any CCS process P that either is nondeterministic or has auto-concurrency can be converted into an equivalent process Q which generates an event structure that is isomorphic, up to relabelling of events, to the one generated by P. Eliminating nondeterminism and auto-concurrency can be done by relabelling events in $\mathcal{P}(P)$, the powerset of CCS processes of P, with an injective map $\theta : \Sigma \rightarrow \Sigma^*$ (where Σ^* is a set of labels and $\Sigma \subseteq \Sigma^*$), and by extending the Synchronization Algebra according to the new labelling of events so as to preserve pairs of (labels of) events that can synchronize. Also notice that the original labelling can always be recovered from the new one, i.e., the one associated with the event structure generated by Q, since θ is injective and hence has inverse $\theta^{-1} : \Sigma^* \rightarrow \Sigma$. In [5,11], deterministic regular event structures are called *trace* event structures.

Finite CCS Processes as Regular Quotient Sets. Call $ESProc(P)$ to the set of configurations of the event structure generated by a CCS process P of the kind described above. The set $ESProc(P)$ together with an equivalence relation between CCS processes $\equiv_{CCS}$ given simply by syntactic equality between them is a regular quotient set representation $(ESProc(P) \ / \ \equiv_{CCS})$ of the event structure generated by P. Notice that since there are finitely many CCS processes, i.e., $\mathcal{P}(P)$ is finite, then the event structure generated by P is of finite-branching and the number of equivalence classes is also bounded. Finally, $\equiv_{CCS}$ is clearly decidable because process P is always associated with configuration $\emptyset$ and any other configuration in $ESProc(P)$ can be associated with only one CCS process in $\mathcal{P}(P)$ as they are deterministic and have no auto-concurrency after relabelling.

Corollary 2. *Model-checking regular trace event structures against Separation Fixpoint Logic specifications is decidable.*

7 Concluding Remarks and Related Work

In this paper we introduced a new kind of model-checking games where both players are allowed to choose *sets* of independent elements in the underlying model. These games, which we call trace LMSO model-checking games, are proved to be *sound* and *complete*, and therefore determined. They can be played on partial order models of concurrency since the one-step interleaving semantics of such models need not be considered. We showed that, similar to [3], by defining infinite games where both players have a *local* second-order power on *conflict-free* sets of transitions, i.e., those in the same *trace*, one can obtain new positive decidability results on the study of partial order models of concurrency. Indeed, we have pushed forward the borderline of the decidability of model-checking event structures. To the best of our knowledge the technique we presented here is the only game-based procedure defined so far that can be used to verify all usual temporal properties of the kind of event structures we studied. We wonder how much further one can go in terms of logical expressive power before reaching the MSO undecidability barrier when model-checking event structures.

Related Work. Model-checking games have been an active area of research in the last decades (cf. [2,10]). Most approaches based on games have considered either only interleaving models or the one-step interleaving semantics of partial order models. Our work differs from these approaches in that we deal with games played on partial order models without considering interleaving simplifications. However, verification procedures in finite partial order models can be undecidable. Nevertheless, the game presented here is *decidable* in the finite case.

Regarding the temporal verification of event structures, previous studies have been done on restricted classes. Closer to our work is [5,7]. Indeed, model-checking regular event structures [11] has turned out to be rather difficult and previous work has shown that verifying MSO properties on these structures is already *undecidable*. For this reason weaker logics have been studied. Unfortunately, although very interesting results have been achieved, especially in [5] where CTL* properties can be verified, previous approaches have not managed to define decidable theories for a logic with enough power to express all usual temporal properties as can be done with $L\mu$ in the interleaving case, and hence with SFL in a noninterleaving setting. The difference between [5] and the approach we presented is that in [5] a *global* second-order quantification on conflict-free sets in the partial order is permitted, whereas only a *local* second-order quantification in the same kind of sets is defined here, but such a second-order power can be embedded into fixpoint specifications, which in turn allows one to express more temporal properties. Therefore, we have improved in terms of temporal expressive power previous results on model-checking regular event structures against a branching-time logic. Our work is the first (local) game approach in doing so.

References

1. Bradfield, J., Stirling, C.: Modal mu-calculi. In: Handbook of Modal Logic, vol. 3, pp. 721–756. Elsevier, Amsterdam (2006)
2. Grädel, E.: Model checking games. Electr. Notes Theor. Comput. Sci. 67 (2002)
3. Gutierrez, J.: Logics and bisimulation games for concurrency, causality and conflict. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 48–62. Springer, Heidelberg (2009)
4. Joyal, A., Nielsen, M., Winskel, G.: Bisimulation from open maps. Inf. Comput. 127(2), 164–185 (1996)
5. Madhusudan, P.: Model-checking trace event structures. In: LICS, pp. 371–380. IEEE Computer Society, Los Alamitos (2003)
6. Nielsen, M., Winskel, G.: Models for concurrency. In: Handbook of Logic in Computer Science, vol. 4, pp. 1–148. Oxford University Press, Oxford (1995)
7. Penczek, W.: Model-checking for a subclass of event structures. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 145–164. Springer, Heidelberg (1997)
8. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
9. Stirling, C.: Local model checking games. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 1–11. Springer, Heidelberg (1995)
10. Stirling, C.: Modal and Temporal Properties of Processes. LNCS. Springer, Heidelberg (2001)
11. Thiagarajan, P.S.: Regular trace event structures. Technical report, BRICS (1996)
12. Winskel, G.: Event structure semantics for ccs and related languages. In: Nielsen, M., Schmidt, E.M. (eds.) ICALP 1982. LNCS, vol. 140, pp. 561–576. Springer, Heidelberg (1982)

Reachability in Succinct and Parametric One-Counter Automata

Christoph Haase, Stephan Kreutzer, Joël Ouaknine, and James Worrell

Oxford University Computing Laboratory, UK
{chrh,kreutzer,joel,jbw}@comlab.ox.ac.uk

Abstract. One-counter automata are a fundamental and widely-studied class of infinite-state systems. In this paper we consider one-counter automata with counter updates encoded in binary—which we refer to as the succinct encoding. It is easily seen that the reachability problem for this class of machines is in PSPACE and is NP-hard. One of the main results of this paper is to show that this problem is in fact in NP, and is thus NP-complete.

We also consider parametric one-counter automata, in which counter updates be integer-valued parameters. The reachability problem asks whether there are values for the parameters such that a final state can be reached from an initial state. Our second main result shows decidability of the reachability problem for parametric one-counter automata by reduction to existential Presburger arithmetic with divisibility.

1 Introduction

Counter automata are a fundamental computational model, known to be equivalent to Turing machines [19], and there has been considerable interest in subclasses of counter machines for which reachability is decidable, such as Petri nets, one-counter automata and flat counter automata [5,18]. As originally conceived by Minsky, counters are updated either by incrementation or decrementation instructions. However, for many applications of counter machines, including modelling computer programs, it is natural to consider more general types of updates, such as adding integer constants to a counter [3,5,16] or adding integer parameters [4,12]. Parametric automata are used in various synthesis problems, and to model open programs, whose behaviour depends on values input from the environment [2]. In [20] parameters are also used to model resources (e.g., time, memory, dollars) consumed by transitions. The reachability problem for parametric counter automata asks whether there exist values of the parameters such that a given configuration is reachable from another given configuration.

In this paper we show NP-completeness of the reachability problem for one-counter automata in which counters can be updated by adding integer constants, where the latter are encoded in binary. We also show decidability of reachability for parametric one-counter automata by reduction to existential Presburger arithmetic with divisibility [17]. We defer consideration of the complexity of the latter problem to the full version of this paper.

1.1 Related Work

The verification literature contains a large body of work on decidability and complexity for various problems on restricted classes of counter automata. The work that is closest to our own is that of Demri and Gascon on model checking extensions of LTL over one-counter automata [8]. They consider automata with one integer-valued counter, with updates encoded in unary, and with sign tests on the counter. They show that reachability in this model is NL-complete. Determining the complexity of reachability when updates are encoded in binary is posed as an open problem by Demri in [7], Page 61, Problem 13. Since this last problem assumes an integer-valued counter with sign tests, it is more general than the one considered in our Theorem 1, and it remains open.

Another work closely related to our own is that of Ibarra, Jiang, Tran and Wang [12], which shows decidability of reachability for a subset of the class of deterministic parametric one-counter automata with sign tests. The decidability of reachability over the whole class of such automata is stated as an open problem in [12]. Note that although we do not allow negative counter values and sign tests, we allow nondeterminism. Thus our Theorem 4 is incomparable with this open problem.

Aside from reachability, similarity and bisimilarity for one-counter automata and one-counter nets have been considered in [1,13,14], among others.

For automata with more than one counter, other restrictions are required to recover decidability of the reachability problem: for example, *flatness* [5,16] and *reversal boundedness* [11]. Bozga, Iosif and Lakhnech [4] show decidability of the reachability problem for flat parametric counter automata with a single loop, by reduction to a decidable problem concerning quadratic diophantine equations. Such systems of equations also feature in the work of Ibarra and Dang [11]. They exhibit a connection between a decidable class of quadratic diophantine equations and a class of counter automata with reversal-bounded counters.

2 One-Counter Automata

A **one-counter automaton** is a nondeterministic finite-state automaton acting on a single counter which takes values in the nonnegative integers. Formally a one-counter automaton is a tuple $\mathcal{C} = (V, E, \lambda)$, where V is a finite set of control locations, $E \subseteq V \times V$ is a finite set of transitions, and $\lambda : E \to Op$ is a function that assigns to each transition an operation from the set $Op = \{\mathsf{zero}\} \cup \{\mathsf{add}(a) : a \in \mathbb{Z}\}$. The operation zero represents a zero test on the counter, whereas $\mathsf{add}(a)$ denotes the operation of adding a to the value of the counter.

A **configuration** of the counter automaton $\mathcal{C}$ is a pair (v, c), where $v \in V$ is a control location and $c \in \mathbb{N}$ is the value of the counter. The transition relation on the locations of $\mathcal{C}$ induces an unlabelled transition relation on configurations in the obvious way: an edge $(v, v') \in E$ with labelled zero yields a single transition $(v, 0) \longrightarrow (v', 0)$, while the same edge with label $\mathsf{add}(a)$ yields a transition $(v, c) \longrightarrow (v', c + a)$, provided that both c and $c + a$ are both non-negative.

A **computation** π of a counter automaton $\mathcal{C}$ is a finite sequence of transitions between configurations

$$\pi = (v_0, c_0) \longrightarrow (v_1, c_1) \longrightarrow \cdots \longrightarrow (v_n, c_n) .$$

We define the length of π to be $\text{length}(\pi) = n$. We sometimes write $\pi : (v_0, c_0) \longrightarrow^* (v_n, c_n)$ or $(v_0, c_0) \xrightarrow{\pi} (v_n, c_n)$ to denote that π is a computation from (v_0, c_0) to (v_n, c_n).

The **reachability problem** asks, given a one-counter automaton $\mathcal{C}$ and configurations (v, c) and (v', c'), whether there is a computation starting in (v, c) and ending in (v', c'). The **control-state reachability problem** asks, given $\mathcal{C}$ and two locations v and v', whether there is a computation from $(v, 0)$ to (v', c') for some counter value c'. It is easily seen that both reachability problems are reducible to each other in logarithmic space.

In determining the complexity of these problems we assume a standard encoding of counter automata and their configurations—in particular, we suppose that integers are encoded in binary. Given a counter machine $\mathcal{C}$, we denote by $|\mathcal{C}|$ the length of the encoding of $\mathcal{C}$. It is easy to see that the shortest computation between two given locations may have length exponential in $|\mathcal{C}|$. For example, in the automaton in Figure 1 the unique path from $(v_0, 0)$ to $(v_1, 0)$ has length 2^n.

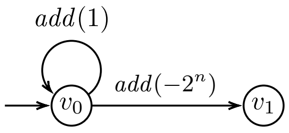


Fig. 1.

The first observation is the following.

Proposition 1. *The reachability problem for one-counter automata is* NP*-hard.*

Proof. The proof is by reduction from the SUBSET SUM problem [9]. Recall that an instance of the latter consists of a set of positive integers $S = \{a_1, a_2, \ldots, a_n\}$ and a goal c, and the question asked is whether there exists a subset $T \subseteq S$ such that $\sum T = c$. This reduces to the question of whether configuration (v_n, c) is reachable from $(v_0, 0)$ in the one-counter automaton in Figure 2. $\square$

Proposition 1 crucially depends on encoding integers in binary. Indeed, it follows from Proposition 2, below, that if integers are encoded in unary then the reachability problem becomes NL-complete, since it reduces to reachability in a polynomial-size graph.

The first main contribution of this paper is to establish an upper bound for the complexity of the reachability problem, matching the lower bound in Proposition 1.

Theorem 1. *The reachability problem for one-counter automata is in* NP*.*

The idea behind the proof of Theorem 1 is as follows. Suppose one is given a one-counter automaton $\mathcal{C}$, and two configurations (v, c) and (v', c'). One can show that if (v', c') is reachable from (v, c), then there is a computation π from (v, c)

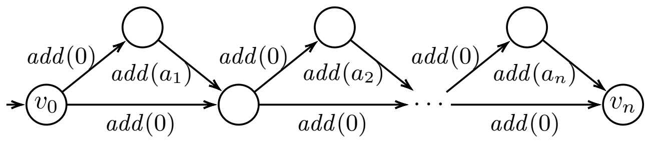


Fig. 2. Reduction from SUBSET SUM to reachability

to (v', c') whose length is bounded by an exponential function in $|\mathcal{C}|$ and the bit lengths of c and c'. This computation has a succinct certificate: a *network flow* which records for each edge of $\mathcal{C}$ how many times it is taken in π. This flow has a polynomial-size description, and so it can be guessed in polynomial time. The main difficulty in fleshing out this idea is the problem of how to validate such a certificate; that is, given a flow, to determine in polynomial time whether it arises from a valid computation of the counter machine. To solve this problem we define a subclass of such flows with certain structural properties, called *reachability certificates*. We show that validating reachability certificates can be done in NP, and that the computation π, above, can, without loss of generality, be divided into sub-computations, each of which generates a reachability certificate.

2.1 Parametric Counter Automata

A **parametric one-counter automaton** is a tuple $\mathcal{C} = (V, E, X, \lambda)$, where the sets V and E of vertices and edges are as in the definition of a one-counter automaton, X is a set of non-negative integer parameters, and the labelling function $\lambda : E \to Op$ has codomain

$$Op = \{\mathsf{zero}\} \cup \{\mathsf{add}(a), \mathsf{add}(x), \mathsf{add}(-x) : a \in \mathbb{Z}, x \in X\} .$$

The only difference with one-counter automata is the ability to add or subtract the value of a parameter $x \in X$ to the counter. Each instantiation of the parameters yields a different one-counter automaton.

The **reachability problem for parametric one-counter automata** asks, given configurations (v, c) and (v', c'), whether there exist values for the parameters such that there is a computation from (v, c) to (v', c'). We exhibit reductions in both directions between this problem and the satisfiability problem for the existential fragment of **Presburger arithmetic with divisibility**, i.e., the existential theory of the structure $(\mathbb{Z}, <, |, +, 0, 1)$, where $|$ is the binary *divides* predicate. Lipshitz [17] gave a procedure for deciding satisfiability of this logic. Thus we obtain our second main result.

Theorem 2. *The reachability problem for parametric one-counter automata is decidable.*

The reduction from existential Presburger arithmetic with divisibility to the reachability problem for parametric one-counter automata is fairly straightforward, and is detailed below. It follows a similar pattern to [2], which reduces existential Presburger arithmetic with divisibility to the reachability problem for two-clock parametric timed automata.

Let φ be a quantifier-free formula of Presburger arithmetic with divisibility. Without loss of generality, assume that φ is a positive Boolean combination of atomic formulas, $A = B$, $A < B$, $A \mid B$ and $\neg(A \mid B)$, where A and B are linear expressions in variables $x_1, \ldots, x_n$. By representing arbitrary integers as differences of positive integers we can also assume that the variables $x_1, \ldots, x_n$ range over the positive integers.

For each such atomic sub-formula ψ we construct a one-counter automaton $\mathcal{C}_\psi$, with parameters $x_1, \ldots, x_n$ and distinguished locations u and v, such that $(v, 0)$ is reachable from $(u, 0)$ iff ψ is satisfied. We can then combine the automata representing atomic sub-formulas using sequential composition to model conjunction and nondeterminism to model disjunction.

For an atomic formula $\psi \equiv A \mid B$, the automaton $\mathcal{C}_\psi$ first guesses the sign of A and B. Assume that A and B are guessed to be non-negative; the remaining cases are similar. In this case the automaton simply loads B into its counter and repeatedly subtracts A until the counter reaches 0.

For an atomic formula $\psi \equiv \neg(A \mid B)$ the automaton $\mathcal{C}_\psi$ first guesses the sign of A and B. Again, assume that A and B are non-negative. Then the automaton loads B into its counter and repeatedly subtracts A until the counter reaches a value strictly between 0 and A. It can be checked whether the counter is strictly between 0 and A by performing the following sequence of transitions: subtract one; add two; add one a nondeterministic number of times; subtract A; test for zero.

Handling the other atomic formulas is equally straightforward.

3 Weighted Graphs and Flow Networks

In this section we recall some standard definitions about weighted graphs and flow networks.

A **weighted graph** is a tuple $G = (V, E, w)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of directed edges, and $w : E \to \mathbb{Z}$ assigns an integer **weight** to each edge. Given such a graph and two distinguished vertices $s, t \in V$, a **path** π from s to t, also called an s-t path, is a sequence of vertices $\pi = v_0 v_1 \ldots v_n$ with $v_0 = s$, $v_n = t$ and $(v_i, v_{i+1}) \in E$ for $0 \leq i < n$. A path with the same first and last vertices is called a **cycle**. To indicate that π is an s-t path we often write $\pi : s \longrightarrow^* t$. If $\pi : s \longrightarrow^* t$ and $\pi' : t \longrightarrow^* u$, then $\pi \cdot \pi'$ denotes the path from s to u obtained by composing π and π'. Given a cycle ℓ on vertex v, we define $\ell^0 = v$ (the trivial cycle on v) and $\ell^{n+1} = \ell^n \cdot \ell$ for $n \in \mathbb{N}$.

The **weight** of a path π, denoted weight(π) is the sum of the weights of the edges in π. If ℓ is a cycle such that weight$(\ell) > 0$ then we say that ℓ is a **positive cycle**, and if weight$(\ell) < 0$ then we say that ℓ is a **negative cycle**.

Given a weighted graph $G = (V, E, w)$, with distinguished vertices s and t, a **flow** from s to t, also called an s-t flow, is a function $f : E \to \mathbb{N}$ satisfying the following flow conservation condition for each vertex $u \in V - \{s, t\}$:

$$\sum_{(v,u) \in E} f(v, u) = \sum_{(u,v) \in E} f(u, v)\,.$$

The **value** $|f|$ of flow f is the net flow out of the source s (equivalently the net flow into the sink t), that is,

$$|f| = \sum_{(s,u)\in E} f(s,u) - \sum_{(u,s)\in E} f(u,s)\,.$$

The **weight** of the flow f is defined to be

$$\mathrm{weight}(f) = \sum_{e\in E} f(e)\cdot w(e)\,.$$

An s-t path π determines an s-t flow f_π, where for each edge $e \in E$, $f(e)$ is defined to be the number of times edge e is taken in π. We call the class of flows that arise in this way **path flows**. Just as paths can be sequentially composed, path flows can be composed by summation: given an s-t path flow f and a t-u path flow g, we define a s-u path flow $f+g$ by $(f+g)(e) = f(e)+g(e)$ for each edge $e \in E$.

The **skew transpose** G^{op} of G is the weighted graph obtained by multiplying all edge weights by -1 and then reversing the direction of each edge. A path flow f from s to t in graph G induces a path flow f^{op} from t to s in G^{op}, where $f^{op}(u,v) = f(v,u)$.

4 Reachability Certificates

The following result [15, Lemma 42] shows that the reachability problem for one-counter machines is in PSPACE.

Proposition 2. *There is a polynomial P such that given a one-counter automaton $\mathcal{C}$ and configurations (v,c) and (v',c'), if (v,c) can reach (v',c') then there is computation from (v,c) to (v',c') of length at most $2^{P(n)}$, where n is the maximum of $|\mathcal{C}|$ and the bit lengths of c and c'.*

Let $\mathcal{C} = (V,E,\lambda)$ be a one-counter automaton. For proving NP-membership of the reachability problem, it is no loss of generality to assume that $\mathcal{C}$ has no zero tests. Indeed, since we may assume that each zero test is taken at most once, by guessing the order in which the zero tests are taken, a reachability query on a one-counter automaton with zero tests can be reduced to a linear number of reachability queries on the same automaton with zero tests erased. Now a one-counter automaton without zero tests is nothing but a weighted graph, where the weight of an edge labelled $\mathsf{add}(a)$ is $a \in \mathbb{Z}$. For emphasis, we denote automaton $\mathcal{C}$ *qua* weighted graph by $G_\mathcal{C}$.

Recall that a computation π of $\mathcal{C}$ determines a path flow f_π in $G_\mathcal{C}$, mapping each edge to its multiplicity in π. If the length of π is bounded by an exponential function in the size of $\mathcal{C}$, then f_π has a description that is polynomial in the size of $\mathcal{C}$. We regard f_π as a polynomial *reachability certificate*. In this section we consider the problem of how to validate such a certificate in polynomial time; that is, given configurations (v,c) and (v',c'), we seek necessary and sufficient conditions on a flow f for there to exist a computation π from (v,c) to (v',c') with $f=f_\pi$, and we require that these conditions be polynomial-time checkable.

As a starting point, we recall the following straightforward variant of Euler's theorem.

Proposition 3. *Given vertices $s \neq t$, an s-t flow f is a path flow if and only if $|f|=1$ and the subgraph induced by the set of edges $\{e \in E : f(e)>0\} \cup \{(t,s)\}$ is strongly connected.*

Proposition 3 gives a way to check in linear time, given a flow f, whether there exists a path π such that $f=f_\pi$. The difficult part is then to determine whether π can be chosen such that it corresponds to a computation between given source and target configurations (v,c) and (v',c'). Informally speaking, we need to know that taking π from (v,c) does not cause the counter to go negative. More formally, given a path $\pi = v_0v_1\dots v_n$, define vertex v_j to be a **minimum** of π if the path $\pi' = v_0v_1\dots v_j$ has minimal weight among all prefixes of π; in this case we define $\text{drop}(\pi)$ to be $\text{weight}(\pi')$. Then π corresponds to a computation from (v,c) to (v',c') if and only if $\text{drop}(\pi) \geq -c$ and $\text{weight}(\pi)=c'-c$.

Given a path π from v to v', if there is a computation over π starting in configuration (v,c) and ending in configuration (v',c'), we say that π can be **taken from** (v,c) and **taken to** (v',c'). Next we introduce two key notions about flows which will help us to state sufficient conditions for a flow to be realisable by a computation between given configurations.

Given a flow f in $G_{\mathcal{C}}$, a **cycle** in f is a cycle ℓ in $G_{\mathcal{C}}$ such that f assigns positive flow to each edge in ℓ. If ℓ has positive (resp. negative) weight, then we speak of f having a positive (resp. negative) cycle.

Let f be a path flow from s to t. A **decomposition** of f consists of an enumeration $v_1,\dots,v_n$ of the set set $\{v : \exists u.f(u,v)>0\}$ of vertices with incoming flow, together with a sequence of flows $f_0,\dots,f_{n-1}$ such that (i) f_0 is a path flow from s to v_1, (ii) f_i is a path flow from v_i to v_{i+1} for $1 \leq i \leq n-1$, (iii) $f = f_0 + f_1 + \dots + f_{n-1}$, and (iv) if $i \leq j$ then f_j directs no flow into vertex v_i.

Proposition 4. *Let (v,c) and (v',c') be configurations of $\mathcal{C}$ and f be a path flow in $G_{\mathcal{C}}$ from v to v' such that* $\text{weight}(f)=c'-c$.

(i) If f has no positive cycles, then $f=f_\pi$ for some computation $\pi : (v,c) \longrightarrow^* (v',c')$ if and only if there is a decomposition $f=f_0+\dots+f_{n-1}$ such that $\sum_{i=0}^{j} \text{weight}(f_i) \geq -c$, $0 \leq j < n$.

(ii) If f has no negative cycles, then $f=f_\pi$ for some computation $\pi : (v,c) \longrightarrow^* (v',c')$ if and only if there is a decomposition $f^{op} = f_0 + \dots + f_{n-1}$ in $G^{op}_{\mathcal{C}}$ such that $\sum_{i=0}^{j} \text{weight}(f_i) \geq -c'$, $0 \leq j < n$.

Proof (sketch)

(i) Since f has no positive cycles, any path π in $G_{\mathcal{C}}$ such that $f=f_\pi$ also has no positive cycles. Thus in a computation along π, the net change in the counter value between consecutive visits to a given location is less than or

equal to 0. Thus to check that the counter never becomes negative, we need only verify that it is non-negative the last time π visits any given location. It is not hard to see that there exists a path π satisfying this last condition if and only if f has a flow decomposition satisfying the condition in *(i)* above.

(ii) This follows by applying the result stated in Part *(i)* to the flow f^{op} on the skew transpose of $G_{\mathcal{C}}$. □

In a slightly different vein to Proposition 4, Proposition 5 gives a simple condition on $G_{\mathcal{C}}$, rather than on the flow f, that guarantees that (v', c') is reachable from (v, c).

Proposition 5. *Let* (v, c) *and* (v', c') *be configurations of* $\mathcal{C}$ *and* f *be a path flow in* $G_{\mathcal{C}}$ *from* v *to* v' *such that* $\text{weight}(f) = c' - c$. *If there is a positive cycle* ℓ *that can be taken from* (v, c)*, and a negative cycle* ℓ' *that can be taken to* (v', c')*, then* (v', c') *is reachable from* (v, c)*.*

Proof (sketch). The idea is simple. By definition, there exists a path π from v to v' in $G_{\mathcal{C}}$ such that $f = f_\pi$. Now π need not yield a computation from (v, c) to (v', c') since it may be that $\text{drop}(\pi) \leq -c$. However we can circumvent this problem, and build a computation from (v, c) to (v', c'), by first *pumping up* the value of the counter by taking the positive cycle ℓ a number of times, then traversing π, and then *pumping down* the value of the counter by taking the negative cycle ℓ' a number of times. Note that if we take the positive cycle $-k \cdot \text{weight}(\ell')$ times, and the negative cycle $k \cdot \text{weight}(\ell)$ times, for some positive integer k, then the net effect on the counter is 0. □

A flow f is called a **reachability certificate** for two configurations (v, c) and (v', c') if there exists a path $\pi : (v, c) \longrightarrow^* (v', c')$ such that $f = f_\pi$ and one of the following three conditions holds: (i) f has no positive cycles; (ii) f has no negative cycles; (iii) there exists a positive cycle ℓ that can be taken *from* (v, c) and a negative cycle ℓ' that can be taken *to* (v', c'). Depending on which of the above three cases holds, we respectively call f_π a type-1, type-2 or type-3 reachability certificate. In any case, we say that the computation π *yields* the reachability certificate f_π. The following corollary of Propositions 4 and 5 gives an upper bound on the complexity of recognising a reachability certificate.

Corollary 1. *Given a one-counter machine* $\mathcal{C}$*, two configurations* (v, c) *and* (v', c')*, and a path flow* f *in* $G_{\mathcal{C}}$*, the problem of deciding whether* f *is a reachability certificate for* (v, c) *and* (v', c') *is in* NP*.*

Proof. It can be checked in polynomial time whether f has any positive cycles or any negative cycles, e.g., using the Bellman-Ford algorithm [6]. If f has no positive cycles, then by Proposition 4(i) to show that f is a type-1 reachability certificate we need only guess a decomposition $f = f_0 + \cdots + f_{n-1}$ such that $\sum_{i=0}^{j} \text{weight}(f_i) \geq -c$, $0 \leq j < n$. The case that f has no negative cycles similarly uses Proposition 4(ii).

It remains to consider type-3 reachability certificates. To this end, observe that there is a positive cycle ℓ that can be taken from (v, c) if and only if there is a positive simple cycle in the same strongly connected component of $G_{\mathcal{C}}$ as v that can be reached and taken from (v, c). This last condition can be checked in polynomial time using a small modification of the Bellman-Ford algorithm. By running the same algorithm on the skew transpose of $G_{\mathcal{C}}$, it can be checked whether there is a negative cycle ℓ' that can be taken to (v', c'). $\square$

Note that we do not assert that the existence of a computation $\pi : (v, c) \longrightarrow^* (v', c')$ guarantees that there is a reachability certificate for (v, c) and (v', c'). However, in the next section we show that the existence of a computation from (v, c) to (v', c') can be witnessed using at most three polynomial-size reachability certificates.

5 NP-Membership

Based on the ideas developed in the previous section, we are interested in paths π for which the associated flow f_π has no positive cycles or no negative cycles. It is important to note here that f_π may have positive cycles even though π itself does not have any positive cycles (and similarly for negative cycles). We will use the following proposition to overcome this problem.

Proposition 6. *Let π be a computation from (v, c) to (v', c') in which all cycles are negative. Then either the corresponding flow f_π has no positive cycles, or there is a computation $\theta = \theta_1 \cdot \theta_2 \cdot \theta_3$ from (v, c) to (v', c') such that* $\mathrm{length}(\theta_1) < \mathrm{length}(\pi)$ *and θ_2 is a positive cycle.*

Proof. Suppose that f_π contains a positive cycle ℓ. Let $u \in V$ be the first vertex of ℓ that π reaches, and let the counter value be y when π first reaches u. We claim that there is a positive cycle in $G_{\mathcal{C}}$ that can be taken from configuration (u, y).

If ℓ cannot be taken from (u, y) then we argue as follows. Factor ℓ as $\ell = u \xrightarrow{\rho_1} w \xrightarrow{\rho_2} u$, with w a minimum of ℓ (cf. Figure 3, which depicts the height of

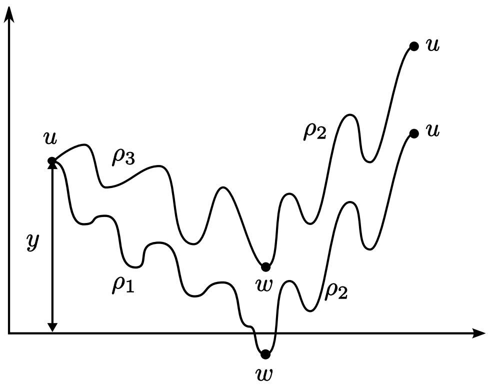


Fig. 3. Decomposition of the loop ℓ

the counter as ℓ is traversed). Then we have $\mathrm{weight}(\rho_1) < -y$. But π must visit w after it first visits u (since u is the first vertex of ℓ visited by π), so there is a path $\rho_3 : u \longrightarrow^* w$ in $G_{\mathcal{C}}$ such that $\mathrm{weight}(\rho_3) \geq \mathrm{drop}(\rho_3) \geq -y > \mathrm{weight}(\rho_1)$. Now consider the cycle $\ell' : u \xrightarrow{\rho_3} w \xrightarrow{\rho_2} u$. The preceding inequality gives

$$\begin{aligned}\mathrm{weight}(\ell') &= \mathrm{weight}(\rho_3) + \mathrm{weight}(\rho_2)\\ &> \mathrm{weight}(\rho_1) + \mathrm{weight}(\rho_2)\\ &= \mathrm{weight}(\ell)\,,\end{aligned}$$

so that ℓ' is a positive cycle. We also have

$$\begin{aligned}\mathrm{drop}(\ell') &\geq \mathrm{drop}(\rho_3) + \mathrm{drop}(\rho_2)\\ &\geq -y + 0\\ &= -y\,,\end{aligned}$$

whence ℓ' can be taken from (u, y). This proves the claim.

Next we observe that the first occurrence of u in π actually lies on a negative cycle in π. This is because π must visit u again, and all cycles in π are negative by assumption. Thus we can factor π as

$$(v, c) \xrightarrow{\pi_1} (u, y) \xrightarrow{\pi_2} (u, y') \xrightarrow{\pi_3} (v', c')$$

such that there is a positive cycle ℓ' that can be taken from (u, y), and with π_2 a negative cycle.

To define the required computation $\theta = \theta_1 \cdot \theta_2 \cdot \theta_3$, we reuse an idea from the proof of Proposition 5. Write $\mathrm{weight}(\ell') = p$ and $\mathrm{weight}(\pi_2) = -q$, where $p, q > 0$. Then define $\theta_1 = \pi_1$, $\theta_2 = (\ell')^q$ and $\theta_3 = (\pi_2)^{p+1} \cdot \pi_3$. Clearly $\mathrm{length}(\theta_1) < \mathrm{length}(\pi)$ and θ_2 is a positive cycle, as required. Since the positive cycle $(\ell')^q$ is cancelled out by the negative cycle $(\pi_2)^p$, θ is a computation from (v, c) to (v', c'). □

We also have the following dual of Proposition 6.

Proposition 7. *Let π be a computation from (v, c) to (v', c') in which all cycles are positive. Then either the corresponding flow f_π has no negative cycles, or there is a computation $\theta = \theta_1 \cdot \theta_2 \cdot \theta_3$ from (v, c) to (v', c') such that θ_2 is a negative cycle and* $\mathrm{length}(\theta_3) < \mathrm{length}(\pi)$.

Next we exploit Propositions 6 and 7 to show that the reachability of a configuration (v', c') from a configuration (v, c) can be witnessed by at most three reachability certificates.

Proposition 8. *If (v', c') is reachable from (v, c), then there exists a computation π from (v, c) to (v', c') that can be written $\pi = \pi_1 \cdot \pi_2 \cdot \pi_3$, such that π_1, π_2 and π_3 each yield reachability certificates.*

Proof. Let $\pi = v_0 v_1 \ldots v_n$ be (the path underlying) a computation from (v, c) to (v', c'). Without loss of generality we assume that π contains no zero-weight cycles. If π contains a positive cycle, then define i_1 such that v_{i_1} is the first vertex that appears in a positive cycle in π; otherwise let $i_1 = n$. Write $\pi_1 = v_0, v_1, \ldots, v_{i_1}$ and assume that π is chosen such that length(π_1) is minimised. Then π_1 contains only negative cycles; thus from Proposition 6 and the minimality of length(π_1) we deduce that the flow f_{π_1} has no positive cycles. We now consider two cases.

Case (i). $i_1 = n$. Then $\pi = \pi_1$, and f_{π_1} is a reachability certificate.

Case (ii). $i_1 < n$. If the terminal segment of π from v_{i_1} to v_n contains a negative cycle, then define $i_2 \geq i_1$ such that v_{i_2} is the last vertex that appears in a negative cycle in π; otherwise let $i_2 = i_1$. Write $\pi_3 = v_{i_2} v_{i_2+1} \ldots v_n$. Assume π is chosen, subject to the original choice to minimise length(π_1), such that length(π_3) is minimised. Then π_3 contains only positive cycles; thus from Proposition 7 and the minimality of length(π_3) we deduce that the flow f_{π_3} has no negative cycles. We now consider two sub-cases.

Case (ii)(a). $i_1 = i_2$. Then $\pi = \pi_1 \cdot \pi_3$, and f_{π_1} and f_{π_3} are both reachability certificates.

Case (ii)(b). $i_1 < i_2$. Then write $\pi_2 = v_{i_1} v_{i_1+1} \ldots v_{i_2}$. Starting in configuration (v, c), let (v_{i_1}, c_{i_1}) be the configuration of $\mathcal{C}$ after executing π_1, and let (v_{i_2}, c_{i_2}) be the configuration of $\mathcal{C}$ after further executing π_2. By definition of π_2 there is a positive cycle that can be taken from (v_{i_1}, c_{i_1}) and a negative cycle that can be taken to (v_{i_2}, c_{i_2}). Thus f_{π_2} is a reachability certificate and $\pi = \pi_1 \cdot \pi_2 \cdot \pi_3$ is the sequential composition of three paths, each of which yields a reachability certificate. $\square$

We can now complete the proof of the first main result of the paper.

Theorem 3. *The reachability problem for one-counter automata is in* NP.

Proof. Let $\mathcal{C}$ be a one-counter automaton with configurations (v, c) and (v', c'). If (v', c') is reachable from (v, c) then, by Proposition 8, there is a computation $\pi = \pi_1 \cdot \pi_2 \cdot \pi_3$ from (v, c) to (v', c') such that π_1, π_2 and π_3 each yield reachability certificates. Moreover we can assume, without loss of generality, that the lengths of π_1, π_2 and π_3 are bounded by 2^P for some polynomial P in $|\mathcal{C}|$ and the bit lengths of c and c'. The bounds on π_1 and π_3 follow from the fact that π_1 has only negative cycles and π_3 has only positive cycles. The bound on π_2 follows from Proposition 2. Thus the reachability certificates corresponding to π_1, π_2 and π_3 all have polynomial size, and, by Corollary 1, can be guessed and verified in polynomial time. $\square$

6 Parametric Counter Automata

In this section we exploit the results developed in Section 5 to show that the reachability problem for parametric one-counter automata can be reduced to the satisfiability problem for a decidable extension of existential Presburger arithmetic.

Let $x_1, \ldots, x_n$ be a set of integer variables. A **linear polynomial** is a polynomial of the form $a_0 + a_1x_1 + \ldots + a_nx_n$, where the a_i are integer coefficients. A **linear constraint** is an inequality of the form $a_0 + a_1x_1 + \ldots + a_nx_n \le 0$. Define $S \subseteq \mathbb{Z}^n$ to be an ($\mathbb{N}$-)**linear set** if there exist vectors $v_0, v_1, \ldots, v_t \in \mathbb{Z}^n$ such that $S = \{v : v = v_0 + b_1v_1 + \ldots + b_nv_n,\, b_i \in \mathbb{N}\}$. A **semilinear set** is a finite union of linear sets.

Presburger arithmetic is the first-order theory of the structure $(\mathbb{Z}, <, +, 0, 1)$. It is well-known that the satisfiability problem for Presburger arithmetic is decidable, and that subsets of $\mathbb{Z}^k$ definable by formulas of Presburger arithmetic are effectively semilinear. Adding multiplication to Presburger arithmetic leads to undecidability, as does adding the *divides* predicate $n \mid m$. However Lipshitz [17] gave a decision procedure for the satisfiability problem for the existential fragment of Presburger arithmetic with divisibility. This last result has been used to show the decidability of certain problems concerning systems of quadratic Diophantine equations [10,11]. We give a simple application of this kind below.

Let $\{y_1, \ldots, y_k\}$ and $\{x_1, \ldots, x_n\}$ be disjoint sets of integer variables. For $1 \le i \le k$ let R_i denote the quadratic polynomial $y_iA_i + B_i$, where A_i and B_i are linear polynomials in $x_1, \ldots, x_n$. Furthermore, let P be a subset of $\mathbb{Z}^k$ defined by a formula of Presburger arithmetic. We consider the following problem:

Problem A. Given $R_1, \ldots, R_k$ and P, are there values for $x_1, \ldots, x_n$ and $y_1, \ldots, y_k$ such that $(R_1, \ldots, R_k) \in P$?

Lemma 1. *Problem A is decidable.*

Proof. The proof is by reduction to the satisfiability problem for the existential fragment of Presburger arithmetic with divisibility.

Note that $P \subseteq \mathbb{Z}^k$, being Presburger definable, is effectively semilinear. By case splitting we may assume that P defines a linear set, say $P = \{v : v = v_0 + a_1v_1 + \ldots + a_tv_t,\, a_i \in \mathbb{N}\}$ where $v_0, \ldots, v_t \in \mathbb{Z}^k$. Thus, introducing new nonnegative integer variables $w_1, \ldots, w_t$, we seek a solution to the following system of equations

$$\begin{aligned} y_1A_1 + B_1 &= v_{0,1} + w_1v_{1,1} + \ldots + w_tv_{t,1} \\ y_2A_2 + B_2 &= v_{0,2} + w_1v_{1,2} + \ldots + w_tv_{t,2} \\ &\vdots \\ y_kA_k + B_k &= v_{0,k} + w_1v_{1,k} + \ldots + w_tv_{t,k} \end{aligned}$$

But this is equivalent to finding a solution to the following formula in Presburger arithmetic with divisibility:

$$\bigwedge_{i=1}^{k} A_i \mid (v_{0,i} + w_1v_{1,i} + \ldots + w_tv_{t,i} - B_i) \wedge \bigwedge_{i=1}^{t} w_i \ge 0\,.$$

Remark 1. Note that in Problem A, each variable y_i occurs in a single quadratic polynomial. It immediately follows from a result of Ibarra and Dang [10] that generalising Problem A to allow the same variable y_i to appear in two separate quadratic polynomials leads to an undecidable problem.

6.1 Reachability

Let $\mathcal{C} = (V, E, X, \lambda)$ be a parametric one-counter automaton, and assume for now that $\mathcal{C}$ does not have any zero tests. Recall that the reachability problem asks whether there is a computation between given configurations (v, c) and (v', c') for *some* instantiation of the parameters. By Proposition 8, the existence of such a computation is witnessed by (at most) three reachability certificates. Thus our strategy to show decidability of reachability is to phrase the existence of each of the three types of reachability certificate as an instance of Problem A, with variables representing the parameters. We illustrate the idea for type-1 certificates, the other cases being very similar.

Recall that a type-1 reachability certificate for configurations (v, c) and (v', c') consists of a path flow f from v to v' such that f has no positive cycles, $\mathit{weight}(f) = c' - c$, and there is a decomposition $f = f_0 + \ldots + f_{n-1}$, such that

$$\bigwedge_{j=0}^{n-1} \left(\sum_{i=0}^{j} \mathit{weight}(f_i) \geq -c \right) . \tag{1}$$

In encoding the existence of such an f, let us temporarily assume that the **support** $E_i \overset{\text{def}}{=} \{e \in E : f_i(e) > 0\}$ of each flow f_i has been fixed beforehand, subject to the requirement that $f = f_0 + \cdots f_{n-1}$ be a flow decomposition. Thus it only remains to determine the flow along each edge of E_i.

Let the set of edges E have cardinality m. We introduce a set of nonnegative integer variables $Y^{(i)} = \{y_1^{(i)}, y_2^{(i)}, \ldots, y_m^{(i)}\}$ to represent the flow f_i, $0 \leq i < n$. The idea is that each variable represents the flow along a given edge. The flow conservation conditions on f_i and the requirement that f_i have support E_i can be expressed as a system $S^{(i)}$ of linear constraints on the set of variables $Y^{(i)}$.

We also have a set of integer variables $X = \{x_1, x_2, \ldots, x_k\}$ representing the parameters of $\mathcal{C}$. The requirement that f have no positive cycles can be expressed as a system of linear constraints:

$$A_j \leq 0 , \quad j = 0, \ldots, t , \tag{2}$$

where A_j is a linear polynomial in the set of variables X, and there is one constraint for each simple cycle of f (exactly which equations need to be written here, which depends on the simple cycles in f, is determined by the supports $E_1, E_2, \ldots, E_{n-1}$.)

The weight of flow f_i can then be expressed as a quadratic expression in the set of variables $X \cup Y^{(i)}$:

$$\mathit{weight}(f_i) = \sum_{j=1}^{m} y_j^{(i)} \alpha_j \quad [\alpha_j \in \mathbb{Z} \cup X] . \tag{3}$$

The next step is to eliminate the system of constraints $S^{(i)}$ by a change of variables. Note that the constraints $S^{(i)}$ on the set of variables $Y^{(i)}$ define a linear set, thus we can introduce a set of nonnegative integer variables $U^{(i)} = \{u_1^{(i)}, u_2^{(i)}, \ldots, u_{l_i}^{(i)}\}$ and linear polynomials $B_j^{(i)}$, $1 \leq j \leq m$, in $U^{(i)}$, such that $(y_1^{(i)}, \ldots, y_m^{(i)})$ satisfies $S^{(i)}$ iff $y_j^{(i)} = B_j^{(i)}$ for some choice of the variables in $U^{(i)}$. Applying this change of variables to Equation (3) and rearranging terms yields

$$weight(f_i) = \sum_{j=1}^{l_i} u_j^{(i)} C_j^{(i)} + D^{(i)}\,, \tag{4}$$

where the $C_j^{(i)}$ and $D^{(i)}$ are linear polynomials in X.

We can now formulate the existence of a type-1 reachability certificate as an instance of Problem A. To this end we introduce a family $R_{i,j}$ of quadratic polynomials over the set of variables $X \cup U^{(i)}$, where $R_{i,j} \overset{\text{def}}{=} u_j^{(i)} C_j$ for $0 \leq i < n$ and $1 \leq j \leq l_i$. By (4) the weight of each flow f_i can be written as a linear expression in $D^{(i)}$ and $R_{i,j}$. Thus requirements (1) and (2) can be expressed as a Presburger definable relation P on the A_i, $D^{(i)}$ and $R_{i,j}$, according to the format of Problem A.

Finally, we note that we can drop our assumption of the fixity of the supports $E_1, E_2, \ldots, E_{n-1}$ by case splitting, using the closure of Presburger definable sets under disjunction. Thus we can phrase the existence of a type-1 reachability certificate between two given configurations as an instance of Problem A.

In a similar fashion, the existence of type-2 and type-3 reachability certificates can also be translated into instances of Problem A. Combining with Proposition 8 we derive our second main result:

Theorem 4. *The reachability problem for parametric one-counter automata is decidable.*

References

1. Abdulla, P.A., Cerans, K.: Simulation is decidable for one-counter nets. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 253–268. Springer, Heidelberg (1998)
2. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Proc. STOC 1993, pp. 592–601. ACM, New York (1993)
3. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
4. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 577–588. Springer, Heidelberg (2006)
5. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427. Springer, Heidelberg (1998)
6. Cormen, T., Leiserson, C., Rivest, R.: Introduction to algorithms. MIT Press and McGraw-Hill (1990)

7. Demri, S.: Logiques pour la spécification et vérification. Mémoire d'habilitation, Université Paris 7 (2007)
8. Demri, S., Gascon, R.: The effects of bounding syntactic resources on Presburger LTL. In: Proc. TIME 2007. IEEE Computer Society Press, Los Alamitos (2007)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
10. Ibarra, O.H., Dang, Z.: On two-way finite automata with monotonic counters and quadratic diophantine equations. Theor. Comput. Sci. 312(2-3), 359–378 (2004)
11. Ibarra, O.H., Dang, Z.: On the solvability of a class of diophantine equations and applications. Theor. Comput. Sci. 352(1), 342–346 (2006)
12. Ibarra, O.H., Jiang, T., Trân, N., Wang, H.: New decidability results concerning two-way counter machines and applications. In: Lingas, A., Carlsson, S., Karlsson, R. (eds.) ICALP 1993. LNCS, vol. 700, pp. 313–324. Springer, Heidelberg (1993)
13. Jančar, P., Kučera, A., Moller, F., Sawa, Z.: DP lower bounds for equivalence-checking and model-checking of one-counter automata. Information Computation 188(1), 1–19 (2004)
14. Kučera, A.: Efficient verification algorithms for one-counter processes. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, p. 317. Springer, Heidelberg (2000)
15. Lafourcade, P., Lugiez, D., Treinen, R.: Intruder deduction for AC-like equational theories with homomorphisms. In: Research Report LSV-04-16. LSV, ENS de Cachan (2004)
16. Leroux, J., Sutre, G.: Flat counter automata almost everywhere! In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 489–503. Springer, Heidelberg (2005)
17. Lipshitz, L.: The diophantine problem for addition and divisibility. Transaction of the American Mathematical Society 235, 271–283 (1976)
18. Mayr, E.W.: An algorithm for the general petri net reachability problem. In: Proc. STOC 1981, pp. 238–246. ACM, New York (1981)
19. Minsky, M.: Recursive unsolvability of post's problem of "tag" and other topics in theory of turing machines. Annals of Math. 74(3) (1961)
20. Xie, G., Dang, Z., Ibarra, O.H.: A solvable class of quadratic diophantine equations with applications to verification of infinite-state systems. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 668–680. Springer, Heidelberg (2003)

Winning Regions of Pushdown Parity Games: A Saturation Method

Matthew Hague and C.-H. Luke Ong

Oxford University Computing Laboratory

Abstract. We present a new algorithm for computing the winning region of a parity game played over the configuration graph of a pushdown system. Our method gives the first extension of the saturation technique to the parity condition. Finite word automata are used to represent sets of pushdown configurations. Starting from an initial automaton, we perform a series of automaton transformations to compute a fixed-point characterisation of the winning region. We introduce notions of under-approximation (soundness) and over-approximation (completeness) that apply to automaton transitions rather than runs, and obtain a clean proof of correctness. Our algorithm is simple and direct, and it permits an optimisation that avoids an immediate exponential blow up.

1 Introduction

Pushdown systems — finite-state transition systems equipped with a stack — are an old model of computation that have recently enjoyed renewed interest from the software verification community. They accurately model the control flow of first-order recursive programs [7] (such as C and Java), and lend themselves readily to algorithmic analysis. Pushdown systems have played a key rôle in the automata-theoretic approach to software model checking [1,5,10,14]. Considerable progress has been made in the implementation of scalable model checkers of pushdown systems. These tools (e.g. Bebop [11] and Moped [10]) are an essential back-end component of such model checkers as SLAM [12].

The modal mu-calculus is a highly expressive language for describing properties of program behaviour (all standard temporal logics in verification are embeddable in it). In a seminal paper [3] at CAV 1996, Walukiewicz showed that *local* modal mu-calculus model checking of pushdown systems, or equivalently [4] the solution of *pushdown parity games* (i.e. parity games over the configuration graphs of pushdown systems), is EXPTIME-complete. His method reduces pushdown parity games to finite parity games by a kind of powerset construction, which is immediately exponential in size. Whilst *local* model checking asks if a designated state (of a pushdown system) satisfies a given property, *global* model checking computes a finite representation of the set of states satisfying the property. The latter is equivalent to computing Éloïse's winning region of a pushdown parity game, which is the problem that we have set ourselves here. It is worth noting that global model checking used to be the norm in verification (CTL and many symbolic model checkers still perform global model checking). While local model checking can be expected to have better complexity, global model checking is important when repeated checks are required (because tests on the representing automata tend to be comparatively cheap), or where the model checking is only a component of the verification process.

Related work. Cachat [13] and Serre [9] have independently generalised Walukiewicz' algorithm to provide solutions to the global model-checking problem: they use the local model-checking algorithm as an oracle to guide the construction of the automaton recognising the winning region. An alternative approach, introduced by Piterman and Vardi [8], uses two-way alternating tree automata to navigate a tree representing all possible stacks: after several reductions, including the complementation of Büchi automata, an automaton accepting the winning region can be constructed.

At CONCUR 1997, Bouajjani *et al.* [1], and, independently, Finkel *et al.* [2] (at INFINITY 1997), introduced a *saturation* technique for global model-checking reachability properties of pushdown systems. From a finite-word automaton recognising a given configuration-set $\mathcal{C}$, they perform a backwards-reachability analysis. By iteratively adding new transitions to the automaton, the set of configurations that can reach some configuration in $\mathcal{C}$ is constructed. Since the number of new transitions is bounded, the iterative process terminates. This approach underpins the acclaimed Moped tool.

Contributions. This paper presents a new algorithm for computing Éloïse's winning region of a pushdown parity game. We represent (regular) configuration sets as alternating multi-automata [1]. Using a modal mu-calculus formula that defines the winning region as a guide, our algorithm iteratively expands (when computing least fixpoints) and contracts (when computing greatest fixpoints) an approximating automaton until the winning region is precisely recognised. Our method is a generalisation of Cachat's for solving Büchi games [13], which is itself a generalisation of the saturation technique for reachability analysis. However, we adopt a different proof strategy which we believe to be cleaner than Cachat's original proof. Our contribution can equivalently be presented as a solution to the *global* model checking problem: given a pushdown system $\mathcal{K}$, a modal mu-calculus formula $\chi(\overline{Y})$, and a regular valuation V, our method can *directly* compute an automaton that recognises the set $[\![\chi(\overline{Y})]\!]_V^{\mathcal{K}}$ of $\mathcal{K}$-configurations satisfying $\chi(\overline{Y})$ with respect to V.

Our algorithm has several advantages:

(i) The algorithm is simple and direct. Even though pushdown graphs are in general infinite, our construction of the automaton that recognises the winning region follows, in outline, the standard pen-and-paper calculation of the semantics of modal mu-calculus formulas in a *finite* transition system. Through the use of *projection*, our algorithm is guaranteed to terminate in a finite number of steps, even though the usual fixpoint calculations may require transfinite iterations. Thanks to projection, the state-sets of the approximating automata are bounded: during expansion, the number of transitions increases, but only up to the bound determined by the finite state-set; during contraction, the number of transitions decreases until it reaches zero or stabilises.

(ii) The correctness proof is simple and easy to understand. A conceptual innovation of the correctness argument are *valuation soundness* and *valuation completeness*. They are respectively under- and over-approximation conditions that apply *locally* to individual transitions of the automaton, rather than *globally* to the extensional behaviour of the automaton (such as runs). By combining these conditions, which reduce the overhead of the proof, we show that our algorithm is both sound and complete in the usual sense.

(iii) Finally, our decision procedure builds on and extends the well-known saturation method, which is the implementation technique of choice of pushdown checkers. In contrast to previous solutions, our algorithm permits a straightforward optimisation that avoids an immediate exponential explosion, which we believe is important for an efficient implementation. Another advantage worth noting is that the automaton representing the winning region is independent of the maximum priority m (even though it takes time exponential in m to construct).

2 Preliminaries

A ***pushdown parity game*** is a parity game defined over a *pushdown graph* (i.e. the configuration graph of a pushdown system). Formally it is a quadruple $(\mathcal{P}, \mathcal{D}, \Sigma_\perp, \Omega)$ where $\mathcal{P} = \mathcal{P}_A \uplus \mathcal{P}_E = \{p^1, \ldots, p^z\}$ is a set of control states partitioned into Abelard's and Éloïse's states, $\Sigma_\perp := \Sigma \cup \{ \perp \}$ is a finite stack alphabet (we assume $\perp \notin \Sigma$), $\mathcal{D} \subseteq \mathcal{P} \times \Sigma_\perp \times \mathcal{P} \times \Sigma_\perp^*$ is a set of pushdown rules and $\Omega : \mathcal{P} \to \{1, \ldots, m\}$ is a function assigning priorities to control states. As is standard, we assume that the bottom-of-stack symbol $\perp$ is neither pushed onto, nor popped from, the stack. We also assume there is a rule for each $p \in \mathcal{P}$ and $a \in \Sigma_\perp$.

A play begins from some configuration $\langle p, a\,w \rangle$. The player controlling p chooses $p\,a \to p'\,w' \in \mathcal{D}$ and the play moves to $\langle p', w'w \rangle$. Then, the player controlling p' chooses a move, and so on, generating an infinite run. The priority of a configuration $\langle p, w \rangle$ is $\Omega(p)$. A priority occurs infinitely often in a play if there are an infinite number of configurations with that priority. Éloïse wins the play if the smallest priority occurring infinitely often is even. Otherwise, Abelard is the winner.

A player's ***winning region*** of a pushdown parity game is the set of configurations from which the player can always win the game, regardless of the other player's strategy. Éloïse's winning region $\mathcal{W}_E$ of a parity game $\mathcal{G}$ is definable in the modal μ-calculus; the following is due to Walukiewicz [3]:

$$\mathcal{W}_E = [\![\mu Z_1.\nu Z_2. \ldots \mu Z_{m-1}.\nu Z_m.\varphi_E(Z_1, \ldots, Z_m)]\!]_V^{\mathcal{G}}$$

where m is the maximum parity (assumed even), V is a valuation of the variables[1], and

$$\varphi_E(Z_1, \ldots, Z_m) := \left(E \Rightarrow \bigwedge_{c \in \{1,\ldots,m\}} (c \Rightarrow \Diamond Z_c) \right) \wedge \left(\neg E \Rightarrow \bigwedge_{c \in \{1,\ldots,m\}} (c \Rightarrow \Box Z_c) \right)$$

where E is an atomic proposition asserting the current configuration is Éloïse's and, for $1 \leq c \leq m$, c asserts that the priority of the current control state is c.

For each $1 \leq c \leq m$, we have a variable Z_c. The odd priorities are bound by μ operators which can be understood intuitively as "finite looping". Dually, even priorities

[1] The valuation is initially empty since the formula has no free variables.

are bound by ν operators and can be understood as "infinite looping". The formula φ_E asserts that a variable Z_c is visited whenever a configuration of priority c is encountered. Thus the full formula asserts that the minimal priority occurring infinitely often must be even — otherwise a variable bound by the μ operator would be passed through infinitely often. It can be shown by a signature lemma that Éloïse has a winning strategy from a configuration satisfying the formula [3]. Since the formula's inverse is a similar formula with μ/ν, and $\Box/\Diamond$ reversed, Abelard has a winning strategy from any configuration not in $\mathcal{W}_E$.

Thanks to the Knaster-Tarski Fixpoint Theorem, the semantics of a fixpoint formula $[\![\sigma Z.\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_V$ where $\sigma \in \{\mu, \nu\}$ can be given as the limit of the sequence of ***α-approximants*** $[\![\sigma^\alpha Z.\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_V$, where α ranges over the ordinals and λ ranges over the limit ordinals:

$$\begin{aligned}
[\![\sigma^0 Z.\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_V &:= \mathit{Init} \\
[\![\sigma^{\alpha+1} Z.\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_V &:= [\![\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_{V[Z \mapsto [\![\sigma^\alpha Z.\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_V]} \\
[\![\sigma^\lambda Z.\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_V &:= \bigcirc_{\alpha<\lambda} [\![\sigma^\alpha Z.\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_V
\end{aligned}$$

where $\mathit{Init} = \emptyset$ and $\bigcirc = \bigcup$ when $\sigma = \mu$, and Init is the set of all configurations and $\bigcirc = \bigcap$ when $\sigma = \nu$. The least ordinal κ such that $[\![\sigma^\kappa Z.\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_V = [\![\sigma Z.\chi(\overline{Y}, Z)]\!]^{\mathcal{G}}_V$ is called the *closure ordinal.*

Example 1. When interpreted in a pushdown graph, $\langle\, \sigma^\alpha Z.\chi(\overline{Y}, Z)\,\rangle_{\alpha \in \mathbf{Ord}}$ may have an infinite closure ordinal. Consider the following pushdown parity graph (which is a dual of an example of Cachat's [13]): all configurations are Abelard's, $\Omega(p) = 1$ and $\Omega(f) = 2$.

$$\begin{tikzcd}
\langle f, \bot \rangle \arrow[loop left] & \langle f, a\bot \rangle \arrow[r] \arrow[d] & \langle f, aa\bot \rangle \arrow[r] \arrow[d] & \cdots \\
\langle p, \bot \rangle \arrow[u] & \langle p, a\bot \rangle \arrow[l] & \langle p, aa\bot \rangle \arrow[l] & \cdots \arrow[l]
\end{tikzcd}$$

In this game, $\mathcal{W}_E = [\![\mu Z_1.\nu Z_2.\varphi_E(Z_1, Z_2)]\!]$ consists of all configurations. However, any $\langle f, a\, a^n \bot \rangle$ for some n only appears in an approximant of the least fixed point when $\langle f, a\, a\, a^n \bot \rangle$ and $\langle p, a\, a^n \bot \rangle$ appear in the previous approximant (since Abelard may move to either of these configurations). Hence, all $\langle p, a^n \bot \rangle$ must appear in the α-approximant before any $\langle f, a^n \bot \rangle$ can appear in the $(\alpha + 1)$-approximant. The first approximant containing all p configurations is the ω-approximant.

We use alternating multi-automata [1] as a representation of (regular) configuration-sets. Given a pushdown system $(\mathcal{P}, \mathcal{D}, \Sigma)$ with $\mathcal{P} = \{p^1, \ldots, p^z\}$, an ***alternating multi-automaton*** A is a tuple $(\mathcal{Q}, \Sigma, \Delta, I, \mathcal{F})$ where $\mathcal{Q}$ is a finite set of states, $\Delta \subseteq \mathcal{Q} \times (\Sigma \cup \{\,\bot\,\}) \times 2^{\mathcal{Q}}$ is a set of transitions (we assume $\bot \notin \Sigma$), $I = \{q^1, \ldots, q^z\} \subseteq \mathcal{Q}$ is a set of initial states, and $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states. Observe that there is an initial state for each control state of the pushdown system. We write $q \xrightarrow{a} Q$ just if $(q, a, Q) \in \Delta$; and define $q \xrightarrow{\varepsilon} \{q\}$; and $q \xrightarrow{aw} Q_1 \cup \cdots \cup Q_n$ just if $q \xrightarrow{a} \{q_1, \ldots, q_n\}$ and $q_k \xrightarrow{w} Q_k$ for all $1 \leq k \leq n$. Finally we define the *language accepted by* A, $\mathcal{L}(A)$, by: $\langle p^j, w \rangle \in \mathcal{L}(A)$ just if $q^j \xrightarrow{w} Q$ for some $Q \subseteq \mathcal{F}$. Henceforth, we shall refer to alternating multi-automata simply as ***automata***.

Reachability and Projection. The formula $\varphi_E(Z_1, \ldots, Z_m)$ asserts reachability in one step, which we compute using the reachability algorithm [1] due to Bouajjani *et al.* Cachat's extension of this algorithm requires a technique called *projection*. Using an example, we briefly introduce the relevant techniques.

Take a PDS with the rules $p^1\, a \rightarrow p^2\, \varepsilon$ and $p^2\, b \rightarrow p^2\, ba$. The automaton A_{eg} in Figure 1 represents a configuration set $\mathcal{C}$. Let $Pre(\mathcal{C})$ be the set of all configurations that can reach $\mathcal{C}$ in exactly one step. To calculate $Pre(\mathcal{C})$ we first add a new set of initial states — since we don't necessarily have $\mathcal{C} \subseteq Pre(\mathcal{C})$. By applying $p^1\, a \rightarrow p^2\, \varepsilon$, any configuration of the form $\langle p^1, aw \rangle$, where w is accepted from q^2 in A_{eg}, can reach $\mathcal{C}$. Hence we add an a-transition from q^1_{new}. (Via the pop transition, we reach $\langle p^2, w \rangle \in \mathcal{L}(A_{eg})$.) Alternatively, via $p^2\, b \rightarrow p^2\, ba$, any configuration of the form $\langle p^2, bw \rangle$, where baw is accepted from q^2 in A_{eg}, can reach $\mathcal{C}$. The push, when applied backwards, replaces ba by b. We add a b-transition from q^2_{new} which skips any run over ba from q^2. Figure 2 (i) shows the resulting automaton.

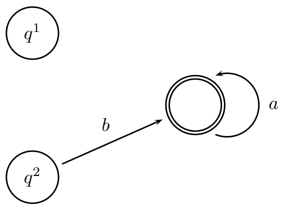


Fig. 1. The automaton A_{eg} accepting $\langle p^2, ba^* \rangle$

To ensure termination of the Büchi construction, Cachat uses *projection*, which replaces a new transition to an old initial state with a transition to the corresponding new state. Hence, the transition in Figure 2 (i) from q^1_{new} is *replaced* by the transition in Figure 2 (ii). The old initial states are then unreachable, and deleted, which, in this case, leaves an automaton with the same states as Figure 1 (modulo the new suffix) but an additional transition. In this sense, the state-set remains fixed.

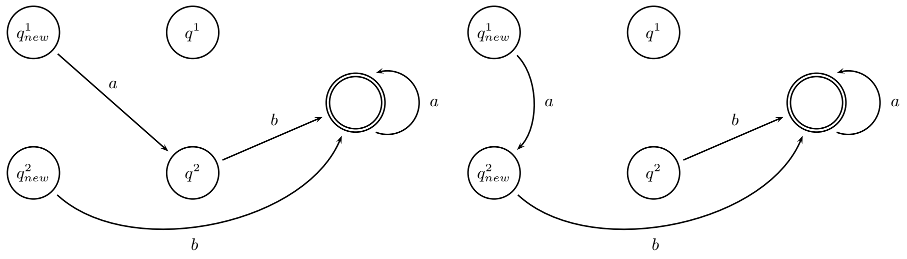


Fig. 2. (i) On the left, A_{eg} updated by the rules $p^1\, a \rightarrow p^2\, \varepsilon$ and $p^2\, a \rightarrow p^2\, ba$; and (ii) on the right, the result of projecting the automaton in (i)

3 An Example

We begin with an intuitive explanation of the algorithm by means of an example. Consider the pushdown game represented in Figure 3. Note that this diagram is a quotient of the infinite state space. Since the aim of this example is to give an overview of the flow of the algorithm, the behaviour of the pushdown system is kept simplistic. The subscripts indicate the priority of a configuration[2] and an arc labelled with $push_w$

[2] Our priorities here begin at 0. This does not change the algorithm significantly.

$$\begin{tikzcd}
\langle p'_E, b\Sigma^*\bot\rangle_0 \arrow[d, bend left, "push_a"] & \langle p'_E, a\Sigma^*\bot\rangle_0 \arrow[l, "push_b"'] & \\
\langle p_E, a\Sigma^*\bot\rangle_1 \arrow[u, bend left, "push_b"] \arrow[r, bend left, "push_a"] & \langle p_A, a\Sigma^*\bot\rangle_1 \arrow[l, bend left, "push_a"] \arrow[r, "push_b"] & \langle p_A, b\Sigma^*\bot\rangle_1 \arrow[loop above, "push_b"] \\
 & & \langle p_E, b\Sigma^*\bot\rangle_1 \arrow[u, "push_b"]
\end{tikzcd}$$

Fig. 3. An example pushdown parity game

indicates a pushdown rule of the form $p\,a \rightarrow p'\,w$ for some p, a and p'. Let $p_E, p'_E \in \mathcal{P}_E$ and $p_A \in \mathcal{P}_A$.

Éloïse can win from configurations of the forms $\langle p'_E, a\Sigma^*\bot\rangle_0$, $\langle p_E, a\Sigma^*\bot\rangle_1$, or $\langle p'_E, b\Sigma^*\bot\rangle_0$. Éloïse can loop between the last two of these configurations, generating a run with priority 0. From elsewhere, Abelard can force play to $\langle p_A, b\Sigma^*\bot\rangle_1$ and generate a run with priority 1. Computing Éloïse's winning region is equivalent to computing $[\![\nu Z_0.\mu Z_1.\varphi_E(Z_0, Z_1)]\!]^{\mathcal{G}}_V$. We illustrate how this is done in the following.

To compute a greatest fixed point, we begin by setting Z_0 to be the set of all configurations. We then calculate the automaton recognising the denotation of $\mu Z_1.\varphi_E(Z_0, Z_1)$ with this value of Z_0. The result is the value of Z_0 for the next iteration. After each iteration the value of Z_0 will be a subset of the previous value. This computation reaches a limit when the value of Z_0 stabilises, which is the denotation of the formula.

Computing the least fixed point proceeds in a similar manner, except that the initial value of Z_1 is set to $\emptyset$. We then compute the (automaton that recognises the) denotation of $\varphi_E(Z_0, Z_1)$, which gives us the next value of Z_1. Dual to the case of greatest fixed points, the value of Z_1 increases with each iteration.

Constructing the Automaton. (We shall often confuse the denotation of a formula with the automaton that recognises it, leaving it to the context to indicate which is intended.) We begin by setting Z_0 to the set of all configurations. The automaton recognising all configurations is shown in Figure 4 (i)[3]. Given this value of Z_0, we compute the denotation of $\mu Z_1.\varphi_E(Z_0, Z_1)$. The first step is to set the initial value of Z_1 to the empty set. The corresponding automaton is also shown in Figure 4 (ii). Observe that we have a separate set of initial states for Z_0 and Z_1.

We now compute $\varphi_E(Z_0, Z_1)$ which will be the next value of Z_1. A configuration $\langle p^j, aw\rangle$ with priority c should be accepted if Éloïse can play - or Abelard must play - a move which reaches some $\langle p^k, w'w\rangle \in V(Z_c)$. The result is Figure 4 (iii).

Observe that the computation of the new automaton has only added transitions. When computing a least fixed point, each generation of initial states has more transitions than the previous generation. In this example the number of possible transitions is finite since

[3] This is a simplification of the automaton defined in Section 4.

all transitions happen to go to q_f^*. Therefore, the automaton must eventually become saturated, causing termination. In the full algorithm, transitions from the new set of initial states to the old are *projected* back onto the new initial states. This ensures that the previous generation is not reachable. Hence, the state-set is fixed. When computing a greatest fixed point, termination can be proved dually: we begin with all transitions and iteratively remove transitions at each stage.

We now compute the next iterate of Z_1. We add a new set of initial states, and perform the reachability analysis, as in Figure 5 (i). If we were to perform another round of the reachability analysis, we would find a fixed point. That is, the transitions from the new initial states corresponding to Z_1 have the same outgoing transitions as the old. This fixed point is the next value of Z_0. Therefore, we set the current initial states of Z_1 to be the new initial states of Z_0. If necessary, we would also perform projections from the old initial states of Z_0 to the new. We then begin evaluating $\mu Z_1.\varphi_E(Z_0, Z_1)$ with our new value of Z_0. The initial value of Z_1 is the empty set, so we introduce new initial states corresponding to Z_1 with no outgoing transitions. Figure 5 (ii) shows the automaton after these steps.

We compute the next iterate of Z_1 as before, as in Figure 6. The second automaton is the fixed point of Z_1, and hence the new iterate of Z_0. Since the new Z_0 is identical to the previous Z_0, we have reached a final fixed point. Setting the initial states of Z_1 to be the initial states of Z_0, and deleting any unreachable states, gives the automaton in Figure 7, which accepts Éloïse's winning region.

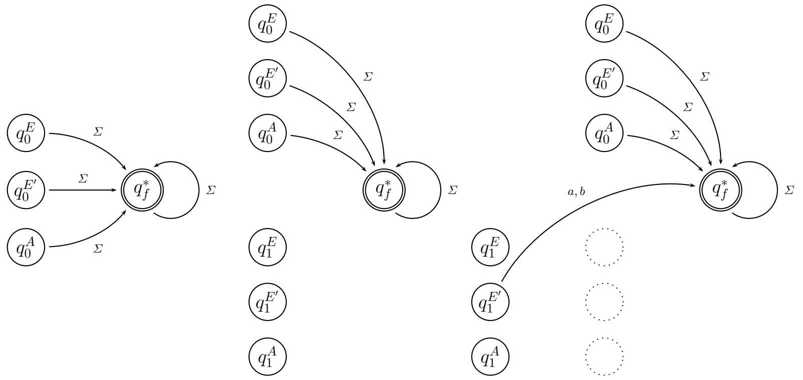


Fig. 4. From left to right, (i) the automaton accepting the initial value of Z_0; (ii) the automaton accepting the initial values of Z_0 and Z_1; and (iii) the automaton after the first round of reachability analysis

4 The Algorithm

Fix a pushdown parity game $\mathcal{G} = (\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$ that has maximum priority m. The algorithm has two key components. The first — $Phi(A)$ — computes an automaton recognising $[\![\varphi_E(Z_1, \ldots, Z_m)]\!]_V^{\mathcal{G}}$, given an automaton A recognising the configuration-sets $V(Z_1), \ldots, V(Z_m)$. The second — $Sig(l, A)$ — computes, for each $1 \leq l \leq m$, an automaton recognising $[\![\sigma Z_l.\chi_{l+1}(Z_1, \ldots, Z_l)]\!]_V^{\mathcal{G}}$ where σ is either μ or ν as appropriate,

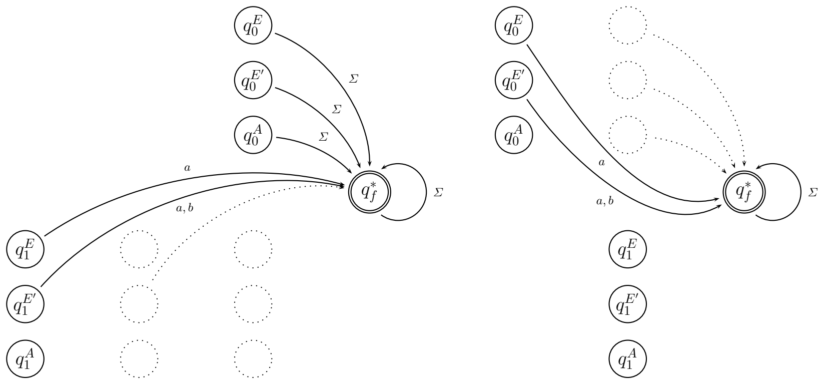


Fig. 5. (i) The automaton after the second round of reachability analysis; and (ii) the automaton with the new value of Z_0 and Z_1 set to the empty set

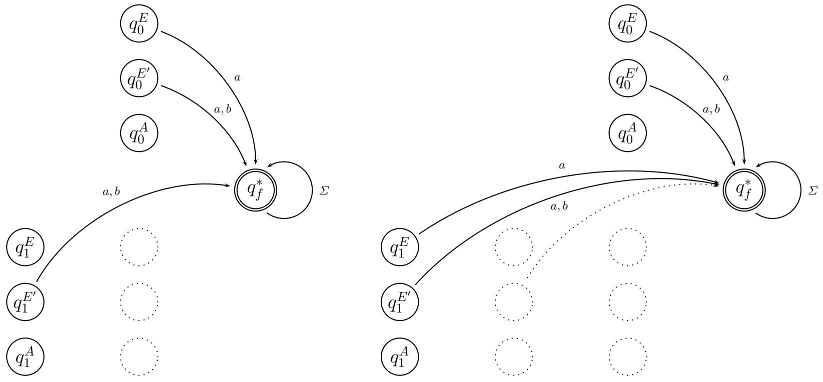


Fig. 6. The automaton after the first round of reachability analysis with the new Z_0; and the automaton after the second round of reachability analysis with the new Z_0

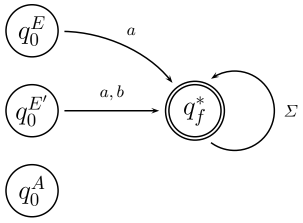


Fig. 7. The automaton accepting the winning region of Éloïse

given an automaton A recognising the configuration-sets $V(Z_1), \ldots, V(Z_{l-1})$, and

$$\chi_{l+1}(Z_1, \ldots, Z_l) := \sigma Z_{l+1} \ldots \sigma Z_m . \varphi_E(Z_1, \ldots, Z_m).$$

Format of the Automata. We describe the format of the automata constructed by the algorithm. Let $\mathcal{Q}_{all} := \{q^*, q^\varepsilon_f\}$, and $\mathcal{Q}_c := \left\{ q^j_c \mid 1 \leq j \leq |\mathcal{P}| \right\}$ for each $1 \leq c \leq m+1$. These states are used to give the valuations of the variables $Z_1, \ldots, Z_m$, and the semantics of $\varphi_E(Z_1, \ldots, Z_m)$ when $c = m+1$.

Let $0 \leq l \leq m+1$. An automaton A is said to be ***type-**l* just if:

(i) the state-set $\mathcal{Q}_A := \mathcal{Q}_1 \cup \cdots \cup \mathcal{Q}_l \cup \mathcal{Q}_{all}$
(ii) every transition of the form $q^j_c \xrightarrow{a} Q$ has the property that $Q \neq \emptyset$, and for all j' and $c' > c$, $q^{j'}_{c'} \notin Q$ (i.e. there are no transitions to states with a higher priority)
(iii) the only final state is q^ε_f, which can only be reached by a $\perp$-transition (i.e. for each $q \xrightarrow{a} Q$, we have $q^\varepsilon_f \in Q$ iff $Q = \{q^\varepsilon_f\}$ iff $a = \perp$); further, q^ε_f has no outgoing transitions
(iv) we have $q^* \xrightarrow{\Sigma} \{q^*\}$ and $q^* \xrightarrow{\perp} \{q^\varepsilon_f\}$, and q^* has no other outgoing transitions.

It follows that there is a unique automaton of type-0.

In the following, let A be a type-l automaton, where $1 \leq c \leq l \leq m+1$. We define $\mathcal{L}_c(A) \subseteq \mathcal{P}\,\Sigma^* \perp$ by: for $1 \leq j \leq |\mathcal{P}|$, $\langle p^j, w\rangle \in \mathcal{L}_c(A)$ just if w is accepted by A from the initial state q^j_c. Thus $\mathcal{L}_c(A)$ is intended to represent the current valuation of the variable Z_c; in case $l = m+1$, $\mathcal{L}_{m+1}(A)$ is intended to represent $[\![\varphi_E(Z_1, \ldots, Z_m)]\!]^{\mathcal{G}}_V$ where the valuation V maps Z_c to $\mathcal{L}_c(A)$. If we omit the subscript and write $\mathcal{L}(A)$, we mean $\mathcal{L}_l(A)$. By abuse of notation, we define $\mathcal{L}_q(A) \subseteq \Sigma^* \perp \cup \{\, \epsilon \,\}$ to be the set of words accepted by A from the state q (note that $\mathcal{L}_{q^*}(A) = \Sigma^* \perp$ and $\mathcal{L}_{q^\varepsilon_f}(A) = \{\, \varepsilon \,\}$).

Definition of the Algorithm. Given a pushdown parity game $\mathcal{G}$, the algorithm presented in Figure 8 computes $\mathcal{W}_E$, the winning region of $\mathcal{G}$:

$$\mathcal{W}_E = [\![\mu Z_1.\nu Z_2.\ldots.\sigma Z_{m-1}.\sigma Z_m.\varphi_E(Z_1, \ldots, Z_m)]\!]^{\mathcal{G}}_{\emptyset}.$$

When computing $[\![\varphi_E(Z_1, \ldots, Z_m)]\!]^{\mathcal{G}}_V$ we may add an exponential number of transitions. To compute $[\![\sigma Z_l. \cdots .\sigma Z_m \,.\, \varphi_E(Z_1, \cdots, Z_m)]\!]^{\mathcal{G}}_V$ we may require an exponential number of iterations. Hence, in the worst case, the algorithm is (singly) exponential in the number of control states and the maximum priority m.

Theorem 1. *Given a pushdown parity game $\mathcal{G} = (\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$, we can construct an automaton recognising the winning region of Éloïse in EXPTIME in $|\mathcal{P}| \cdot m$ where m is the maximum priority.*

The alternating multi-automaton returned by the algorithm, $\mathit{Sig}(1, A_0)$, has $n = |\mathcal{P}|+2$ states. The number of transitions is bounded by $n \cdot |\Sigma| \cdot 2^n$, which is independent of m.

5 Termination and Correctness

Termination. First an auxiliary notion of monotonicity for automaton constructions. Let $1 \leq l,\ l' \leq m+1$, and A and A' be type-l automata. We write $A \preceq A'$ to mean: for all q, a and Q, if $q \xrightarrow{a} Q$ is an A-transition then it is an A'-transition. We

```
Input: A pushdown parity game $\mathcal{G} = (\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$ with maximum priority $m$.
Output: A type-1 automaton recognising $[\![\chi_1]\!]^{\mathcal{G}}$, the winning region of $\mathcal{G}$.
begin
   return $Sig(1, A_0)$        % $A_0$ is the unique type-0 automaton.
end

procedure $Sig(l, A)$
Input: $1 \le l \le m + 1$;
        a type-$(l-1)$ automaton $A$ as valuation of $Z_1, \cdots, Z_{l-1}$.
Output: A type-$l$ automaton denoting $\sigma Z_l \cdots \sigma Z_m . \varphi_E(\overline{Z})$, relative to $A$.
 1. if $l = m + 1$ then return $Phi(A)$
 2. $A^0 := \begin{cases} A \text{ with new states } \mathcal{Q}_l\text{, but no new transitions} & \text{if } \sigma Z_l = \mu Z_l \\ A \text{ with new states } \mathcal{Q}_l\text{, and all outgoing} & \text{if } \sigma Z_l = \nu Z_l \\ \quad \text{transitions obeying the format of the automata.} & \end{cases}$
 3. for $i = 0$ to $\infty$ do
 4.       $B^i := Sig(l + 1, A^i)$
 5.       $A^{i+1} := Proj(l, B^i)$
 6.       if $A^i = A^{i+1}$ then return $A^i$

procedure $Phi(A)$
Input: A type-$m$ automaton $A$ as valuation of $\overline{Z} = Z_1, \cdots, Z_m$.
Output: A type-$(m+1)$ automaton denoting $\varphi_E(\overline{Z})$, relative to $A$.

 1. (1-Step Reachability) Construct the automaton $A'$ by adding new states
    $\{q^1_{m+1}, \ldots, q^{|\mathcal{P}|}_{m+1}\}$ and the following transitions to $A$. For each $1 \le j \le |\mathcal{P}|$,
    set $c := \Omega(p^j)$, and
     – if $p^j \in \mathcal{P}_E$ then $q^j_{m+1} \xrightarrow{a} Q$ if $q^k_c \xrightarrow{w} Q$ and $(p^k, w) \in Next(p^j, a)$
     – if $p^j \in \mathcal{P}_A$ then $q^j_{m+1} \xrightarrow{a} Q_1 \cup \cdots \cup Q_n$ if $q^{k_1}_c \xrightarrow{w_1} Q_1, \ldots, q^{k_n}_c \xrightarrow{w_n}$
       $Q_n$, and $Next(p^j, a) = \{(p^{k_1}, w_1), \ldots, (p^{k_n}, w_n)\}$
    where $Next(p^j, a) := \{\ (p^k, w) \mid p^j\ a \to p^k\ w\ \in\ \mathcal{D}\ \}$.
 2. return $A'$.

procedure $Proj(l, A)$
Input: $1 \le l \le m$; a type-$(l+1)$ automaton $A$.
Output: A type-$l$ automaton.

 1. For each $j$, replace each transition $q^j_{l+1} \xrightarrow{a} Q$ with $q^j_{l+1} \xrightarrow{a} \pi^l(Q)$ where
    $\pi^l(Q) := \{q^{j'}_{l+1} \mid q^{j'}_l \in Q\} \cup (Q - \mathcal{Q}_l)$.
 2. For each $j$, remove the state $q^j_l$.
 3. For each $j$, rename the state $q^j_{l+1}$ to $q^j_l$.
```

Fig. 8. Algorithm for computing winning region of a pushdown parity game

consider automaton constructions $\mathcal{T}$ (such as Sig, Phi and Proj) that transform type-l automata to type-l' automata. We say that $\mathcal{T}$ is *monotone* just if $\mathcal{T}(A) \preceq \mathcal{T}(A')$ whenever $A \preceq A'$.

To show that our winning-region construction procedure terminates, it suffices to prove the following.

Theorem 2 (Termination). *For every $1 \leq l \leq m+1$ and every type-$(l-1)$ automaton A, the procedure $\mathit{Sig}(l, A)$ terminates.*

We prove the theorem by induction on l. It is straightforward to establish the base case of $l = m+1$: $\mathit{Phi}(A)$ (where A is type-m) terminates. For the inductive case of $\mathit{Sig}(l, -)$ where $1 \leq l \leq m$, since $\mathit{Sig}(l+1, -)$ terminates by the induction hypothesis, and $\mathit{Proj}(l, -)$ clearly terminates, it remains to check that in the computation of $\mathit{Sig}(l, A)$ where A is type-$(l-1)$, there exists an $i \geq 0$ such that $A^i = A^{i+1}$. Since all automata of the same type have the same finite state-set (and $A^0, A^1, \ldots$ are all type-l) , it suffices to show (i) of the following Lemma.

Lemma 1 (Monotonicity). *We have the following properties.*

(i) Let $1 \leq l \leq m$ and A be a type-$(l-1)$ automaton. In $\mathit{Sig}(l, A)$:
   a. if $\sigma Z_l = \mu Z_l$ then $A^i \preceq A^{i+1}$ for all $i \geq 0$
   b. if $\sigma Z_l = \nu Z_l$ then $A^{i+1} \preceq A^i$ for all $i \geq 0$.

(ii) For every $1 \leq l \leq m+1$, the construction $\mathit{Sig}(l, -)$ is monotone.

(iii) For every $1 \leq l \leq m$, the construction $\mathit{Proj}(l, -)$ is monotone.

Correctness. To prove correctness, we introduce the notions of *valuation soundness* and *completeness*. Fix a pushdown parity game $\mathcal{G} = (\mathcal{P}, \mathcal{D}, \Sigma, \Omega)$. A *valuation profile* is a vector $\overline{S} = (S_1, \ldots, S_l)$ of configuration-sets (i.e. vertex-sets of the underlying configuration graph). We define the valuation $V_{\overline{S}} : Z_c \mapsto S_c$ induced by $\overline{S}$, which we extend to a map $V_{\overline{S}} : \mathcal{Q}_A \longrightarrow 2^{\Sigma^* \bot}$ on the states of a type-l automaton as follows:

$$V_{\overline{S}} := \begin{cases} q_c^j \mapsto \{\, w \mid \langle p^j, w \rangle \in S_c \,\} & 1 \leq j \leq |\mathcal{P}|,\ 1 \leq c \leq l \\ q^* \mapsto \Sigma^* \bot \\ q_f^\varepsilon \mapsto \{\, \varepsilon \,\} \end{cases}$$

Definition 1. Given a valuation profile $\overline{S}$ of length l, we say that a type-l automaton A is ***$\overline{S}$-sound*** just if, for all q, a and w, if A has a transition $q \xrightarrow{a} Q$ such that $w \in V_{\overline{S}}(q')$ for all $q' \in Q$, then $a\, w \in V_{\overline{S}}(q)$.

By induction on the length of the word, valuation soundness extends to runs of a multi-automaton. We then obtain that all accepting runs are sound.

Lemma 2. *Let A be a $\overline{S}$-sound automaton.*

(i) For all q, w and w', if A has a run $q \xrightarrow{w} Q$ such that $w' \in V_{\overline{S}}(q')$ for all $q' \in Q$, then $w\, w' \in V_{\overline{S}}(q)$.

(ii) For all $q \in \mathcal{Q}_A$, $\mathcal{L}_q(A) \subseteq V_{\overline{S}}(q)$.

Proof

(i) We prove by induction on the length of the word w. When $w = a$, the property is just $\overline{S}$-soundness. Take $w = au$ and some run $q \xrightarrow{a} Q \xrightarrow{u} Q'$ such that for all $q' \in Q'$, we have $w \in V_{\overline{S}}(q')$. By the induction hypothesis, we have the property for the run $Q \xrightarrow{u} Q'$. Hence, we have for all $q' \in Q$ that, $uw' \in V_{\overline{S}}(q')$. Thus, from $\overline{S}$-soundness, we have $auw' \in V_{\overline{S}}(q)$.

(ii) Take an accepting run $q \xrightarrow{w} Q_f$ of A. We have for all $q' \in Q_f = \{q_f^{\varepsilon}\}, \varepsilon \in V_{\overline{S}}(q')$. Thanks to (i), we have $w \in V_{\overline{S}}(q)$. □

Definition 2. Given a valuation profile $\overline{S}$ of length l, we say that a type-l automaton A is ***$\overline{S}$-complete*** just if, for all q, a and w, if $a\,w \in V_{\overline{S}}(q)$ then A has a transition $q \xrightarrow{a} Q$ such that $w \in V_{\overline{S}}(q')$ for all $q' \in Q$.

By induction on the length of the word, valuation completeness extends to runs. Furthermore, an accepting run always exists when required.

Lemma 3. *Let A be an $\overline{S}$-complete automaton.*

(i) For all q, w and w', if $w\,w' \in V_{\overline{S}}(q)$ then A has a run $q \xrightarrow{w} Q$ such that $w' \in V_{\overline{S}}(q')$ for all $q' \in Q$.
(ii) For all $q \in \mathcal{Q}_A$, $V_{\overline{S}}(q) \subseteq \mathcal{L}_q(A)$.

Notation. Recall $\chi_l(Z_1, \ldots, Z_{l-1}) := \sigma Z_l \cdots Z_m.\varphi_E(Z_1, \ldots, Z_m)$ where $1 \leq l \leq m+1$. Thus we have $\chi_1 = \mu Z_1 \ldots \sigma Z_m.\varphi_E(\overline{Z})$ and $\chi_{m+1}(Z_1, \ldots, Z_m) = \varphi_E(\overline{Z})$. Let $\overline{S} = (S_1, \ldots, S_{l-1})$; we write $(\overline{S}, T)$ to mean $(S_1, \ldots, S_{l-1}, T)$. Thus we write $\chi_l(\overline{S})$ to mean $\chi_l(S_1, \cdots, S_{l-1})$, and $\chi_{l+1}(\overline{S}, Z_l)$ to mean $\chi_{l+1}(S_1, \ldots, S_{l-1}, Z_l)$.

Proposition 1 (Main). *Let $1 \leq l \leq m+1$, A be a type-$(l-1)$ automaton, and $\overline{S}$ be a valuation profile of length $l-1$.*

*(i) (**Soundness Preservation**) If A is $\overline{S}$-sound, then Sig(l, A) is a type-l automaton which is $(\overline{S}, [\![\chi_l(\overline{S})]\!])$-sound.*[4]
*(ii) (**Completeness Preservation**) If A is $\overline{S}$-complete, then Sig(l, A) is a type-l automaton which is $(\overline{S}, [\![\chi_l(\overline{S})]\!])$-complete.*

Since the type-0 automaton A_0 is trivially sound and complete with respect to the empty valuation profile, we obtain the following as an immediate corollary.

Theorem 3 (Correctness). *The procedure call Sig$(1, A_0)$ terminates and returns a type-1 automaton which is $([\![\chi_1]\!])$-sound and $([\![\chi_1]\!])$-complete. Hence, thanks to Lemmas 2 and 3, for each $1 \leq j \leq |\mathcal{P}|$, $V_{[\![\chi_1]\!]}(q_1^j) = \mathcal{L}_{q_1^j}(Sig(1, A_0))$ i.e. the automaton Sig$(1, A_0)$ recognises the configuration set $[\![\chi_1]\!]$, which is the winning region of the pushdown parity game $\mathcal{G}$.*

[4] By $[\![\chi_l(S_1, \cdots, S_{l-1})]\!]$ we mean $[\![\chi_l(Z_1, \cdots, Z_{l-1})]\!]_V$ where V maps Z_c to S_c.

Proof of the Main Proposition. We prove Proposition 1 by induction on l. First the base case: $l = m + 1$.

Lemma 4. *Let $\overline{S}$ be a valuation profile of length m, and A a type-m automaton.*

(i) $Phi(A)$ is a type-$(m+1)$ automaton.
(ii) If A is $\overline{S}$-sound then $Phi(A)$ is $(\overline{S}, [\![\varphi_E(\overline{S})]\!])$-sound.
(iii) If A is $\overline{S}$-complete then $Phi(A)$ is $(\overline{S}, [\![\varphi_E(\overline{S})]\!])$-complete.

Proof

(i) We omit the straightforward proof.
(ii) Let $\overline{S'} = (\overline{S}, [\![\varphi_E(\overline{S})]\!])$ and $\Omega(p^j) = c$. Take any transition $q^j_{m+1} \xrightarrow{a} Q$ in $Phi(A)$ and stack w such that for all $q^{j'}_{c'} \in Q$, $\langle p^{j'}, w\rangle \in V_{\overline{S'}}(Z_{c'})$. For an Éloïse position, we abuse notation by interpreting $Next(p^j, a)$ as the singleton set containing the rule that led to the introduction of the new transition. Essentially, we present the proof for an Abelard position, which can be easily applied to Éloïse's positions. Since A is $\overline{S}$-sound and for all $(p^k, w_k) \in Next(p^j, a)$ we have $q^k_c \xrightarrow{w_k} Q_k \subseteq Q$, we know that $\langle p^k, w_k w\rangle \in V_{\overline{S'}}(Z_c)$. Hence all $\langle p^k, w_k w\rangle$ are in $V_{\overline{S'}}(Z_c)$, and $\langle p^j, aw\rangle \in V_{\overline{S'}}(Z_{m+1}) = [\![\varphi_E(\overline{Z})]\!]^{\mathcal{G}}_{V_{\overline{S'}}}$, since all moves, in the case of Abelard, and a move in the case of Éloïse, reach configurations in Z_c.
(iii) Take any configuration $\langle p^j, aw\rangle \in V_{\overline{S'}}(Z_{m+1}) = [\![\varphi_E(\overline{Z})]\!]^{\mathcal{G}}_{V_{\overline{S'}}}$. Let $\Omega(p^j) = c$. There exists an appropriate assignment $\{(p^{k_1}, w_1), \ldots, (p^{k_n}, w_n)\}$ to $Next(p^j, a)$ (as before) such that $\langle p^{k_h}, w_h w\rangle \in V_{\overline{S'}}(Z_c)$ for all $h \in \{1, \ldots, n\}$. Since A is assumed to be $\overline{S}$-complete, it follows that all $\langle p^{k_h}, w_h w\rangle$ have a complete run. In particular, we have a complete run $q^{k_h}_c \xrightarrow{w_h} Q_h$ for all h. Hence, by the definition of $Phi(A)$, there exists a transition $p^j \xrightarrow{a} Q$ that is complete. □

For the inductive case of $1 \leq l \leq m$, we present the proof when $\sigma Z_l = \mu Z_l$. The case of $\sigma Z_l = \nu Z_l$ is exactly dual (in outline, the soundness and completeness proofs are interchanged). Recall that $\chi_l(Z_1, \ldots, Z_{l-1}) := \sigma Z_l.\chi_{l+1}(Z_1, \cdots, Z_l)$.

Lemma 5. *Suppose $\sigma Z_l = \mu Z_l$. Let $\overline{S}$ be a valuation profile of length $l-1$, and A be a type-$(l-1)$ automaton; set $\theta = [\![\mu Z_l.\chi_{l+1}(\overline{S}, Z_l)]\!]$.*

(i) $Sig(l, A)$ is a type-l automaton.
(ii) If A is $\overline{S}$-sound, then $Sig(l, A)$ is $(\overline{S}, \theta)$-sound.
(iii) If A is $\overline{S}$-complete, then $Sig(l, A)$ is $(\overline{S}, \theta)$-complete.

Proof

(i) The result of the recursive call to $Sig(l+1, A)$ combined with the call to $Proj$ ensures the property.
(ii) Let $\overline{S'} := (\overline{S}, \theta)$. It is straightforward to see that A^0 is $\overline{S'}$-sound, since it did not add any transitions to A, which is assumed to be $\overline{S}$-sound. Hence, we assume by induction A^i is $\overline{S'}$-sound and argue the case for A^{i+1}.
Take a transition $q^j_l \xrightarrow{a} Q$ in A^{i+1} such that for all $q^k_{l'} \in Q$ we have $\langle p^k, w\rangle \in V_{\overline{S'}}(Z_{l'})$. Take the corresponding transition $q^j_{l+1} \xrightarrow{a} Q'$ in $Sig(l+1, A^i)$ before

the projection. In particular, for every $q_l^k \in Q$ we have q_l^k or q_{l+1}^k in Q'. By the induction hypothesis, we know $Sig(l+1, A^i)$ is $(\overline{S'}, [\![\chi_{l+1}(\overline{S'})]\!])$-sound. Furthermore, $V_{\overline{S'}}(Z_l) = \theta = [\![\chi_{l+1}(\overline{S}, \theta)]\!] = V_{\overline{S'}}(Z_{l+1})$. Since $Sig(l+1, A^i)$ is $(\overline{S'}, [\![\chi_{l+1}(\overline{S'})]\!])$-sound, we have $\langle p^j, aw\rangle \in V_{\overline{S'}}(Z_{l+1}) = V_{\overline{S'}}(Z_l)$ as required.

(iii) Let A be a type-$(l-1)$ automaton which is $\overline{S}$-complete. We use the shorthand $\theta^\alpha = [\![\mu^\alpha Z_l.\chi_{l+1}(\overline{S}, Z_l)]\!]$. We first show that if the type-l A^i is $(\overline{S}, \theta^\alpha)$-complete for some α then A^{i+1} is $(\overline{S}, \theta^{\alpha+1})$-complete. By the induction hypothesis, $B^i := Sig(l+1, A^i)$ is $(\overline{S}, \theta^\alpha, \theta^{\alpha+1})$-complete, since $\theta^{\alpha+1} = [\![\chi_{l+1}(\overline{S}, \theta^\alpha)]\!]$. We need to show that, after the projection, $A^{i+1} := Proj(l, B^i)$ is $\overline{S'}$-complete, where $\overline{S'} := (\overline{S}, \theta^{\alpha+1})$. Take some $\langle p^j, aw\rangle \in V_{\overline{S'}}(Z_l)$. We know B^i has a transition $q_{l+1}^j \xrightarrow{a} Q$ satisfying completeness. If Q contains no states of the form q_l^k, then the transition $q_l^j \xrightarrow{a} Q$ satisfies completeness in A^{i+1}. If Q contains states q_l^k, then $\langle p^k, w\rangle \in \theta^\alpha \subseteq \theta^{\alpha+1} = V_{\overline{S'}}(Z_l)$. Hence, we have a required complete transition after the projection, and so, A^{i+1} is $\overline{S'}$-complete. We require that $Sig(l, A)$ be $(\overline{S}, [\![\mu Z_l.\chi_{l+1}(\overline{S}, Z_l)]\!])$-complete. Take i such that $A^i = A^{i+1} = Sig(l, A)$. Trivially $Sig(l, A)$ is $(\overline{S}, \theta^0)$-complete. We proceed by transfinite induction. For a successor ordinal we know by induction that A^i is $(\overline{S}, \theta^\alpha)$-complete and from the above that A^{i+1} is $(\overline{S}, \theta^{\alpha+1})$-complete. Since $Sig(l, A) = A^i = A^{i+1}$ we are done. For a limit ordinal λ, we have that $Sig(l, A)$ is $(\overline{S}, \theta^\alpha)$-complete for all $\alpha < \lambda$. Since $\theta^\lambda = \bigcup_{\alpha<\lambda} \theta^\alpha$, the result follows because each configuration in the limit appears in some smaller approximant, and the transition witnessing completeness for the approximant witnesses completeness for the limit. □

6 Optimisation

In the procedure $Sig(l, A)$, in case $\sigma Z_l = \nu Z_l$, our definition of A^0 contains all allowable transitions, and hence is immediately exponential. However, if we have $q \xrightarrow{a} Q$ and $q \xrightarrow{a} Q'$ with $Q \subseteq Q'$, then acceptance from Q' implies acceptance from Q. That is, the transition to Q' is redundant. Furthermore, acceptance from any q_c^j implies acceptance from q^* (trivially). Using these observations, we can optimise our automaton. In the following definition, $Q \ll Q'$ can be taken to mean an accepting run from Q' implies an accepting run from Q.

Definition 3. *For all non-empty sets of states Q and Q', we define*

$$Q \ll Q' \;:=\; \big((q^* \in Q \Rightarrow \exists q.q \neq q_f^\varepsilon \wedge q \in Q') \;\wedge\; (\forall q \neq q^*.q \in Q \Rightarrow q \in Q')\big)$$

and $\textsc{Expand}(A) := \{\, q \xrightarrow{a} Q' \mid q \xrightarrow{a} Q$ *in* A *and* $Q \ll Q' \,\}$.

By specifying monotonicity with respect to $\textsc{Expand}(A)$ rather than A, A^0 (in case $\sigma Z_l = \nu Z_l$) only needs transitions to q^* and q_f^ε, which is linear. When this optimisation is used in the case of a one-player game, the constructed automaton will not use any alternating transitions. Furthermore, we can remove redundant transitions at every stage of the algorithm. Since a transition to $\{q^*\}$ is powerful with respect to $\ll$ we expect to keep the automaton small. However, this will have to be confirmed experimentally.

To test termination of $Sig(A, l)$, we check if $\textsc{Expand}(A^{i+1}) = \textsc{Expand}(A^i)$.

Lemma 6. $\textsc{Expand}(A) \preceq \textsc{Expand}(A')$ *if and only if whenever* $q \xrightarrow{a} Q$ *in* A *then there is some* $Q' \ll Q$ *with* $q \xrightarrow{a} Q'$ *in* A'.

By induction, we extend the property to runs. Hence $\textsc{Expand}(A) \preceq \textsc{Expand}(A')$ implies $\mathcal{L}(A) \subseteq \mathcal{L}(A')$. Finally, we have:

Lemma 7. *The optimisation preserves monotonicity and both valuation soundness and valuation completeness.*

Conclusion. We have proposed a new, simple and direct algorithm for computing the winning region of a pushdown parity game. The algorithm uses a mu-calculus formula that characterises Éloïse's winning region as a guide to construct the required automaton. We have identified an optimisation that avoids an immediate exponential blow up. An interesting open problem is to construct winning strategies using our approach.

Acknowledgments. This work is supported by EPSRC (EP/F036361). We are greatly indebted to Arnaud Carayol for his invaluable assistance.

References

1. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
2. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. In: INFINITY (1997)
3. Walukiewicz, I.: Pushdown processes: Games and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 62–74. Springer, Heidelberg (1996)
4. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS 1991, pp. 368–377 (1991)
5. Esparza, J., Kučera, A., Schwoon, S.: Model-checking LTL with regular valuations for pushdown systems. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 316–339. Springer, Heidelberg (2001)
6. Hague, M.: Saturation methods for global model-checking pushdown systems. PhD. Thesis, University of Oxford (2009)
7. Jones, N., Muchnick, S.: Even simple programs are hard to analyse. JACM 24, 338–350 (1977)
8. Piterman, N., Y. Vardi, M.: Global model-checking of infinite-state systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 387–400. Springer, Heidelberg (2004)
9. Serre, O.: Note on winning positions on pushdown games with ω-regular conditions. Information Processing Letters 85, 285–291 (2003)
10. Schwoon, S.: Model-checking Pushdown Systems. PhD thesis, Tech. Univ., Munich (2002)
11. Ball, T., Rajamani, S.K.: Bebop: A Symbolic Model Checker for Boolean Programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
12. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: POPL, pp. 1–3 (2002)
13. Cachat, T.: Games on Pushdown Graphs and Extensions. PhD thesis, RWTH Aachen (2003)
14. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. (2005)

Concurrent Kleene Algebra

C.A.R. Tony Hoare[1], Bernhard Möller[2], Georg Struth[3], and Ian Wehrman[4]

[1] Microsoft Research, Cambridge, UK
[2] Universität Augsburg, Germany
[3] University of Sheffield, UK
[4] University of Texas at Austin, USA

Abstract. A concurrent Kleene algebra offers, next to choice and iteration, operators for sequential and concurrent composition, related by an inequational form of the exchange law. We show applicability of the algebra to a partially-ordered trace model of program execution semantics and demonstrate its usefulness by validating familiar proof rules for sequential programs (Hoare triples) and for concurrent ones (Jones's rely/guarantee calculus). This involves an algebraic notion of invariants; for these the exchange inequation strengthens to an equational distributivity law. Most of our reasoning has been checked by computer.

1 Introduction

Kleene algebra [7] has been recognised and developed [18,19,8] as an algebraic framework (or structural equivalence) that unifies diverse theories for conventional sequential programming by axiomatising the fundamental concepts of choice, sequential composition and finite iteration. Its many familiar models include relations under union, relational composition and reflexive transitive closure, as well as formal languages under union, concatenation and Kleene star. This paper defines a 'double' Kleene algebra, which adds an operator for concurrent composition. Sequential and concurrent composition are related by an inequational weakening of the equational exchange law $(a \circ b) \bullet (c \circ d) \;=\; (a \bullet c) \circ (b \bullet d)$ of two-category or bicategory theory (e.g. [20]). Under certain conditions, this is strengthened to an equational law, by which concurrent composition distributes through sequential. The axioms proposed for a concurrent Kleene algebra are catalogued in Section 4.

The interest of concurrent Kleene algebra (CKA) is two-fold. Firstly, it expresses only the essential properties of program execution; indeed, it represents just those properties which are preserved even by architectures with weakly-ordered memory access, unreliable communications and massively re-ordering program optimisers. Secondly, the modelled properties, though unusually weak, are strong enough to validate the main structural laws of assertional reasoning about program correctness, both in sequential style [12] (as described in Section 5) and in concurrent style [17] (as described in Section 8).

The purpose of the paper is to introduce the basic operations and their laws, both in a concrete representation and in abstract, axiomatic form. We hope in future research to relate CKA to various familiar process algebras, such as the π-calculus or CSP, and to clarify the links between their many variants.

Before we turn to the abstract treatment, Section 2 presents our weak semantic model which also is a concrete model of the notion of a CKA. A program is identified with the set of traces of all the executions it may evoke. Each trace consists of the set of events that occur during a single execution. When two subprograms are combined, say in a sequential or a concurrent combination, each event that occurs is an event in the trace of exactly one of the subprograms. Each trace of the combination is therefore the disjoint union of a trace of one of the sub-programs with a trace of the other. Our formal definitions of the program combinators identify them as a kind of separating conjunction [25].

We use a primitive dependence relation between the events of a trace. Its transitive closure represents a direct or indirect chain of dependence. In a sequential composition, it is obviously not allowed for an event occurring in execution of the first operand to depend (directly or indirectly) on an event occurring in execution of the second operand. We take this as our definition of a very weak form of sequential composition. Concurrent composition places no such restriction, and allows dependence in either direction. The above-mentioned exchange law seems to generally capture the interrelation between sequential and concurrent composition in adequate inequational form.

The dependence primitive is intended to model a wide range of computational phenomena, including control dependence (arising from program structure) and data dependence (arising from flow of data). There are many forms of data flow. Flow of data across time is usually mediated by computer memory, which may be private or shared, strongly or only weakly consistent. Flow of data across space is usually mediated by a real or simulated communication channel, which may be buffered or synchronised, double-ended or multiplexed, reliable or lossy, and perhaps subject to stuttering or even re-ordering of messages.

Obviously, it is only weak properties of a program that can be proved without knowing more of the properties of the memory and communication channels involved. The additional properties are conveniently specified by additional axioms, like those used by hardware architects to describe specific weak memory models (e.g. [22]). Fortunately, as long as they are consistent with our fundamental theory, they do not invalidate our development and hence do not require fresh proofs of any of our theorems.

In this paper we focus on the basic concrete CKA model and the essential laws; further technical details are given in the companion paper [15]. The formal proofs of our results can be found in [14]; there we also show a typical input file for the automated theorem prover Prover9 [28] with which all the purely algebraic proofs have been reconstructed automatically.

2 Operators on Traces and Programs

We assume a set EV of *events*, i.e., occurrences of primitive actions, and a *dependence relation* $\rightarrow \subseteq EV \times EV$ between them: $p \rightarrow q$ indicates occurrence of a data flow or control flow from event p to event q.

A *trace* is a set of events. We use *sets* of events to have greater flexibility, in particular, to accommodate non-interleaving semantics; the more conventional *sequences* can be recovered by adding time stamps or the like to events. A *program* is a set of traces. For example, the program skip, which does nothing, is defined as $\{\emptyset\}$, and the program $[p]$, which has the only event p, is $\{\{p\}\}$. The program $\mathsf{false} =_{df} \emptyset$ has no traces, and therefore cannot be executed at all. It serves the rôle of the 'miracle' [23] in the development of programs by stepwise refinement. We have $\mathsf{false} \subseteq P$ for all programs P.

Following [16] we will define four operators on programs P and Q:

$P * Q$	fine-grain concurrent composition, allowing dependences between P and Q ;
$P \,;\, Q$	weak sequential composition, forbidding dependence of P on Q ;
$P \parallel Q$	disjoint parallel composition, with no dependence in either direction;
$P \,[]\, Q$	alternation – exactly one of P or Q is executed, whenever possible.

For the formal definition let $\rightarrow^+$ be the transitive closure of the dependence relation $\rightarrow$ and let, for trace tp, be $dep(tp) =_{df} \{q \mid \exists\, p \in tp : q \rightarrow^+ p\}$. Thus, $dep(tp)$ is the set of events on which some event in tp depends. Therefore, trace tp is independent of trace tq iff $dep(tp) \cap tq = \emptyset$. The use of the transitive closure $\rightarrow^+$ seems intuitively reasonable; an algebraic justification is given in Section 7.

Definition 2.1. Consider the schematic combination function

$$\mathrm{COMB}(P, Q, C) =_{df} \{tp \cup tq \mid tp \in P \wedge tq \in Q \wedge tp \cap tq = \emptyset \wedge C(tp, tq)\}$$

with programs P, Q and a predicate C in the trace variables tp and tq. Then the above operators are given by

$$\begin{array}{lll} P * Q & =_{df} & \mathrm{COMB}(P, Q, \mathsf{TRUE}) , \\ P \;;\; Q & =_{df} & \mathrm{COMB}(P, Q, dep(tp) \cap tq = \emptyset) , \\ P \parallel Q & =_{df} & \mathrm{COMB}(P, Q, dep(tp) \cap tq = \emptyset \wedge dep(tq) \cap tp = \emptyset) , \\ P \,[]\, Q & =_{df} & \mathrm{COMB}(P, Q, tp = \emptyset \vee tq = \emptyset) . \end{array}$$

Example 2.2. We illustrate the operators with a mini-example. We assume a set EV of events the actions of which are simple assignments to program variables. We consider three particular events ax, ay, az associated with the assignments $x := x + 1, y := y + 2, z := x + 3$, resp. There is a dependence arrow from event p to event q iff $p \neq q$ and the variable assigned to in p occurs in the assigned expression in q. This means that for our three events we have exactly $ax \rightarrow az$. We form the corresponding single-event programs $P_x =_{df} [ax], P_y =_{df} [ay], P_z =_{df} [az]$. To describe their compositions we extend the notation for single-event programs and set $[p_1, \ldots, p_n] =_{df} \{\{p_1, \ldots, p_n\}\}$ (for uniformity we sometimes also write $[\,]$ for skip). Figure 1 lists the composition tables for our operators on these programs. They show that the operator $*$ allows forming parallel programs with race conditions, whereas ; and $\parallel$ respect dependences. □

$*$	P_x	P_y	P_z
P_x	$\emptyset$	$[ax, ay]$	$[ax, az]$
P_y	$[ax, ay]$	$\emptyset$	$[ay, az]$
P_z	$[ax, az]$	$[ay, az]$	$\emptyset$

$;$	P_x	P_y	P_z
P_x	$\emptyset$	$[ax, ay]$	$[ax, az]$
P_y	$[ax, ay]$	$\emptyset$	$[ay, az]$
P_z	$\emptyset$	$[ay, az]$	$\emptyset$

$\|$	P_x	P_y	P_z
P_x	$\emptyset$	$[ax, ay]$	$\emptyset$
P_y	$[ax, ay]$	$\emptyset$	$[ay, az]$
P_z	$\emptyset$	$[ay, az]$	$\emptyset$

$[\!]$	P_x	P_y	P_z
P_x	$\emptyset$	$\emptyset$	$\emptyset$
P_y	$\emptyset$	$\emptyset$	$\emptyset$
P_z	$\emptyset$	$\emptyset$	$\emptyset$

Fig. 1. Composition tables

It is straightforward from the definitions that $*, \|$ and $[\!]$ are commutative and that $[\!] \subseteq \| \subseteq ; \subseteq *$ where for $\circ, \bullet \in \{*, ;, \|, [\!]\}$ the formula $\circ \subseteq \bullet$ abbreviates $\forall\, P, Q : P \circ Q \subseteq P \bullet Q$. Further useful laws are the following.

Lemma 2.3. *Let* $\circ, \bullet \in \{*, ;, \|, [\!]\}$.

1. $\circ$ *distributes through arbitrary unions; in particular,* false *is an annihilator for* $\circ$, *i.e.,* $\mathsf{false} \circ P = \mathsf{false} = P \circ \mathsf{false}$. *Moreover,* $\circ$ *is isotone w.r.t.* $\subseteq$ *in both arguments.*
2. skip *is a neutral element for* $\circ$, *i.e.,* $\mathsf{skip} \circ P = P = P \circ \mathsf{skip}$.
3. *If* $\bullet \subseteq \circ$ *and* $\circ$ *is commutative then* $(P \circ Q) \bullet (R \circ S) \subseteq (P \bullet R) \circ (Q \bullet S)$.
4. *If* $\bullet \subseteq \circ$ *then* $P \bullet (Q \circ R) \subseteq (P \bullet Q) \circ R$.
5. *If* $\bullet \subseteq \circ$ *then* $(P \circ Q) \bullet R \subseteq P \circ (Q \bullet R)$.
6. $\circ$ *is associative.*

The proofs either can be done by an easy adaptation of the corresponding ones in [16] or follow from more general results in [15]. A particularly important special case of Part 3 is the exchange law

$$(P * Q) \,;\, (R * S) \subseteq (P \,;\, R) * (Q \,;\, S) \tag{1}$$

In the remainder of this paper we shall mostly concentrate on the more interesting operators $*$ and $;$.

Another essential operator is union which again is $\subseteq$-isotone and distributes through arbitrary unions. However, it is *not* false-strict.

By the Tarski-Kleene fixpoint theorems all recursion equations involving only the operators mentioned have $\subseteq$-least solutions which can be approximated by the familiar fixpoint iteration starting from false. Use of union in such a recursions enables non-trivial fixpoints, as will be seen in the next section.

3 Quantales, Kleene and Omega Algebras

We now abstract from the concrete case of programs and embed our model into a more general algebraic setting.

Definition 3.1. A *semiring* is a structure $(S, +, 0, \cdot, 1)$ such that $(S, +, 0)$ is a commutative monoid, $(S, \cdot, 1)$ is a monoid, multiplication distributes over addition in both arguments and 0 is a left and right annihilator with respect to multiplication ($a \cdot 0 = 0 = 0 \cdot a$). A semiring is *idempotent* if its addition is.

The operation $+$ denotes an abstract form of nondeterministic choice; in the concrete case of programs it will denote union (of sets of traces). This explains why $+$ is required to be associative, commutative and idempotent. Its neutral element 0 will take the rôle of the miraculous program $\emptyset$.

In an idempotent semiring, the relation $\leq$ defined by $a \leq b \Leftrightarrow_{df} a + b = b$ is a partial ordering, in fact the only partial ordering on S for which 0 is the least element and for which addition and multiplication are isotone in both arguments. It is therefore called the *natural ordering* on S. This makes S into a semilattice with addition as join and least element 0.

Definition 3.2. A *quantale* [24] or *standard Kleene algebra* [7] is an idempotent semiring that is a complete lattice under the natural order and in which multiplication distributes over arbitrary suprema. The infimum and the supremum of a subset T are denoted by $\sqcap T$ and $\sqcup T$, respectively. Their binary variants are $x \sqcap y$ and $x \sqcup y$ (the latter coinciding with $x + y$).

In particular, quantale composition is *continuous*, i.e., distributes through suprema of arbitrary, not just countable, chains. As an idempotent semiring, every quantale has 0 as its least element. As a complete lattice, it also has a greatest element $\top$. Quantales have been used in many contexts other than that of program semantics, see e.g. the c-semirings of [3] or the general reference [29].

Let now $PR(EV) =_{df} \mathcal{P}(\mathcal{P}(EV))$ denote the set of all programs over the event set EV. From the observations in Section 2 the following is immediate:

Lemma 3.3. $(PR(EV), \cup, \mathsf{false}, *, \mathsf{skip})$ *and* $(PR(EV), \cup, \mathsf{false}, ;, \mathsf{skip})$ *are quantales. In each of them* $\top = \mathcal{P}(EV)$ *is the most general program over* EV.

Definition 3.4. In a quantale S, the finite and infinite iterations a^* and a^ω of an element $a \in S$ are defined by $a^* = \mu x \,.\, 1 + a \cdot x$ and $a^\omega = \nu x \,.\, a \cdot x$, where μ and ν denote the least and greatest fixpoint operators. The star here should not be confused with the separation operator $*$ above.

It is well known that $(S, +, 0, \cdot, 1, ^*)$ forms a Kleene algebra [18]. From this we obtain many useful laws for free. As examples we mention

$$1 \leq a^* \;, \quad a \leq a^* \;, \quad a^* \cdot a^* = (a^*)^* = a^* \;, \quad (a+b)^* = a^* \cdot (b \cdot a^*)^* \;. \qquad \text{(KA)}$$

The finite non-empty iteration of a is defined as $a^+ =_{df} a \cdot a^* = a^* \cdot a$. Again, the plus in a^+ should not be confused with the plus of semiring addition.

Since in a quantale the function defining star is continuous, Kleene's fixpoint theorem shows that $a^* = \bigsqcup_{i \in \mathbb{N}} a^i$. Moreover, we have the star induction rules

$$b + a \cdot x \leq x \Rightarrow a^* \cdot b \leq x \;, \qquad b + x \cdot a \leq x \Rightarrow b \cdot a^* \leq x \;. \qquad (2)$$

Our main reason for using quantales rather than an extension of conventional Kleene algebra (see e.g. the discussion on Priscariu's synchronous Kleene algebras [27] in Section 9) is the availability of general fixpoint calculus there. A number of our proofs need the principle of fixpoint fusion which is a second-order principle; in the first-order setting of conventional Kleene algebra only special cases of it, like the above induction rules can be added as axioms.

We now explain the behaviour of iteration in our program quantales. For a program P, the program P^* taken in the quantale $(PR(EV), \cup, \mathsf{false}, ;, \mathsf{skip})$ consists of all sequential compositions of finitely many traces in P; it is denoted by P^∞ in [16]. The program P^* taken in $(PR(EV), \cup, \mathsf{false}, *, \mathsf{skip})$ consists of all disjoint unions of finitely many traces in P; it may be considered as describing all finite parallel spawnings of traces in P. The disjointness requirement that is built into the definition of $*$ and ; does not mean that an iteration cannot repeat primitive actions: the iterated program just needs to supply sufficiently many (e.g., countably many) events having the actions of interest as labels. Then in each round of iteration a fresh one of these can be used.

Example 3.5. With the notation of Example 2.2 let $P =_{df} P_x \cup P_y \cup P_z$. We first look at its powers w.r.t. $*$ composition:

$$\begin{aligned} P^2 &= P * P = [ax, ay] \cup [ax, az] \cup [ay, az] \ , \\ P^3 &= P * P * P = [ax, ay, az] \ . \end{aligned}$$

Hence P^2 and P^3 consist of all programs with exactly two and three events from $\{ax, ay, az\}$, respectively. Since none of the traces in P is disjoint from the one in P^3, we have $P^4 = P^3 * P = \emptyset$, and hence strictness of $*$ w.r.t. $\emptyset$ implies $P^n = \emptyset$ for all $n \geq 4$. Therefore P^* consists of all traces with at most three events from $\{ax, ay, az\}$ (the empty trace is there, too, since by definition skip is contained in every program of the form Q^*). It coincides with the set of all possible traces over the three events; this connection will be taken up again in Section 6.

It turns out that for the powers of P w.r.t. the ; operator we obtain exactly the same expressions, since for every program $Q = [p] \cup [q]$ with $p \neq q$ we have

$$Q\,;Q = ([p] \cup [q])\,;([p] \cup [q]) = [p]\,;[p] \cup [p]\,;[q] \cup [q]\,;[p] \cup [q]\,;[q] = [p, q] = Q * Q \ ,$$

provided $p \not\to^+ q$ or $q \not\to^+ p$, i.e., provided the trace $[p, q]$ is consistent with the dependence relation. Only in case of a cyclic dependence $p \to^+ q \to^+ p$ we have $Q\,;Q = \emptyset$, whereas still $Q * Q = [p, q]$.

□

If the complete lattice $(S, \leq)$ in a quantale is completely distributive, i.e., if $+$ distributes over arbitrary infima, then $(S, +, 0, \cdot, 1, {}^*, {}^\omega)$ forms an omega algebra in the sense of [6]. Again this entails many useful laws, e.g.,

$$1^\omega = \top \ , \qquad (a \cdot b)^\omega = a \cdot (b \cdot a)^\omega \ , \qquad (a + b)^\omega = a^\omega + a^* \cdot b \cdot (a + b)^\omega \ .$$

Since $PR(EV)$ is a power set lattice, it is completely distributive. Hence both program quantales also admit infinite iteration with all its laws. The infinite iteration P^ω w.r.t. the composition operator $*$ is similar to the unbounded parallel spawning $!P$ of traces in P in the π-calculus (e.g. [30]).

4 Concurrent Kleene Algebras

That $PR(EV)$ is a double quantale motivates the following abstract definition.

Definition 4.1. By a *concurrent Kleene algebra* (CKA) we mean a structure $(S,+,0,*,;,1)$ such that $(S,+,0,*,1)$ and $(S,+,0,;,1)$ are quantales linked by the exchange axiom $(a*b);(c*d) \leq (b;c)*(a;d)$.

This definition implies, in particular, that $*$ and $;$ are isotone w.r.t. $\leq$ in both arguments. Compared to the original exchange law (1) this one has its free variables in a different order. This does no harm, since the concrete $*$ operator on programs is commutative and hence satisfies the above law as well.

Corollary 4.2. *$(PR(EV), \cup, \mathsf{false}, *, ;, \mathsf{skip})$ is a CKA.*

The reason for our formulation of the exchange axiom here is that this form of the law implies commutativity of $*$ as well as $a;b \leq a*b$ and hence saves two axioms. This is shown by the following result.

Lemma 4.3. *In a CKA the following laws hold.*
1. $a*b = b*a$.
2. $(a*b);(c*d) \leq (a;c)*(b;d)$.
3. $a;b \leq a*b$.
4. $(a*b);c \leq a*(b;c)$.
5. $a;(b*c) \leq (a;b)*c$.

The notion of a CKA abstracts completely from traces and events; in the companion paper [15] we show how to retrieve these notions algebraically using the lattice-theoretic concept of atoms.

5 Hoare Triples

In [16] Hoare triples relating programs are defined by $P\,\{Q\}\,R \Leftrightarrow_{df} P;Q \subseteq R$. Again, it is beneficial to abstract from the concrete case of programs.

Definition 5.1. An *ordered monoid* is a structure $(S,\leq,\cdot,1)$ such that $(S,\cdot,1)$ is a monoid with a partial order $\leq$ and $\cdot$ is isotone in both arguments. In this case we define the *Hoare triple* $a\,\{b\}\,c$ by

$$a\,\{b\}\,c \Leftrightarrow_{df} a\cdot b \leq c\ .$$

Lemma 5.2. *Assume an ordered monoid $(S,\leq,\cdot,1)$.*
1. $a\,\{1\}\,c \Leftrightarrow a \leq c$; *in particular,* $a\,\{1\}\,a \Leftrightarrow \mathsf{TRUE}$. *(skip)*
2. $(\forall a,c : a\,\{b\}\,c \Rightarrow a\,\{b'\}\,c) \Leftrightarrow b' \leq b$. *(antitony)*
3. $(\forall a,c : a\,\{b\}\,c \Leftrightarrow a\,\{b'\}\,c) \Leftrightarrow b = b'$. *(extensionality)*
4. $a\,\{b\cdot b'\}\,c \Leftrightarrow \exists\, d : a\,\{b\}\,d \wedge d\,\{b'\}\,c$. *(composition)*
5. $a \leq d \wedge d\,\{b\}\,e \wedge e \leq c \Rightarrow a\,\{b\}\,c$. *(weakening)*

If $(S,\cdot,1)$ is the multiplicative reduct of an idempotent semiring $(S,+,0,\cdot,1)$ and the order used in the definition of Hoare triples is the natural semiring order then we have in addition

6. $a\,\{0\}\,c \Leftrightarrow \mathsf{TRUE}$, *(failure)*
7. $a\,\{b+b'\}\,c \Leftrightarrow a\,\{b\}\,c \wedge a\,\{b'\}\,c$. *(choice)*
If that semiring is a quantale then we have in addition
8. $a\,\{b\}\,a \Leftrightarrow a\,\{b^+\}\,a \Leftrightarrow a\,\{b^*\}\,a$. *(iteration)*

Lemma 5.2 can be expressed more concisely in relational notation. Define for $b \in S$ the relation $\{b\} \subseteq S \times S$ between preconditions a and postconditions c by

$$\forall\, a, c : a\,\{b\}\,c \Leftrightarrow_{df} a \cdot b \le c\ .$$

Then the above properties rewrite into

1. $\{1\} = \le$.
2. $\{b\} \subseteq \{b'\} \Leftrightarrow b' \le b$.
3. $\{b\} = \{b'\} \;\Leftrightarrow\; b = b'$.
4. $\{b \cdot b'\} = \{b\} \circ \{b'\}$ where $\circ$ means relational composition.
5. $\le \circ \{b\} \circ \le\; \subseteq\; \{b\}$.
6. $\{0\} = \mathbb{T}$ where $\mathbb{T}$ is the universal relation.
7. $\{b+b'\} = \{b\} \cap \{b'\}$.
8. $\{b\} \cap I = \{b^+\} \cap I = \{b^*\} \cap I$ where I is the identity relation.

Properties 4 and 2 allow us to determine the weakest premise ensuring that two composable Hoare triples establish a third one:

Lemma 5.3. *Assume again an ordered monoid* $(S, \le, \cdot, 1)$. *Then*

$$(\forall\, a, d, c : a\,\{b\}\,d \wedge d\,\{b'\}\,c \Rightarrow a\,\{e\}\,c) \;\Leftrightarrow\; e \le b \cdot b'\ .$$

Next we present two further rules that are valid in CKAs when the above monoid operation is specialised to sequential composition:

Lemma 5.4. *Let* $S = (S, +, 0, *, ;, 1)$ *be a CKA and* $a, a', b, b', c, c', d \in S$ *with* $a\,\{b\}\,c$ *interpreted as* $a\,;b \le c$.
1. $a\,\{b\}\,c \wedge a'\,\{b'\}\,c' \Rightarrow (a * a')\,\{b * b'\}\,(c * c')$. *(concurrency)*
2. $a\,\{b\}\,c \Rightarrow (d * a)\,\{b\}\,(d * c)$. *(frame rule)*

Let us interpret these results in our concrete CKA of programs. It may seem surprising that so many of the standard basic laws of Hoare logic should be valid for such a weak semantic model of programs. E.g., the definition of weak sequential composition allows all standard optimisations by compilers which shift independent commands between the operands of a semicolon. What is worse, weak composition does not require any data to flow from an assignment command to an immediately following read of the assigned variable. The data may flow to a different thread, which assigns a different value to the variable. In fact, weak sequential composition is required for any model of modern architectures, which allow arbitrary race conditions between fine-grain concurrent threads.

The validity of Hoare logic in this weak model is entirely due to a cheat: that we use the same model for our assertions as for our programs. Thus any weakness of the programming model is immediately reflected in the weakness of the assertion language and its logic. In fact, conventional assertions mention the current values of single-valued program variables; and this is not adequate for reasoning about general fine-grain concurrency. To improve precision here, assertions about the history of assigned values would seem to be required.

6 Invariants

We now deal with the set of events a program may use.

Definition 6.1. A *power invariant* is a program R of the form $R = \mathcal{P}(E)$ for a set $E \subseteq EV$ of events.

It consists of all possible traces that can be formed from events in E and hence is the most general program using only those events. The smallest power invariant is $\mathsf{skip} = \mathcal{P}(\emptyset) = \{\emptyset\}$. The term “invariant” expresses that often a program relies on the assumption that its environment only uses events from a particular subset, i.e., preserves the invariant of staying in that set.

Example 6.2. Consider again the event set EV form Example 2.2. Let V be a certain subset of the variables involved and let E be the set of all events that assign to variables in V. Then the environment Q of a given program P can be constrained to assign at most to the variables in V by requiring $Q \subseteq R$ with the power invariant $R =_{df} \mathcal{P}(E)$. The fact that we want P to be executed only in such environments is expressed by forming the parallel composition $P * R$. $\square$

If E is considered to characterise the events that are admissible in a certain context, a program P can be confined to using only admissible events by requiring $P \subseteq R$ for $R = \mathcal{P}(E)$. In the rely/guarantee calculus of Section 8 invariants will be used to express properties of the environment on which a program wants to rely (whence the name R).

Power invariants satisfy a rich number of useful laws (see [15] for details). The most essential ones for the purposes of the present paper are the following straightforward ones for arbitrary invariant R:

$$\mathsf{skip} \subseteq R\ , \qquad R * R \subseteq R\ . \tag{3}$$

We now again abstract from the concrete case of programs. It turns out that the properties in (3) largely suffice for characterising invariants.

Definition 6.3. An *invariant* in a CKA S is an element $r \in S$ satisfying $1 \leq r$ and $r * r \leq r$. These two axioms can equivalently be combined into $1 + r * r \leq r$. The set of all invariants of S is denoted by $I(S)$.

We now first give a number of algebraic properties of invariants that are useful in proving the soundness of the rely/guarantee-calculus in Section 8.

Theorem 6.4. *Assume a CKA S, an $r \in I(S)$ and arbitrary $a, b \in S$.*
1. $a \leq r \circ a$ *and* $a \leq a \circ r$.
2. $r\,;r \leq r$.
3. $r * r = r = r\,;r$.
4. $r\,;(a * b) \leq (r\,;a) * (r\,;b)$ *and* $(a * b)\,;r \leq (a\,;r) * (b\,;r)$.
5. $r\,;a\,;r \leq r * a$.
6. $a \in I(S) \Leftrightarrow a = a^*$, *where* * *is taken w.r.t.* $*$ *composition.*
7. *The least invariant comprising* a *is* a^* *where* * *is taken w.r.t.* $*$ *composition.*

Next we discuss the lattice structure of the set $I(S)$ of invariants.

Theorem 6.5. *Assume again a CKA S.*

1. $(I(S), \leq)$ *is a complete lattice with least element* 1 *and greatest element* $\top$.
2. *For* $r, r' \in I(S)$ *we have* $r \leq r' \Leftrightarrow r * r' = r'$. *This means that* $\leq$ *coincides with the natural order induced by the associative, commutative and idempotent operation* $*$ *on* $I(S)$.
3. *For* $r, r' \in I(S)$ *the infimum* $r \sqcap r'$ *in* S *coincides with the infimum of* r *and* r' *in* $I(S)$.
4. $r * r'$ *is the supremum of* r *and* r' *in* $I(S)$. *In particular,* $r \leq r'' \wedge r' \leq r'' \Leftrightarrow r * r' \leq r''$ *and* $r' \sqcap (r * r') = r'$.
5. *Invariants are downward closed:* $r * r' \leq r'' \Rightarrow r \leq r''$.
6. $I(S)$ *is even closed under arbitrary infima, i.e., for a subset* $U \subseteq I(S)$ *the infimum* $\sqcap U$ *taken in* S *coincides with the infimum of* U *in* $I(S)$.

We conclude this section with two laws about iteration.

Lemma 6.6. *Assume a CKA* S *and let* $r \in I(S)$ *be an invariant and* $a \in S$ *be arbitrary. Let the finite iteration* * *be taken w.r.t.* $*$ *composition. Then*
1. $(r * a)^* \leq r * a^*$.
2. $r * a^* = r * (r * a)^*$.

7 Single-Event Programs and Rely/Guarantee-CKAs

We will now show that our definitions of $*$ and ; for concrete programs in terms of transitive closure of the dependence relation $\rightarrow$ entail two important further laws that are essential for the rely/guarantee calculus to be defined below. In the following theorem they are presented as inclusions; the reverse inclusions already follow from Theorem 6.4.4 for Part 1 and from Lemma 4.3.5, 4.3.4, 4.3.1 and Theorem 6.4.3 for Part 2. Informally, Part 1 means that for acyclic $\rightarrow$ parallel composition of an invariant with a singleton program can be always sequentialised. Part 2 means that for invariants a kind of converse to the exchange law of Lemma 4.3.2 holds.

Theorem 7.1. *Let* $R = \mathcal{P}(E)$ *be a power invariant in* $PR(EV)$.
1. *If* $\rightarrow$ *is acyclic and* $p \in EV$ *then* $R * [p] \subseteq R\,;[p]\,;R$.
2. *For all* $P, Q \in PR(EV)$ *we have* $R * (P\,;Q) \subseteq (R * P)\,;(R * Q)$.

In the companion paper [15] we define the composition operators ; and $\|$ in terms of $\rightarrow$ rather than $\rightarrow^+$ and show a converse of Theorem 7.1:

– If Part 1 is valid then $\rightarrow$ is weakly acyclic, viz.

$$\forall\, p, q \in EV : p \rightarrow^+ q \rightarrow^+ p \;\Rightarrow\; p = q\ .$$

This means that $\rightarrow$ allows at most immediate self-loops which cannot be "detected" by our definitions of the operators that require disjointness of the operands. It is easy to see that $\rightarrow$ is weakly acyclic iff its reflexive-transitive closure $\rightarrow^*$ is a partial order.

– If Part 2 is valid then $\rightarrow$ is weakly transitive, i.e.,

$$p \rightarrow q \rightarrow r \;\Rightarrow\; p = r \vee p \rightarrow r\ .$$

This provides the formal justification why in the present paper we right away defined our composition operators in terms of $\to^+$ rather than just $\to$.

As before we abstract the above results into general algebraic terms. The terminology stems from the applications in the next section.

Definition 7.2. A *rely/guarantee-CKA* is a pair (S, I) such that S is a CKA and $I \subseteq I(S)$ is a set of invariants such that $1 \in I$ and for all $r, r' \in I$ also $r \sqcap r' \in I$ and $r * r' \in I$, in other words, I is a sublattice of $I(S)$. Moreover, all $r \in I$ and $a, b \in S$ have to satisfy $r * (a \,;\, b) \leq (r * a) \,;\, (r * b)$.

Together with the exchange law in Lemma 4.3.2, $\circ$-idempotence of r and commutativity of $*$ this implies

$$r * (b \circ c) = (r * b) \circ (r * c) \qquad (*\text{-}distributivity)$$

for all invariants $r \in I$ and operators $\circ \in \{*, ;\}$.

The restriction that I be a sublattice of $I(S)$ is motivated by the rely/guarantee-calculus in Section 8 below.

Using Theorem 7.1 we can prove

Lemma 7.3. *Let $I =_{df} \{\mathcal{P}(E) \,|\, E \subseteq EV\}$ be the set of all power invariants over EV. Then $(PR(EV), I)$ is a rely-guarantee-CKA.*

Proof. We only need to establish closure of $\mathcal{P}(\mathcal{P}(EV))$ under $*$ and $\cap$. But straightforward calculations show that $\mathcal{P}(E) * \mathcal{P}(F) = \mathcal{P}(E \cup F)$ and $\mathcal{P}(E) \cap \mathcal{P}(F) = \mathcal{P}(E \cap F)$ for $E, F \subseteq EV$. □

We now can explain why it was necessary to introduce the subset I of invariants in a rely/guarantee-CKA. Our proof of $*$-distributivity used downward closure of power invariants. Other invariants in $PR(EV)$ need not be downward closed and hence $*$-distributivity need not hold for them.

Example 7.4. Assume an event set EV with three different events $p, q, r \in EV$ and dependences $p \to r \to q$. Set $P =_{df} [p, q]$. Then $P * P = \emptyset$ and hence $P^i = \emptyset$ for all $i > 1$. This means that the invariant $R =_{df} P^* = \mathsf{skip} \cup P = [\,] \cup [p, q]$ is not downward closed. Indeed, $*$-distributivity does not hold for it: we have $R * [r] = [r] \cup [p, q, r]$, but $R \,;\, [r] \,;\, R = [r]$. □

The property of $*$-distributivity implies further iteration laws.

Lemma 7.5. *Assume a rely/guarantee-CKA (S, I), an invariant $r \in I$ and an arbitrary $a \in S$ and let the finite iteration * be taken w.r.t. $\circ \in \{*, ;\}$.*
1. $r * a^* \;=\; (r * a)^* \circ r \;=\; r \circ (r * a)^*$.
2. $(r * a)^+ \;=\; r * a^+$.

8 Jones's Rely/Guarantee-Calculus

In [17] Jones has presented a calculus that considers properties of the environment on which a program wants to rely and the ones it, in turn, guarantees for the environment. We now provide an abstract algebraic treatment of this calculus.

Definition 8.1. We define, abstracting from [16], the *guarantee relation* by setting for arbitrary element b and invariant g

$$b \mathrel{\mathsf{guar}} g \;\Leftrightarrow_{df}\; b \leq g \;.$$

A slightly more liberal formulation is discussed in [15].

Example 8.2. With the notation $P_u =_{df} [au]$ for $u \in \{x, y, z\}$ of Example 2.2 we have $P_u \mathrel{\mathsf{guar}} G_u$ where $G_u =_{df} P_u \cup \mathsf{skip} = [au] \cup []$. $\square$

We have the following properties.

Theorem 8.3. *Let a, b, b' be arbitrary elements and g, g' be invariants of a CKA. Let $\circ \in \{*, ;\}$ and $^{*\circ}$ be the associated iteration operator.*

1. $1 \mathrel{\mathsf{guar}} g$.
2. $b \mathrel{\mathsf{guar}} g \wedge b' \mathrel{\mathsf{guar}} g' \;\Rightarrow\; (b \circ b') \mathrel{\mathsf{guar}} (g * g')$.
3. $a \mathrel{\mathsf{guar}} g \;\Rightarrow\; a^{*\circ} \mathrel{\mathsf{guar}} g$.
4. *For the concrete case of programs let $G = \mathcal{P}(E)$ for some set $E \subseteq EV$ and $p \in EV$. Then $[p] \mathrel{\mathsf{guar}} G \Leftrightarrow p \in E$.*

Using the guarantee relation, Jones quintuples can be defined, as in [16], by

$$a\; r\, \{b\}\, g\, s \;\Leftrightarrow_{df}\; a\, \{r * b\}\, s \;\wedge\; b \mathrel{\mathsf{guar}} g \;,$$

where r and g are invariants and Hoare triples are again interpreted in terms of sequential composition ; .

The first rule of the rely/guarantee calculus concerns parallel composition.

Theorem 8.4. *Consider a CKA S. For invariants $r, r', g, g' \in I(S)$ and arbitrary $a, a', b, b', c, c' \in S$,*

$$\begin{aligned} &a\;\, r\, \{b\}\, g\, c \;\wedge\; a'\;\, r'\, \{b'\}\, g'\, c' \;\wedge\; g' \mathrel{\mathsf{guar}} r \;\wedge\; g \mathrel{\mathsf{guar}} r' \;\Rightarrow \\ &\quad (a \sqcap a')\; (r \sqcap r')\, \{b * b'\}\, (g * g')\, (c \sqcap c') \;. \end{aligned}$$

Note that $r \sqcap r'$ and $g * g'$ are again invariants by Lemma 6.5.3 and 6.5.4.

For sequential composition one has

Theorem 8.5. *Assume a rely/guarantee-CKA (S, I). Then for invariants r, r', $g, g' \in I$ and arbitrary a, b, b', c, c',*

$$a\;\, r\, \{b\}\, g\, c \wedge c\;\, r'\, \{b'\}\, g'\, c' \;\Rightarrow\; a\;\, (r \sqcap r')\, \{b\, ;\, b'\}\, (g * g')\, c'$$

Next we give rules for 1, union and singleton event programs.

Theorem 8.6. *Assume a rely/guarantee-CKA (S, I). Then for invariants $r, g \in I$ and arbitrary $s \in S$,*

1. $a\;\, r\, \{1\}\, g\, s \;\Leftrightarrow\; a\, \{r\}\, s$.
2. $a\;\, r\, \{b + b'\}\, g\, s \;\Leftrightarrow\; a\;\, r\, \{b\}\, g\, s \wedge a\;\, r\, \{b'\}\, g\, s$.
3. *Assume power invariants $R = \mathcal{P}(E)$, $G = \mathcal{P}(F)$ for $E, F \subseteq EV$, event $p \notin E$ and let $\rightarrow$ be acyclic. Then $P\;\, R\, \{[p]\}\, G\, S \Leftrightarrow P\, \{R\, ;\, [p]\, ;\, R\}\, S \wedge [p] \mathrel{\mathsf{guar}} G$.*

Finally we give rely/guarantee rules for iteration.

Theorem 8.7. *Assume a rely/guarantee-CKA* (S, I) *and let* * *be finite iteration w.r.t.* $\circ \in \{*, ;\}$. *Then for invariants* $r, g \in I$ *and arbitrary elements* $a, b \in S$,

$$a\ r\,\{b\}\,g\ a\ \Rightarrow\ a\ r\,\{b^+\}\,g\ a\ ,$$
$$a\,\{r\}\,a\ \wedge\ a\ r\,\{b\}\,g\ a\ \Rightarrow\ a\ r\,\{b^*\}\,g\ a\ .$$

We conclude this section with a small example of the use of our rules.

Example 8.8. We consider again the programs $P_u = [au]$ and invariants $G_u = P_u \cup \mathsf{skip}$ $(u \in \{x, y\})$ from Example 8.2. Moreover, we assume an event av with $v \neq x, y$, $ax \not\to av$ and $ay \not\to av$ and set $P_v =_{df} [av]$. We will show

$$P_v\ \mathsf{skip}\,\{P_x * P_y\}\,(G_x * G_y)\,[av, ax, ay]$$

holds. In particular, the parallel execution of the assignments $x := x + 1$ and $y := y + 2$ guarantees that at most x and y are changed. We set $R_x =_{df} G_y$ and $R_y =_{df} G_x$. Then

(a) P_x guar G_x guar R_y , (b) P_y guar G_y guar R_x .

Define the postconditions

$$S_x =_{df} [av, ax] \cup [av, ax, ay] \quad \text{and} \quad S_y =_{df} [av, ay] \cup [av, ax, ay]\ .$$

Then

(c) $S_x \cap S_y = [av, ax, ay]$, (d) $R_x \cap R_y = \mathsf{skip}$.

From the definition of Hoare triples we calculate

$$P_v\,\{R_x\}\,([av] \cup [av, ay]) \qquad ([av] \cup [av, ay])\,\{P_x\}\,S_x \qquad S_x\,\{R_x\}\,S_x\ ,$$

since $[av, ax, ay] * [ay] = \emptyset$. Combining the three clauses by Lemma 5.2.4 we obtain

$$P_v\,\{R_x\,;\,P_x\,;\,R_x\}\,S_x\ .$$

By Theorem 8.6.3 we obtain $P_v\ R_y\,\{P_x\}\,G_x\ S_x$ and, similarly, $P_v\ R_x\,\{P_y\}\,G_y\ S_y$. Now the claim follows from the clauses (a),(b),(c),(d) and Theorem 8.4. □

In a practical application of the theory of Kleene algebras to program correctness, the model of a program trace will be much richer than ours. It will certainly include labels on each event, indicating which atomic command of the program is responsible for execution of the event. It will include labels on each data flow arrow, indicating the value which is 'passed along' the arrow, and the identity of the variable or communication channel which mediated the flow.

9 Related Work

Although our basic model and its algebraic abstraction reflect a non-interleaving view of concurrency, we try to set up a connection with familiar process algebras such as ACP [1], CCS[21], CSP[13], mCRL2 [11] and the π-calculus [30].

It is not easy to relate their operators to those of CKA. The closest analogies seem to be the following ones.

CKA operator	corresponding operator
$+$	non-deterministic choice in CSP
$*$	parallel composition $\mid$ in ACP, π-calculus and CCS
$\|$	interleaving $\|\|$ in CSP
;	sequential composition ; in CSP and $\cdot$ in ACP
▯	choice $+$ in CCS and internal choice $\Box$ in CSP
1	`SKIP` in CSP
0	this is the miracle and cannot be represented in any implementable calculus

However, there are a number of laws which show the inaccuracy of this table. For instance, in CSP we have `SKIP` $\Box\, P \neq P$, whereas CKA satisfies 1 ▯ $P = P$. A similarly different behaviour arises in CCS, ACP and the π-calculus concerning distributivity of composition over choice.

As the discussion after Theorem 7.1 shows, our basic model falls into the class of partial-order models for true concurrency. Of the numerous works in that area we discuss some approaches that have explicit operators for composition related to our $*$ and ;. Whereas we assume that our dependence relation is fixed a priori, in the pomset approach [10,9,26] is is constructed by the composition operators. The operators there are sequential and concurrent composition; there are no choice and iteration, though. Moreover, no laws are given for the operators. In Winskel's event structures [31] there are choice (sum) and concurrent composition, but no sequential composition and iteration. Again, there are no interrelating laws. Another difference to our approach is that the "traces" are required to observe certain closure conditions.

Among the axiomatic approaches to partial order semantics we mention the following ones. Boudol and Castellani [4] present the notion of trioids, which are algebras offering the operations of choice, sequential and concurrent composition. However, there are no interrelating laws and no iteration. Chothia and Kleijn07 [5] use a double semiring with choice, sequential and concurrent composition, but again no interrelating laws and no iteration. The application is to model quality of service, not program semantics.

The approach closest in spirit to ours are Prisacariu's synchronous Kleene algebras (SKA) [27]. The main differences are the following. SKAs are not quantale-based, but rather an enrichment of conventional Kleene algebras. They are restricted to a finite alphabet of actions and hence have a complete and even decidable equational theory. There is only a restricted form of concurrent composition, and the exchange law is equational rather than equational. Iteration is present but not used in an essential way. Nevertheless, Prisacariu's paper is the only of the mentioned ones that explicitly deals with Hoare logic. It does so using the approach of Kleene algebras with tests [19]. This is not feasible in our basic model, since tests are required to be below the element 1, and 0 and 1 are the only such elements. Note, however, that Mace4 [28] quickly shows that this is not a consequence of the CKA axioms but holds only for the particular model.

10 Conclusion and Outlook

The study in this paper has shown that even with the extremely weak assumptions of our trace model many of the important programming laws can be shown, mostly by very concise and simple algebraic calculations. Indeed, the rôle of the axiomatisation was precisely to facilitate these calculations: rather than verifying the laws laboriously in the concrete trace model, we can do so much more easily in the algebraic setting of Concurrent Kleene Algebras. This way many new properties of the trace model have been shown in the present paper. Hence, although currently we know of no other interesting model of CKA than the trace model, the introduction of that structure has already been very useful.

The discussion in the previous section indicates that CKA is not a direct abstraction of the familiar concurrency calculi. Rather, we envisage that the trace model and its abstraction CKA can serve as a basic setting into which many of the existing other calculi can be mapped so that then their essential laws can be proved using the CKA laws. A first experiment along these lines is a trace model of a core subset of the π-calculus in [16]. An elaboration of these ideas will be the subject of further studies.

Acknowledgement. We are grateful for valuable comments by J. Desharnais, H.-H. Dang, R. Glück, W. Guttmann, P. Höfner, P. O'Hearn, H. Yang and by the anonymous referees of CONCUR09.

References

1. Bergstra, J.A., Bethke, I., Ponse, A.: Process algebra with iteration and nesting. The Computer Journal 37(4), 243–258 (1994)
2. Birkhoff, G.: Lattice Theory, 3rd edn. Amer. Math. Soc. (1967)
3. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM 44(2), 201–236 (1997)
4. Boudol, G., Castellani, I.: On the semantics of concurrency: partial orders and transition systems. In: Ehrig, H., Levi, G., Montanari, U. (eds.) CAAP 1987 and TAPSOFT 1987. LNCS, vol. 249, pp. 123–137. Springer, Heidelberg (1987)
5. Chothia, T., Kleijn, J.: Q-Automata: modelling the resource usage of concurrent components. Electr. Notes Theor. Comput. Sci. 175(2), 153–167 (2007)
6. Cohen, E.: Separation and reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000)
7. Conway, J.: Regular Algebra and Finite Machines. Chapman & Hall, Sydney (1971)
8. Desharnais, J., Möller, B., Struth, G.: Kleene Algebra with domain. Trans. Computational Logic 7, 798–833 (2006)
9. Gischer, J.: Partial orders and the axiomatic theory of shuffle. PhD thesis, Stanford University (1984)
10. Grabowski, J.: On partial languages. Fundamenta Informaticae 4(1), 427–498 (1981)
11. Groote, J., Mathijssen, A., van Weerdenburg, M., Usenko, Y.: From μCRL to mCRL2: motivation and outline. In: Proc. Workshop Essays on Algebraic Process Calculi (APC 25). ENTCS, vol. 162, pp. 191–196 (2006)

12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12, 576–585 (1969)
13. Hoare, C.A.R.: Communicating sequential processes. Prentice Hall, Englewood Cliffs (1985)
14. Hoare, C.A.R., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene Algebra. Institut für Informatik, Universität Augsburg, Technical Report 2009-04 (April 2009)
15. Hoare, C.A.R., Möller, B., Struth, G., Wehrman, I.: Foundations of Concurrent Kleene Algebra. In: Berghmmar, R., Jaoua, A., Möller, B. (eds.) Relations and kleene Algebra in Computer Science. Proc. 11th International Conference on Relational Methods in Computer Science (RelMiCS 11) and 6th International Conference on Applications of Kleene Algebra (AKA 6), Doha, Qatar, November 1–5. LNCS, vol. 5827. Springer, Heidelberg (2009) (forthcoming)
16. Hoare, C.A.R., Wehrman, I., O'Hearn, P.: Graphical models of separation logic. In: Proc. Marktoberdorf Summer School (forthcoming, 2008)
17. Jones, C.: Development methods for computer programs including a notion of interference. PhD Thesis, University of Oxford. Programming Research Group, Technical Monograph 25 (1981)
18. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Information and Computation 110, 366–390 (1994)
19. Kozen, D.: Kleene algebra with tests. Trans. Programming Languages and Systems 19, 427–443 (1997)
20. Mac Lane, S.: Categories for the working mathematician, 2nd edn. Springer, Heidelberg (1998)
21. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
22. Misra, J.: Axioms for memory access in asynchronous hardware systems. ACM Trans. Program. Lang. Syst. 8, 142–153 (1986)
23. Morgan, C.: Programming from Specifications. Prentice Hall, Englewood Cliffs (1990)
24. Mulvey, C.: Rendiconti del Circolo Matematico di Palermo 12, 99–104 (1986)
25. O'Hearn, P.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375, 271–307 (2007)
26. Pratt, V.R.: Modelling concurrency with partial orders. Journal of Parallel Programming 15(1) (1986)
27. Prisacariu, C.: Extending Kleene lgebra with synchrony — technicalities. University of Oslo, Department of Informatics, Research Report No. 376 (October 2008)
28. McCune, W.: Prover9 and Mace4, http://www.prover9.org/ (accessed March 1, 2009)
29. Rosenthal, K.: Quantales and their applications. Pitman Research Notes in Math. No. 234. Longman Scientific and Technical (1990)
30. Sangiorgi, D., Walker, D.: The π-calculus — A theory of mobile processes. Cambridge University Press, Cambridge (2001)
31. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)

Concavely-Priced Probabilistic Timed Automata

Marcin Jurdziński[1], Marta Kwiatkowska[2], Gethin Norman[2] and Ashutosh Trivedi[2]

[1] Department of Computer Science,University of Warwick, UK
[2] Computing Laboratory, University of Oxford, UK

Abstract. Concavely-priced probabilistic timed automata, an extension of probabilistic timed automata, are introduced. In this paper we consider expected reachability, discounted, and average price problems for concavely-priced probabilistic timed automata for arbitrary initial states. We prove that these problems are EXPTIME-complete for probabilistic timed automata with two or more clocks and PTIME-complete for automata with one clock. Previous work on expected price problems for probabilistic timed automata was restricted to expected reachability for linearly-priced automata and integer valued initial states. This work uses the boundary region graph introduced by Jurdziński and Trivedi to analyse properties of concavely-priced (non-probabilistic) timed automata.

1 Introduction

Markov decision processes [27] (MDPs) extend finite automata by providing a probability distribution over successor states for each transition. Timed automata [1] extend finite automata by providing a mechanism to constrain the transitions with real-time. Probabilistic timed automata (PTAs) [20,15,3] generalise both timed automata and MDPs by allowing both probabilistic and real-time behaviour.

Priced timed automata are timed automata with (time-dependent) prices attached to locations. Optimisation problems on priced timed automata are fundamental to the *verification* of (quantitative timing) properties of systems modelled as timed automata. In linearly-priced timed automata [2,24] the price information is given by a real-valued function over locations which returns the price to be paid for each time-unit spent in the location. Jurdziński and Trivedi [18] have recently proposed a generalisation to *concave prices*, where the price of remaining in a location is a concave function over time, and demonstrated that, for such prices, a number of optimisation problems including reachability-price, discounted-price, and average-price are PSPACE-complete.

In this paper we present *concavely-priced probabilistic timed automata*, that is, probabilistic timed automata with concave prices. Concave functions appear frequently in many modelling scenarios such as in economics when representing resource utilization, sales or productivity. Examples include:

1. when renting equipment the rental rate decreases as the rental duration increases [11];
2. the expenditure by *vacation travellers* in an airport is typically a concave function of the waiting time [28];
3. the price of perishable products with fixed stock over a finite time period is usually a concave function of time [12], e.g., the price of a low-fare air-ticket as a function of the time remaining before the departure date.

Contribution. We show that finite-horizon expected total price, and infinite-horizon expected reachability, discounted price and average price objectives on concavely-priced PTAs are decidable. We also show that the complexity of solving expected reachability price, expected discounted price and expected average price problems are EXPTIME-complete for concavely-priced PTAs with two or more clocks and PTIME-complete for concavely-priced PTAs with one clock. An important contribution of this paper is the proof techniques which complement the techniques of [18]. We extend the boundary region graph construction for timed automata [18] to PTAs and demonstrate that all the optimisation problems considered can be reduced to similar problems on the boundary region graph. We characterise the values of the optimisation problems by *optimality equations* and prove the correctness of the reduction by analysing the solutions of the optimality equations on the boundary region graph. This allows us to obtain an efficient algorithm matching the EXPTIME lower bound for the problems. Complete proofs can be found in the technical report version of this paper [16].

Related Work. For a review of work on optimisation problems for (non-probabilistic) timed automata we refer the reader to [18]. For priced probabilistic timed automata work has been limited to considering linearly-priced PTAs. Based on the digital clocks approach [13], Kwiatkowska et. al. [19] present a method for solving infinite-horizon expected reachability problems for a subclass of probabilistic timed automata. In [6] an algorithm for calculating the maximal probability of reaching some goal location within a given cost (and time) bound is presented. The algorithm is shown to be partially correct in that if it terminates it terminates with the correct value. Following this, [5] demonstrates the undecidability of this problem. We also mention the approaches for analysing unpriced probabilistic timed automata against temporal logic specifications based on the region graph [20,15] and either forwards [20] or backwards [21] reachability. The complexity of performing such verification is studied in [23,17].

2 Preliminaries

We assume, wherever appropriate, sets $\mathbb{N}$ of non-negative integers, $\mathbb{R}$ of reals and $\mathbb{R}_\oplus$ of non-negative reals. For $n \in \mathbb{N}$, let $[\![n]\!]_\mathbb{N}$ and $[\![n]\!]_\mathbb{R}$ denote the sets $\{0, 1, \ldots, n\}$, and $\{r \in \mathbb{R} \mid 0 {\le} r {\le} n\}$ respectively. A *discrete probability distribution* over a countable set Q is a function $\mu : Q \to [0, 1]$ such that $\sum_{q \in Q} \mu(q){=}1$. For a possible uncountable set Q', we define $\mathcal{D}(Q')$ to be the set of functions $\mu : Q' \to [0, 1]$ such that the set *supp*$(\mu) \stackrel{\text{def}}{=} \{q \in Q \mid \mu(q){>}0\}$ is countable and, over the domain *supp*(μ), μ is a distribution. We say that $\mu \in \mathcal{D}(Q)$ is a *point distribution* if $\mu(q){=}1$ for some $q \in Q$.

A set $\mathcal{D} \subseteq \mathbb{R}^n$ is *convex* if $\theta x{+}(1{-}\theta)y \in \mathcal{D}$ for all $x, y \in \mathcal{D}$ and $\theta \in [0, 1]$. A function $f : \mathbb{R}^n \to \mathbb{R}$ is *concave* (on its domain $\text{dom}(f) \subseteq \mathbb{R}^n$), if $\text{dom}(f) \subseteq \mathbb{R}^n$ is a convex set and $f(\theta{\cdot}x{+}(1{-}\theta){\cdot}y) \ge \theta{\cdot}f(x){+}(1{-}\theta){\cdot}f(y)$ for all $x, y \in \text{dom}(f)$ and $\theta \in [0, 1]$. We require the following well known [8] properties of concave functions.

Lemma 1

1. *If* $f_1, \ldots, f_k : \mathbb{R}^n \to \mathbb{R}$ *are concave and* $w_1, \ldots, w_k \in \mathbb{R}_\oplus$*, then* $\sum_{i=1}^{k} w_i{\cdot}f_i : \mathbb{R}^n \to \mathbb{R}$ *is concave on the domain* $\bigcap_{i=1}^{k} \text{dom}(f_i)$.
2. *If* $f : \mathbb{R}^n \to \mathbb{R}$ *is concave and* $g : \mathbb{R}^m \to \mathbb{R}^n$ *linear, then* $x \mapsto f(g(x))$ *is concave.*

3. *If* $f_1, \dots f_k : \mathbb{R}^n \to \mathbb{R}$ *are concave, then* $x \mapsto \min_{i=1}^k f_i(x)$ *is concave on the domain* $\bigcap_{i=1}^k \mathrm{dom}(f_i)$*.*
4. *If* $f_i : \mathbb{R}^n \to \mathbb{R}$ *is concave for* $i \in \mathbb{N}$*, then* $x \mapsto \lim_{i\to\infty} f_i(x)$ *is concave.*

Lemma 2. *If* $f : (a,b) \to \mathbb{R}$ *is a concave function and* $\overline{f}$ *is the (unique) continuous extension of* f *to the closure of the interval* (a,b)*, then* $\inf_{x\in(a,b)} f(x) = \min\{\overline{f}(a), \overline{f}(b)\}$*.*

A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is *Lipschitz-continuous* on its domain, if there exists a constant $K{\geq}0$, called a Lipschitz constant of f, such that $\|f(x){-}f(y)\|_\infty \leq K\|x{-}y\|_\infty$ for all $x, y \in \mathrm{dom}(f)$; we then also say that f is K-continuous.

3 Markov Decision Processes

In this section we introduce Markov decision processes (MDPs), a form of transition systems which exhibit both probabilistic and nondeterministic behaviour.

Definition 3. *A* priced Markov decision process *is a tuple* $\mathcal{M} = (S, A, p, \pi)$ *where:*

- S *is the set of* states;
- A *is the set of* actions;
- $p : S \times A \to \mathcal{D}(S)$ *is a partial function called the* probabilistic transition function;
- $\pi : S{\times}A \to \mathbb{R}$ *is a bounded and measurable* price function *assigning real-values to state-action pairs.*

We write $A(s)$ for the set of actions available at s, i.e., the set of actions a for which $p(s,a)$ is defined. For technical convenience we assume that $A(s)$ is nonempty for all $s \in S$. We say $\mathcal{M}$ is finite, if the sets S and A are finite.

In the priced MDP $\mathcal{M}$, if the current state is s, then a strategy chooses an action $a \in A(s)$ after which a probabilistic transition is made according to the distribution $p(s,a)$, i.e., state $s' \in S$ is reached with probability $p(s'|s,a) \overset{\text{def}}{=} p(s,a)(s')$. We say that (s,a,s') is a transition of $\mathcal{M}$ if $p(s'|s,a){>}0$ and a *run* of $\mathcal{M}$ is a sequence $\langle s_0, a_1, s_1, \dots\rangle \in S{\times}(A{\times}S)^*$ such that (s_i, a_{i+1}, s_{i+1}) is a transition for all $i{\geq}0$. We write $\mathit{Runs}^{\mathcal{M}}$ ($\mathit{Runs}^{\mathcal{M}}_{\mathit{fin}}$) for the sets of infinite (finite) runs and $\mathit{Runs}^{\mathcal{M}}(s)$ ($\mathit{Runs}^{\mathcal{M}}_{\mathit{fin}}(s)$) for the sets of infinite (finite) runs starting from state s. For a finite run $r{=}\langle s_0, a_1, \dots, s_n\rangle$ we write $\mathit{last}(r){=}s_n$ for the last state of the run. Furthermore, let X_i and Y_i denote the random variables corresponding to ith state and action of a run.

A *strategy* in $\mathcal{M}$ is a function $\sigma : \mathit{Runs}^{\mathcal{M}}_{\mathit{fin}} \to \mathcal{D}(A)$ such that $\mathit{supp}(\sigma(r)) \subseteq A(\mathit{last}(r))$ for all $r \in \mathit{Runs}^{\mathcal{M}}_{\mathit{fin}}$. Let $\mathit{Runs}^{\mathcal{M}}_{\sigma}(s)$ denote the subset of $\mathit{Runs}^{\mathcal{M}}(s)$ which correspond to the strategy σ when starting in state s. Let $\Sigma_{\mathcal{M}}$ be the set of all strategies in $\mathcal{M}$. We say that a strategy σ is *pure* if $\sigma(r)$ is a point distribution for all $r \in \mathit{Runs}^{\mathcal{M}}_{\mathit{fin}}$. We say that a strategy σ is *stationary* if $\mathit{last}(r){=}\mathit{last}(r')$ implies $\sigma(r){=}\sigma(r')$ for all $r, r' \in \mathit{Runs}^{\mathcal{M}}_{\mathit{fin}}$. A strategy σ is *positional* if it is both pure and stationary. To analyse the behaviour of an MDP $\mathcal{M}$ under a strategy σ, for each state s of $\mathcal{M}$, we define a probability space $(\mathit{Runs}^{\mathcal{M}}_{\sigma}(s), \mathcal{F}_{\mathit{Runs}^{\mathcal{M}}_{\sigma}(s)}, \mathrm{Prob}^{\sigma}_{s})$ over the set of infinite runs of σ with s as the initial state. For details on this construction see, for example, [27]. Given a *real-valued random variable* over the set of infinite runs $f : \mathit{Runs}^{\mathcal{M}} \to \mathbb{R}$, using standard techniques from probability theory, we can define the expectation of this variable $\mathbb{E}^{\sigma}_{s}\{f\}$ with respect to σ when starting in s.

Performance Objectives. For a priced MDP $\mathcal{M} = (S, A, p, \pi)$, under any strategy σ and starting from any state s, there is a sequence of random prices $\{\pi(X_{i-1}, Y_i)\}_{i\geq 1}$. Depending on the problem under study there are a number of different performance objectives that can be studied. Below are the objectives most often used.

1. *Expected Reachability Price* (with *target set* F):

$$\text{EReach}_{\mathcal{M}}(F)(s,\sigma) \stackrel{\text{def}}{=} \mathbb{E}_s^\sigma \left\{ \sum_{i=1}^{\min\{i \,|\, X_i \in F\}} \pi(X_{i-1}, Y_i) \right\}.$$

2. *Expected Total Price* (with *horizon* N):

$$\text{ETotal}_{\mathcal{M}}(N)(s,\sigma) \stackrel{\text{def}}{=} \mathbb{E}_s^\sigma \left\{ \sum_{i=1}^{N} \pi(X_{i-1}, Y_i) \right\}.$$

3. *Expected Discounted Price* (with *discount factor* $\lambda \in (0,1)$):

$$\text{EDisct}_{\mathcal{M}}(\lambda)(s,\sigma) \stackrel{\text{def}}{=} \mathbb{E}_s^\sigma \left\{ (1-\lambda) \sum_{i=1}^{\infty} \lambda^{i-1} \pi(X_{i-1}, Y_i) \right\} .$$

4. *Expected Average Price:*

$$\text{EAvg}_{\mathcal{M}}(s,\sigma) \stackrel{\text{def}}{=} \limsup_{n\to\infty} \frac{1}{n} \mathbb{E}_s^\sigma \left\{ \sum_{i=1}^{n} \pi(X_{i-1}, Y_i) \right\}.$$

For an objective $\text{ECost}_{\mathcal{M}}$ and state s we let $\text{ECost}^*_{\mathcal{M}}(s) = \inf_{\sigma\in\Sigma_M} \text{ECost}_{\mathcal{M}}(s,\sigma)$. A strategy σ of $\mathcal{M}$ is *optimal* for $\text{ECost}_{\mathcal{M}}$ if $\text{ECost}_{\mathcal{M}}(s,\sigma) = \text{ECost}^*_{\mathcal{M}}(s)$ for all $s \in S$. Note that an optimal strategy need not exist, and in such cases one can consider, for each $\varepsilon{>}0$, a ε-optimal strategy, that is, a strategy σ such that $\text{ECost}^*_{\mathcal{M}}(s) \geq \text{ECost}_{\mathcal{M}}(s,\sigma) - \varepsilon$ for all $s \in S$. For technical convenience we make the follows assumptions for reachability objectives [9].

Assumption 1. If $s \in F$, then s is *absorbing* and *price free*, i.e. $p(s|s,a){=}1$ and $\pi(s,a){=}0$ for all $a \in A(s)$.

Assumption 2. For all $\sigma \in \Sigma_M$ and $s \in S$ we have $\lim_{i\to\infty} \text{Prob}_s^\sigma(X_i \in F) = 1$.

Optimality Equations. We now review optimality equations for determining the objectives given above.

1. Let $P : S \to \mathbb{R}$ and $F \subseteq S$; we write $P \models \text{Opt}_R^F(\mathcal{M})$, and we say that P is a solution of optimality equations $\text{Opt}_R^F(\mathcal{M})$ if, for all $s \in S$, we have:

$$P(s) = \begin{cases} 0 & \text{if } s \in F \\ \inf_{a\in A(s)} \left\{ \pi(s,a) + \sum_{s'\in S} p(s'|s,a) \cdot P(s') \right\} & \text{otherwise.} \end{cases}$$

2. Let $T_0, \ldots, T_N : S \to \mathbb{R}$; we say that $\langle T_i \rangle_{i=0}^N$ is a solution of optimality equations $\text{Opt}_T^N(\mathcal{M})$ if, for all $s \in S$, we have:

$$T_i(s) = \begin{cases} 0 & \text{if } i = 0 \\ \inf_{a\in A(s)} \left\{ \pi(s,a) + \sum_{s'\in S} p(s'|s,a) \cdot T_{i-1}(s') \right\} & \text{otherwise.} \end{cases}$$

3. Let $D : S \to \mathbb{R}$; we write $D \models \mathrm{Opt}^{\lambda}_{D}(\mathcal{M})$, and we say that D is a solution of optimality equations $\mathrm{Opt}^{\lambda}_{D}(\mathcal{M})$ if, for all $s \in S$, we have:
$$D(s) = \inf_{a \in A(s)} \left\{ (1-\lambda) \cdot \pi(s,a) + \lambda \cdot \textstyle\sum_{s' \in S} p(s'|s,a) \cdot D(s') \right\}.$$
4. Let $G : S \to \mathbb{R}$ and $B : S \to \mathbb{R}$; we write $(G, B) \models \mathrm{Opt}_{A}(\mathcal{M})$, and we say that (G, B) is a solution of optimality equations[1] $\mathrm{Opt}_{A}(\mathcal{M})$, if for all $s \in S$, we have:
$$G(s) = \inf_{a \in A(s)} \left\{ \textstyle\sum_{s' \in S} p(s'|s,a) \cdot G(s') \right\}$$
$$B(s) = \inf_{a \in A(s)} \left\{ \pi(s,a) - G(s) + \textstyle\sum_{s' \in S} p(s'|s,a) \cdot B(s') \right\}$$

The proof of the following proposition is routine and for details see, for example, [10].

Proposition 4. *Let $\mathcal{M}$ be a priced MDP.*

1. *If $P \models \mathrm{Opt}^{F}_{R}(\mathcal{M})$, then $P(s) = \mathrm{EReach}^{*}_{\mathcal{M}}(F)(s)$ for all $s \in S$.*
2. *If $\langle T_i \rangle_{i=0}^{N} \models \mathrm{Opt}^{N}_{R}(\mathcal{M})$, then $T_i(s) = \mathrm{ETotal}^{*}_{\mathcal{M}}(i)(s)$ for all $i \leq N$ and $s \in S$.*
3. *If $D \models \mathrm{Opt}^{\lambda}_{D}(\mathcal{M})$ and D is bounded, then $D(s) = \mathrm{EDisct}^{*}_{\mathcal{M}}(\lambda)(s)$ for all $s \in S$.*
4. *If $(G, B) \models \mathrm{Opt}_{A}(\mathcal{M})$ and G, B are bounded, then $G(s) = \mathrm{EAvg}^{*}_{\mathcal{M}}(s)$ for all $s \in S$.*

Notice that, for each objective, if, for every state $s \in S$, the infimum is attained in the optimality equations, then there exists an optimal positional strategy. An important class of MDPs with this property are finite MDPs, which gives us the following proposition.

Proposition 5. *For every finite MDP, the existence of a solution of the optimality equations for the expected reachability, discounted and average price implies the existence of a positional optimal strategy for the corresponding objective.*

For a finite MDP $\mathcal{M}$ a solution of optimality equations for expected reachability, total, discounted, and average price objectives can be obtained by value iteration or strategy improvement algorithms [27].

Proposition 6. *For every finite MDP, there exist solutions of the optimality equations for expected reachability, total, discounted, and average price objectives.*

Proposition 5 together with Proposition 6 provide a proof of the following well-known result for priced MDPs [27].

Theorem 7. *For a finite priced MDP the reachability, discounted, and average price objectives each have an optimal positional strategy.*

Notice that the total price objective need not have an optimal positional strategy since, unlike the other objectives, it is a finite horizon problem. However, for this reason, its analysis concerns only finitely many strategies.

[1] These optimality equations are slightly different from Howard's optimality equations for expected average price, and correspond to Puterman's [27] modified optimality equations.

4 Concavely-Priced Probabilistic Timed Automata

In this section we introduce concavely-priced probabilistic timed automata and begin by defining clocks, clock valuations, clock regions and zones.

4.1 Clocks, Clock Valuations, Regions and Zones

We fix a constant $k \in \mathbb{N}$ and finite set of *clocks* $\mathcal{C}$. A (k-bounded) *clock valuation* is a function $\nu : \mathcal{C} \to [\![k]\!]_{\mathbb{R}}$ and we write V for the set of clock valuations.

Assumption 3. Although clocks in (probabilistic) timed automata are usually allowed to take arbitrary non-negative values, we have restricted the values of clocks to be bounded by some constant k. More precisely, we have assumed the models we consider are *bounded* probabilistic timed automata. This standard restriction [7] is for technical convenience and comes without significant loss of generality.

If $\nu \in V$ and $t \in \mathbb{R}_{\oplus}$ then we write $\nu{+}t$ for the clock valuation defined by $(\nu{+}t)(c) = \nu(c){+}t$, for all $c \in \mathcal{C}$. For $C \subseteq \mathcal{C}$ and $\nu \in V$, we write $\nu[C{:=}0]$ for the clock valuation where $\nu[C{:=}0](c) = 0$ if $c \in C$, and $\nu[C{:=}0](c) = \nu(c)$ otherwise.

The set of *clock constraints* over $\mathcal{C}$ is the set of conjunctions of *simple constraints*, which are constraints of the form $c \bowtie i$ or $c{-}c' \bowtie i$, where $c, c' \in \mathcal{C}$, $i \in [\![k]\!]_{\mathbb{N}}$, and $\bowtie \in \{<, >, =, \leq, \geq\}$. For every $\nu \in V$, let SCC(ν) be the set of simple constraints which hold in ν. A *clock region* is a maximal set $\zeta \subseteq V$, such that SCC(ν)=SCC(ν') for all $\nu, \nu' \in \zeta$. Every clock region is an equivalence class of the indistinguishability-by-clock-constraints relation, and vice versa. Note that ν and ν' are in the same clock region if and only if the integer parts of the clocks and the partial orders of the clocks, determined by their fractional parts, are the same in ν and ν'. We write $[\nu]$ for the clock region of ν and, if $\zeta{=}[\nu]$, write $\zeta[C{:=}0]$ for the clock region $[\nu[C{:=}0]]$.

A *clock zone* is a convex set of clock valuations, which is a union of a set of clock regions. We write $\mathcal{Z}$ for the set of clock zones. For any clock zone W and clock valuation ν, we use the notation $\nu \lhd W$ to denote that $[\nu] \in W$. A set of clock valuations is a clock zone if and only if it is definable by a clock constraint. For $W \subseteq V$, we write $\overline{W}$ for the smallest closed set in V containing W. Observe that, for every clock zone W, the set $\overline{W}$ is also a clock zone.

4.2 Probabilistic Timed Automata

We are now in a position to introduce probabilistic timed automata.

Definition 8. *A* probabilistic timed automaton $\mathsf{T} = (L, \mathcal{C}, inv, Act, E, \delta)$ *consists of:*

- *a finite set of locations L;*
- *a finite set of clocks $\mathcal{C}$;*
- *an* invariant condition $inv : L \to \mathcal{Z}$*;*
- *a finite set of* actions Act*;*
- *an* action enabledness function $E : L{\times}Act \to \mathcal{Z}$*;*
- *a* transition probability function $\delta : (L{\times}Act) \to \mathcal{D}(2^{\mathcal{C}}{\times}L)$.

When we consider a probabilistic timed automaton as an input of an algorithm, its size should be understood as the sum of the sizes of encodings of L, $\mathcal{C}$, inv, Act, E, and δ. A *configuration* of a probabilistic timed automaton T is a pair (ℓ, ν), where $\ell \in L$ is a location and $\nu \in V$ is a clock valuation over $\mathcal{C}$ such that $\nu \triangleleft inv(\ell)$. For any $t \in \mathbb{R}$, we let $(\ell, \nu)+t$ equal the configuration $(\ell, \nu+t)$. Informally, the behaviour of a probabilistic timed automaton is as follows. In configuration (ℓ, ν) time passes before an available action is triggered, after which a discrete probabilistic transition occurs. Time passage is available only if the invariant condition $inv(\ell)$ is satisfied while time elapses, and the action a can be chosen after time t if the action is enabled in the location ℓ, i.e., if $\nu+t \triangleleft E(\ell, a)$. Both the amount of time and the action chosen are nondeterministic. If the action a is chosen, then the probability of moving to the location ℓ' and resetting all of the clocks in C to 0 is given by $\delta[\ell, a](C, \ell')$.

Formally, the semantics of a probabilistic timed automaton is given by an MDP which has both an infinite number of states and an infinite number of transitions.

Definition 9. *Let* $\mathsf{T} = (L, \mathcal{C}, inv, Act, E, \delta)$ *be a probabilistic timed automaton. The semantics of* T *is the MDP* $[\![\mathsf{T}]\!] = (S_\mathsf{T}, A_\mathsf{T}, p_\mathsf{T})$ *where*

- $S_\mathsf{T} \subseteq L \times V$ *such that* $(\ell, \nu) \in S_\mathsf{T}$ *if and only if* $\nu \triangleleft inv(\ell)$*;*
- $A_\mathsf{T} = \mathbb{R}_\oplus \times Act$*;*
- *for* $(\ell, \nu) \in S_\mathsf{T}$ *and* $(t, a) \in A_\mathsf{T}$*, we have* $p_\mathsf{T}((\ell, \nu), (t, a)) = \mu$ *if and only if*
  - $\nu+t' \triangleleft inv(\ell)$ *for all* $t' \in [0, t]$*;*
  - $\nu+t \triangleleft E(\ell, a)$*;*
  - $\mu(\ell', \nu') = \sum_{C \subseteq \mathcal{C} \wedge (\nu+t)[C:=0]=\nu'} \delta[\ell, a](C, \ell')$ *for all* $(\ell', \nu') \in S$.

We assume the following—standard and easy to syntactically verify—restriction on PTAs which ensures *time divergent* behaviour.

Assumption 4. We restrict attention to *structurally non-Zeno* probabilistic timed automata [29,17]. A PTA is structurally non-Zeno if, for any run $\langle s_0, (t_1, a_1), \ldots, s_n \rangle$, such that $s_0 = (\ell_0, \nu_0)$, $s_n = (\ell_n, \nu_n)$ and $\ell_0 = \ell_n$ (i.e., the run forms a cycle in the finite graph of the locations and transitions of the automaton) we have $\sum_{i=1}^n t_i \geq 1$.

4.3 Priced Probabilistic Timed Automata

We now introduce priced probabilistic timed automata which extend probabilistic timed automata with price functions over state and time-action pairs.

Definition 10. *A* priced probabilistic timed automaton $\mathcal{T} = (\mathsf{T}, \pi)$ *consists of a probabilistic timed automaton* T *and a price function* $\pi : (L \times V) \times (\mathbb{R}_\oplus \times Act) \to \mathbb{R}$.

The semantics of a priced PTA $\mathcal{T}$ is the priced MDP $[\![\mathcal{T}]\!] = ([\![\mathsf{T}]\!], \pi)$ where $\pi(s, (t, a))$ is the price of taking the action (t, a) from state s in $[\![\mathsf{T}]\!]$. In a *linearly-priced* PTA [19], the price function is represented as a function $r : L \cup Act \to \mathbb{R}$, which gives a *price rate* to every location ℓ, and a price to every action a; the price of taking the timed move (a, t) from state (ℓ, ν) is then defined by $\pi((\ell, \nu), (t, a)) = r(\ell) \cdot t + r(a)$.

In this paper we restrict attention to *concave* price functions requiring that for any location $\ell \in L$ and action $a \in Act$ the function $\pi((\ell, \cdot), (\cdot, a)) : V \times \mathbb{R}_\oplus \to \mathbb{R}$ is concave. However, the results in this paper for PTAs with more than 1 clock, also hold for the more general *region-wise concave* price functions of [18]. Notice that every linearly-priced PTA is also concavely-priced.

Considering the optimisation problems introduced for priced MDPs in Section 3, the following is the main result of the paper.

Theorem 11. *The minimisation problems for reachability, total, discounted, and average cost functions for concavely-priced PTAs are decidable.*

In the next section we introduce the boundary region graph, an abstraction whose size is exponential in the size of PTA. In Section 6 we show that to solve the above mentioned optimisation problems on concavely-priced PTAs, it is sufficient to solve them on the corresponding boundary region graph.

5 Boundary Region Graph Construction

Before introducing the boundary region graph we review the standard region graph construction for timed automata [1] extended in [20] to probabilistic timed automata.

5.1 The Region Graph

A *region* is a pair (ℓ, ζ), where ℓ is a location and ζ is a clock region such that $\zeta \subseteq inv(\ell)$. For any $s{=}(\ell, \nu)$, we write $[s]$ for the region $(\ell, [\nu])$ and $\mathcal{R}$ for the set of regions. A set $Z \subseteq L{\times}V$ is a *zone* if, for every $\ell \in L$, there is a clock zone W_ℓ (possibly empty), such that $Z = \{(\ell, \nu) \mid \ell \in L \wedge \nu \lhd W_\ell\}$. For a region $R{=}(\ell, \zeta) \in \mathcal{R}$, we write $\overline{R}$ for the zone $\{(\ell, \nu) \mid \nu \in \overline{\zeta}\}$, recall $\overline{\zeta}$ is the smallest closed set in V containing ζ.

For $R, R' \in \mathcal{R}$, we say that R' is in the future of R, or that R is in the past of R', if there is $s \in R$, $s' \in R'$ and $t \in \mathbb{R}_\oplus$ such that $s' = s{+}t$; we then write $R \rightarrow_* R'$. We say that R' is the *time successor* of R if $R \rightarrow_* R'$, $R{\neq}R'$, and $R \rightarrow_* R'' \rightarrow_* R'$ implies $R''{=}R$ or $R''{=}R'$ and write $R \rightarrow_{+1} R'$ and $R' \leftarrow_{+1} R$. Similarly we say that R' is the n^{th} successor of R and write $R \rightarrow_{+n} R'$, if there is a sequence of regions $\langle R_0, R_1, \ldots, R_n\rangle$ such that $R_0{=}R$, $R_n{=}R'$ and $R_i \rightarrow_{+1} R_{i+1}$ for every $0{\leq}i{<}n$.

The region graph is an MDP with both a finite number of states and transitions.

Definition 12. *Let* $\mathsf{T} = (L, \mathcal{C}, inv, Act, E, \delta)$ *be a probabilistic timed automaton. The* region graph *of* T *is the MDP* $\mathsf{T}_{\text{RG}} = (S_{\text{RG}}, A_{\text{RG}}, p_{\text{RG}})$ *where:*

- $S_{\text{RG}} = \mathcal{R}$;
- $A_{\text{RG}} \subseteq \mathbb{N}{\times}Act$ *such that if* $(n, a) \in A_{\text{RG}}$ *then* $n \leq (2{\cdot}|\mathcal{C}|)^k$;
- *for* $(\ell, \zeta) \in S_{\text{RG}}$ *and* $(n, a) \in A_{\text{RG}}$ *we have* $p_{\text{RG}}((\ell, \zeta), (n, a)) = \mu$ *if and only if*
  - $(\ell, \zeta) \rightarrow_{+n} (\ell, \zeta_n)$;
  - $\zeta_n \lhd E(\ell, a)$;
  - $\mu(\ell', \zeta') = \sum_{C \subseteq \mathcal{C} \wedge \zeta_n[C:=0]=\zeta'} \delta[\ell, a](C, l')$ *for all* $(\ell', \zeta') \in S_{\text{RG}}$.

Since the region graph abstracts away the precise timing information, it can not be used to solve expected reachability, total, discounted, and average price problems for the original probabilistic timed automaton. In the next section, we define a new abstraction of probabilistic timed automata, called the boundary region graph, which retains sufficient timing information to solve these performance objectives.

5.2 The Boundary Region Graph

We say that a region $R \in \mathcal{R}$ is *thin* if $[s] \neq [s{+}\varepsilon]$ for every $s \in R$ and $\varepsilon{>}0$; other regions are called *thick*. We write $\mathcal{R}_{\text{Thin}}$ and $\mathcal{R}_{\text{Thick}}$ for the sets of thin and thick regions, respectively. Note that if $R \in \mathcal{R}_{\text{Thick}}$ then, for every $s \in R$, there is an $\varepsilon > 0$, such that $[s] = [s{+}\varepsilon]$. Observe that the time successor of a thin region is thick, and vice versa.

We say $(\ell, \nu) \in L{\times}V$ is in the *closure of the region* (ℓ, ζ), and we write $(\ell, \nu) \in \overline{(\ell, \zeta)}$, if $\nu \in \overline{\zeta}$. For any $\nu \in V$, $b \in [\![k]\!]_{\mathbb{N}}$ and $c \in \mathcal{C}$ such that $\nu(c){\leq}b$, we let $time(\nu, (b, c)) \overset{\text{def}}{=} b{-}\nu(c)$. Intuitively, $time(\nu, (b, c))$ returns the amount of time that must elapse in ν before the clock c reaches the integer value b. Note that, for any $(\ell, \nu) \in L{\times}V$ and $a \in Act$, if $t = time(\nu, (b, c))$ is defined, then $(\ell, [\nu{+}t]) \in \mathcal{R}_{\text{Thin}}$ and $supp(p_{\mathsf{T}}(\cdot \mid (\ell, \nu), (t, a))) \subseteq \mathcal{R}_{\text{Thin}}$. Observe that, for every $R' \in \mathcal{R}_{\text{Thin}}$, there is a number $b \in [\![k]\!]_{\mathbb{N}}$ and a clock $c \in \mathcal{C}$, such that, for every $R \in \mathcal{R}$ in the past of R', we have that $s \in R$ implies $(s{+}(b{-}s(c)) \in R'$; and we write $R \rightarrow_{b,c} R'$.

The motivation for the boundary region graph is the following. Let $a \in A$, $s = (\ell, \nu)$ and $R = (\ell, \zeta) \rightarrow_{*} R' = (\ell, \zeta')$ such that $s \in R$ and $R' \lhd E(\ell, a)$.

- If $R' \in \mathcal{R}_{\text{Thick}}$, then there are infinitely many $t \in \mathbb{R}_{\oplus}$ such that $s{+}t \in R'$. One of the main results that we establish is that in the state s, amongst all such t's, for one of the boundaries of ζ', the closer $\nu{+}t$ is to this boundary, the 'better' the timed action (t, a) becomes for each performance objective. However, since R' is a thick region, the set $\{t \in \mathbb{R}_{\oplus} \mid s{+}t \in R'\}$ is an open interval, and hence does not contain its boundary values. Observe that the infimum equals $b_{-}{-}\nu(c_{-})$ where $R \rightarrow_{b_{-},c_{-}} R_{-} \rightarrow_{+1} R'$ and the supremum equals $b_{+}{-}\nu(c_{+})$ where $R \rightarrow_{b_{+},c_{+}} R_{+} \leftarrow_{+1} R'$. In the boundary region graph we include these 'best' timed action through the actions $((b_{-}, c_{-}, a), R')$ and $((b_{+}, c_{+}, a), R')$.
- If $R' \in \mathcal{R}_{\text{Thin}}$, then there exists a unique $t \in \mathbb{R}_{\oplus}$ such that $(\ell, \nu{+}t) \in R'$. Moreover since R' is a thin region there exists a clock $c \in C$ and a number $b \in \mathbb{N}$ such that $R \rightarrow_{b,c} R'$ and $t = b{-}\nu(c)$. In the boundary region graph we summarize this 'best' timed action from region R via region R' through the action $((b, c, a), R')$.

With this intuition in mind, let us present the definition of a boundary region graph.

Definition 13. *Let* $\mathsf{T} = (L, \mathcal{C}, inv, A, E, \delta)$ *be a probabilistic timed automaton. The* boundary region graph *of* T *is defined as the MDP* $\mathsf{T}_{\text{BRG}} = (S_{\text{BRG}}, A_{\text{BRG}}, p_{\text{BRG}})$ *such that:*

- $S_{\text{BRG}} = \{((\ell, \nu), (\ell, \zeta)) \mid (\ell, \zeta) \in \mathcal{R} \wedge \nu \in \overline{\zeta}\}$*;*
- $A_{\text{BRG}} \subseteq ([\![k]\!]_{\mathbb{N}}{\times}\mathcal{C}{\times}Act){\times}\mathcal{R}$*;*
- *for any state* $((\ell, \nu), (\ell, \zeta)) \in S_{\text{BRG}}$ *and action* $((b, c, a), (\ell, \zeta_a)) \in A_{\text{BRG}}$ *we have* $p_{\text{BRG}}((\ell, \nu), (\ell, \zeta), ((b, c, a), (\ell, \zeta_a))) = \mu$ *if and only if*

$$\mu((\ell', \nu'), (\ell', \zeta')) = \textstyle\sum_{C \subseteq \mathcal{C} \wedge \nu_a[C:=0]=\nu' \wedge \zeta_a[C:=0]=\zeta'} \delta[\ell, a](C, l')$$

for all $((\ell', \nu'), (\ell', \zeta')) \in S_{\text{BRG}}$ *where* $\nu_a = \nu{+}time(\nu, (b, c))$ *and one of the following conditions holds:*
  - $(\ell, \zeta) \rightarrow_{b,c} (\ell, \zeta_a)$ *and* $\zeta_a \lhd E(\ell, a)$
  - $(\ell, \zeta) \rightarrow_{b,c} (\ell, \zeta_{-}) \rightarrow_{+1} (\ell, \zeta_a)$ *for some* (ℓ, ζ_{-}) *and* $\zeta_a \lhd E(\ell, a)$
  - $(\ell, \zeta) \rightarrow_{b,c} (\ell, \zeta_{+}) \leftarrow_{+1} (\ell, \zeta_a)$ *for some* (ℓ, ζ_{+}) *and* $\zeta_a \lhd E(\ell, a)$.

Although the boundary region graph is infinite, for a fixed initial state we can restrict attention to a finite state subgraph, thanks to the following observation [18].

Proposition 14. *For every state $s \in S_{\text{BRG}}$ of a boundary region graph T_{BRG}, the reachable sub-graph $\mathsf{T}^{s}_{\text{BRG}}$ is a finite MDP.*

Proposition 15. *If $s = ((\ell,\nu),(\ell,\zeta)) \in S_{\text{BRG}}$ is such that ν is an integer valuation, then the MDP $\mathsf{T}^{s}_{\text{BRG}}$ is equivalent to the digital clock semantics [19] of T and extends the corner point abstraction of [7] to the probabilistic setting.*

Definition 16. *Let $\mathcal{T} = (\mathsf{T},\pi)$ be a priced probabilistic timed automaton. The* priced boundary region graph *of $\mathcal{T}$ equals the priced MDP $\mathcal{T}_{\text{BRG}} = (\mathsf{T}_{\text{BRG}},\pi_{\text{BRG}})$ where for any state $((\ell,v),(\ell,\zeta)) \in S_{\text{BRG}}$ and action $((b,c,a),(\ell,\zeta')) \in A_{\text{BRG}}$ available in the state:*

$$\pi_{\text{BRG}}\big(((\ell,v),(\ell,\zeta)),((b,c,a),(\ell,\zeta'))\big) = \pi\big((\ell,\nu),(\mathit{time}(\nu,(b,c)),a)\big)\,.$$

6 Correctness of the Reduction to Boundary Region Automata

For the remainder of this section we fix a concavely-priced PTA $\mathcal{T}$. Proposition 14 together with Proposition 6 yield the following important result.

Proposition 17. *For the priced MDP $\mathcal{T}_{\text{BRG}}$ there exist solutions of the optimality equations for expected reachability, total, discounted, and average price objectives.*

We say that a function $f : S_{\text{BRG}} \to \mathbb{R}$ is *regionally concave* if for all $(\ell,\zeta) \in \mathcal{R}$ the function $f(\cdot,(\ell,\zeta)) : \{(\ell,\nu) \,|\, \nu \in \overline{\zeta}\} \to \mathbb{R}$ is concave.

Lemma 18. *Assume that $P \models \mathrm{Opt}^{F}_{R}(\mathcal{T}_{\text{BRG}})$, $\langle T_i \rangle_{i=1}^{N} \models \mathrm{Opt}^{N}_{T}(\mathcal{T}_{\text{BRG}})$, and $D \models \mathrm{Opt}^{\lambda}_{D}(\mathcal{T}_{\text{BRG}})$ We have that P, T_N, and D are regionally concave.*

Proof. (Sketch.) Using an elementary, but notationally involved, inductive proof we can show that T_N is regionally concave. The proof uses closure properties of concave functions (see Lemma 1), along with the fact that price functions π are concave. The concavity of P and D follows from the observation that they can be characterised as the limit (concave due to Lemma 1) of certain optimal expected total price objectives. □

For any function $f : S_{\text{BRG}} \to \mathbb{R}$, we define $\widetilde{f} : S_{\mathsf{T}} \to \mathbb{R}$ by $\widetilde{f}(\ell,\nu) = f((\ell,\nu),(\ell,[\nu]))$.

Lemma 19. *If $P \models \mathrm{Opt}^{F}_{R}(\mathcal{T}_{\text{BRG}})$, then $\widetilde{P} \models \mathrm{Opt}^{F}_{R}([\![\mathcal{T}]\!])$.*

Proof. Assuming $P \models \mathrm{Opt}^{F}_{R}(\mathcal{T}_{\text{BRG}})$, to prove this proposition it is sufficient to show that for any $s{=}(\ell,\nu) \in S_{\mathsf{T}}$ we have:

$$\widetilde{P}(s) = \inf_{(t,a)\in A(s)} \{\pi(s,(t,a)) + \sum_{(C,\ell')\in 2^{\mathcal{C}}\times L} \delta[\ell,a](C,\ell')\cdot\widetilde{P}(\ell',(\nu{+}t)[C{:=}0])\}. \quad (1)$$

We therefore fix a state $s{=}(\ell,\nu) \in S_{\mathsf{T}}$ for the remainder of the proof. For any $a \in \mathit{Act}$, let $\mathcal{R}^{a}_{\text{Thin}}$ and $\mathcal{R}^{a}_{\text{Thick}}$ denote the set of thin and think regions respectively that are successors of $[\nu]$ and are subsets of $E(\ell,a)$. Considering the RHS of (1) we have:

$$\text{RHS of (1)} = \min_{a\in \mathit{Act}} \{T_{\text{Thin}}(s,a), T_{\text{Thick}}(s,a)\}, \quad (2)$$

where $T_{\text{Thin}}(s,a)$ ($T_{\text{Thick}}(s,a)$) is the infimum of the RHS of (1) over all actions (t,a) such that $[\nu{+}t] \in \mathcal{R}^a_{\text{Thin}}$ ($[\nu{+}t] \in \mathcal{R}^a_{\text{Thick}}$). For the first term we have:

$$T_{\text{Thin}}(s,a) = \min_{(\ell,\zeta)\in\mathcal{R}^a_{\text{Thin}}} \inf_{\substack{t\in\mathbb{R}\wedge\\ \nu+t\in\zeta}} \left\{ \pi(s,(t,a)) + \sum_{(C,\ell')\in 2^{\mathcal{C}}\times L} \delta[\ell,a](C,\ell')\cdot\widetilde{P}(\ell',\nu^t_C) \right\}$$

$$= \min_{(\ell,\zeta)\in\mathcal{R}^a_{\text{Thin}}} \inf_{\substack{t\in\mathbb{R}\wedge\\ \nu+t\in\zeta}} \left\{ \pi(s,(t,a)) + \sum_{(C,\ell')\in 2^{\mathcal{C}}\times L} \delta[\ell,a](C,\ell')\cdot P((\ell',\nu^t_C),(\ell',\zeta_C)) \right\}$$

$$= \min_{(\ell,\zeta)\in\mathcal{R}^a_{\text{Thin}}} \left\{ \pi(s,(t^{(\ell,\zeta)},a)) + \sum_{(C,\ell')\in 2^{\mathcal{C}}\times L} \delta[\ell,a](C,\ell')\cdot P((\ell',\nu^{t^{(\ell,\zeta)}}_C),(\ell',\zeta^C)) \right\}$$

where ν^t_C denote the clock valuation $(\nu{+}t)[C{:=}0]$, $t^{(\ell,\zeta)}$ the time to reach the region R from s and ζ^C the region $\zeta[C{:=}0]$. Considering the second term of (2) we have

$$T_{\text{Thick}}(s,a) = \min_{(\ell,\zeta)\in\mathcal{R}^a_{\text{Thick}}} \inf_{\substack{t\in\mathbb{R}\wedge\\ \nu+t\in\zeta}} \left\{ \pi(s,(t,a)) + \sum_{(C,\ell')\in 2^{\mathcal{C}}\times L} \delta[\ell,a](C,\ell')\cdot\widetilde{P}(\ell',\nu^t_C) \right\}$$

$$= \min_{(\ell,\zeta)\in\mathcal{R}^a_{\text{Thick}}} \inf_{\substack{t\in\mathbb{R}\wedge\\ \nu+t\in\zeta}} \left\{ \pi(s,(t,a)) + \sum_{(C,\ell')\in 2^{\mathcal{C}}\times L} \delta[\ell,a](C,\ell')\cdot P((\ell',\nu^t_C),(\ell',\zeta^C)) \right\}$$

$$= \min_{(\ell,\zeta)\in\mathcal{R}^a_{\text{Thick}}} \inf_{\substack{t^s_{R_-}<t<t^s_{R_+}\\ R\leftarrow_{+1}R_-\\ R\rightarrow_{+1}R_+}} \left\{ \pi(s,(t,a)) + \sum_{(C,\ell')\in 2^{\mathcal{C}}\times L} \delta[\ell,a](C,\ell')\cdot P((\ell',\nu^t_C),(\ell',\zeta^C)) \right\}$$

From Lemma 18 we have that $P((\ell',\cdot),(\ell',\zeta^C))$ is concave and, from Lemma 1, since ν^t_C is an affine mapping and $\delta[\ell,a](C,\ell'){\geq}0$ for all $2^{\mathcal{C}}\times L$, the weighted sum over (C,ℓ') of the functions $P((\ell',\nu^t_C),(\ell',\zeta^C))$ is concave on the domain $\{t \,|\, \nu{+}t \in \zeta\}$. From the concavity assumption of price functions, $\pi(s,(\cdot,a))$ is concave over the domain $\{t \,|\, \nu{+}t \in \zeta\}$, and therefore, again using Lemma 1, we have that the function:

$$\pi(s,(t,a)) + \sum_{(C,\ell')\in 2^{\mathcal{C}}\times L} \delta[\ell,a](C,\ell')\cdot P((\ell',\nu^t_C),(\ell',\zeta^C))$$

is concave over $\{t \,|\, \nu{+}t \in \zeta\}$. Therefore using Lemma 2 we have $T_{\text{Thick}}(s,a)$ equals

$$\min_{(\ell,\zeta)\in\mathcal{R}^a_{\text{Thick}}} \min_{\substack{t=t^s_{R_-},t^s_{R_+}\\ (\ell,\zeta)\longleftarrow_{+1}R_-\\ (\ell,\zeta)\longrightarrow_{+1}R_+}} \left\{ \overline{\pi}(s,(t,a)) + \sum_{(C,\ell')\in 2^{\mathcal{C}}\times L} \delta[\ell,a](C,\ell')\cdot P((\ell',\nu^t_C),(\ell',\zeta^C)) \right\}$$

Substituting the values of $T_{\text{Thin}}(s,a)$ and $T_{\text{Thick}}(s,a)$ into (2) and observing that for any thin region $(\ell,\zeta) \in \mathcal{R}^a_{\text{Thin}}$ there exist $b \in \mathbb{Z}$ and $c \in C$ such that $\nu{+}(b{-}\nu(c)) \in \zeta$, it follows from Definition 13 that RHS of (1) equals:

$$\min_{(\alpha,R)\in A_{\text{BRG}}(s,[s])} \left\{ \pi_{\text{BRG}}((s,[s]),(\alpha,R)) + \sum_{(s',R')\in S_{\text{BRG}}} p_{\text{BRG}}((s',R')|(s,[s]),(\alpha,R))\cdot P(s',R') \right\}$$

which by definition equals $\widetilde{P}(s)$ as required. □

Lemma 20. *If* $\langle T_i \rangle_{i=1}^{N} \models \mathrm{Opt}_T^N(\mathcal{T}_{\mathrm{BRG}})$, *then* $\langle \widetilde{T}_i \rangle_{i=0}^{N} \models \mathrm{Opt}_T^N([\![\mathcal{T}]\!])$.

Lemma 21. *If* $D \models \mathrm{Opt}_D^{\lambda}(\mathcal{T}_{\mathrm{BRG}})$, *then* $\widetilde{D} \models \mathrm{Opt}_D^{\lambda}([\![\mathcal{T}]\!])$.

Since it is not known to us whether there exists a solution $(G, B) \models \mathrm{Opt}_A(\mathcal{T}_{\mathrm{BRG}})$ such that both G and B are regionally concave, we can not show that $(G, B) \models \mathrm{Opt}_A(\mathcal{T}_{\mathrm{BRG}})$, implies $(G, B) \models \mathrm{Opt}_A([\![\mathcal{T}]\!])$. Instead, we use the following result to reduce the average price problem on PTA to that over the corresponding boundary region graph.

Lemma 22. *If* $(G, B) \models \mathrm{Opt}_A(\mathcal{T}_{\mathrm{BRG}})$, *then* $\widetilde{G} = \mathrm{EAvg}^*_{[\![\mathcal{T}]\!]}$.

The proof of this result follows from Lemma 24 and Corollary 26 below.

Lemma 23. *For an arbitrary priced MDP* $\mathcal{M} = (S, A, p, \pi)$ *and state* $s \in S$, *the following inequality holds:*

$$\inf_{\sigma\in\Sigma_M} \limsup_{n\to\infty} \frac{1}{n}\cdot\mathbb{E}_s^{\sigma}\left\{\textstyle\sum_{i=1}^{n}\pi(X_{i-1},Y_i)\right\} \geq \limsup_{n\to\infty} \inf_{\sigma\in\Sigma_M} \frac{1}{n}\cdot\mathbb{E}_s^{\sigma}\left\{\textstyle\sum_{i=1}^{n}\pi(X_{i-1},Y_i)\right\}$$

Lemma 24. *For every state* $s \in S_{\mathsf{T}}$ *we have* $\mathrm{EAvg}^*_{[\![\mathcal{T}]\!]}(s) \geq \mathrm{EAvg}^*_{\mathcal{T}_{\mathrm{BRG}}}(s, [s])$.

Proof. Consider any $s \in S_{\mathsf{T}}$, using Lemma 23 we have:

$$\begin{aligned}
\mathrm{EAvg}^*_{[\![\mathcal{T}]\!]}(s) \;&\geq\; \limsup_{n\to\infty} \inf_{\sigma\in\Sigma_{[\![\mathcal{T}]\!]}} \frac{1}{n}\cdot\mathbb{E}_s^{\sigma}\{\textstyle\sum_{i=1}^{n}\pi(X_{i-1},Y_i)\} && \\
&= \limsup_{n\to\infty} \frac{1}{n}\cdot\mathrm{ETotal}^*_{[\![\mathcal{T}]\!]}(n)(s) && \text{by definition of } \mathrm{ETotal}^*_{[\![\mathcal{T}]\!]}(n) \\
&\geq \limsup_{n\to\infty} \frac{1}{n}\cdot\mathrm{ETotal}^*_{\mathcal{T}_{\mathrm{BRG}}}(n)((s,[s])) && \text{by Lemma 20 and Proposition 4} \\
&= \limsup_{n\to\infty} \frac{1}{n}\cdot \inf_{\sigma\in\Sigma_{\mathcal{T}_{\mathrm{BRG}}}} \mathbb{E}^{\sigma}_{(s,[s])}\{\textstyle\sum_{i=1}^{n}\pi(X_{i-1},Y_i)\} && \text{by definition} \\
&= \inf_{\sigma\in\Sigma_{\mathcal{T}_{\mathrm{BRG}}}} \limsup_{n\to\infty} \frac{1}{n}\cdot\mathbb{E}^{\sigma}_{(s,[s])}\{\textstyle\sum_{i=1}^{n}\pi(X_{i-1},Y_i)\} && \text{since } [\mathcal{T}_{\mathrm{BRG}},(s,[s])] \text{ is finite} \\
&= \mathrm{EAvg}^*_{\mathcal{T}_{\mathrm{BRG}}}(s,[s]) && \text{as required} \quad \square
\end{aligned}$$

Using the Lipschitz-continuity of price functions and a slight variant of Lemma 3 of [7] we show that the following proposition and corollary hold.

Proposition 25. *For every* $\varepsilon{>}0$, $\sigma \in \Sigma_{\mathcal{T}_{\mathrm{BRG}}}$ *and* $s \in S_{\mathsf{T}}$, *there exists* $\sigma_\varepsilon \in \Sigma_{[\![\mathcal{T}]\!]}$ *such that* $|\,\mathrm{ETotal}_{\mathcal{T}_{\mathrm{BRG}}}(N)((s,[s]),\sigma) - \mathrm{ETotal}_{[\![\mathcal{T}]\!]}(N)(s,\sigma_\varepsilon)\,| \leq N{\cdot}\varepsilon$ *for all* $N \in \mathbb{N}$.

Corollary 26. *For every* $\varepsilon{>}0$ *and* $s \in S_{\mathsf{T}}$ *we have* $\mathrm{EAvg}^*_{[\![\mathcal{T}]\!]}(s) \leq \mathrm{EAvg}^*_{\mathcal{T}_{\mathrm{BRG}}}(s,[s]){+}\varepsilon$.

7 Complexity

To show EXPTIME-hardness we present a reduction to the EXPTIME-complete problem of solving countdown games [17]. The lemma concerns only expected reachability price problem as similar reductions follow for the other problems.

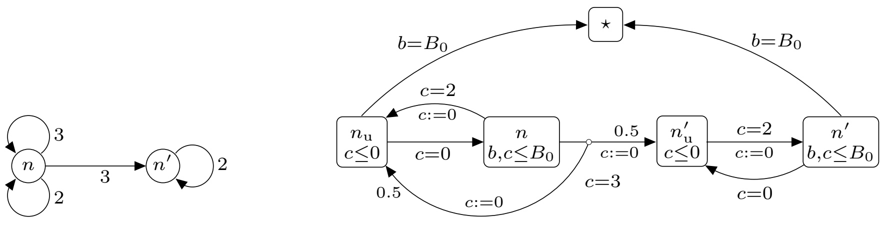


Fig. 1. Countdown Game and the corresponding probabilistic timed automata

Lemma 27. *The expected reachability problem is EXPTIME-hard for concavely-priced PTA with two or more clocks.*

Proof. Let $\mathcal{G} = (N, M, \pi_{\mathcal{G}}, n_0, B_0)$ be a countdown game. N is a finite set of nodes; $M \subseteq N{\times}N$ is a set of moves; $\pi_{\mathcal{G}} : M \to \mathbb{N}_+$ assigns a positive integer to every move; $(n_0, B_0) \in N{\times}\mathbb{N}_+$ is the initial configuration. From $(n, B) \in N{\times}\mathbb{N}_+$, a move consists of player 1 choosing $k \in \mathbb{N}_+$, such that $k{\le}B$ and $\pi_{\mathcal{G}}(n, n'){=}k$ for some $(n, n') \in M$, then player 2 choosing $(n, n'') \in M$ such that $\pi_{\mathcal{G}}(n, n''){=}k$; the new configuration is $(n'', B{-}k)$. Player 1 wins if a configuration of the form $(n, 0)$ is reached, and loses when a configuration (n, B) is reached such that $\pi_{\mathcal{G}}(n, n'){>}B$ for all $(n, n') \in M$.

Given a countdown game $\mathcal{G}$ we define the PTA $\mathsf{T}_{\mathcal{G}} = (L, \mathcal{C}, \mathit{inv}, \mathit{Act}, E, \delta)$ where $L = \{\star\}{\cup}N{\cup}N_{\mathrm{u}}$ where $N_{\mathrm{u}} = \{n_{\mathrm{u}} \mid n \in N\}$; $\mathcal{C} = \{b, c\}$; $\mathit{inv}(n) = \{\nu \mid 0{\le}\nu(b){\le}B_0 \wedge 0{\le}\nu(c){\le}B_0\}$ and $\mathit{inv}(n_{\mathrm{u}}) = \{\nu \mid \nu(c){=}0\}$ for any $n \in N$; $\mathit{Act} = \{\star, \mathrm{u}\} \cup \{k \mid \exists m \in M.\, \pi_{\mathcal{G}}(m){=}k\}$; for any $\ell \in L$ and $a \in \mathit{Act}$:

$$E(\ell, a) = \begin{cases} \{\nu \mid \exists n' \in N.\, (\pi_{\mathcal{G}}(\ell, n'){=}k \wedge \nu(c){=}k)\} & \text{if } \ell \in N \text{ and } a{=}k \in \mathbb{N}_+ \\ \{\nu \mid \nu(b){=}B_0\} & \text{if } \ell \in N_{\mathrm{u}} \text{ and } a = \star \\ \{\nu \mid \nu(c) = 0\} & \text{if } \ell \in N_{\mathrm{u}} \text{ and } a = \mathrm{u} \\ \emptyset & \text{otherwise} \end{cases}$$

and for any $n \in N$, $a \in \mathit{Act}(n)$, $C \subseteq \mathcal{C}$ and $\ell' \in L$:

$$\delta(n, a)(C, \ell') = \begin{cases} \frac{1}{|\{n'' \mid \pi_{\mathcal{G}}(n,n''){=}a\}|} & \text{if } a \in \mathbb{N}_+,\, C{=}\{c\},\, \ell' = n'_{\mathrm{u}} \text{ and } \pi_{\mathcal{G}}(n, n'){=}a \\ 0 & \text{otherwise} \end{cases}$$

$$\delta(n_{\mathrm{u}}, a)(C, \ell') = \begin{cases} 1 \text{ if } C = \emptyset,\, \ell' = n \text{ and } a = \mathrm{u} \\ 1 \text{ if } C = \emptyset,\, \ell' = \star \text{ and } a = \star \\ 0 \text{ otherwise.} \end{cases}$$

An example of a reduction is shown in Figure 1. For the price function $\pi_{\mathcal{G}}(s, (t, a)) = t$, it routine to verify that the optimal expected reachability price for target $F = \{\star\}{\times}V$ equals B_0 when starting from $(n_0, (0, 0))$ in the concavely-priced PTA $(\mathsf{T}_{\mathcal{G}}, \pi_{\mathcal{G}})$ if and only if player 1 has a winning strategy in the countdown game $\mathcal{G}$.

□

On the other hand, we can solve each problem in EXPTIME because:

- we can reduce each problem on probabilistic timed automata to a similar problem on the boundary region graph (see Lemmas 19–22);
- the boundary region graph has exponential-size and can be constructed in exponential time in the size of the PTA;
- on the boundary region graph (a finite state MDP) we can solve each minimisation problem using a polynomial-time algorithm (see, e.g., [27]) in the size of the graph.

For *one clock* concavely-priced PTAs expected reachability, discounted, and average-price problems are PTIME-hard as these problems are PTIME-complete [26] even for finite MDPs (i.e. PTAs with no clocks). To show PTIME-membership one can adapt the construction of [22]—which shows the NLOGSPACE-membership of the reachability problem for one clock timed automata—to obtain an abstraction similar to the boundary region graph whose size is polynomial in the size of probabilistic timed automata, and then run polynomial-time algorithms to solve this finite MDP.

Theorem 28. *The exact complexity of solving expected reachability, discounted and average price problems is EXPTIME-complete for concavely-priced PTA with two or more clocks and PTIME-complete for concavely-priced PTA with one clock.*

8 Conclusion

We presented a probabilistic extension of the boundary region graph originally defined in the case of timed automata for PTAs. We characterize expected total (finite horizon), reachability, discounted and average price using optimality equations. By analysing properties of the solutions of these optimality equations on boundary region graphs, we demonstrated that solutions on the boundary region graph are also solutions to the corresponding optimality equations on the original priced PTA. Using this reduction, we then showed that the exact complexity of solving expected reachability, discounted, and average optimisation problems on concavely-priced PTAs is EXPTIME-complete.

Although the computational complexity is very high, we feel motivated by the success of quantitative analysis tools like UPPAAL [4] and PRISM [14]. We wish to develop more efficient symbolic zone-based algorithms for the problems considered in this paper. Another direction for future work is to consider more involved objectives like price-per-reward average [7,18] and multi-objective optimisation [25].

Acknowledgments. The authors are supported in part by EPSRC grants EP/D076625 and EP/F001096.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126 (1994)
2. Alur, R., La Torre, S., Pappas, G.: Optimal paths in weighted timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, p. 49. Springer, Heidelberg (2001)

3. Beauquier, D.: Probabilistic timed automata. Theoretical Computer Science 292(1) (2003)
4. Behrmann, G., David, A., Larsen, K., Möller, O., Pettersson, P., Yi, W.: UPPAAL - present and future. In: Proc. CDC 2001, vol. 3. IEEE, Los Alamitos (2001)
5. Berendsen, J., Chen, T., Jansen, D.: Undecidability of cost-bounded reachability in priced probabilistic timed automata. In: Chen, T., Cooper, S.B. (eds.) TAMC 2009. LNCS, vol. 5532, pp. 128–137. Springer, Heidelberg (2009)
6. Berendsen, J., Jansen, D., Katoen, J.-P.: Probably on time and within budget - on reachability in priced probabilistic timed automata. In: Proc. QEST 2006. IEEE, Los Alamitos (2006)
7. Bouyer, P., Brinksma, E., Larsen, K.: Optimal infinite scheduling for multi-priced timed automata. Formal Methods in System Design 32(1) (2008)
8. Boyd, S., Vandenberghe, L.: Convex Optimization. CUP, Cambridge (2004)
9. de Alfaro, L.: Computing minimum and maximum reachability times in probabilistic systems. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, p. 66. Springer, Heidelberg (1999)
10. Dynkin, E., Yushkevich, A.: Controlled Markov Processes. Springer, Heidelberg (1979)
11. Falk, J., Horowitz, J.: Critical path problems with concave cost-time curves. Management Science 19(4) (1972)
12. Feng, Y., Xiao, B.: A Continuous-Time Yield Management Model with Multiple Prices and Reversible Price Changes. Management Science 46(5) (2000)
13. Henzinger, T., Manna, Z., Pnueli, A.: What good are digital clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
14. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
15. Jensen, H.: Model checking probabilistic real time systems. In: Proc. 7th Nordic Workshop on Programming Theory, Report 86, pp. 247–261. Chalmers University of Technology (1996)
16. Jurdzinski, M., Kwiatkowska, M., Norman, G., Trivedi, A.: Concavely-priced probabilistic timed automata. Technical Report RR-09-06. Oxford University Computing Laboratory (2009)
17. Jurdziński, M., Sproston, J., Laroussinie, F.: Model checking probabilistic timed automata with one or two clocks. Logical Methods in Computer Science 4(3) (2008)
18. Jurdziński, M., Trivedi, A.: Concavely-priced timed automata. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 48–62. Springer, Heidelberg (2008)
19. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods in System Design 29 (2006)
20. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theoretical Computer Science 282 (2002)
21. Kwiatkowska, M., Norman, G., Sproston, J., Wang, F.: Symbolic model checking for probabilistic timed automata. Information and Computation 205(7) (2007)
22. Laroussinie, F., Markey, N., Schnoebelen, P.: Model checking timed automata with one or two clocks. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 387–401. Springer, Heidelberg (2004)
23. Laroussinie, F., Sproston, J.: State explosion in almost-sure probabilistic reachability. Information Processing Letters 102(6) (2007)
24. Larsen, K., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.: As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 493. Springer, Heidelberg (2001)

25. Larsen, K., Rasmussen, J.: Optimal reachability for multi-priced timed automata. Theoretical Computer Science 390(2-3) (2008)
26. Papadimitriou, C., Tsitsiklis, J.: The complexity of Markov decision processes. Mathematics of Operations Research 12 (1987)
27. Puterman, M.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Chichester (1994)
28. Torres, E., Dominguez, J., Valdes, L., Aza, R.: Passenger waiting time in an airport and expenditure carried out in the commercial area. Journal of Air Transport Management 11(6) (2005)
29. Tripakis, S.: Verifying progress in timed systems. In: Katoen, J.-P. (ed.) AMAST-ARTS 1999, ARTS 1999, and AMAST-WS 1999. LNCS, vol. 1601, p. 299. Springer, Heidelberg (1999)

Compositional Control Synthesis for Partially Observable Systems

Wouter Kuijper[*] and Jaco van de Pol

University of Twente Formal Methods and Tools
Dept. of EEMCS
{W.Kuijper,J.C.vandePol}@ewi.utwente.nl

Abstract. We present a compositional method for deriving control constraints on a network of interconnected, partially observable and partially controllable plant components. The constraint derivation method works in conjunction with an antichain–based, symbolic algorithm for computing weakest strategies in safety games of imperfect information. We demonstrate how the technique allows a reactive controller to be synthesized in an incremental manner, exploiting locality and independence in the problem specification.

1 Introduction

Control Synthesis [23] is the idea of automatically synthesizing a controller for enforcing some desired behaviour in a plant. This problem can be phrased logically as follows. Given a plant description P and a desired property ϕ, construct a controller C such that $P \| C \vDash \phi$. Control synthesis is a close cousin of the model checking problem. Where model checking is about establishing whether or not a model supports a given property, control synthesis is about generating a model on which the property will hold.

The main difficulty that any effective procedure for controller synthesis must face is that the uncontrolled state space generated by the plant description is typically large. This is mainly due to concurrency in the model, which is a central issue also in model checking. However, for synthesis the problem is amplified by two additional, complicating factors. First, we typically see a higher degree of non–determinism because a priori no control constraints are given. Second, it is often the case that the state of the plant P is only partially observable for the controller C. Resolving this may incur another exponential blowup. On instances, this blowup may be avoided by using smart, symbolic methods [26].

Contribution. In this paper we focus on the compositional synthesis of a reactive controller under the assumption of partial observability. Our main contributions are a compositional framework for describing control synthesis problems as a network of interconnected, partially controllable, partially observable plant components, and a compositional method for synthesizing safety controllers over such a plant model.

We believe there are at least two novel aspects to our approach. First, there is the combination of imperfect information with compositionality. In particular, we make

[*] This author is supported by NWO project 600.065.120.24N20.

sure that context assumptions take into account partial observability of the components. Second, our framework ensures context assumptions gradually shift in the direction of control constraints as the scope widens. In this way we avoid, to some extent, unrealistic assumptions, and generally obtain a less permissive context. Note that the size of the context assumptions is an important factor [19] in the efficiency of assume–guarantee based methods.

This work is complementary to our work in [13]. The game solving algorithm we present there is designed to be applied in a compositional setting. It is an efficient counterexample driven algorithm for computing sparse context assumptions. As such, it is especially suitable for the higher levels in the composition hierarchy where abstraction is often possible (and useful). However, its successful application hinges on the fact that control constraints local to any of the considered subcomponents should already be incorporated into the game board. Here we show how this can be achieved by proceeding compositionally.

Related Work. Synthesis of reactive systems was first considered by Church [10] who suggested the problem of finding a set of restricted recursion equivalences mapping an input signal to an output signal satisfying a given requirement [25]. The classical solutions to Church's Problem [4,22] in principle solve the synthesis problem for omega–regular specifications. Since then, much of the subsequent work has focused on extending these results to richer classes of properties and systems, and on making synthesis more scalable.

Pioneering work on synthesis of *closed* reactive systems [18,11] uses a reduction to the satisfiability of a temporal logic formula. That it is also possible to synthesize *open* reactive systems is shown [20,21] using a reduction to the satisfiability of a CTL* formula, where path operators force alternation between the system and the environment. Around the same time, another branch of work [23,16] considers the synthesis problem specifically in the context of control of discrete event systems, this introduces many important control theoretical concepts, such as *observability*, *controllability*, and the notion of a *control hierarchy*.

More recently several contributions have widened the scope of the field and addressed several scalabilty issues. Symbolic methods, already proven successful in a verification setting, can be applied also for synthesis [2]. Symbolic techniques also enable synthesis for hybrid systems which incorporate continuous as well as discrete behaviour [1]. Controller synthesis under partial information can be solved by a reduction to the emptiness of an alternating tree automaton [15]. This method is very general and works in branching and linear settings. However, scalability issues remain as the authors note that most of the combinatorial complexity is shifted to the emptiness check for the alternating tree automaton. In [14] a compositional synthesis method is presented that reduces the synthesis problem to the emptiness of a non–deterministic Büchi tree automaton. The authors of [17] consider the specific case of hard real time systems. They argue that the full expressive power of omega regular languages may not be necessary in such cases, since a bounded response requirement can be expressed as a safety property.

Even the earliest solutions to Church's problem, essentially involve solving a game between the environment and the control [24]. As such there is a need to study the (symbolic) representation and manipulation of games and strategies as first class citizens. In [7] a symbolic algorithm for games with imperfect information is developed based on fixed point iteration over antichains. In [5] there are efficient on–the–fly algorithms for solving games of imperfect information. In [13] we propose a counter example driven algorithm for computing weakest strategies in safety games of imperfect information.

Compositionality adds another dimension to the synthesis problem: for reasons of scalability it is desirable to solve the synthesis problem in an incremental manner, treating first subproblems in isolation before combining their results. In general this requires a form of assume–guarantee reasoning. There exists an important line of related work that addresses such issues.

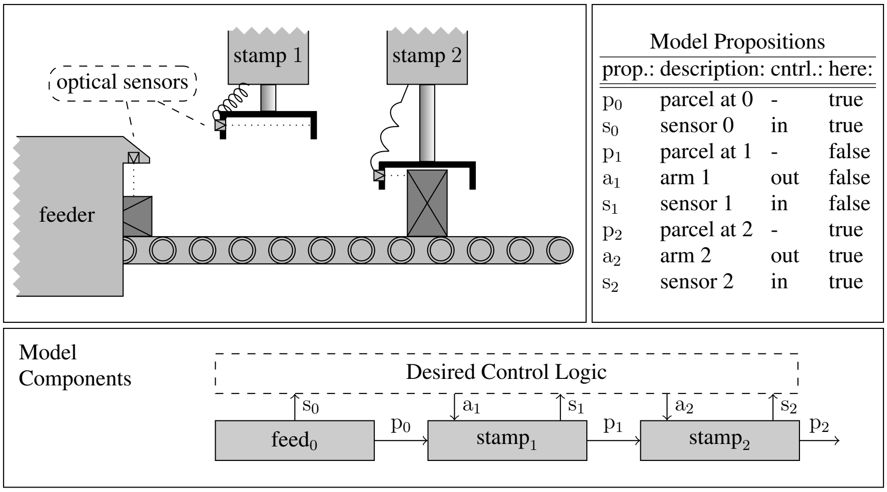
One such recent development that aims to deal with component based designs is the work on interface automata [12]. This work introduces *interfaces* as a set of behavioural assumptions/guarantees between components. Composition of interfaces is *optimistic*: two interfaces are *compatible* iff there exists an environment in which they could be made to work together. In this paper, we work from a similar optimistic assumption while deriving local control constraints, i.e.: a local transition should not be disabled as long as there exists a safe context which would allow it to be taken. A synchronous, bidirectional interface model is presented in [6]. Our component model is similar, but differs on the in–/output partition of the variables to be able to handle partially observable systems. Besides providing a clean theory of important software engineering concepts like *incremental design* and *independent implementability*, interfaces also have nice algorithmic properties allowing for instance automated refinement and compatibility checking. Several algorithms for interface synthesis are discussed in [3].

Authors in [9] describe a co–synthesis method based on assume–guarantee reasoning. Their solution is interesting in that it addresses non–zero–sum games where processes compete, but not at all cost. In [8] the same authors explore ways in which to compute environment assumptions for specifications involving liveness properties, where removal of unsafe transitions constitutes a pre–processing step. Note that, for a liveness property, in general, there does not exist a unique weakest assumption on the context, so this is a non–trivial problem.

Structure. The rest of the paper is structured as follows. In Section 2 we build up a game–theoretic, semantical framework for control synthesis problems. We also present our running example used to illustrate all the definitions. In Section 3 we propose a sound and complete compositional method for control synthesis. We demonstrate the method on the running example. In Section 4 we conclude with a summary of the contribution, and perspectives on future work.

2 Compositional Framework

In this section we give a formal semantics for our central object of interest, which is the *plant under control*, or PuC. First we give an example.



Model Propositions

prop.:	description:	cntrl.:	here:
p_0	parcel at 0	-	true
s_0	sensor 0	in	true
p_1	parcel at 1	-	false
a_1	arm 1	out	false
s_1	sensor 1	in	false
p_2	parcel at 2	-	true
a_2	arm 2	out	true
s_2	sensor 2	in	true

Fig. 1. A modular parcel stamping plant. (top left) The legend (top right) shows all the propositions we used to model this example. The decomposition structure (bottom) shows the components we identified in the model and their interconnections, in terms of input and output propositions.

A Motivating Example. We consider the *parcel plant* illustrated in Figure 1. This fictive plant consists of a *feeder* and two *stamps* connected together with a conveyor belt. A parcel is fed onto the belt by the feeder. The belt transports the parcel over to stamp 1 which prints a tracking code. The belt then transports the parcel over to stamp 2 which stamps the shipping company's logo. Next to the illustration in Figure 1 we list all the *propositions* we used to model this particular example. For each proposition we indicate whether it is a control output/input, and whether or not it holds for the current state of the plant as it is shown in the picture. We specify the behaviour of the three components in the parcel stamp using Kripke structures over these atomic propositions in Figure 2.

Definition 1 (Kripke Structures). We let $\mathcal{X}$ be a background set of *propositions*. For a given (finite) subset $X \subseteq \mathcal{X}$ of propositions we define $\mathcal{S}[X] = 2^X$ as the set of *states*, or *valuations* over X. We define shorthand $\mathcal{S} = \mathcal{S}[\mathcal{X}]$. A *Kripke structure* is a tuple $A = (L, X, \gamma, \delta, L^{\text{init}})$, consisting of a set of *locations* L, a finite set of *relevant propositions* $X \subseteq \mathcal{X}$, a *propositional labeling* $\gamma : L \to \mathcal{S}[X]$, a *transition relation* $\delta \subseteq L \times L$, and a set of *initial locations* $L^{\text{init}} \subseteq L$. For any two Kripke structures A_1, A_2 the *composition* $A_{12} = A_1 \| A_2$ is defined with $L_{12} = \{(\ell_1, \ell_2) \in L_1 \times L_2 \mid \gamma_1(\ell_1) \cap X_2 = \gamma_2(\ell_2) \cap X_1\}$, $X_{12} = X_1 \cup X_2$, for all $(\ell_1, \ell_2) \in L_{12}$ it holds $\gamma_{12}(\ell_1, \ell_2) = \gamma_1(\ell_1) \cup \gamma_2(\ell_2)$, for all $(\ell_1, \ell_2), (\ell_1', \ell_2') \in L_{12}$ it holds $(\ell_1, \ell_2)\delta_{12}(\ell_1', \ell_2')$ iff $\ell_1 \delta_1 \ell_1'$ and $\ell_2 \delta_2 \ell_2'$, and, for the initial locations, it holds $L_{12}^{\text{init}} = (L_1^{\text{init}} \times L_2^{\text{init}}) \cap L_{12}$. Note that L_{12} contains all the pairs of locations in the Kripke structures that are *consistent*, meaning that they agree on the truth of all *shared propositions* $X_1 \cap X_2$. $\triangleleft$

We note that, in our notation, we use a horizontal bar to denote the negation of a proposition letter, i.e.: $\overline{a_1}$ should be read as *not* a_1 which, in turn, is interpreted as *stamp*

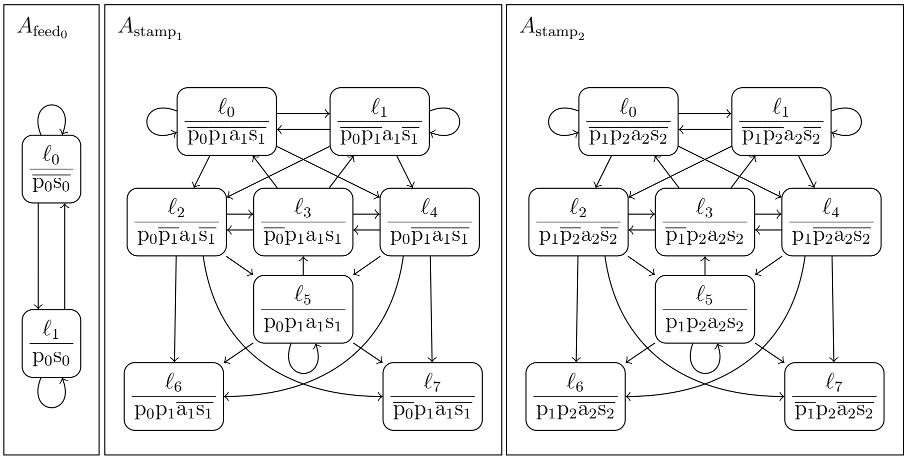

A_{feed_0}
ℓ_0
$\overline{\mathrm{p}_0}\,\overline{\mathrm{s}_0}$
ℓ_1
$\mathrm{p}_0\mathrm{s}_0$
A_{stamp_1}
ℓ_0
$\overline{\mathrm{p}_0}\,\overline{\mathrm{p}_1}\,\overline{\mathrm{a}_1}\,\overline{\mathrm{s}_1}$
ℓ_1
$\overline{\mathrm{p}_0}\,\overline{\mathrm{p}_1}\mathrm{a}_1\overline{\mathrm{s}_1}$
ℓ_2
$\mathrm{p}_0\overline{\mathrm{p}_1}\mathrm{a}_1\overline{\mathrm{s}_1}$
ℓ_3
$\overline{\mathrm{p}_0}\mathrm{p}_1\mathrm{a}_1\mathrm{s}_1$
ℓ_4
$\mathrm{p}_0\overline{\mathrm{p}_1}\,\overline{\mathrm{a}_1}\,\overline{\mathrm{s}_1}$
ℓ_5
$\mathrm{p}_0\mathrm{p}_1\mathrm{a}_1\mathrm{s}_1$
ℓ_6
$\mathrm{p}_0\mathrm{p}_1\overline{\mathrm{a}_1}\,\overline{\mathrm{s}_1}$
ℓ_7
$\overline{\mathrm{p}_0}\mathrm{p}_1\overline{\mathrm{a}_1}\,\overline{\mathrm{s}_1}$
A_{stamp_2}
ℓ_0
$\overline{\mathrm{p}_1}\,\overline{\mathrm{p}_2}\,\overline{\mathrm{a}_2}\,\overline{\mathrm{s}_2}$
ℓ_1
$\overline{\mathrm{p}_1}\,\overline{\mathrm{p}_2}\mathrm{a}_2\overline{\mathrm{s}_2}$
ℓ_2
$\mathrm{p}_1\overline{\mathrm{p}_2}\mathrm{a}_2\overline{\mathrm{s}_2}$
ℓ_3
$\overline{\mathrm{p}_1}\mathrm{p}_2\mathrm{a}_2\mathrm{s}_2$
ℓ_4
$\mathrm{p}_1\overline{\mathrm{p}_2}\,\overline{\mathrm{a}_2}\,\overline{\mathrm{s}_2}$
ℓ_5
$\mathrm{p}_1\mathrm{p}_2\mathrm{a}_2\mathrm{s}_2$
ℓ_6
$\mathrm{p}_1\mathrm{p}_2\overline{\mathrm{a}_2}\,\overline{\mathrm{s}_2}$
ℓ_7
$\overline{\mathrm{p}_1}\mathrm{p}_2\overline{\mathrm{a}_2}\,\overline{\mathrm{s}_2}$

Fig. 2. Kripke structures modeling the feeder (left), stamp 1 (center), and stamp 2 (right) of the parcel plant in Figure 1.

number 1 has not activated its stamping arm. Next, we note that the models for the stamps contain deadlock locations. We use this to encode a *safety property* into the model. The safety property here simply says that the stamp *must* activate whenever there is a parcel present on the belt, otherwise the system will deadlock.

Game Semantics. We now turn to assigning a standard game semantics to the type of plant models like the parcel plant that we have just introduced. We start with some prerequisites.

Definition 2 (Strategy). A *strategy* $f : \mathcal{S}^* \to 2^{\mathcal{S}}$ is a function mapping *histories* of states to the sets of allowed successor states. With $f_\top$ we denote the strategy that maps all histories to $\mathcal{S}$. A *trace* $\sigma = s_0 \dots s_n \in \mathcal{S}^+$ is *consistent* with f iff $f(\epsilon) \ni s_0$ and for all $0 < i \leq n$ it holds $f(s_0 \dots s_{i-1}) \ni s_i$. With $\mathsf{Reach}(f)$ we denote the set of traces consistent with f. We require strategies to be *prefix–closed* meaning that $f(\sigma) \neq \varnothing$ and $\sigma \neq \epsilon$ implies $\sigma \in \mathsf{Reach}(f)$. A strategy f is *safe* iff for all $\sigma \in \mathsf{Reach}(f)$ it holds $f(\sigma) \neq \varnothing$. With Safe we denote the set of all safe strategies. $\lhd$

Any given Kripke structure can be viewed as a strategy by considering the restriction that the Kripke structure places on the next state given the history of states that came before.

Definition 3 (Regular Strategies). To a given Krikpke structure A, we assign the strategy $[\![A]\!] : \mathcal{S}^* \to 2^{\mathcal{S}}$ such that for the empty trace ϵ it holds $s \in [\![A]\!](\epsilon)$ iff there exists an initial location $\ell_0 \in L_A^{\mathrm{init}}$ such that ℓ_0 is consistent with s, and for a given history $\sigma = s_0 \dots s_n \in \mathcal{S}^+$ it holds $s' \in [\![A]\!](\sigma)$ iff there exists a computation $\ell_0 \dots \ell_{n+1}$ in the Kripke structure such that $\ell_0 \in L_A^{\mathrm{init}}$, each location ℓ_i for $i \leq n$ is consistent with s_i, and s' is consistent with ℓ_{n+1}. A strategy $f \in \mathcal{F}$ is regular iff there exists a finite Kripke structure A such that $f = [\![A]\!]$. $\lhd$

It is often important that a Kripke structure A is input enabled for a subset of the propositions $X_A^{\mathrm{i}} \subseteq X_A$, as is the case for the Kripke structures we defined for the parcel stamp. In practice this is easy to enforce using some syntactic criteria on the specification. On a semantic level we prefer to abstract from the particular way the strategy is represented. For this reason we introduce the notion of *permissibility*.

Definition 4 (Permissibility). For a given set of propositions $X \subseteq \mathcal{X}$ we define the *indistinguishability relation* $\sim_X \subseteq \mathcal{S} \times \mathcal{S}$ such that $s \sim_X s'$ iff $s \cap X = s' \cap X$, we lift $\sim_X$ to *traces* of states $\sigma = s_1 \dots s_n \in \mathcal{S}^*$ and $\sigma' = s'_1 \dots s'_m \in \mathcal{S}^*$ such that $\sigma \sim_X \sigma'$ iff $n = m$ and for all $0 < i \leq n$ it holds $s_i \sim_X s'_i$. A *signature* is a pair $[X^{\mathrm{i}} \to X^{\mathrm{o}}]$ such that $X^{\mathrm{i}} \subseteq^{\mathrm{finite}} \mathcal{X}$ and $X^{\mathrm{o}} \subseteq^{\mathrm{finite}} \mathcal{X}$, we let $X^{\mathrm{io}} = X^{\mathrm{i}} \cup X^{\mathrm{o}}$. A strategy $f \in \mathcal{F}$ is *permissible* for a signature $[X^{\mathrm{i}} \to X^{\mathrm{o}}]$ iff for all $\sigma, \sigma' \in \mathcal{S}^*$ such that $\sigma \sim_{X^{\mathrm{io}}} \sigma'$ and all $s, s' \in \mathcal{S}$ such that $s \sim_{X^{\mathrm{o}}} s'$ we have $s \in f(\sigma)$ iff $s' \in f(\sigma')$. With $\mathcal{F}[X^{\mathrm{i}} \to X^{\mathrm{o}}]$ we denote the set of strategies that are permissible for $[X^{\mathrm{i}} \to X^{\mathrm{o}}]$. $\lhd$

To illustrate, we can now formalize a suitable notion of *input enabledness* for Kripke structures in terms of permissibility. We say that A is *input enabled* for a subset of the relevant propositions $X_A^{\mathrm{i}} \subseteq X_A$ iff $[\![A]\!] \in \mathcal{F}[X_A^{\mathrm{i}} \to X_A^{\mathrm{o}}]$ where $X_A^{\mathrm{o}} = X_A \setminus X_A^{\mathrm{i}}$.

Definition 5 (Lattice of Strategies). We fix a partial order $\sqsubseteq$ on the set of strategies such that $f \sqsubseteq f'$ iff for all $\sigma \in \mathcal{S}^*$ it holds $f(\sigma) \subseteq f'(\sigma)$ we say f' is *more permissive* or *weaker* than f. The set of strategies ordered by permissiveness forms a complete lattice. The *join* of a set of strategies $F \subseteq \mathcal{F}$ is denoted $\sqcup F$ and is defined as $f \in \mathcal{F}$ such that for all $\sigma \in \mathcal{S}^*$ it holds $f(\sigma) = \cup_{f' \in F} f'(\sigma)$. The *meet* is denoted $\sqcap F$ and is defined dually. $\lhd$

We have now all the prerequisites to introduce the concept of a *plant under control* or PuC. A PuC is a semantic object that encodes the behaviour of a system of interacting plant components. In addition it also specifies a set of control localities which are selected subsets of plant components that are of special interest because of their control dependencies. For each such control locality the PuC encodes context assumptions and control constraints. Later we will show how these assumptions and constraints can be automatically derived by solving games.

Definition 6 (Plant under Control). A *plant under control (PuC)* M is a tuple

$$M = (P, \{f_p, [X_p^{\mathrm{i}} \to X_p^{\mathrm{o}}]\}_{p \in P}, C, \{g_K, h_K\}_{K \in C})$$

Consisting of a finite set of *plant components* P, for each plant component $p \in P$ the PuC represents the *component behaviour* as the strategy $f_p \in \mathcal{F}[X_p^{\mathrm{i}} \to X_p^{\mathrm{o}}]$. We require for all $p_1, p_2 \in P$ such that $p_1 \neq p_2$ it holds $X_{p_1}^{\mathrm{i}} \cap X_{p_2}^{\mathrm{i}} = X_{p_1}^{\mathrm{o}} \cap X_{p_2}^{\mathrm{o}} = \varnothing$. The PuC has a selected set of *control localities* $C \subseteq 2^P$ such that $P \in C$. For a given control locality $K \in C$ we define $X_K^{\mathrm{i}} = \cup_{p \in K} X_p^{\mathrm{i}}$, and $X_K^{\mathrm{o}} = \cup_{p \in K} X_p^{\mathrm{o}}$, and $f_K = \sqcap_{p \in K} f_p$. We define the *control signature* $[X^{\mathrm{ci}} \to X^{\mathrm{co}}]$ such that $X^{\mathrm{ci}} = X_P^{\mathrm{o}} \setminus X_P^{\mathrm{i}}$ and $X^{\mathrm{co}} = X_P^{\mathrm{i}} \setminus X_P^{\mathrm{o}}$. For each control locality $K \in C$, the PuC represents the current *context assumptions* as the strategy $g_K \in \mathcal{F}[X_K^{\mathrm{o}} \setminus X_K^{\mathrm{i}} \to X_K^{\mathrm{i}} \setminus X_K^{\mathrm{o}}]$, and the current *control constraints* as the strategy $h_K \in \mathcal{F}[X_K^{\mathrm{o}} \cap X^{\mathrm{ci}} \to X_K^{\mathrm{i}} \cap X^{\mathrm{co}}]$. $\lhd$

In this definition we are assuming a set of interacting plant components that communicate to each other and to the controller by means of their signature of input/output propositions. If a proposition is both input and output to the same component we say it is *internal* to the plant component. The definition ensures that no other plant component may synchronize on such an internal proposition. We assume that all non–internal propositions that are not used for synchronization among plant components are control propositions (open plant output propositions are control input propositions, and open plant input propositions are control output propositions).

Note that we are assuming a *given* set of control localities. This information should be added to an existing componentized model of the plant. Either manually or by looking for interesting clusters of plant components that have some mutual dependencies. Such an automated clustering approach has already been investigated for compositional verification in [19].

Example 1 (Parcel Stamp). We define a PuC M_{parcel} for the parcel stamp example. We first fix the plant components $P_{\mathrm{parcel}} = \{\mathrm{feed}_0, \mathrm{stamp}_1, \mathrm{stamp}_2\}$. Their signatures are $X^{\mathrm{o}}_{\mathrm{feed}_0} = \{\mathrm{p}_0, \mathrm{s}_0\}$, $X^{\mathrm{i}}_{\mathrm{feed}_0} = \varnothing$, $X^{\mathrm{o}}_{\mathrm{stamp}_1} = \{\mathrm{p}_1, \mathrm{s}_1\}$, $X^{\mathrm{i}}_{\mathrm{stamp}_1} = \{\mathrm{p}_0, \mathrm{a}_1\}$, $X^{\mathrm{o}}_{\mathrm{stamp}_2} = \{\mathrm{p}_2, \mathrm{s}_2\}$, $X^{\mathrm{i}}_{\mathrm{stamp}_2} = \{\mathrm{p}_1, \mathrm{a}_2, \mathrm{p}_2\}$, Note that we make p_2 an internal variable of stamp_2 since it is not input to any other component, in this way the control signature becomes $X^{\mathrm{ci}} = \{\mathrm{s}_0, \mathrm{s}_1, \mathrm{s}_0\}$ and $X^{\mathrm{co}} = \{\mathrm{a}_1, \mathrm{a}_2\}$. The component behaviour is given by the Kripke structures in Figure 2, $f_{\mathrm{feed}_0} = [\![A_{\mathrm{feed}_0}]\!]$, and $f_{\mathrm{stamp}_1} = [\![A_{\mathrm{stamp}_1}]\!]$, and $f_{\mathrm{stamp}_2} = [\![A_{\mathrm{stamp}_2}]\!]$. We define the control localities

$$\begin{aligned} C_{\mathrm{parcel}} = \{&\{\mathrm{feed}_0, \mathrm{stamp}_1, \mathrm{stamp}_2\}, \\ &\{\mathrm{feed}_0, \mathrm{stamp}_1\}, \{\mathrm{stamp}_1, \mathrm{stamp}_2\}, \\ &\{\mathrm{feed}_0\}, \{\mathrm{stamp}_1\}, \{\mathrm{stamp}_2\}\} \end{aligned}$$

The context assumptions g_K and control guarantees h_K for each locality $K \in C$ are initially set to the vacuous strategy $g_K = h_K = f_\top$. $\triangleleft$

Global Control Constraints. For a given PuC M we are interested in computing the weakest global control constraints $\hat{h}_P$ such that $f_P \sqcap \hat{h}_P \in \mathsf{Safe}$. In principle this can be done by viewing the PuC as a safety game of imperfect information where the safety player may, at each turn, observe the value of the control input propositions and determine the value of the control output propositions. In this way we obtain a game graph that can be solved using conventional game solving algorithms. The result will be the weakest strategy $\hat{h}_P$ for the safety player.

Definition 7 (Weakest Safe Global Strategy). For a given PuC M we define the *weakest safe global control constraints* $\hat{h}_P$ as follows

$$\hat{h}_P = \sqcup\{h \in \mathcal{F}[X^{\mathrm{ci}} \to X^{\mathrm{co}}] \mid f_P \sqcap h \in \mathsf{Safe}\}$$

i.e. the weakest global control strategy that is sufficient to keep the system safe. $\triangleleft$

Computing $\hat{h}_P$ directly by solving the global safety game does not scale very well to larger systems. For this reason we want to proceed compositionally and start with smaller control localities $K \in C$ such that $K \subset P$ before treating the plant $P \in C$ as a whole. As it turns out solving the local safety game over the control signature

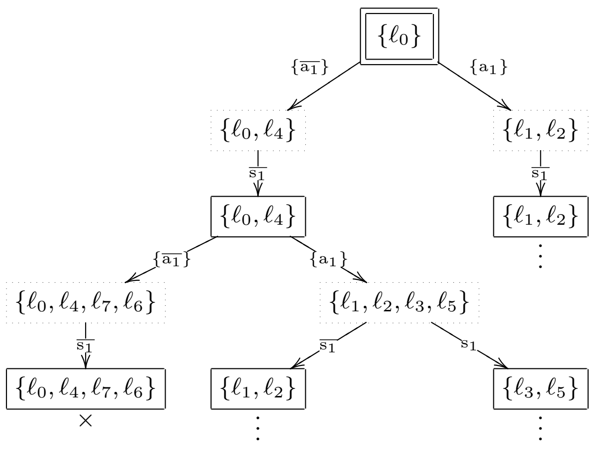


Fig. 3. Partial game tree for A_{stamp_1}, where the safety player is restricted to use the control signature $[X^{\text{o}}_{\text{stamp}_1} \cap X^{\text{ci}} \to X^{\text{i}}_{\text{stamp}_1} \cap X^{\text{co}}] = [\{s_1\} \to \{a_1\}]$

$[X^{\text{o}}_K \cap X^{\text{ci}} \to X^{\text{i}}_K \cap X^{\text{co}}]$ will yield control constraints that are too strong in the sense that not every possible safe control solution on the global level will be preserved. In Example 2 we illustrate this phenomenon.

Example 2 (Overrestrictive Control). In Figure 3 we show a partial unravelling of the game board A_{stamp_1} into a *game tree*. The nodes in the game tree are partitioned into nodes for the *safety player* shown as solid boxes and nodes for the *reachability player* shown as dotted boxes. Edges originating at nodes for the safety player are annotated with allow sets. For these examples it suffices to consider only allow sets consisting of a single, concrete control output. Edges originating at the nodes for the reachability player are annotated with concrete control inputs. Since we consider only singleton allow sets it is not necessary to also include the concrete control output on these edges.

The nodes for the safety player are annotated with the *knowledge* or *information set* that the safety player has given the observation history. The nodes for the reachability player are labeled with the *forcing sets* which are all locations to which there exists a trace that is consistent with the observation history *and* the control output as allowed by the safety player.

From a forcing set, the reachability player fixes the control input by choosing one of the locations in the forcing set as the concrete successor location. Note that the subset construction does not show the concrete successor locations. Rather it shows the resulting information set for the safety player, which is the smallest set of locations that are consistent with the input/output pair that has been exchanged.

As can be seen the safety player is forced, after 1 iteration, to always play a_1. Meaning that, she is *always* activating the stamp. She cannot, based on her observations, determine for sure whether or not there is a parcel present. Note however, if we would have taken also the feeder component A_{feed_0} into account, it would have been possible for the safety player to deduce this information based on the sensor in the feeder. So the

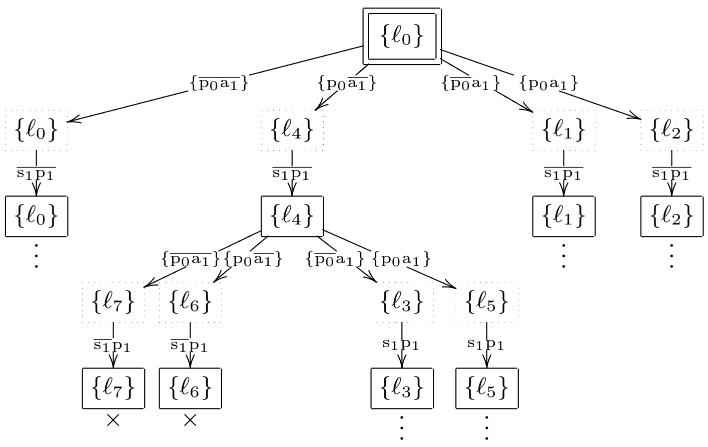


Fig. 4. Partial game tree for A_{stamp_1}, where the safety player is allowed to use the control signature $[X^{\mathrm{o}}_{\mathrm{stamp}_1} \setminus X^{\mathrm{i}}_{\mathrm{stamp}_1} \to X^{\mathrm{i}}_{\mathrm{stamp}_1} \cap X^{\mathrm{o}}_{\mathrm{stamp}_1}] = [\{\mathrm{s}_1, \mathrm{p}_1\} \to \{\mathrm{p}_0, \mathrm{a}_1\}]$.

strategy for A_{stamp_1} that we got from solving this game does not respect the strategy for $A_{\mathrm{feed}_0} \| A_{\mathrm{stamp}_1}$ which activates only if the optical sensor in the feeder is triggered. $\lhd$

3 Compositional Synthesis Method

Our solution approach to the problems sketched in the previous section is based on an over approximation of the allowable behaviour followed by an under approximation of the denyable behaviour. The soundness of our approach rests on the notion of *conservativity*.

Definition 8 (Conservative Systems). A PuC is *conservatively constrained* iff for all $K \in C$ both the local assumptions g_K and the local constraints h_K are *conservative*, meaning that $f_P \sqcap \hat{h}_P \sqsubseteq g_K$ and $f_P \sqcap \hat{h}_P \sqsubseteq h_K$, i.e.: both the local assumptions and the local constraints allow all the behaviour that would be allowed by the weakest safe global control constraints. A system that is not conservatively constrained is *over constrained*. $\lhd$

For a conservatively constrained PuC we may always take into account the existing control constraints and context assumptions while computing new control constraint or context assumptions. This unlocks possibilities that allows more efficient symbolic game solving. For larger systems there may exist control localities that need highly non–trivial context assumptions which require a lot of computation time and storage space [19]. This problem is sometimes referred to as the problem of *assumption explosion*.

To prevent this we rely on two mechanisms. The first is the fact that the signature for the context assumptions g_K converges to the signature for the control constraints h_K as K approaches P. Note that, at the highest level of composition, P, the signatures for the control constraints and the context assumptions coincide. This means that context assumptions become more and more like control constraints as we progress upward in the decomposition hierarchy C. The second mechanism we rely on is a synergy between context assumptions and control constraints. In particular, for conservative systems, it is possible while computing weakest context assumptions for a control locality $K \in C$ to take into account the conservative control constraints of all lower control localities $K' \subset K$ that have been previously computed. We refer to this as *subordinate* control.

Definition 9 (Conservative Local Context Assumptions). For a PuC M and control locality $K \in C$ we define the *subordinate localities* $K{\downarrow} = \{K' \in C \mid K' \subset K\}$, and the *subordinate control* $h_{K\downarrow} = \sqcap_{K' \in K\downarrow} h_{K'}$. Now $\mathsf{WeakestContext}_K(M) = M'$ such that

$$g'_K = \sqcup\{g' \in \mathcal{F}[X^{\mathrm{o}}_K \setminus X^{\mathrm{i}}_K \to X^{\mathrm{i}}_K \setminus X^{\mathrm{o}}_K] \mid (f_K \sqcap h_{K\downarrow}) \sqcap g' \in \mathsf{Safe}\}$$

and M' is equal to M otherwise. $\lhd$

Lemma 1 (WeakestContext). *The operation* $\mathsf{WeakestContext}_K(\cdot)$ *preserves conservativity.* $\lhd$

Proof Sketch. We define $\hat{g}_K = \sqcap\{g \in \mathcal{F}[X^{\mathrm{o}}_K \setminus X^{\mathrm{i}}_K \to X^{\mathrm{i}}_K \setminus X^{\mathrm{o}}_K] \mid (f_P \sqcap \hat{h}_P) \sqsubseteq g\}$. For this context assumption we can prove that it is conservative and safe in the sense that $(f_K \sqcap h_{K\downarrow}) \sqcap \hat{g}_K \in \mathsf{Safe}$, it follows, by Definition 9, that $\hat{g}_K \sqsubseteq g'_K$, hence g'_K is also conservative. $\square$

Example 3 (Computing Conservative Local Context Assumptions). In Figure 4 we show a partial unravelling of the game board A_{stamp_1} into a game tree, this time for the control signature $[X^{\mathrm{o}}_{\mathrm{stamp}_1} \setminus X^{\mathrm{i}}_{\mathrm{stamp}_1} \to X^{\mathrm{i}}_{\mathrm{stamp}_1} \setminus X^{\mathrm{o}}_{\mathrm{stamp}_1}] = [\{\mathrm{s}_1, \mathrm{p}_1\} \to \{\mathrm{a}_1, \mathrm{p}_0\}]$.

When we solve this game and determine the weakest safe strategy we obtain the strategy which, in modal logic notation, can be defined as follows: $\mathrm{p}_0 \to \bigcirc \mathrm{a}_1$, i.e.: when there is a parcel in the feeder the stamp *must* activate in the next state. This regular strategy is shown in Figure 5 (left) encoded as a Kripke structure.

Note however, that this strategy relies on observation of p_0 which is not in the control signature. This means that this strategy encodes an *assumption* on the context, rather than a *guarantee* on the control. Assumptions may or may not be realizable depending on the rest of the plant. In this example, for this particular constraint, a control is realizable because the feeder component indeed provides us with an observation s_0 that allows the control to derive the status of p_0 by causality. $\lhd$

To fully exploit the synergy between context assumptions and control constraints we need to obtain also control constraints on a local level. So far we have shown (in Example 3) how local context assumptions can be computed, at the same time we have shown (in Example 2) that the direct approach to computing local control guarantees breaks down because it may yield constraints that are not conservative. However, as it turns out, it is possible to approximate conservative control constraints based on conservative context assumptions. Intuitively, this is done by *under approximating* the denyable behaviour.

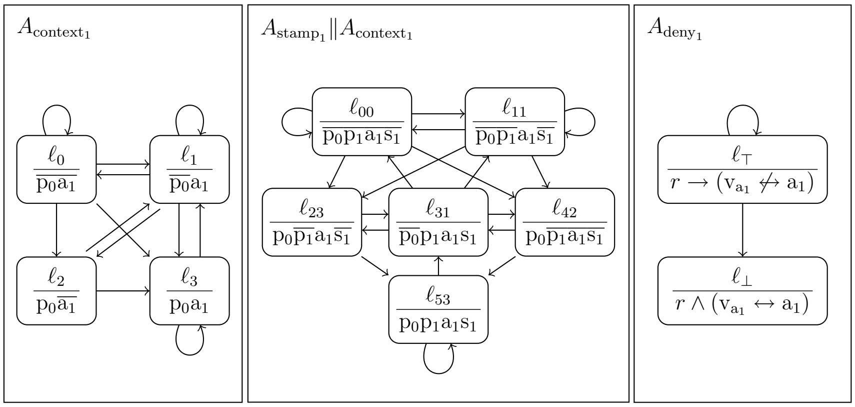

A_{context_1}
ℓ_0
$\overline{\mathrm{p}_0\mathrm{a}_1}$
ℓ_1
$\overline{\mathrm{p}_0}\mathrm{a}_1$
ℓ_2
$\mathrm{p}_0\overline{\mathrm{a}_1}$
ℓ_3
$\mathrm{p}_0\mathrm{a}_1$
$A_{\mathrm{stamp}_1} \| A_{\mathrm{context}_1}$
ℓ_{00}
$\overline{\mathrm{p}_0\mathrm{p}_1\mathrm{a}_1\mathrm{s}_1}$
ℓ_{11}
$\overline{\mathrm{p}_0\mathrm{p}_1}\mathrm{a}_1\overline{\mathrm{s}_1}$
ℓ_{23}
$\mathrm{p}_0\overline{\mathrm{p}_1}\mathrm{a}_1\overline{\mathrm{s}_1}$
ℓ_{31}
$\overline{\mathrm{p}_0}\mathrm{p}_1\mathrm{a}_1\mathrm{s}_1$
ℓ_{42}
$\mathrm{p}_0\overline{\mathrm{p}_1\mathrm{a}_1\mathrm{s}_1}$
ℓ_{53}
$\mathrm{p}_0\mathrm{p}_1\mathrm{a}_1\mathrm{s}_1$
A_{deny_1}
$\ell_\top$
$r \to (\mathrm{v}_{\mathrm{a}_1} \not\leftrightarrow \mathrm{a}_1)$
$\ell_\bot$
$r \wedge (\mathrm{v}_{\mathrm{a}_1} \leftrightarrow \mathrm{a}_1)$

Fig. 5. Context assumptions for stamp_1 component (left), the behaviour of the stamp_1 component in its idealized context (center), the special Kripke structure encoding the rules of the dual deny game for the stamp_1 component (right).

Definition 10 (Conservative Local Control). For a given PuC M and a control locality $K \in C$ we define $\mathsf{StrongestControl}_K(M) = M'$ such that

$$h'_K = \sqcap\{h' \in \mathcal{F}[X^{\mathrm{o}}_K \cap X^{\mathrm{ci}} \to X^{\mathrm{i}}_K \cap X^{\mathrm{co}}] \mid (f_K \sqcap h_{K\downarrow}) \sqcap g_K \sqsubseteq h'\}$$

and M' is equal to M otherwise. $\lhd$

Lemma 2 (StrongestControl). *The operation* $\mathsf{StrongestControl}_K(\cdot)$ *preserves conservativity.* $\lhd$

Proof. By conservativity of M it holds $f_P \sqcap \hat{h}_P \sqsubseteq (f_K \sqcap h_{K\downarrow}) \sqcap g_K$. By Definition 10 it holds $(f_K \sqcap h_{K\downarrow}) \sqcap g_K \sqsubseteq h'_K$. It follows $f_P \sqcap \hat{h}_P \sqsubseteq h'_K$. $\square$

Example 4 (Computing Strongest Local Control). We can compute an approximate local control strategy by exploiting a natural duality that exists between *allow strategies* and *deny strategies*. Where allow strategies determine what is the set of *allowed* control outputs based on the observation history, deny strategies work by mapping an observation history to the set of *denied* control outputs, which is just the complement of the allowed set. We can exploit this duality because the *weakest conservative deny strategy* is the *strongest conservative allow strategy*.

The construction that turns an *allow game* into a *deny game* is then done as follows. First we turn the control outputs $a_1, a_2 \in \mathrm{X}^{\mathrm{co}}$ into control inputs $a_1, a_2 \in \mathrm{X}^{\mathrm{ci}'}$ and introduce, specifically for the control outputs on which we want to derive the strongest local constraints, a fresh set of *deny outputs* $\mathrm{v}_{a_1}, \mathrm{v}_{a_2} \dots \in X^{\mathrm{co}'}$. We add one special control output $\mathrm{r} \in X^{\mathrm{co}'}$ which is called *restrict*. The rules of the game are as follows: if the safety player plays $\overline{r}$ the next state is *not restricted*. And the plant in its idealized context progresses normally. If the safety player plays r, *restrict*, we require that, in the next state, *at least one of the deny outputs* $\mathrm{v}_{a_1}, \mathrm{v}_{a_2}, \dots$ *differs from the original control*

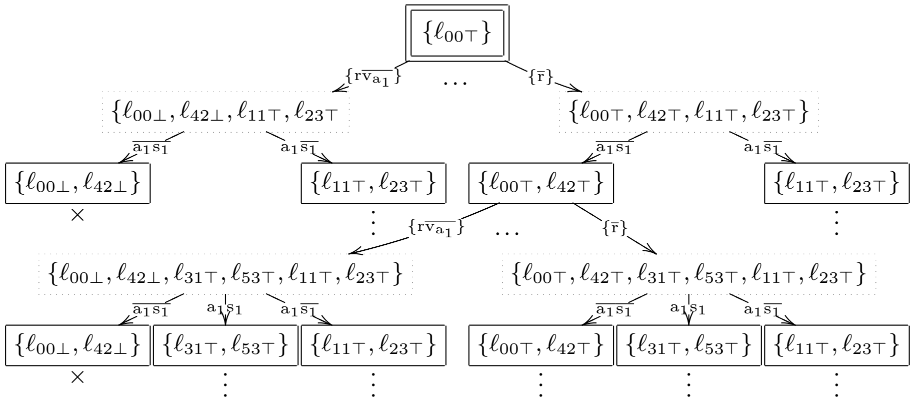

$\{\ell_{00\top}\}$
$\{r\overline{v_{a_1}}\}$
$\{\bar{r}\}$
$\{\ell_{00\bot}, \ell_{42\bot}, \ell_{11\top}, \ell_{23\top}$
$\{\ell_{00\top}, \ell_{42\top}, \ell_{11\top}, \ell_{23\top}\}$
$\overline{a_1 s_1}$
$a_1\overline{s_1}$
$\{\ell_{00\bot}, \ell_{42\bot}\}$
$\{\ell_{11\top}, \ell_{23\top}\}$
$\{\ell_{00\top}, \ell_{42\top}\}$
$\{\ell_{00\bot}, \ell_{42\bot}, \ell_{31\top}, \ell_{53\top}, \ell_{11\top}, \ell_{23\top}\}$
$\{\ell_{00\top}, \ell_{42\top}, \ell_{31\top}, \ell_{53\top}, \ell_{11\top}, \ell_{23\top}\}$
$a_1 s_1$
$\{\ell_{31\top}, \ell_{53\top}\}$

Fig. 6. Game tree for the deny game $A_{\mathrm{stamp}_1} \| A_{\mathrm{context}_1} \| A_{\mathrm{deny}_1}$, over the control signature $[\{a_1, s_1\} \to \{r, v_{a_1}\}]$, here $r\overline{v_{a_1}}$ means *restrict by denying* $\overline{a_1}$, and $\bar{r}$ means *unrestricted.*

outputs $a_1, a_2 \ldots$ as chosen by the plant in its idealized context. In this way the safety player is forced to be conservative in the restriction that she puts since she can only deny some sequence of control outputs whenever she is sure that the idealized context will never allow this sequence.

We may construct the game board for this deny game by taking the composition of the Kripke structures for the plant, the context strategy, and the rule that forces at least one of the deny outputs to be distinct from the control output as chosen by the plant in its idealized context. For stamp_1 this is $A_{\mathrm{stamp}_1} \| A_{\mathrm{context}_1} \| A_{\mathrm{deny}_1}$. A partial unraveling of the resulting game tree over this product game is shown in Figure 6. When we work this out further we quickly see that, in this game, the safety player is always forced to play $\bar{r}$. This means she cannot put any restriction on the control outputs. Which, in turn, means that the resulting control strategy (after projecting out r, and projecting the temporary v_{a_1} back to a_1) will be $f_\top$, i.e.: we cannot put any control constraints using the control signature for this locality.

Note that it is possible to repeat this procedure several times with different sets of control outputs, each time requiring the player to come up with new deny outputs. In this way we can build up the new control constraints incrementally in accordance with the control hierarchy. $\lhd$

Compositional Controller Synthesis Algorithm. We have now established all prerequisites to present Compositional Controller Synthesis Algorithm 1 (COCOS). The algorithm starts with a PuC M that is initially unconstrained, that is: the context assumptions and control constraints for each control locality are vacuous. The algorithm then works by making a single bottom–up pass over the control localities. It will start at the lowest localities (for which $K{\downarrow} = \varnothing$) progressing up to the highest control locality P. For each locality the weakest local context assumptions are computed (simplified using the subordinate control constraints) and subsequently the strongest local control constraints are computed (based on the weakest local control assumptions and the

Algorithm 1. COCOS (Compositional Control Synthesis)

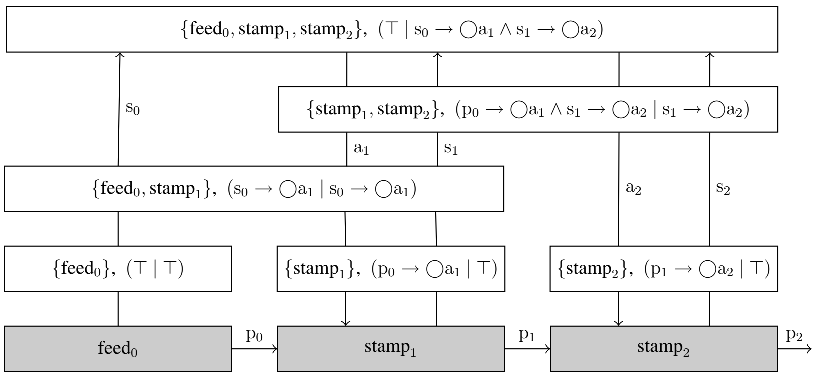
```
Data: A PuC $M = (P, \{f_p, [X^{\mathrm{i}}_p \to X^{\mathrm{o}}_p]\}_{p\in P}, C, \{g_K, h_K\}_{K\in C})$ such that for all
      $K \in C$ it holds $g_k = h_K = f_\top$
Result: The maximally permissive, safe control strategy for the given system of plant
        components.
1 Visited $\leftarrow \varnothing$
2 while $P \notin$ Visited do
3     select some $K \in C$ such that $K \notin$ Visited and $K{\downarrow} \subseteq$ Visited
4     $M \leftarrow \mathsf{StrongestControl}_K(\mathsf{WeakestContext}_K(M))$
5     Visited $\leftarrow$ Visited $\cup \{K\}$
6 return $h_P$
```

subordinate control constraints). The while loop terminates when the highest control locality $P \in C$ has been visited. At this point it holds: $(f_P \sqcap h_P) = (f_P \sqcap \hat{h}_P)$.

Theorem 3 (Correctness). *Algorithm 1 always terminates, and after termination it will hold* $(f_P \sqcap h_P) = (f_P \sqcap \hat{h}_P)$. ◁

Proof. Completeness follows from the fact that there are only a finite number of control localities. Soundness follows from Lemmas 1 and 2, and the fact that for the highest control locality P the signatures of the weakest context assumptions, the strongest control constraints and the global control constraints $\hat{h}_P$ coincide. □

Example 5 (Compositional Controller Synthesis). In Figure 7 we show how COCOS treats the PuC M_{parcel} as defined in Example 1. The plant components are shown as gray boxes connected by horizontal arrows denoting the unchecked plant propositions. The control localities are shown as white boxes connected to the plant components by



$\{\text{feed}_0, \text{stamp}_1, \text{stamp}_2\},\ (\top \mid s_0 \to \bigcirc a_1 \wedge s_1 \to \bigcirc a_2)$
$\{\text{stamp}_1, \text{stamp}_2\},\ (p_0 \to \bigcirc a_1 \wedge s_1 \to \bigcirc a_2 \mid s_1 \to \bigcirc a_2)$
s_0
a_1
s_1
$\{\text{feed}_0, \text{stamp}_1\},\ (s_0 \to \bigcirc a_1 \mid s_0 \to \bigcirc a_1)$
a_2
s_2
$\{\text{feed}_0\},\ (\top \mid \top)$
$\{\text{stamp}_1\},\ (p_0 \to \bigcirc a_1 \mid \top)$
$\{\text{stamp}_2\},\ (p_1 \to \bigcirc a_2 \mid \top)$
feed_0
p_0
stamp_1
p_1
stamp_2
p_2

Fig. 7. A run of COCOS on the PuC M_{parcel} from example 1 each locality is annotated with: (weakest local context assumptions | strongest local control guarantees)

vertical arrows denoting the control propositions. Each control locality is labeled with the weakest local context assumptions and the strongest local control constraints in modal logic notation. Since the algorithm performs a single, bottom–up pass over the control localities this picture represents the entire run.

For locality $\{\mathrm{feed}_0\}$ we get two vacuous strategies. The reason is that the feeder plant component does not contain any deadlocks. As such it needs no assumptions or control constraints to function safely. Control locality $\{\mathrm{stamp}_1\}$ has been treated more extensively as the running example in the previous sections. It requires a single context assumption saying that the arm will activate when there is a parcel queuing in the feeder. As we have seen in the previous example we cannot enforce this assumption on a local level, yet. The situation for $\{\mathrm{stamp}_2\}$ is completely symmetrical. After treating the lower localities COCOS proceeds with the intermediate two localities.

For locality $\{\mathrm{feed}_0, \mathrm{stamp}_1\}$ new context assumptions are computed. This computation cannot yet benefit from subordinate control, since subordinate control is still vacuous. However, we do note that, since the signature of the context assumptions changes non–monotonically, the results will be different this time. In particular the p_0 proposition has become an internal plant proposition which is no longer visible to the context. At the same time the feeder component has added the control input proposition s_0. This means that the weakest context assumption has changed from $\mathrm{p}_0 \rightarrow \bigcirc \mathrm{a}_1$ into $\mathrm{s}_0 \rightarrow \bigcirc \mathrm{a}_1$. Intuitively, by restricting the context signature as much as we can (without sacrificing conservativity) the context assumptions have shifted into the direction of something that can be turned into a control constraint.

For locality $\{\mathrm{stamp}_1, \mathrm{stamp}_2\}$ the situation is almost symmetrical except for the fact that the assumption $\mathrm{p}_0 \rightarrow \bigcirc \mathrm{a}_1$ on stamp_1 still has to be made by the context. Note that, even though the weakest local context assumptions are over approximating the allowable behaviour by assuming plant proposition p_0 to be observable, COCOS is still able to recover the control constraint $\mathrm{s}_1 \rightarrow \bigcirc \mathrm{a}_2$ which has a clear causal dependency on the plant proposition p_0.

Finally, we treat the topmost locality $\{\mathrm{feed}_0, \mathrm{stamp}_1, \mathrm{stamp}_2\} = P$. The weakest context assumptions for this locality are vacuous, since subordinate control already ensures safety. In this case, the strongest control constraints are simply the conjunction of the subordinate control constraints. Note that this is where compositionality really helps us: computing the assumptions for higher control localities becomes much easier in the presence of subordinate control, especially using a counterexample driven algorithm. In this case subordinate control ensures that there are no unsafe transitions anymore at the highest level of composition. $\triangleleft$

4 Conclusion

We have presented a semantical framework for compositional control synthesis problems. Based on the framework we have developed a compositional control synthesis method that is sound and complete for computing most permissive safety controllers on regular models under the assumption of partial observation. The novel aspects of the method are:

1. The signature of the context assumptions changes non–monotonically with increasing scope, converging to the signature of control constraints. In this way we obtain more realistic assumptions as the scope widens.
2. Local context assumptions are simplified with respect to control constraints that were previously derived for lower control localities. In this way, we make it possible to efficiently apply a counterexample driven refinement algorithm for finding the weakest context assumptions.
3. Local control constraints are approximated based on local context assumptions and control constraints that were previously derived for lower control localities. In this way, we enable a synergy between local context assumptions and local control constraints.

Bidirectionality and Control Hierarchy. To simplify exposition in this paper we forbid the situation where a proposition is output by more than one plant component. For some applications, such as circuit synthesis, or hardware protocol analysis [6], it may be desirable to relax this restriction.

As a simple example of bidirectionality consider the plant component over the single proposition x that, at *even* clockcycles, treats x as *output* by writing a random bit $s_{2j}(x) \in \{0,1\}$ to the controller, and, at *odd* clockcycles, treats x as *input* by reading $s_{2j+1}(x) \in \{0,1\}$ back from the controller. Next consider a safety invariant that requires: $s_{2j+1}(x) = s_{2j}(x)$, i.e. the controller must simply echo back the bits written by the component in the previous clockcycle.

In the resulting game, the safety player cannot always choose *concrete* outputs for x at each clockcycle t. Since at even clockcycles $t = 2j$ she cannot predict what the component is going to write as output. Note that, since we consider moves as *allow sets*, this is not a problem in principle: at even clockcycles the safety player may simply allow x to vary freely by allowing x to be either high or low: $h(\epsilon) = h(s_0 \dots s_{2j+1}) = \{x, \overline{x}\}$, and at odd clockcycles the safety player may restrict x depending on what she observed in the previous state: $h(s_0 \dots s_{2j}) = \{x\}$ in case $s_{2j}(x) = 1$ or $h(s_0 \dots s_{2j}) = \{\overline{x}\}$ in case $s_{2j}(x) = 0$.

Note that we do allow bidirectionality on the level of control localities' context assumptions and control constraints. For the control, bidirectionality is important, not so much for reasons of expressivity, but because it allows the synthesis algorithm to achieve a form of *abstraction* for the higher control localities. By abstraction we mean simplifying the context assumptions with respect to the control constraints already imposed by the lower control localities. Intuitively, the synthesized context uses its output propositions in a bidirectional fashion by "listening" to the plant being controlled by the various subcontrollers that are already in place. The higher control locality then only has to impose more constraints in those situations where the constraints imposed by the subcontrollers are not already sufficient to keep the plant safe.

Future Work. We are working to validate COCOS by applying it in a case study for deriving a reactive controller in a setting with limited observability. In order to do this a prototype toolchain is being developed. Implementation of COCOS requires efficient symbolic manipulation of strategies. We have described a way of deriving context assumptions in an efficient counterexample driven manner [13]. However, it would certainly be interesting to compare with various other game solving algorithms [7,5]. The intuition here is that lower levels of the control hierarchy may benefit from forward rather than backward algorithms, and possibly even explicit state rather than symbolic representations since these games are typically much simpler. At higher levels of composition backward symbolic algorithms may be more beneficial since there is much more room for abstraction and potential benefit in terms of alleviating the state explosion problem.

Other directions for future research include applications in design, modelchecking, static analysis, optimization and testing, extension to asynchronous and timed systems, and parallelized versions of COCOS.

References

1. Asarin, E., Bournez, O., Dang, T., Maler, O., Pnueli, A.: Effective synthesis of switching controllers for linear systems. Proceedings of the IEEE 88(7), 1011–1025 (2000)
2. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1994. LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995)
3. Beyer, D., Henzinger, T.A., Singh, V.: Algorithms for interface synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 4–19. Springer, Heidelberg (2007)
4. Buchi, J.R., Landweber, L.H.: Solving sequential conditions by Finite-State strategies. Transactions of the American Mathematical Society 138, 295–311 (1969)
5. Cassez, F.: Efficient On-the-Fly algorithms for partially observable timed games. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 5–24. Springer, Heidelberg (2007)
6. 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, vol. 2404, pp. 414–745. Springer, Heidelberg (2002)
7. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for Omega-Regular games with imperfect information. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 287–302. Springer, Heidelberg (2006)
8. Chatterjee, K., Henzinger, T., Jobstmann, B.: Environment assumptions for synthesis. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 147–161. Springer, Heidelberg (2008)
9. Chatterjee, K., Henzinger, T.A.: Assume-Guarantee synthesis. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 261–275. Springer, Heidelberg (2007)
10. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. In: Summaries of the Summer Institute of Symbolic Logic, Cornell Univ., Ithaca, NY, vol. 1, pp. 3–50 (1957)
11. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using Branching-Time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
12. de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Softw. Eng. Notes 26(5), 109–120 (2001)
13. Kuijper, W., van de Pol, J.: Computing weakest strategies for safety games of imperfect information. In: Kowalewski, S., Phillippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 92–106. Springer, Heidelberg (2009)
14. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006)

15. Kupferman, O., Vardi, M.Y.: Synthesis with incomplete information. In: Advances in Temporal Logic. Applied Logic Series, vol. 16, pp. 109–127. Kluwer, Dordrecht (2000)
16. Lin, F., Wonham, W.M.: Decentralized control and coordination of discrete-event systemswith partial observation. IEEE Transactions on automatic control 35(12), 1330–1337 (1990)
17. Maler, O., Nickovic, D., Pnueli, A.: On synthesizing controllers from Bounded-Response properties. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 95–107. Springer, Heidelberg (2007)
18. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) 6(1), 68–93 (1984)
19. Nam, W., Madhusudan, P., Alur, R.: Automatic symbolic compositional verification by learning assumptions. Form. Methods Syst. Des. 32(3), 207–234 (2008)
20. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Austin, Texas, United States, pp. 179–190. ACM, New York (1989)
21. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989)
22. Rabin, M.O.: Automata on infinite objects and Church's problem. American Mathematical Society (1972)
23. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proceedings of the IEEE 77(1), 81–98 (1989)
24. Thomas, W.: On the synthesis of strategies in infinite games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, p. 12. Springer, Heidelberg (1995)
25. Thomas, W.: Facets of synthesis: Revisiting church's problem. In: de Alfaro, L. (ed.) FOSSACAS 2009. LNCS, vol. 5504, p. 1. Springer, Heidelberg (2009)
26. De Wulf, M., Doyen, L., Raskin, J.-F.: A lattice theory for solving games of imperfect information. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 153–168. Springer, Heidelberg (2006)

Howe's Method for Calculi with Passivation

Sergueï Lenglet[1], Alan Schmitt[2], and Jean-Bernard Stefani[2]

[1] Université Joseph Fourier, Grenoble, France
[2] INRIA Rhône-Alpes, France

Abstract. We show how to apply Howe's method for the proof of congruence of early bisimilarities in a higher-order process calculus with passivation. This involves the introduction of a proof technique based on a new kind of transition system and bisimilarity, called *complementary semantics*. We show that complementary semantics is equivalent to contextual semantics, originally introduced by Sangiorgi, that relies on classical transition systems for higher-order calculi and context bisimilarity. We apply this technique to obtain the first characterization of weak barbed congruence for such a higher-order calculus with passivation.

1 Introduction

Motivation. A natural notion of program equivalence in concurrent languages is a form of contextual equivalence called *barbed congruence*, introduced by Milner and Sangiorgi [17]. Roughly, given an operational semantics defined by means of a small-step reduction relation, two processes are barbed congruent if they have the same reductions and the same observables (or *barbs*), under any context.

The definition of barbed congruence, however, is impractical to use in proofs because of its quantification on contexts. An important question, therefore, is to find more effective characterizations of barbed congruence. A powerful method for proving program equivalence is the use of coinduction with the definition of an appropriate notion of *bisimulation*. The question of characterizing barbed congruence to enable the use of coinduction becomes that of finding appropriate bisimulation relations such that their resulting behavioral equivalences, called *bisimilarities*, are *sound* (i.e., included in barbed congruence) and *complete* (i.e., containing barbed congruence) with respect to barbed congruence.

For first-order languages, such as CCS or the π-calculus, the behavioral theory and the associated proof techniques, e.g., for proving congruence, are well developed [23]. Characterizing barbed congruence in these languages is a reasonably well understood proposition. The situation is less satisfactory for higher-order concurrent languages. Bisimilarity relations that coincide with barbed congruence have only been given for some higher-order concurrent languages. They usually take the form of *context bisimilarities*, building on a notion of *context bisimulation* introduced by Sangiorgi for a higher-order π-calculus, HOπ [21]. Context bisimilarity has been proven to coincide with barbed congruence for higher-order variants of the π-calculus: Sangiorgi's HOπ [20,21,12], a concurrent ML with local names [11], a higher-order distributed π-calculus called SafeDpi

[8], Mobile Ambients [16], and some of Mobile Ambients's variants such as Boxed Ambients [3]. A sound but incomplete form of context bisimilarity has been proposed for the Seal calculus [5]. For the Homer calculus [6], strong context bisimilarity is proven sound and complete, but weak context bisimilarity is not complete. A sound and complete context bisimilarity has been defined for the Kell calculus [24], but for the strong case only.

The key step in proving the soundness of candidate bisimilarities in higher-order concurrent calculi is to show that they are congruences. A systematic technique for proving the congruence of bisimilarity relations is Howe's method [10,1,7]. Unfortunately, Howe's method is originally well suited for bisimulations that are defined in both a *late* and a *delay* style, either of which generally breaks the correspondence with barbed congruence. For Homer, Howe's method has been extended to a version of context bisimulation in an input-early style [6], but the resulting weak bisimilarity is not complete with respect to weak barbed congruence.

Contributions. In this paper, we show how to apply Howe's method to deal with bisimulations defined in an early (and non-delay) style. This involves the introduction of *complementary* semantics, a labelled transition system and its associated bisimulation. This semantics is designed to avoid the key difficulty in applying Howe's method to a bisimulation defined in an early style. We use complementary semantics as a proof technique to obtain a characterization of weak barbed congruence in a concurrent calculus called HOπP. HOπP is a calculus introduced in [15] to study the behavioral theory of *passivation* in a simpler setting than in Homer [9] or the Kell calculus [24]. Passivation allows a named locality to be stopped and its state captured for further handling. It can be used to model process failures, strong process mobility, and "thread thunkification" as in the Acute programming language [25] (see [24] for discussion and motivation). To our knowledge, this is the first characterization of weak barbed congruence for a concurrent calculus with both restriction and passivation.

Outline. In Section 2, we present the syntax, contextual semantics, and known bisimilarity results for HOπP. In Section 3, we explain why Howe's method fails with early context bisimilarities, and present the intuition behind our approach. We propose in Section 4 a new semantics and associated bisimilarities for HOπP, called *complementary semantics*. We prove that the semantics are equivalent, that complementary bisimilarity coincides with early context bisimilarity, and that complementary bisimilarity is a congruence using Howe's method. We discuss related work in Section 5. Section 6 concludes the paper. Proofs and additional details are available in the draft of the full paper [14].

2 HOπP Contextual Semantics

HOπP (Higher-Order π with Passivation) [15] extends the higher-order calculus HOπ [21] with localities $a[P]$, that are passivation units. We write names $a, b \ldots$,

Syntax: $P ::= \mathbf{0} \;|\; X \;|\; P \mid P \;|\; a(X)P \;|\; \overline{a}\langle P\rangle P \;|\; \nu a.P \;|\; a[P]$

Agents:

Processes	P, Q, R, S	
Abstractions	F, G	$::= (X)P$
Concretions	C, D	$::= \langle P\rangle Q \;|\; \nu a.C$
Agents	A, B	$::= P \;|\; F \;|\; C$

Extension of operators to all agents

$$(X)Q \mid P \triangleq (X)(Q \mid P)$$
$$P \mid (X)Q \triangleq (X)(P \mid Q)$$
$$a[(X)Q] \triangleq (X)a[Q]$$
$$\nu a.(X)Q \triangleq (X)\nu a.P$$

$$(\nu\widetilde{b}.\langle Q\rangle R) \mid P \triangleq \nu\widetilde{b}.\langle Q\rangle(R \mid P)$$
$$P \mid (\nu\widetilde{b}.\langle Q\rangle R) \triangleq \nu\widetilde{b}.\langle Q\rangle(P \mid R)$$
$$a[\nu\widetilde{b}.\langle Q\rangle R] \triangleq \nu\widetilde{b}.\langle Q\rangle a[R]$$
$$\nu a.(\nu\widetilde{b}.\langle Q\rangle R) \triangleq \nu\widetilde{b}, a.\langle Q\rangle R \text{ if } a \in \mathrm{fn}(Q)$$
$$\nu a.(\nu\widetilde{b}.\langle Q\rangle R) \triangleq \nu\widetilde{b}.\langle Q\rangle \nu a.R \text{ if } a \notin \mathrm{fn}(Q)$$

Pseudo-application and process application

$$(X)P \bullet \nu\widetilde{b}.\langle R\rangle Q \triangleq \nu\widetilde{b}.(P\{R/X\} \mid Q) \qquad (X)P \circ Q \triangleq P\{Q/X\}$$

LTS rules

$$a(X)P \xrightarrow{a} (X)P \;\textsc{Abstr} \qquad \overline{a}\langle Q\rangle P \xrightarrow{\overline{a}} \langle Q\rangle P \;\textsc{Concr} \qquad a[P] \xrightarrow{\overline{a}} \langle P\rangle \mathbf{0} \;\textsc{Passiv}$$

$$\frac{P \xrightarrow{\alpha} A}{a[P] \xrightarrow{\alpha} a[A]}\;\textsc{Loc} \qquad \frac{P \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} A \mid Q}\;\textsc{Par} \qquad \frac{P \xrightarrow{\alpha} A \quad \alpha \notin \{a, \overline{a}\}}{\nu a.P \xrightarrow{\alpha} \nu a.A}\;\textsc{Restr}$$

$$\frac{P \xrightarrow{a} F \quad Q \xrightarrow{\overline{a}} C}{P \mid Q \xrightarrow{\tau} F \bullet C}\;\text{HO}$$

Fig. 1. Contextual Labeled Transition System for HOπP

conames $\overline{a}, \overline{b} \ldots$, and process variables $X, Y, \ldots$. Sets $\{x_1 \ldots x_n\}$ are written $\widetilde{x}$. We let γ range over names and conames.

Convention. We identify processes up to α-conversion of names and variables: processes and agents are always chosen such that their bound names and variables are distinct from free names and variables. We also assume that bound names and variables of any process or actions under consideration are chosen to be different from free names and variables of any other entities under consideration. Note that with this convention, we have $(X)Q \mid P \triangleq (X)(Q \mid P)$ without qualification on the free variables of P.

Syntax and contextual semantics of the calculus can be found in Figure 1, with the exception of the symmetric rules for Par and HO. In process $a(X)P$, the variable X is bound. Similarly, in process $\nu a.P$, the name a is bound. We write $\mathrm{fv}(P)$ for the free variables of a process P, $\mathrm{fn}(P)$ for its free names, and $\mathrm{bn}(P)$ for its bound names. We write $P\{Q/X\}$ for the capture-free substitution of X by Q in P. A *closed process* has no free variable.

Processes may evolve towards a process (internal actions $P \xrightarrow{\tau} P'$), an *abstraction* (message input $P \xrightarrow{a} F = (X)Q$), or a *concretion* (message output $P \xrightarrow{\overline{a}} C = \nu\widetilde{b}.\langle R\rangle Q$). Transition $P \xrightarrow{a} (X)Q$ means that P may receive process R on a to continue as $Q\{R/X\}$. Transition $P \xrightarrow{\overline{a}} \nu\widetilde{b}.\langle R\rangle Q$ means that P may send process R on a and continue as Q, and the scope of names $\widetilde{b}$ has to be expanded to encompass the recipient of R. A synchronous higher-order communication takes place when a concretion interacts with an abstraction (rule HO).

A locality $a[P]$ is a transparent evaluation context: process P may evolve by itself and communicate freely with processes outside of locality a (rule Loc). At any time, passivation may be triggered and locality $a[P]$ becomes a concretion $\langle P\rangle\mathbf{0}$ (rule Passiv). Rule Loc implies that the scope of restricted names may cross locality boundaries. Scope extrusion outside localities is performed "by need" when a communication takes place, as defined in the extension of restriction to concretions in Fig. 1. Note that with this semantics, the interaction between passivation and restriction is not benign: in general processes $b[\nu a.P]$ and $\nu a.b[P]$ are not barbed congruent (see [14] for more details).

Remark 1. In HOπP, process $a(X)P$ is used for message input and process passivation, while Homer and the Kell calculus use two different receiving patterns. We chose to keep HOπP syntax as simple as possible; adding a specific input for passivation does not change our results.

Contextual Equivalences

Barbed congruence is the usual reduction-based behavioral equivalence. We identify reduction with τ-transition $\longrightarrow \overset{\Delta}{=} \xrightarrow{\tau}$ and define weak reduction $\Longrightarrow$ as the reflexive and transitive closure of $\longrightarrow$. Observables γ of a process P, written $P\downarrow_\gamma$, are unrestricted names or conames on which a communication may immediately occur ($P \xrightarrow{\gamma} A$, for some agent A). Contexts $\mathbb{C}$ are terms with a hole $\square$. A relation $\mathcal{R}$ is a *congruence* iff $P \mathrel{\mathcal{R}} Q$ implies $\mathbb{C}\{P\} \mathrel{\mathcal{R}} \mathbb{C}\{Q\}$ for all contexts $\mathbb{C}$.

Definition 1. *A symmetric relation $\mathcal{R}$ on closed processes is a strong (resp. weak) barbed bisimulation iff $P \mathrel{\mathcal{R}} Q$ implies:*

– *for all $P\downarrow_\gamma$, we have $Q\downarrow_\gamma$ (resp. $Q \Longrightarrow\downarrow_\gamma$);*
– *for all $P \longrightarrow P'$, there exists Q' such that $Q \longrightarrow Q'$ (resp. $Q \Longrightarrow Q'$) and $P' \mathrel{\mathcal{R}} Q'$;*

Strong (resp. weak) barbed congruence $\sim_b$ (resp. $\approx_b$) is the largest congruence that is a strong (resp. weak) barbed bisimulation.

A relation $\mathcal{R}$ is *sound* with respect to $\sim_b$ (resp. $\approx_b$) iff $\mathcal{R}\subseteq\sim_b$ (resp. $\mathcal{R}\subseteq\approx_b$); $\mathcal{R}$ is *complete* with respect to $\sim_b$ (resp. $\approx_b$) iff $\sim_b\subseteq\mathcal{R}$ (resp. $\approx_b\subseteq\mathcal{R}$).

As in HOπ [21], we characterize strong barbed congruence using an early strong context bisimilarity. As explained in [15], bisimilarities in HOπP require more discriminating power than in HOπ, as the passivation of enclosing localities has to be taken into account. Let *bisimulation contexts* $\mathbb{E}$ be evaluation contexts, i.e., contexts that allow transitions at the hole position, used for observational purposes.

$$\mathbb{E} ::= \square \mid \nu a.\mathbb{E} \mid \mathbb{E} \mid P \mid P \mid \mathbb{E} \mid a[\mathbb{E}]$$

We write bn($\mathbb{E}$) the names bound at the hole position by the context $\mathbb{E}$: a name $a \in \mathrm{bn}(\mathbb{E}) \cap \mathrm{fn}(P)$ is free in P and becomes bound in $\mathbb{E}\{P\}$.

Definition 2. *Early strong context bisimilarity $\sim$ is the largest symmetric relation on closed processes $\mathcal{R}$ such that $P \mathrel{\mathcal{R}} Q$ implies $fn(P) = fn(Q)$ and:*

– *for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \mathrel{\mathcal{R}} Q'$;*
– *for all $P \xrightarrow{a} F$, for all C, there exists F' such that $Q \xrightarrow{a} F'$ and $F \bullet C \mathrel{\mathcal{R}} F' \bullet C$;*
– *for all $P \xrightarrow{\overline{a}} C$, for all F, there exists C' such that $Q \xrightarrow{\overline{a}} C'$ and for all $\mathbb{E}$, $F \bullet \mathbb{E}\{C\} \mathrel{\mathcal{R}} F \bullet \mathbb{E}\{C'\}$.*

Note. The late variant of strong context bisimulation can simply be obtained by changing the order of quantifications on concretions and abstractions in the above clauses. Thus the clause for input in late style would be: *for all $P \xrightarrow{a} F$, there exists F' such that $Q \xrightarrow{a} F'$ and for all C, we have $F \bullet C \mathrel{\mathcal{R}} F' \bullet C$.*

Condition $\mathrm{fn}(P) = \mathrm{fn}(Q)$ is required because of lazy scope extrusion: two bisimilar processes with different free names may be distinguished by scope extrusion and passivation [15,13].

Theorem 1. *Relation $\sim$ is a congruence and $\sim = \sim_b$.*

The proof of congruence in Theorem 1 is the same as in the Kell calculus [24] (see [14] for details). Unfortunately, this technique fails with weak relations. We write $\overset{\tau}{\Rightarrow}$ for the reflexive and transitive closure of $\xrightarrow{\tau}$. For every higher-order name or coname γ, we write $\overset{\gamma}{\Rightarrow}$ for $\Rightarrow\xrightarrow{\gamma}$. As higher order steps result in concretions and abstractions, they may not reduce further; silent steps after this reduction are taken into account in the definition of weak bisimulation.

Definition 3. *Early weak context bisimilarity $\approx$ is the largest symmetric relation on closed processes $\mathcal{R}$ such that $P \mathrel{\mathcal{R}} Q$ implies $fn(P) = fn(Q)$ and:*

– *for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \overset{\tau}{\Rightarrow} Q'$ and $P' \mathrel{\mathcal{R}} Q'$;*
– *for all $P \xrightarrow{a} F$, for all C, there exist F', Q' such that $Q \overset{a}{\Rightarrow} F'$, $F' \bullet C \overset{\tau}{\Rightarrow} Q'$, and $F \bullet C \mathrel{\mathcal{R}} Q'$;*
– *for all $P \xrightarrow{\overline{a}} C$, for all F, there exists C' such that $Q \overset{\overline{a}}{\Rightarrow} C'$ and for all $\mathbb{E}$, there exists Q' such that $F \bullet \mathbb{E}\{C'\} \overset{\tau}{\Rightarrow} Q'$ and $F \bullet \mathbb{E}\{C\} \mathrel{\mathcal{R}} Q'$.*

Barbed congruence and context bisimilarities are extended to open processes via the notion of open extension $\mathcal{R}^\circ$ of a relation $\mathcal{R}$ on closed processes.

Definition 4. *Let P and Q be two open processes. We have $P \mathrel{\mathcal{R}^\circ} Q$ iff $P\sigma \mathrel{\mathcal{R}} Q\sigma$ for all process substitutions σ that close P and Q.*

In the following section, we present a congruence proof technique, called Howe's method, and then explain why it fails with early context bisimilarities.

3 Howe's Method

The essence of the method. Howe's method [10,1,7] is a systematic proof technique to show that a bisimilarity $\mathcal{R}$ (and its open extension $\mathcal{R}^\circ$) is a congruence. The method can be divided in three steps: first, prove some basic properties on the *Howe's closure* $\mathcal{R}^\bullet$ of the relation. By construction, $\mathcal{R}^\bullet$ contains $\mathcal{R}^\circ$ and is a congruence. Second, prove a simulation-like property for $\mathcal{R}^\bullet$. Finally, prove that $\mathcal{R}$ and $\mathcal{R}^\bullet$ coincide on closed processes. Since $\mathcal{R}^\bullet$ is a congruence, conclude that $\mathcal{R}$ is a congruence.

Howe's closure is inductively defined as the smallest congruence which contains $\mathcal{R}^\circ$ and is closed under right composition with $\mathcal{R}^\circ$.

Definition 5. *Howe's closure $\mathcal{R}^\bullet$ of a relation $\mathcal{R}$ is the smallest relation verifying:*

- $\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$*;*
- $\mathcal{R}^\bullet \mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$*;*
- *for all operators op of the language, if $\widetilde{P}\ \mathcal{R}^\bullet\ \widetilde{Q}$, then $op(\widetilde{P})\ \mathcal{R}^\bullet\ op(\widetilde{Q})$.*

By definition, $\mathcal{R}^\bullet$ is a congruence, and the composition with $\mathcal{R}^\circ$ allows some transitivity and gives additional properties to the relation.

Remark 2. In the literature (e.g., [10,7,9]) Howe's closure is usually inductively defined by the following rule for all operators *op* in the language:

$$\frac{\widetilde{P}\ \mathcal{R}^\bullet\ \widetilde{R} \qquad op(\widetilde{R})\ \mathcal{R}^\circ\ Q}{op(\widetilde{P})\ \mathcal{R}^\bullet\ Q}$$

Both definitions are equivalent (see [7] for the proof). We believe that Definition 5 is easier to understand and to work with in the proofs.

By definition, we have $\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$. To have the reverse inclusion, we prove that $\mathcal{R}^\bullet$ is a bisimulation. To this end, we need the following classical properties of Howe's closure of any equivalence $\mathcal{R}$.

Lemma 1. *Let $\mathcal{R}$ be an equivalence.*

- *If $P\ \mathcal{R}^\bullet\ Q$ and $R\ \mathcal{R}^\bullet\ S$, then we have $P\{R/X\}\ \mathcal{R}^\bullet\ Q\{S/X\}$.*
- *The reflexive and transitive closure $(\mathcal{R}^\bullet)^*$ of $\mathcal{R}^\bullet$ is symmetric.*

The first property is typically used to establish the simulation-like result (second step of the method). Then one proves that the restriction of $(\mathcal{R}^\bullet)^*$ to closed terms is a bisimulation. Consequently we have $\mathcal{R} \subseteq \mathcal{R}^\bullet \subseteq (\mathcal{R}^\bullet)^* \subseteq \mathcal{R}$ on closed terms, and we conclude that bisimilarity $\mathcal{R}$ is a congruence.

The main difficulty of the method lies in the proof of the simulation-like property for Howe's closure. In the following, we explain why we cannot directly use Howe's method with early context bisimilarity (Definition 2).

Communication problem. To prove that $\mathcal{R}^\bullet$ is a simulation, we need to establish a stronger result, to avoid transitivity issues which otherwise would arise and make the method fail in the weak case [14]. Given a bisimilarity $\mathcal{R}$ based on a LTS $P \xrightarrow{\lambda} A$, the simulation-like result follows the pattern below, similar to a higher-order bisimilarity clause, such as the one for Plain CHOCS [26].

Let $P \mathcal{R}^\bullet Q$. If $P \xrightarrow{\lambda} A$, then for all $\lambda \mathcal{R}^\bullet \lambda'$, there exists B such that $Q \xrightarrow{\lambda'} B$ and $A \mathcal{R}^\bullet B$.

Early bisimulations are those where all the information about a step on one side is known before providing a matching step. In the higher-order setting with concretions and abstractions, it means that when an output occurs, the abstraction that will consume the output is specified before the matching step is given. In fact, the matching step may very well be different for a given output when the abstraction considered is different. Symmetrically, in the case of an input, the matching step is chosen depending on the input and the actual concretion that is provided. In both cases, this amounts to putting the abstraction in the label in the case of an output, and the concretion in the label in case of an input. One is thus lead to prove the following simulation property.

Conjecture 1. If $P \mathcal{R}^\bullet Q$, then:

- for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \mathcal{R}^\bullet Q'$;
- for all $P \xrightarrow{a} F$, for all $C \mathcal{R}^\bullet C'$, there exists F' such that $Q \xrightarrow{a} F'$ and $F \bullet C \mathcal{R}^\bullet F' \bullet C'$;
- for all $P \xrightarrow{\overline{a}} C$, for all $F \mathcal{R}^\bullet F'$ there exists C' such that $Q \xrightarrow{\overline{a}} C'$ and for all $\mathbb{E}$, we have $F \bullet \mathbb{E}\{C\} \mathcal{R}^\bullet F' \bullet \mathbb{E}\{C'\}$.

These clauses raise several issues. First, we have to find extensions of Howe's closure to abstractions and concretions which fit an early style. Even assuming such extensions, there are issues in the inductive proof of conjecture 1 with higher-order communication. The reasoning is by induction on $P \mathcal{R}^\bullet Q$. Suppose we are in the parallel case, i.e., we have $P = P_1 \mid P_2$ and $Q = Q_1 \mid Q_2$, with $P_1 \mathcal{R}^\bullet Q_1$ and $P_2 \mathcal{R}^\bullet Q_2$. Suppose that we have $P \xrightarrow{\tau} P'$, and the transition comes from rule HO: we have $P_1 \xrightarrow{a} F$, $P_2 \xrightarrow{\overline{a}} C$ and $P' = F \bullet C$. We want to find Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \mathcal{R}^\bullet Q'$. We also want to use the same rule HO, hence we have to find F', C' such that $Q \xrightarrow{\tau} F' \bullet C'$. However we cannot use the input clause of the induction hypothesis with P_1, Q_1: to have a F' such that $Q_1 \xrightarrow{a} F'$, we have to find first a concretion C' such that $C \mathcal{R}^\bullet C'$. We cannot use the output clause with P_2, Q_2 either: to have a C' such that $Q_2 \xrightarrow{\overline{a}} C'$, we have to find first an abstraction F' such that $F \mathcal{R}^\bullet F'$. We cannot bypass this mutual dependency and the inductive proof of conjecture 1 fails.

Remark 3. Note that the reasoning depends more on the bisimilarity than on the calculus: the same problem occurs with early context bisimilarities for HOπ, Homer, and the Kell calculus.

The intuition behind our approach. A simple way to break the mutual dependency between concretions and abstractions is to give up on the early style. An approach, used in [6], is to change the output case to a late style (hence the name, *input-early*, of their bisimulation): an output is matched by another output *independently of the abstraction that receives it*. This breaks the symmetry and allows us to proceed forward: first find the matching output C', then for this C' find the matching input using the input-early relation $\sim_{ie}$. Howe's closure is then extended to concretions $C \sim_{ie}^{\bullet} C'$ and a simulation-like property similar to Conjecture 1 is shown, except that the output clause is changed into:

– for all $P \xrightarrow{\overline{a}} C$, there exists C' such that $Q \xrightarrow{\overline{a}} C'$ and $C \sim_{ie}^{\bullet} C'$.

However, in the weak case, this input-early approach does not result in a sound and complete characterization of weak barbed congruence. Definition of weak input-early bisimilarity has to be written in the *delay* style: internal actions are not allowed after a visible action. The delay style is necessary to keep the concretion clause independent from abstractions. It is not satisfactory since delay bisimilarities are generally not complete with respect to weak barbed congruence.

We thus propose a different approach, detailed in Section 4, that works with weak bisimulations defined in the early non-delay style. In our solution, the output clause is not late, just a little less early. More precisely, instead of requiring the abstraction before providing a matching output, we only require the process that will do the reception (that will reduce to the abstraction). This may seem a very small change, yet it is sufficient to break the symmetry. We return to the communication problem where $P_1 \mid P_2$ is in relation with $Q_1 \mid Q_2$. The concretion C' from Q_2 matching the $P_2 \xrightarrow{\overline{a}} C$ step depends only on P_1, which is known, and not on some unknown abstraction. We can then obtain the abstraction F' from Q_2 that matches the $P_1 \xrightarrow{a} F$ step. This abstraction depends fully on C', in the usual early style. Technically, we do not use concretions and abstractions anymore. In the LTS, when a communication between P and Q occurs, this becomes a transition from P using Q as a label (rule HO_τ in Fig. 2). Higher in the derivation, the actual output from P is discovered, and we switch to dealing with the input knowing exactly the output (rule $\textsc{Out}_o$ in Fig. 3). The proof of the bisimulation property for the candidate relation relies on this serialization of the LTS, which illustrates the break in the symmetry. On the other hand, the gap between a completely early relation and this one is small enough to let us prove that they actually coincide.

4 Complementary Semantics for HOπP

We define a new semantics for HOπP that coincides with the contextual one, yet allows the use of Howe's method to prove the soundness of early bisimilarities.

4.1 Complementary LTS

We define a LTS $P \overset{\lambda}{\longmapsto} P'$ where processes always evolve to other processes. We have three kinds of transitions: internal actions $P \overset{\tau}{\longmapsto} P'$, message input $P \overset{a,R}{\longmapsto}$

$$a(X)P \xmapsto{a,R} P\{R/X\} \ \textsc{In}_i \qquad \frac{P \xmapsto{\mu} P'}{P \mid Q \xmapsto{\mu} P' \mid Q}\ \textsc{Par}_{i\tau} \qquad \frac{P \xmapsto{\mu} P'}{a[P] \xmapsto{\mu} a[P']}\ \textsc{Loc}_{i\tau}$$

$$\frac{P \xmapsto{\mu} P' \qquad a \neq n(\mu)}{\nu a.P \xmapsto{\mu} \nu a.P'}\ \textsc{Restr}_{i\tau} \qquad \frac{P \xmapsto{\overline{a},Q,\square}_{\widetilde{b}} P'}{P \mid Q \xmapsto{\tau} P'}\ \mathrm{HO}_{\tau}$$

Fig. 2. Complementary LTS for HOπP: Internal and Message Input Actions

P', and message output $P \xmapsto{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$. We call this new LTS *complementary* since labels λ contain contexts complementing and interacting with P for observational purposes. They are used to establish bisimilarity.

Rules for internal actions $P \xmapsto{\tau} P'$ are similar to the ones for the contextual LTS $P \xrightarrow{\tau} P'$, except for higher-order communication since message output is different; we detail rule HO$_\tau$ later. Message input $P \xmapsto{a,R} P'$ means that process P may receive the process R as a message on a and becomes P'. In the contextual style, it means that $P \xrightarrow{a} F$ and $P' = F \circ R$ for some F. Complementary message input is simply contextual input written in the early style. We let $\xmapsto{\mu}$ range over $\xmapsto{\tau}$ and $\xmapsto{a,R}$. For higher-order input, we define $n(a, R) = a$. Rules can be found in Figure 2 except for the symmetric counterparts of rules Par$_{i\tau}$ and HO$_\tau$.

We now detail output actions $P \xmapsto{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$. Rules can be found in Fig. 3, except for the symmetric of rule Par$_o$. Context bisimilarity (Definition 2) compares message outputs by making them react with an abstraction F and a context $\mathbb{E}$. In complementary semantics, we consider a *receiving process* Q instead of F, i.e., a process able to receive the message emitted by P on a. Transition $P \xmapsto{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$ means that P is put under context $\mathbb{E}$ and emits a message on a, which is received by Q: we have $\mathbb{E}\{P\} \mid Q \xmapsto{\tau} P'$ by communication on a. In the contextual style, it means that there exist F, C such that $P \xrightarrow{\overline{a}} C$, $Q \xrightarrow{a} F$, and $P' = F \bullet \mathbb{E}\{C\}$. It is not however a simple rewrite of contextual transitions in an early style as the abstraction F is not fixed by the rule. Consider rule Out$_o$ for message output. Unlike contextual rule Concr, it needs a premise $Q \xmapsto{a,R} Q'$, to check that Q is able to receive on a the emitted process R. The resulting process Q' is then run in parallel with the continuation S under context $\mathbb{E}$. By hypothesis of rule Out$_o$, the context does not capture any free name of R. We explain below why we choose to first deal with capture-free contexts. For such contexts, we introduce the transition $\overline{a}\langle R\rangle S \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} Q' \mid \mathbb{E}\{S\}$. In the contextual semantics, we have $\overline{a}\langle R\rangle S \xrightarrow{\overline{a}} \langle R\rangle S$, and context bisimilarity tests $F \bullet \mathbb{E}\{\langle R\rangle S\}$ in the output clause, which is equal to $F \circ R \mid \mathbb{E}\{S\}$ for capture-free contexts. Since Q' may be rewritten as $F \circ R$ for some F, the complementary transition mimics exactly the context bisimilarity output clause.

Note that the message no longer appears in the label of output transitions. We thus need additional information to deal with scope extrusion in the rules

$$\frac{\mathrm{fn}(R)=\widetilde{b} \qquad Q \xmapsto{a,R} Q' \qquad \mathrm{bn}(\mathbb{E})\cap\widetilde{b}=\emptyset}{\overline{a}\langle R\rangle S \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} Q' \mid \mathbb{E}\{S\}}\ \textsc{Out}_o$$

$$\frac{\mathrm{fn}(P)=\widetilde{b} \qquad Q \xmapsto{b,P} Q' \qquad \mathrm{bn}(\mathbb{E})\cap\widetilde{b}=\emptyset}{b[P] \xhookrightarrow{\overline{b},Q,\mathbb{E}}_{\widetilde{b}} Q' \mid \mathbb{E}\{\mathbf{0}\}}\ \textsc{Passiv}_o \qquad \frac{P \xhookrightarrow{\overline{a},Q,\mathbb{E}\{b[\square]\}}_{\widetilde{b}} P'}{b[P] \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'}\ \textsc{Loc}_o$$

$$\frac{P_1 \xhookrightarrow{\overline{a},Q,\mathbb{E}\{\square\mid P_2\}}_{\widetilde{b}} P'}{P_1 \mid P_2 \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'}\ \textsc{Par}_o \qquad \frac{P \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P' \qquad c\neq a \qquad c\in\widetilde{b}}{\nu c.P \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}\backslash c} \nu c.P'}\ \textsc{Extr}_o$$

$$\frac{P \xhookrightarrow{\overline{a},Q,\mathbb{E}\{\nu c.\square\}}_{\widetilde{b}} P' \qquad c\neq a \qquad c\notin\widetilde{b}}{\nu c.P \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'}\ \textsc{Restr}_o \qquad \frac{P \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'}{P \xmapsto{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'}\ \textsc{CFree}_o$$

$$\frac{P \xmapsto{\overline{a},Q,\mathbb{E}\{\mathbb{F}\}}_{\widetilde{b}} P' \qquad c\in\widetilde{b}}{P \xmapsto{\overline{a},Q,\mathbb{E}\{\nu c.\mathbb{F}\}}_{\widetilde{b}} \nu c.P'}\ \textsc{Capt}_o$$

Fig. 3. Complementary LTS for HOπP: Message Output Actions

for name restriction. To this end, rule $\textsc{Out}_o$ stores the free names $\widetilde{b}$ of R in the label. Scope extrusion may happen in the process under consideration (e.g., $P=\nu c.\overline{a}\langle R\rangle S$ with $c\in\mathrm{fn}(R)$) or because of the bisimulation context $\mathbb{E}$ (e.g., $P=\overline{a}\langle R\rangle S$ and $\mathbb{E}=d[\nu c.(\square\mid\overline{c}\langle\mathbf{0}\rangle\mathbf{0})]$ with $c\in\mathrm{fn}(R)$). Notice that premise $\mathrm{fn}(\mathbb{E})\cap\widetilde{b}=\emptyset$ in rule $\textsc{Out}_o$ forbids the latter kind of capture. We thus first define auxiliary transitions $P \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$ where $\mathbb{E}$ is a capture-free context, and we then give rules for all contexts $P \xmapsto{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$.

We now explain the restriction rules on an example; let $P=\overline{a}\langle R\rangle S$ and $c\in\mathrm{fn}(R)$. Process $\nu c.P$ may emit R on a, but the scope of c has to be expanded to encompass the recipient of R. First premise of rule $\textsc{Extr}_o$ checks that P may output a message; here we have $\overline{a}\langle R\rangle S \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} \mathbb{E}\{S\}\mid Q'$ with $\widetilde{b}=\mathrm{fn}(R)$. In conclusion, we have $\nu c.\overline{a}\langle R\rangle S \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}\backslash c} \nu c.(\mathbb{E}\{S\}\mid Q')$. Scope of c includes Q' as expected. For a concretion $C=\nu\widetilde{a}.\langle P_1\rangle P_2$, the names $\widetilde{b_C}$ that may be extruded are the free names of P_1 which are not already bound in $\widetilde{a}$, i.e., $\widetilde{b_C}=\mathrm{fn}(P_1)\setminus\widetilde{a}$.

Suppose now that $P=\overline{a}\langle R\rangle S$ with $c\notin\mathrm{fn}(R)$. Process $\nu c.P$ may emit a message, but the scope of c has to encompass only the continuation S: we want to obtain $\nu c.P \xhookrightarrow{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} \mathbb{E}\{\nu c.S\}\mid Q'$. To this end, we consider $P \xhookrightarrow{\overline{a},Q,\mathbb{E}\{\nu c.\square\}}_{\widetilde{b}} P'$ as a premise of rule $\textsc{Restr}_o$. In process P', the continuation is put under $\mathbb{E}\{\nu c.\square\}$, hence we obtain $\overline{a}\langle R\rangle S \xhookrightarrow{\overline{a},Q,\mathbb{E}\{\nu c.\square\}}_{\widetilde{b}} \mathbb{E}\{\nu c.S\}\mid Q'=P'$, as expected and reflected in the conclusion of the rule.

Rule for passivation $\textsc{Passiv}_o$ is similar to rule $\textsc{Out}_o$, while rules $\textsc{Loc}_o$ and $\textsc{Par}_o$ follow the same pattern as rule $\textsc{Restr}_o$. Rule $\textsc{CFree}_o$ simply means that a transition with a capture-free context is a message output transition. We now explain how to deal with context capture with rule $\textsc{Capt}_o$. Suppose $P = \overline{a}\langle R\rangle S$ and $\mathbb{E}' = d[\nu c.(\square \mid \overline{c}\langle \mathbf{0}\rangle\mathbf{0})]$ with $c \in \text{fn}(R)$; we want to obtain $P \xmapsto{\overline{a},Q,\mathbb{E}'}_{\widetilde{b}} \nu c.(d[S \mid \overline{c}\langle\mathbf{0}\rangle\mathbf{0}] \mid Q')$ (with the scope of c extended out of d). We first consider the transition $P \xmapsto{\overline{a},Q,\mathbb{E}\{\mathbb{F}\}}_{\widetilde{b}} P'$ without capture on c; in our case we have $P \xmapsto{\overline{a},Q,d[\square|\overline{c}\langle\mathbf{0}\rangle\mathbf{0}]}_{\widetilde{b}} d[S \mid \overline{c}\langle\mathbf{0}\rangle\mathbf{0}] \mid Q' = P'$ with $\mathbb{E} = d[\square]$ and $\mathbb{F} = \square \mid \overline{c}\langle\mathbf{0}\rangle\mathbf{0}$. According to the rule, we have $P \xmapsto{\overline{a},Q,\mathbb{E}\{\nu c.\mathbb{F}\}}_{\widetilde{b}} \nu c.P'$, that is $P \xmapsto{\overline{a},Q,\mathbb{E}'}_{\widetilde{b}} \nu c.(d[S \mid \overline{c}\langle\mathbf{0}\rangle\mathbf{0}] \mid Q')$. The scope of c is extended outside $\mathbb{E}$ and includes the recipient of the message, as required.

Premise $P \xmapsto{\overline{a},Q,\square}_{\widetilde{b}} P'$ of rule HO_τ (Figure 2) means that process P sends a message on a to Q without any context around P, and the result is P'. Consequently we have $P \mid Q \xmapsto{\tau} P'$ by communication on a, which is the expected conclusion. Names $\widetilde{b}$ may no longer be potentially extruded, so we simply forget them.

4.2 Complementary Bisimilarity

We now define complementary bisimilarity and prove its soundness using Howe's method. Strong complementary bisimilarity for HOπP is simply the classic bisimilarity associated to the complementary LTS with an additional condition on free names.

Definition 6. *Strong complementary bisimilarity $\sim_m$ is the largest symmetric relation $\mathcal{R}$ such that $P \mathcal{R} Q$ implies fn(P) = fn(Q) and for all $P \xmapsto{\lambda} P'$, there exists $Q \xmapsto{\lambda} Q'$ such that $P' \mathcal{R} Q'$.*

We define $\mathbb{E} \sim_m^\bullet \mathbb{E}'$ as the smallest congruence that extends $\sim_m^\bullet$ with $\square \sim_m^\bullet \square$. We also extend $\sim_m^\bullet$ to labels λ in the following way: we have $\lambda \sim_m^\bullet \lambda'$ iff $\lambda = \lambda' = \tau$, or $\lambda = (a, R)$, $\lambda' = (a, R')$ with $R \sim_m^\bullet R'$, or $\lambda = (\overline{a}, R, \mathbb{E}, \widetilde{b})$, $\lambda' = (\overline{a}, R', \mathbb{E}', \widetilde{b})$ with $R \sim_m^\bullet R'$ and $\mathbb{E} \sim_m^\bullet \mathbb{E}'$. We prove the following simulation-like property for $\sim_m^\bullet$:

Lemma 2. *Let P, Q be closed processes. If $P \sim_m^\bullet Q$ and $P \xmapsto{\lambda} Q$, then for all $\lambda \sim_m^\bullet \lambda'$, there exists Q' such that $Q \xmapsto{\lambda'} Q'$ and $P' \sim_m^\bullet Q'$.*

The higher-order communication problem of Section 3 is avoided. We recall that in this case, we have $P_1 \mid P_2 \sim_m^\bullet Q_1 \mid Q_2$ with $P_1 \sim_m^\bullet Q_1$, $P_2 \sim_m^\bullet Q_2$ and $P_1 \xmapsto{\overline{a},P_2,\square}_{\widetilde{b}} P'$. We can apply directly the message output clause of the induction hypothesis: there exists Q' such that $Q_1 \xmapsto{\overline{a},Q_2,\square}_{\widetilde{b}} Q'$ and $P' \sim_m^\bullet Q'$. We conclude that $Q_1 \mid Q_2 \xmapsto{\tau} Q'$ (by rule HO_τ) with $P' \sim_m^\bullet Q'$ as wished.

Theorem 2. *Relation $\sim_m$ is a congruence.*

We now turn to the relationship between context and complementary bisimilarities. We show that they actually coincide.

Theorem 3. *We have* $\sim = \sim_m$.

In the message input case, complementary bisimilarity tests with a process while context bisimilarity tests with a concretion. Both testings are equivalent because of the congruence of $\sim_m$ (Theorem 2). See [13] for more details. The output clause of complementary bisimilarity requires that transition $P \xmapsto{\overline{a},T,\mathbb{E}}_{\widetilde{b}} P'$ has to be matched by a transition $Q \xmapsto{\overline{a},T,\mathbb{E}}_{\widetilde{b}} Q'$ with the same set of names $\widetilde{b}$ which may be extruded. At first glance, we do not have this requirement for the early strong context bisimilarity, hence we first have to prove that it is the case before proving Theorem 3.

Correspondence also holds in the weak case. We write $\overset{\tau}{\Mapsto}$ the reflexive and transitive closure of $\overset{\tau}{\mapsto}$. We define $\overset{a,R}{\Longmapsto}$ as $\overset{\tau}{\Mapsto}\xmapsto{a,R}\overset{\tau}{\Mapsto}$. In the weak case, two processes P and Q may evolve independently before interacting with each other. Since a transition $P \xmapsto{\overline{a},Q,\mathbb{E}}_{\widetilde{b}} P'$ includes a communication between P and Q, we have to authorize Q to perform τ-actions before interacting with P in the weak output transition. We define $P \overset{\overline{a},Q,\mathbb{E}}{\Longmapsto}_{\widetilde{b}} P'$ as $P \overset{\tau}{\Mapsto}\xmapsto{\overline{a},Q',\mathbb{E}}_{\widetilde{b}}\overset{\tau}{\Mapsto} P'$ with $Q \overset{\tau}{\Mapsto} Q'$.

Definition 7. *Weak complementary bisimilarity* $\approx_m$ *is the largest symmetric relation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$ *implies* $fn(P) = fn(Q)$ *and for all* $P \overset{\lambda}{\mapsto} P'$, *there exists* $Q \overset{\lambda}{\Mapsto} Q'$ *such that* $P' \mathcal{R} Q'$.

Using the same proofs techniques as in the strong case, we have the following results.

Theorem 4. *Relation* $\approx_m$ *is a congruence.*

Theorem 5. *We have* $\approx_m = \approx$.

Bisimilarity $\approx_m$ coincides with $\approx_b$ on *image-finite* processes. The limitation on image-finite processes is usual and can be found in π-calculus [23] for instance. A closed process P is image finite iff for every label λ, the set $\{P', P \overset{\lambda}{\Mapsto} P'\}$ is finite. Using the same proof technique as in [23], we have the following completeness result.

Theorem 6. *Let* P, Q *be image-finite processes.* $P \approx_b Q$ *if and only if* $P \approx_m Q$.

5 Related Work

Howe's method. This method has been used originally to prove congruence in a lazy functional programming language [10]. Baldamus and Frauenstein [2] are the first to adapt Howe's method to process calculi. They prove congruence of a late delay context bisimilarity, and of late and early delay higher-order bisimilarities in variants of Plain CHOCS [26]. Hildebrandt and Godskesen use Howe's method for their calculus Homer to prove congruence for late delay [9] and input-early delay [6] context bisimilarities. In [15], we have used Howe's method to prove congruence of a weak higher-order bisimilarity for a calculus featuring passivation but without restriction.

Behavioral equivalences in higher-order calculi. Very few higher-order calculi feature a coinductive characterization of weak barbed congruence. It is the case in HOπ and in variants of Mobile Ambients. Sangiorgi introduces context bisimilarities as characterizations of barbed congruence for its calculus HOπ [21]. Mobile Ambients [4] is a calculus with hierarchical localities and subjective linear process mobility. Contextual characterizations of weak barbed congruence have been defined for Mobile Ambients [16] and its variant NBA [3].

Difficulties arise in more expressive process calculi. The Seal calculus [5] is a calculus with objective, non linear process mobility, which requires synchronization between three processes (a process sending a name a, a receiving process, and a locality named a). Castagna et al. define a sound weak delay context bisimilarity in [5] called *Hoe bisimilarity* for the Seal calculus. The authors point out that their notion of context bisimilarity, Hoe bisimilarity, is not complete, not only because of the delay style, but also because of the labels introduced for partial synchronization which are most likely not all observable.

The Kell calculus [24] and Homer [9] are two higher-order calculi featuring passivation. They differ in how they handle communication; in particular, the Kell calculus allows join patterns while Homer does not. Sound and complete context bisimilarities have been defined for both calculi in the strong case. As stated before, a weak delay input-early bisimilarity has been proved sound in Homer using Howe's method. In [15] we have studied various calculi with passivation in order to find *normal* bisimilarities for those calculi, i.e., relations with finite testing at each step. In particular we have studied HOπP and showed that its behavioral theory raises similar difficulties as the Homer and Kell ones.

One can also establish soundness by writing transitions in a special rule format that ensures that the corresponding bisimilarity is a congruence. For higher-order calculi, Mousavi et al. [18] propose Promoted (resp. Higher-Order) PANTH format which guarantees that the corresponding strong (resp. higher-order) bisimilarity is a congruence. Another method is to generate the LTS from the reduction such that the corresponding bisimilarity is a congruence [19]. To this date, both methods have been applied in the strong case only. One can also prove congruence using environmental bisimulations [22]. We have not been able to apply this method to a calculus with passivation yet.

6 Conclusion and Future Work

Contextual LTS (based on abstractions and/or concretions) are not well suited to prove congruence using Howe's method. The method relies on a simulation-like property, which is hard to establish with early context bisimilarities that are the usual candidate relations for characterization of barbed congruence. The main issue is the mutual dependency between message input and output clauses of the context bisimilarity. In our complementary semantics, the message output clause depends on a process which may receive the message (i.e., a process which evolves towards an abstraction), instead of an abstraction which directly receives the message. This proof technique allows Howe's method to work with early weak (non delay) bisimilarity.

We have defined complementary semantics for HOπP, an extension of HOπ with passivation. In Section 4, we have defined a weak complementary bisimilarity and proved its soundness using Howe's method. We have also proved that it coincides with weak barbed congruence on image-finite processes, yielding the first characterization result in a calculus featuring passivation and restriction. This approach may be applied to other calculi: in [13] we present a sound weak complementary bisimilarity for HOπ and for the Seal calculus [5].

An immediate future work would be to define a complementary semantics for process calculi with no characterization result for weak barbed congruence, such as Seal, Homer [6], and the Kell calculus [24], and to prove that complementary bisimilarity in these calculi yields the required characterization. It should be easy for Homer since the HOπP semantics is close to the Homer one. For Seal, it remains to show that the complementary semantics in [13] is indeed complete. The Kell calculus may prove to be more challenging because of join patterns: to complement an emitting process P, we need a receiving process Q, but also other emitting processes $\widetilde{R}$ to match the receiving pattern of Q. Another future work is to define a LTS rule format which guarantees that Howe's method works with the corresponding bisimilarity, possibly extending the Promoted or Higher-Order PANTH format for higher-order calculi proposed by Mousavi et al. [18].

References

1. Baldamus, M.: Semantics and Logic of Higher-Order Processes: Characterizing Late Context Bisimulation. PhD thesis, Berlin University of Technology (1998)
2. Baldamus, M., Frauenstein, T.: Congruence proofs for weak bisimulation equivalences on higher–order process calculi. Technical report, Berlin University of Technology (1995)
3. Bugliesi, M., Crafa, S., Merro, M., Sassone, V.: Communication and mobility control in boxed ambients. Information and Computation 202 (2005)
4. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)
5. Castagna, G., Vitek, J., Zappa Nardelli, F.: The Seal Calculus. Information and Computation 201(1), 1–54 (2005)
6. Godskesen, J.C., Hildebrandt, T.: Extending howe's method to early bisimulations for typed mobile embedded resources with local names. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005, vol. 3821, pp. 140–151. Springer, Heidelberg (2005)
7. Gordon, A.D.: Bisimilarity as a theory of functional programming. Mini-course. Notes Series NS-95-3, BRICS, University of Cambridge Computer Laboratory, iv+59 pp. (July 1995)
8. Hennessy, M., Rathke, J., Yoshida, N.: Safedpi: a language for controlling mobile code. Acta Inf. 42(4-5) (2005)
9. Hildebrandt, T., Godskesen, J.C., Bundgaard, M.: Bisimulation congruences for Homer — a calculus of higher order mobile embedded resources. Technical Report ITU-TR-2004-52, IT University of Copenhagen (2004)
10. Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Information and Computation 124(2), 103–112 (1996)
11. Jeffrey, A., Rathke, J.: A theory of bisimulation for a fragment of concurrent ML with local names. Theoretical Computer Science 323, 1–48 (2004)

12. Jeffrey, A., Rathke, J.: Contextual equivalence for higher-order pi-calculus revisited. Logical Methods in Computer Science 1(1) (2005)
13. Lenglet, S., Schmitt, A., Stefani, J.B.: Howe's method for early bisimilarities. Technical Report RR 6773, INRIA (2008)
14. Lenglet, S., Schmitt, A., Stefani, J.B.: Characterizing contextual equivalence in calculi with passivation (2009), http://sardes.inrialpes.fr/~aschmitt/papers/hop_howe_long.pdf
15. Lenglet, S., Schmitt, A., Stefani, J.-B.: Normal bisimulations in process calculi with passivation. In: FoSSaCS 2009. LNCS, vol. 5504, pp. 257–271. Springer, Heidelberg (2009)
16. Merro, M., Zappa Nardelli, F.: Behavioral theory for mobile ambients. Journal of the ACM 52(6), 961–1023 (2005)
17. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623. Springer, Heidelberg (1992)
18. Mousavi, M., Gabbay, M.J., Reniers, M.A.: Sos for higher order processes (extended abstract). In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 308–322. Springer, Heidelberg (2005)
19. Rathke, J., Sobocinski, P.: Deconstructing behavioural theories of mobility. In: IFIP TCS. IFIP, vol. 273, pp. 507–520. Springer, Heidelberg (2008)
20. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis, Department of Computer Science, University of Edinburgh (1992)
21. Sangiorgi, D.: Bisimulation for higher-order process calculi. Information and Computation 131(2), 141–178 (1996)
22. Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. In: LICS 2007, pp. 293–302. IEEE Computer Society, Los Alamitos (2007)
23. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
24. A. Schmitt and J.-B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing 2004 workshop*, volume 3267 of *LNCS*, 2004.
25. Sewell, P., Leifer, J., Wansbrough, K., Zappa Nardelli, F., Allen-Willians, M., Habouzit, P., Vafeiadis, V.: Acute: High-level programming language design for distributed computation. Journal of Functional Programming 17(4-5) (2007)
26. Thomsen, B.: Plain chocs: A second generation calculus for higher order processes. Acta Informatica 30(1), 1–59 (1993)

On the Relationship between π-Calculus and Finite Place/Transition Petri Nets

Roland Meyer[1] and Roberto Gorrieri[2]

[1] LIAFA, Paris Diderot University & CNRS
roland.meyer@liafa.jussieu.fr
[2] Department of Computing Science, University of Bologna
gorrieri@cs.unibo.it

Abstract. We clarify the relationship between π-calculus and finite p/t Petri nets. The first insight is that the concurrency view to processes taken in [Eng96, AM02, BG09] and the structural view in [Mey09] are orthogonal. This allows us to define a new concurrency p/t net semantics that can be combined with the structural semantics in [Mey09]. The result is a more expressive mixed semantics, which translates precisely the so-called mixed-bounded processes into finite p/t nets. Technically, the translation relies on typing of restricted names. As second main result we show that mixed-bounded processes form the borderline to finite p/t nets. For processes just beyond this class reachability becomes undecidable and so no faithful translation into finite p/t nets exists.

1 Introduction

There has been considerable recent interest in verification techniques for mobile systems that are based on automata-theoretic and in particular Petri net translations [AM02, FGMP03, KKN06, DKK08, MKS09, BG09, Mey09]. Most verification approaches and tool implementations have been developed for finite place/transition (p/t) Petri nets. This raises the question for a characterisation of the processes that can be verified with the help of finite p/t Petri net semantics. Despite the efforts in the semantics community dating back to [BG95, MP95, Eng96, Pis99] the problem is still open and we tackle it in this paper. The contribution is the precise borderline that separates π-calculus processes from finite p/t Petri nets in terms of computational expressiveness.

In structurally stationary processes [Mey09] restricted names generate a finite number of connection graphs at runtime. Although the constraint is semantical and thus undecidable, large syntactic subclasses of structurally stationary processes exist, e.g. finite control [Dam96] and restriction-free processes [AM02]. The so-called structural semantics represents structurally stationary processes as finite p/t nets and is sound wrt. the reaction semantics [Mey09]. Therefore, we take structurally stationary processes as starting point in our quest for the borderline to finite p/t nets.

Mobile systems are concurrent systems where the connections between components evolve over time. While the structural semantics focuses on these connections, concurrency semantics highlight the interactions between components.

In [BG95, BG09], we defined a concurrency semantics that reflects the intended causality semantics for π-calculus. It is finite for the semantic class of restriction-bounded processes, which generate a finite number of restricted names. In this work, we present the largest class of processes with a finite p/t net semantics that is sound wrt. the reaction behaviour. We find it as follows.

We observe that the structural view to processes is orthogonal to the concurrency view. The main result is that an appropriately defined concurrency semantics can be combined with the structural semantics to a more expressive mixed translation. Intuitively, the new translation mirrors the interactions between groups of components. Technically, the combination is achieved by typing restricted names, and the type determines the semantics that handles a name.

We prove the mixed semantics to be finite precisely for mixed-bounded processes, which combine the previous finiteness characterisations, i.e., they form finitely many connection graphs with names of type one and generate finitely many restricted names of type two. Again, mixed boundedness is an undecidable semantic condition, but it inherits all syntactic subclasses of structurally stationary and restriction-bounded processes.

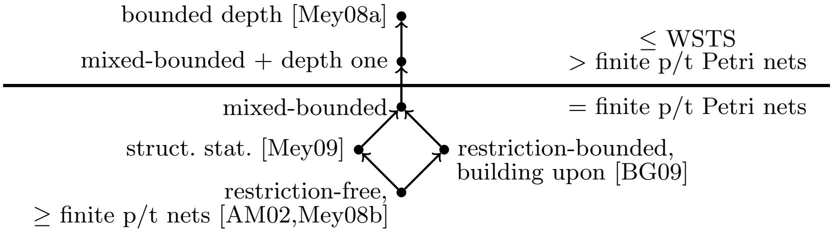


Fig. 1. Hierarchy of process classes and relation to finite p/t Petri nets ($\rightarrow := \subseteq$)

We then show that mixed-bounded processes form the *borderline* between π-calculus and finite p/t nets, since in a minimal extension of the class reachability becomes undecidable. Unfortunately, this extension lies within the processes of bounded depth, which are known to have well-structured transition systems (WSTS) and so, e.g. a decidable termination problem [Mey08a]. Hence, our results dash hope of a finite p/t net translation for this interesting class.

Note that every finite p/t net can be represented by a restriction-free process of linear size with contraction-isomorphic transition system [AM02, Mey08b]. Hence, all process classes we consider are at least as expressive as finite p/t nets. Figure 1 illustrates the elaborated relationships. We summarise our contribution.

– We define a new concurrency semantics for the π-calculus, which satisfies three indispensable quality criteria. It yields a bisimilar transition system, translates processes with restricted names, and enjoys an intuitive finiteness characterisation. The technical tool that facilitates the definition is a new *name-aware transition system*, which manages the use of restricted names.

– We combine the concurrency semantics with the structural semantics and prove the resulting translation finite precisely for so-called *mixed-bounded processes*. The idea to combine the semantics is to type restricted names. Technically, the definition also relies on a new normal form for processes under structural congruence.
– We prove that mixed-bounded processes form the borderline to finite p/t nets. If we relax the requirement slightly, reachability becomes undecidable.

Related Work. Although several automata-theoretic semantics for the π-calculus have been proposed [Eng96, MP95, Pis99, AM02, BG95, BG09, KKN06, DKK08], we found them all defective in the sense that they do not satisfy the criteria we require for a semantics to be usable for verification. We discuss their problems.

Engelfriet [Eng96] translates processes with replication into p/t nets that are bisimilar to the reaction semantics. Since the Petri net representation is infinite as soon as the replication operator is used, the requirement for finiteness is not satisfied. In subsequent papers [EG99, EG04], Engelfriet and Gelsema show that the discriminating power of their semantics corresponds to extended and decidable versions of structural congruence. In the proofs, they exploit normal forms for processes similar to the restricted form we present in Section 2.

Montanari and Pistore propose history dependent automata (HDA) as semantic domain, finite automata where states are labelled by sets of names that represent the restrictions in use [MP95, Pis99]. The ground and early labelled transition semantics are translated into HDA in a way that bisimilarity on the automata coincides with the corresponding bisimilarity on processes. The translations yield finite HDA only for finitary processes, the subclass of structurally stationary processes obtained by bounding the degree of concurrency [Mey09].

Amadio and Meyssonnier [AM02] translate restriction-free processes into bisimilar (reaction semantics) and finite p/t nets. To deal with a class of processes that contain restrictions, a second translation identifies restricted names as unsed and replaces them by generic free names. Since the number of processes to be modified by replacement is not bounded, these authors rely on Petri nets with transfer as semantic domain. Having an undecidable reachability problem, transfer nets are strictly more expressive than finite p/t nets [DFS98].

Our work [BG95] translates the early labelled transition relation of restriction-bounded processes into finite p/t Petri nets, but fails to prove bisimilarity. Based on that translation, [BG09] studies non-interleaving and causal semantics for the π-calculus and provides decidability results for model checking.

Koutny et. al. [DKK08] achieve a bisimilar translation of the indexed labelled transition system into finite but high-level Petri nets, thus relying on a Turing complete formalism where automatic analyses are necessarily incomplete. The main contribution is compositionality, for every π-calculus operator there is a corresponding net operator and in many cases the size of the net is linear in the size of the process. In [KKN06], the translation is extended by an unfolding-based model checker. To avoid the undecidability the verification approach is restricted to recursion-free processes, a class of limited practical applicability.

2 Preliminaries

We recall the basics on π-calculus, p/t Petri nets, and the structural semantics.

π-Calculus. We use a π-calculus with parameterised recursion as proposed in [SW01]. Let the set $\mathcal{N} := \{a, b, x, y, \ldots\}$ of *names* contain the channels, which are also the possible messages, that occur in communications. During a process execution the *prefixes* π are successively removed from the process to communicate with other processes or to perform silent actions. The *output action* $\pi = \overline{a}\langle b\rangle$ sends the name b along channel a. The *input action* $\pi = a(x)$ receives a name that replaces x on a. Prefix $\pi = \tau$ performs a *silent action*.

To denote recursive processes, we use *process identifiers* $K, L, \ldots$ A process identifier is defined by an equation $K(\tilde{x}) := P$, where $\tilde{x}$ is a short-hand notation for $x_1, \ldots, x_k$. When the identifier is *called*, $K\lfloor\tilde{a}\rfloor$, it is replaced by process P with the names $\tilde{x}$ changed to $\tilde{a}$. More precisely, a *substitution* $\sigma = \{\tilde{a}/\tilde{x}\}$ is a function that maps the names in $\tilde{x}$ to $\tilde{a}$, and is the identity for all the names not in $\tilde{x}$. The *application of a substitution* is denoted by $P\sigma$ and defined in the standard way [SW01]. A π-calculus process is either a call to an identifier, $K\lfloor\tilde{a}\rfloor$, a *choice process* deciding between prefixes, $M + N$, a *parallel composition* of processes, $P_1 \mid P_2$, or the *restriction* of a name in a process, $\nu a.P$:

$$M ::= \mathbf{0} \;\mid\; \pi.P \;\mid\; M + N \qquad\qquad P ::= M \;\mid\; K\lfloor\tilde{a}\rfloor \;\mid\; P_1 \mid P_2 \;\mid\; \nu a.P.$$

Writing π for $\pi.\mathbf{0}$ we omit pending $\mathbf{0}$ processes. By $M^{=\mathbf{0}}$ we denote choice compositions of stop processes $\mathbf{0} + \ldots + \mathbf{0}$. To indicate a choice composition contains at least one term $\pi.P$ we denote it by $M^{\neq\mathbf{0}}$. Processes $M^{\neq\mathbf{0}}$ and $K\lfloor\tilde{a}\rfloor$ are called *sequential*, and they are the basic processes that produce reactions, either alone or by synchronisation of two of them, in the reaction relation defined below.

A restriction νa that is not covered by a prefix π is *active* and the *set of active restrictions* in a process is $arn(P)$. For example, $arn(\nu a.\overline{a}\langle b\rangle.\nu c.\overline{a}\langle c\rangle) = \{a\}$. The input action $a(b)$ and the restriction $\nu c.P$ *bind* the names b and c, respectively. The *set of bound names* in a process P is $(arn(P) \subseteq)\ bn(P)$. A name which is not bound is *free* and the *set of free names* in P is $fn(P)$. We permit α-conversion of bound names. Therefore, wlog. we assume that a name is bound at most once in a process and that $bn(P) \cap fn(P) = \emptyset$. Moreover, if a substitution $\sigma = \{\tilde{a}/\tilde{x}\}$ is applied to a process P, we assume $bn(P) \cap (\tilde{a} \cup \tilde{x}) = \emptyset$.

We use the *structural congruence* relation in the definition of the behaviour of a process term. It is the smallest congruence where α-conversion of bound names is allowed, $+$ and $\mid$ are commutative and associative with $\mathbf{0}$ as the neutral element, and the following laws for restriction hold:

$$\nu x.\mathbf{0} \equiv \mathbf{0} \qquad \nu x.\nu y.P \equiv \nu y.\nu x.P \qquad \nu x.(P \mid Q) \equiv P \mid (\nu x.Q),\ \text{if } x \notin fn(P).$$

The last rule is called *scope extrusion*. The behaviour of π-calculus processes is then determined by the *reaction relation* $\rightarrow\ \subseteq \mathcal{P} \times \mathcal{P}$ defined by the rules in Table 1. By $Reach(P)$ we denote the *set of all processes reachable from* P by the reaction relation. The *transition system* of process P factorises the reachable

Table 1. Rules defining the reaction relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$

$$\text{(Tau)}\quad \tau.P + M \rightarrow P \qquad \text{(React)}\quad x(y).P + M \mid \overline{x}\langle z\rangle.Q + N \rightarrow P\{z/y\} \mid Q$$

$$\text{(Const)}\quad K\lfloor\tilde{a}\rfloor \rightarrow P\{\tilde{a}/\tilde{x}\},\ \text{if } K(\tilde{x}) := P$$

$$\text{(Par)}\quad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{(Res)}\quad \frac{P \rightarrow P'}{\nu a.P \rightarrow \nu a.P'} \qquad \text{(Struct)}\quad \frac{Q \equiv P \rightarrow P' \equiv Q'}{Q \rightarrow Q'}$$

processes along structural congruence, $\mathcal{T}(P) := (\mathit{Reach}(P)/_{\equiv}, \rightarrow_{\mathcal{T}}, [P])$ with the transition relation $[P] \rightarrow_{\mathcal{T}} [Q]$ defined by $P \rightarrow Q$.

Our theory employs two normal forms for processes. The classical *standard form* of Milner [Mil99] maximises the scopes of active restricted names and removes unused restrictions and $\mathbf{0}$ processes. For example, the standard form of $\nu a.K\lfloor a\rfloor \mid \nu b.0 \mid K\lfloor c\rfloor$ is $\nu a.(K\lfloor a\rfloor \mid K\lfloor c\rfloor)$. We compute it with the function sf, which is the identity on sequential processes. For a restriction $\nu a.P$ we have $\mathit{sf}(\nu a.P) := \nu a.\mathit{sf}(P)$ if $a \in \mathit{fn}(P)$ and $\mathit{sf}(\nu a.P) := \mathit{sf}(P)$ otherwise. The parallel composition of P and Q with $\mathit{sf}(P) = \nu\tilde{a}_P.P^{\neq\nu}$ and $\mathit{sf}(Q) = \nu\tilde{a}_Q.Q^{\neq\nu}$ maximises the scopes, $\mathit{sf}(P \mid Q) := \nu\tilde{a}_P.\nu\tilde{a}_Q.(P^{\neq\nu} \mid Q^{\neq\nu})$. Of course, the sequences of names $\tilde{a}_P$ and $\tilde{a}_Q$ may be empty and furthermore $P^{\neq\nu}$ and $Q^{\neq\nu}$ denote parallel compositions of sequential processes $K\lfloor a\rfloor$ and $M^{\neq\mathbf{0}}$.

Dual to the standard form, the *restricted form* [Mey09] minimises the scopes of active restrictions and also removes unused restrictions and processes congruent to $\mathbf{0}$. For example, the restricted form of $\nu a.(K\lfloor a\rfloor \mid \nu b.0 \mid K\lfloor c\rfloor)$ is $\nu a.K\lfloor a\rfloor \mid K\lfloor c\rfloor$. Technically, the restricted form relies on the notion of *fragments* F, G, H built inductively in two steps. Sequential processes $K\lfloor\tilde{a}\rfloor$ and $M^{\neq\mathbf{0}}$ are called *elementary fragments* F^e, G^e and form the basis. General fragments are defined by the grammar

$$F^e ::= K\lfloor\tilde{a}\rfloor \;\mid\; M^{\neq\mathbf{0}} \qquad\qquad F ::= F^e \;\mid\; \nu a.(F_1 \mid \ldots \mid F_n)$$

with $a \in \mathit{fn}(F_i)$ for all i. A *process P^{rf} in restricted form* is now a parallel compositions of fragments, $P^{\mathit{rf}} = \Pi_{i\in I}F_i$, and we refer to the fragments in P^{rf} by $\mathit{fg}(P^{\mathit{rf}}) := \bigcup_{i\in I}\{F_i\}$. The *decomposition function* $\mathit{dec}(P^{\mathit{rf}})$ counts the number of fragments in P^{rf} that are in a given class, e.g. $P^{\mathit{rf}} = F \mid G \mid F'$ with $F \equiv F' \not\equiv G$ yields $(\mathit{dec}(P^{\mathit{rf}}))([F]) = 2$, $(\mathit{dec}(P^{\mathit{rf}}))([G]) = 1$, and $(\mathit{dec}(P^{\mathit{rf}}))([H]) = 0$ for $F \not\equiv H \not\equiv G$. The function characterises structural congruence: $P^{\mathit{rf}} \equiv Q^{\mathit{rf}}$ is equivalent to $\mathit{dec}(P^{\mathit{rf}}) = \mathit{dec}(Q^{\mathit{rf}})$ [Mey09]. This is crucial for the well-definedness of the concurrency semantics we investigate in Section 3.

For a fragment F, we let $\|F\|_\nu$ denote the *nesting of active restrictions.* For example, $\|\nu a.(\nu b.K\lfloor a,b\rfloor \mid \nu c.\nu d.L\lfloor a,c,d\rfloor)\|_\nu = 3$. The *depth of a fragment* F is then the nesting of active restrictions in the flattest representation, $\|F\|_{\mathcal{D}} := \min\{\|G\|_\nu \mid G \equiv F\}$. A process is *bounded in depth* if there is a bound $k \in \mathbb{N}$ on the depth of all reachable fragments [Mey08a].

For a given process, function rf computes a structurally congruent one in restricted form [Mey09]. It is the identity on sequential processes and a homomorphism for the the parallel composition, $\mathit{rf}(P \mid Q) := \mathit{rf}(P) \mid \mathit{rf}(Q)$. In case of restriction $\nu a.P$, we first compute the restricted form $rf(P) = \Pi_{i \in I} F_i$. Then the scope of νa is restricted to the fragments where a is a free name (let the set I_a contain their indices), $rf(\nu a.P) := \nu a.(\Pi_{i \in I_a} F_i) \mid \Pi_{i \in I \setminus I_a} F_i$.

Petri Nets. An *(unmarked) Petri net* is a triple (S, T, W) with disjoint and potentially infinite sets of *places* S and *transitions* T, and a *weight function* $W : (S \times T) \cup (T \times S) \to \mathbb{N} := \{0, 1, 2, \ldots\}$. A Petri net is *finite* if S and T are finite sets. A *marking* of the net is a function $M : S \to \mathbb{N}$ and its *support* $supp(M)$ are the elements mapped to a value greater zero. As usual, places are represented by circles, transitions by boxes, the weight function by arcs with numbers (missing arcs have weight 0, unlabelled arcs have weight 1), and markings by tokens within the circles. We denote *pre-* and *postset* of $z \in S \cup T$ by ${}^\bullet z := \{y \mid W(y, z) > 0\}$ and $z^\bullet := \{y \mid W(z, y) > 0\}$. A *(marked) Petri net* is a pair $N = (S, T, W, M_0)$ of an unmarked net (S, T, W) and an *initial* marking M_0. The *set of all marked Petri nets* is denoted by $\mathcal{PN}$.

A transition $t \in T$ is *enabled* in marking M if $M(s) \geq W(s, t)$ for every $s \in {}^\bullet t$. *Firing* an enabled transition leads to marking $M'(s) := M(s) - W(s, t) + W(t, s)$ for every $s \in S$, and is denoted by $M[t\rangle M'$. In the transition system we work with the unlabelled relation $M \to M'$, which means $M[t\rangle M'$ for some $t \in T$.

To relate a process and its Petri net representation, we rely on the notion of bisimilarity [Mil89]. Two transition systems $\mathcal{T}_i = (St_i, \to_i, s_i^0)$ with $i = 1, 2$ are *bisimilar*, $\mathcal{T}_1 \approx \mathcal{T}_2$, if there is a bisimulation relation $\mathcal{R} \subseteq St_1 \times St_2$ that contains the initial states, $(s_1^0, s_2^0) \in \mathcal{R}$. In a *bisimulation relation* $\mathcal{R}$, containment $(s_1, s_2) \in \mathcal{R}$ requires (1) that for every $t_1 \in St_1$ with $s_1 \to_1 t_1$ there is a $t_2 \in St_2$ with $s_2 \to_2 t_2$ and $(t_1, t_2) \in \mathcal{R}$ and (2) similar for transitions from s_2.

Structural Semantics. We recall the translation of π-calculus processes into Petri nets defined in [Mey09]. The idea is to have a separate place for each reachable group of processes connected by restricted names, i.e., the notion of fragments plays a crucial role. The algorithm takes a π-calculus process P and computes a Petri net $\mathcal{N}_{\mathcal{S}}[\![P]\!]$, called the *structural semantics* of P, as follows.

The places are the fragments of all reachable processes Q. More precisely, we take the structural congruence classes of fragments, $fg(rf(Q))/_{\equiv}$.

There are two disjoint sets of transitions. Transitions $t = ([F], [Q])$ model reactions inside fragment F that lead to process Q. More formally, $[F]$ is a place and $F \to Q$ holds. These reactions are communications within F, silent actions, or calls to process identifiers. There is an arc weighted one from place $[F]$ to t.

Transitions $t = ([F_1 \mid F_2], [Q])$ model reactions between reachable fragments along public channels: $[F_1]$ and $[F_2]$ are places and $F_1 \mid F_2 \to Q$. To define the preset of t, consider place $[F]$. If F_1, F_2, and F are structurally congruent there is an arc weighted two from $[F]$ to t. If F is structurally congruent with either F_1 or F_2, there is an arc weighted one. Otherwise there is no arc.

The postset of a transition $([F], [Q])$ or $([F_1 \mid F_2], [Q])$ are the reachable fragments of Q. If fragment G occurs (up to $\equiv$) $k \in \mathbb{N}$ times in $rf(Q)$, then there is an arc weighted k from $([F], [Q])$ to $[G]$. For example, from the transition $([\tau.(K\lfloor a \rfloor \mid K\lfloor a \rfloor)], [K\lfloor a \rfloor \mid K\lfloor a \rfloor])$ there is an arc weighted two to place $[K\lfloor a \rfloor]$.

The initial marking of place $[F]$ in $\mathcal{N}_{\mathcal{S}}[\![P]\!]$ is determined by the number of fragments in $rf(P)$ that are congruent to F. We illustrate the translation on our running example: a model of a bag data structure that takes values from a fill process on channel *in* and emits them on channel *out* in any order [Fok07].

Example 1. Consider $FILL\lfloor in\rfloor \mid BAG\lfloor in, out\rfloor$ with the equations $FILL(in) := \nu val.\overline{in}\langle val\rangle.FILL\lfloor in\rfloor$ and $BAG(in, out) := in(y).(\overline{out}\langle y\rangle \mid BAG\lfloor in, out\rfloor)$. The structural semantics $\mathcal{N}_{\mathcal{S}}[\![FILL\lfloor in\rfloor \mid BAG\lfloor in, out\rfloor]\!]$ is the Petri net in Figure 2 without $s_6, \ldots, s_9$ and with the following places:

$$s_1 := [FILL\lfloor in\rfloor] \qquad s_2 := [BAG\lfloor in, out\rfloor] \qquad s_5 := [\nu val.\overline{out}\langle val\rangle]$$
$$s_3 := [\nu val.\overline{in}\langle val\rangle.FILL\lfloor in\rfloor] \qquad s_4 := [in(y).(\overline{out}\langle y\rangle \mid BAG\lfloor in, out\rfloor)].$$

Note that the structural semantics remains finite if we restrict the name *in* while it becomes infinite if we restrict *out*. A restricted name *out*, distributed to an unbounded number of processes $\nu val.\overline{out}\langle val\rangle$, would create an unbounded number of places (unboundedness in breadth [Mey09]).

A π-calculus process and its structural semantics have isomorphic transition systems [Mey09]. Moreover, the structural semantics is finite iff the translated process is structurally stationary, i.e., generates finitely many types of fragments.

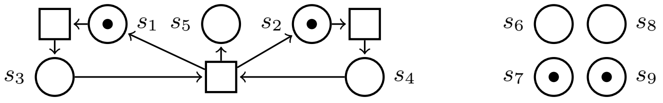


Fig. 2. Petri net representation of the different versions of the bag data structure introduced in the Examples 1, 2, and 3. The places and the usage of $s_6, \ldots, s_9$ depend on the semantics under consideration and are explained in the examples

3 A Sound Concurrency Semantics for the π-Calculus

Concurrency semantics highlight the communications between sequential processes. As opposed to the structural semantics, the scopes of restricted names are not important. The key idea to define a concurrency semantics is to use designated free names for active restrictions. To generate these names systematically, we define the so-called *name-aware transition system*. It facilitates the first bisimilarity proof $P \approx \mathcal{N}_{\mathcal{C}}[\![P]\!]$ for a concurrency semantics that handles name creation and allows for a finiteness characterisation. Surprisingly, all earlier attempts lacking this technical tool failed as explained in the introduction.

For a smooth definition of the name-aware transition system we assume that restricted names have the form a_m, i.e., they carry an *index* $m \in \mathbb{N}$. α-conversion of a restricted name a_m changes the index m but not the name a. Wlog., for a process P of interest we assume the indices to be zero. If P uses defining equations $K_i(\tilde{x}) := P_i$, then the indices in the P_i are zero as well. For a restriction a_m the *increment operation* yields $a_m + 1 := a_{m+1}$. Its application to sets is defined elementwise, for example $\{a_3, b_2, c_5\} + 1 = \{a_4, b_3, c_6\}$.

In the name-aware transition system, the states are *name-aware processes* of the form $(P^{\neq\nu}, \tilde{a})$. Intuitively, in an execution sequence leading to process $(P^{\neq\nu}, \tilde{a})$ the active restrictions $\tilde{a}$ have been found (the set may be empty, $\tilde{a} = \emptyset$). The names $\tilde{a}$ are not chosen arbitrarily but are computed by incrementing the indices. For example, the name-aware process $(\tau.\nu a_0.K\lfloor a_0 \rfloor, \{a_0, a_1, a_2\})$ consumes a τ-action and generates the restricted name a_3. Formally, the behaviour of name-aware processes is captured by the *name-aware reaction relation*

$$(P^{\neq\nu}, \tilde{a}) \rightarrow^{na} (Q^{\neq\nu}, \tilde{a} \uplus \tilde{b}) :\Leftrightarrow \begin{array}{l} P^{\neq\nu} \rightarrow \nu\tilde{b}.Q^{\neq\nu} \text{ in standard form and} \\ \forall b_k \in \tilde{b} : k - 1 = max\{i \mid b_i \in \tilde{a}\}, \end{array} \qquad (\clubsuit)$$

where we choose zero as index if there is no name $b_i \in \tilde{a}$. The set of all processes reachable from $(P^{\neq\nu}, \tilde{a})$ by the name-aware reaction relation is $Reach_{na}(P^{\neq\nu}, \tilde{a})$. For the example process above, the definition in fact yields the name-aware reaction $(\tau.\nu a_0.K\lfloor a_0 \rfloor, \{a_0, a_1, a_2\}) \rightarrow^{na} (K\lfloor a_3 \rfloor, \{a_0, a_1, a_2, a_3\})$. The *name-aware transition system* $\mathcal{T}_{na}(P^{\neq\nu}, \tilde{a})$ is again defined by factorising the reachable processes along structural congruence, $Reach_{na}(P^{\neq\nu}, \tilde{a})/_{\equiv}$. The name-aware reaction relation is lifted to process classes $([P^{\neq\nu}], \tilde{a})$, accordingly. Lemma 1 states bisimilarity of the name-aware and the standard transition system of a process.

Lemma 1. *For every process $P \in \mathcal{P}$ with standard form $sf(P) = \nu\tilde{a}.P^{\neq\nu}$ the bisimilarity $\mathcal{T}(P) \approx \mathcal{T}_{na}(P^{\neq\nu}, \tilde{a})$ holds.*

Proof. The relation that connects

$$([Q^{\neq\nu}], \tilde{b}) \in Reach_{na}(P^{\neq\nu}, \tilde{a})/_{\equiv} \qquad \text{and} \qquad [\nu\tilde{b}.Q^{\neq\nu}] \in Reach(P)/_{\equiv}$$

is a bisimulation and relates the initial processes in both transition systems.

Two basic ideas underly our concurrency semantics. First, as usual we let tokens reflect the numbers of processes in a state. Second and unusual, we use additional *name places* to imitate the generation of free names in the name-aware transition system; a construction that has precursor in our earlier works [BG95, BG09]. If a place, say a_3, is marked, the names a_0, a_1, a_2 have already been generated and a_3 is the next one to be invented. The transition that corresponds to the reaction $(\tau.\nu a_0.K\lfloor a_0 \rfloor, \{a_0, a_1, a_2\}) \rightarrow^{na} (K\lfloor a_3 \rfloor, \{a_0, a_1, a_2, a_3\})$ above moves the token from name place a_3 to a_4.

Technically, we start with the name-aware transition system and compute the two disjoint sets of *name* and *process places*. The name places are precisely the names $\tilde{b}$ in all reachable name-aware processes $([Q^{\neq\nu}], \tilde{b})$. The process places are given by the (structural congruence classes of) sequential processes in $Q^{\neq\nu}$.

Let $([P^{\neq\nu}], \tilde{a})$ be the initial process in the name-aware transition system. Also the initial marking is composed out of two disjoint functions. Function $M_0^{\mathcal{P}}$ marks the process places as required by the sequential processes in $P^{\neq\nu}$. Marking $M_0^{\mathcal{N}}$ puts a single token on all name places with index zero—except $\tilde{a}$. If $a_0 \in \tilde{a}$ the name is already in use and a_1 is the next to be generated. Therefore, name place a_1 is marked by one token. In fact, all name places are 1-safe.

Like for the structural semantics we have two disjoint sets of transitions. The first set contains transitions $t = ([M^{\neq \mathbf{0}}], \tilde{b}, [Q^{\neq\nu}])$ with the constraint that $[M^{\neq \mathbf{0}}]$ and $\tilde{b}$ are places and $M^{\neq \mathbf{0}} \rightarrow \nu\tilde{b}.Q^{\neq\nu}$ in standard form. The preset of t are the process place $[M^{\neq \mathbf{0}}]$ and the name places $\tilde{b}$. Hence, names can only be generated if their places are marked. The postset is given by the reachable sequential processes in $Q^{\neq\nu}$ and the names $\tilde{b}+1$. Thus, the transition moves a token from $b_k \in \tilde{b}$ to b_{k+1} as was explained. Similar transitions exist for $K\lfloor\tilde{c}\rfloor$.

The second set of transitions models communications between sequential processes. Here we have transitions $t = ([M_1^{\neq \mathbf{0}} \mid M_2^{\neq \mathbf{0}}], \tilde{b}, [Q^{\neq\nu}])$ with the condition that $M_1^{\neq \mathbf{0}} \mid M_2^{\neq \mathbf{0}}$ reacts to $\nu\tilde{b}.Q^{\neq\nu}$ in standard form. There is an arc weighted two from place $[N^{\neq \mathbf{0}}]$ to t if $M_1^{\neq \mathbf{0}} \equiv N^{\neq \mathbf{0}} \equiv M_2^{\neq \mathbf{0}}$. In this case, two structurally congruent processes communicate. If place $[N^{\neq \mathbf{0}}]$ is only one of the sequential processes, $N^{\neq \mathbf{0}} \equiv M_1^{\neq \mathbf{0}}$ xor $N^{\neq \mathbf{0}} \equiv M_2^{\neq \mathbf{0}}$, we draw an arc weighted one from the place to the transition. In any other case there is no arc, which means transition t represents a reaction where process $[N^{\neq \mathbf{0}}]$ is not involved in. Like for the first set of transitions, the places $\tilde{b}$ in the preset of t ensure restricted names are invented in the correct order. The postset is similar as well. We illustrate the definition of the concurrency semantics on the bag data structure.

Example 2. Consider the process $\nu in_0.\nu out_0.(FILL_2\lfloor in_0, val\rfloor \mid BAG\lfloor in_0, out_0\rfloor)$ where different from Example 1 data value val is not restricted, $FILL_2(in_0, val) := \overline{in_0}\langle val\rangle.FILL_2\lfloor in_0, val\rfloor$. The equation for BAG is as before. The concurrency semantics $\mathcal{N}_{\mathcal{C}}[\![\nu in_0.\nu out_0.(FILL_2\lfloor in_0, val\rfloor \mid BAG\lfloor in_0, out_0\rfloor)]\!]$ is the Petri net in Figure 2 with the process places

$$s_1 := [FILL_2\lfloor in_0, val\rfloor] \qquad s_2 := [BAG\lfloor in_0, out_0\rfloor] \qquad s_5 := [\overline{out_0}\langle val\rangle]$$
$$s_3 := [\overline{in_0}\langle val\rangle.FILL_2\lfloor in_0, val\rfloor] \qquad s_4 := [in_0(y).(\overline{out_0}\langle y\rangle \mid BAG\lfloor in_0, out_0\rfloor)]$$

and the name places $s_6 := in_0$, $s_7 := in_1$, $s_8 := out_0$, and $s_9 := out_1$. Note that place in_1 is initially marked as in_0 is active in the initial process. Moreover, if we restricted val the concurrency semantics would have an unbounded number of val_i places and hence be infinite.

Formally, the *concurrency semantics* is the function $\mathcal{N}_{\mathcal{C}} : \mathcal{P} \rightarrow \mathcal{PN}$. It assigns to a process $P_0 \in \mathcal{P}$ with $sf(P_0) = \nu\tilde{a}_0.P_0^{\neq\nu}$ the Petri net $\mathcal{N}_{\mathcal{C}}[\![P_0]\!] = (S, T, W, M_0)$ as follows. The set of places is the disjoint union $S := S^{\mathcal{P}} \uplus S^{\mathcal{N}}$ of the process places $S^{\mathcal{P}}$ and the name places $S^{\mathcal{N}}$:

$$S^{\mathcal{P}} := fg(Reach_{na}(P_0^{\neq\nu}, \tilde{a}_0))/_{\equiv}$$
$$S^{\mathcal{N}} := nms(Reach_{na}(P_0^{\neq\nu}, \tilde{a}_0)) \cup nms(Reach_{na}(P_0^{\neq\nu}, \tilde{a}_0)) + 1,$$

where $fg(P^{\neq\nu}, \tilde{a}) := fg(P^{\neq\nu})$ and $nms(P^{\neq\nu}, \tilde{a}) := \tilde{a}$ for a name-aware process $(P^{\neq\nu}, \tilde{a})$. Note that since $P^{\neq\nu}$ is a parallel composition of elementary fragments, it is in restricted form and we can access its elements via $fg(P^{\neq\nu})$.

To define the set T of transitions, consider the process places $[F^e], [F_1^e], [F_2^e]$ and the name places $\tilde{a}$ in S. We have

$$\begin{aligned} T := \quad & \{([F^e], \tilde{a}, [Q^{\neq\nu}]) \mid F^e \rightarrow \nu\tilde{a}.Q^{\neq\nu} \text{ in standard form}\} \\ \cup\, & \{([F_1^e \mid F_2^e], \tilde{a}, [Q^{\neq\nu}]) \mid F_1^e \mid F_2^e \rightarrow \nu\tilde{a}.Q^{\neq\nu} \text{ in standard form}\}. \end{aligned}$$

We define the weight function for transitions $t = ([F_1^e \mid F_2^e], \tilde{a}, [Q^{\neq\nu}])$, for transitions $t' = ([F^e], \tilde{a}, [Q^{\neq\nu}])$ the definition is similar. Consider places a and $[G^e]$ and let condition $a \in \tilde{a}$ or $a \in (\tilde{a}+1)$ yield 1 if it is satisfied and 0 otherwise:

$$W([G^e], t) := (dec(F_1^e \mid F_2^e))([G^e]) \qquad W(a, t) := a \in \tilde{a}$$
$$W(t, [G^e]) := (dec(Q^{\neq\nu}))([G^e]) \qquad W(t, a) := a \in (\tilde{a}+1).$$

The initial marking is the disjoint union $M_0 := M_0^{\mathcal{P}} \uplus M_0^{\mathcal{N}}$. Since name places receive a single token, we define $M_0^{\mathcal{N}}$ by the set of marked places:

$$M_0^{\mathcal{P}} := dec(P_0^{\neq\nu}) \qquad M_0^{\mathcal{N}} := (\{a_0 \in S\} \setminus \tilde{a}_0) \cup (\tilde{a}_0 + 1).$$

This definition in fact mirrors the name-aware transition system.

Lemma 2. *For every process $P \in \mathcal{P}$ with $sf(P) = \nu\tilde{a}.P^{\neq\nu}$ the bisimilarity $\mathcal{T}(\mathcal{N}_{\mathcal{C}}[\![P]\!]) \approx \mathcal{T}_{na}(P^{\neq\nu}, \tilde{a})$ holds.*

Proof. The lemma can be established by showing that

$$\mathcal{R} := \{(M^{\mathcal{P}} \uplus M^{\mathcal{N}}, ([Q^{\neq\nu}], \tilde{b})) \mid Q^{\neq\nu} \equiv \Pi_{[F^e] \in supp(M^{\mathcal{P}})} \Pi^{M^{\mathcal{P}}([F^e])} F^e \text{ and } \tilde{b} = \{b_i \in S \mid M^{\mathcal{N}}(b_k) = 1 \text{ with } i < k\}\}$$

is a bisimulation relation that connects the initial state of the name-aware transition system and the initial marking of the concurrency semantics.

By transitivity of bisimilarity, the Lemmas 1 and 2 prove our first main result. The transition system of a process and that of its concurrency semantics are bisimilar and the reachable processes can be recomputed from the markings using the composed bisimulation relation.

Theorem 1 (Retrievability). *For every $P \in \mathcal{P}$ we have $\mathcal{T}(\mathcal{N}_{\mathcal{C}}[\![P]\!]) \approx \mathcal{T}(P)$.*

Our second result is a finiteness characterisation. The concurrency semantics is a finite Petri net if and only if the process generates finitely many restricted names. We call those processes *restriction bounded*. Since in the standard transition system unused restrictions can be removed by $\nu a.P \equiv P$ if $a \notin fn(P)$, we again rely on the name-aware transition system to define restriction boundedness.

Definition 1. *Consider $P \in \mathcal{P}$ with $sf(P) = \nu\tilde{a}.P^{\neq\nu}$. We call P* restriction bounded *if there is a finite set of names $\tilde{m} \subseteq \mathcal{N}$ so that for every reachable name-aware process $(Q^{\neq\nu}, \tilde{b})$ the inclusion $\tilde{b} \subseteq \tilde{m}$ holds.*

If the process is not restriction bounded, clearly the concurrency semantics is infinite as every restricted name yields a place. Theorem 2 shows that also the reverse holds, i.e., a bounded number of restricted names implies finiteness of the Petri net. The proof uses the theory of derivatives [Mey09, Proposition 3].

Theorem 2 (Finiteness Characterisation). *For any process $P \in \mathcal{P}$, the concurrency semantics $\mathcal{N}_{\mathcal{C}}[\![P]\!]$ is finite if and only if P is restriction bounded.*

The Examples 1 and 2 show that structurally stationary and restriction-bounded processes are incomparable. Section 4 explains how to unify the classes.

4 Combining Structural and Concurrency Semantics

To combine the structural and the concurrency semantics we type restricted names, and the type determines the semantics that handles a name. More precisely, restricted names νa may carry a tag $\mathcal{C}$. Tagged names $\nu a^{\mathcal{C}}$ are translated by the concurrency semantics while names νa without tag are handled by the structural semantics. Hence, tagged names yield name places in the combined semantics and untagged names form fragments that replace the process places. Like in the concurrency semantics, we assume that tagged names have indices.

Technically, the idea of adding tags raises two problems that need to be addressed. (1) We need to define a name-aware transition system to generate tagged names systematically. (2) We need to compute the fragments formed by untagged names. The solution to both problems is a normal form for processes, which combines the standard and the restricted form. Before we turn to its definition, we illustrate the use of tags on our running example.

Example 3. Tagging the names in_0 and out_0 in the bag example yields process $\nu in_0^{\mathcal{C}}.\nu out_0^{\mathcal{C}}.(FILL\lfloor in_0^{\mathcal{C}}\rfloor \mid BAG\lfloor in_0^{\mathcal{C}}, out_0^{\mathcal{C}}\rfloor)$ with the equations in Example 1. Since channel $out_0^{\mathcal{C}}$ is shared by arbitrarily many processes, we tag it to treat it by the concurrency semantics. Vice versa, as arbitrarily many instances of *val* are created we omit the tag to handle the name by the structural semantics.

The mixed translation will result in the Petri net in Figure 2 with the name places $s_6 := in_0^{\mathcal{C}}$, $s_7 := in_1^{\mathcal{C}}$, $s_8 := out_0^{\mathcal{C}}$, and $s_9 := out_1^{\mathcal{C}}$. Different from the concurrency semantics, the mixed semantics has *fragment places*

$$s_1 := [FILL\lfloor in_0^{\mathcal{C}}\rfloor] \qquad s_2 := [BAG\lfloor in_0^{\mathcal{C}}, out_0^{\mathcal{C}}\rfloor] \qquad s_5 := [\nu val.\overline{out_0^{\mathcal{C}}}\langle val\rangle]$$
$$s_3 := [\nu val.\overline{in_0^{\mathcal{C}}}\langle val\rangle.FILL\lfloor in_0^{\mathcal{C}}\rfloor] \qquad s_4 := [in_0^{\mathcal{C}}(y).(\overline{out_0^{\mathcal{C}}}\langle y\rangle \mid BAG\lfloor in_0^{\mathcal{C}}, out_0^{\mathcal{C}}\rfloor)].$$

Observe that neither the structural nor the concurrency semantics finitely represent the process. It is also worth comparing the places with those in the Examples 1 and 2 to see how the mixed semantics unifies the previous translations.

4.1 Mixed Normal Form

The idea of the mixed normal form is to *maximise* the scopes of tagged active restrictions $\nu a^{\mathcal{C}}$ and to *minimise* the scopes of untagged ones νa. In the resulting process $P^{mf} = \nu\tilde{a}^{\mathcal{C}}.P^{rf}$ the tagged names $\nu\tilde{a}^{\mathcal{C}}$ surround a process P^{rf} in restricted form, which only contains untagged active restrictions. We call P^{mf} a process *in mixed normal form* and denote the *set of all processes in mixed normal form* by $\mathcal{P}_{mf}$. To give an example, process

$$P = \nu in_0^{\mathcal{C}}.\nu out_0^{\mathcal{C}}.(FILL\lfloor in_0^{\mathcal{C}}\rfloor \mid BAG\lfloor in_0^{\mathcal{C}}, out_0^{\mathcal{C}}\rfloor \mid \overline{out_0^{\mathcal{C}}}\langle val\rangle)$$

is in mixed normal form while $\nu val.P$ is not as the scope of νval is not minimal.

For every tagged process, the function $mf : \mathcal{P} \to \mathcal{P}_{mf}$ computes a structurally congruent process in mixed normal form. Empty sums $M^{=\mathbf{0}}$ are mapped to $\mathbf{0}$

and sequential processes are left unchanged. For the parallel composition $P \mid Q$, we recursively compute the mixed normal forms $mf(P) = \nu\tilde{a}_P^{\mathcal{C}}.P^{rf}$ and $mf(Q) = \nu\tilde{a}_Q^{\mathcal{C}}.Q^{rf}$, where $\tilde{a}_P^{\mathcal{C}}$ and $\tilde{a}_Q^{\mathcal{C}}$ may be empty. Like the standard form, the mixed normal form extrudes the scopes of $\tilde{a}_P^{\mathcal{C}}$ and $\tilde{a}_Q^{\mathcal{C}}$,

$$mf(P \mid Q) := \nu\tilde{a}_P^{\mathcal{C}}.\nu\tilde{a}_Q^{\mathcal{C}}.(P^{rf} \mid Q^{rf}).$$

Tagged active restricted names are handled like in the standard form, i.e., we have $mf(\nu a^{\mathcal{C}}.P) := \nu a^{\mathcal{C}}.mf(P)$ if $a^{\mathcal{C}} \in fn(P)$ and $mf(\nu a^{\mathcal{C}}.P) := mf(P)$ otherwise. For a process $\nu a.P$ with an untagged name νa, we recursively compute $mf(P) = \nu\tilde{a}^{\mathcal{C}}.P^{rf}$. This singles out the tagged names $\tilde{a}^{\mathcal{C}}$ in P. Commuting νa with $\nu\tilde{a}^{\mathcal{C}}$ yields $\nu\tilde{a}^{\mathcal{C}}.\nu a.P^{rf}$. Let $P^{rf} = \Pi_{i\in I}F_i$ and let I_a contain the indices of the fragments that have a as a free name. Shrinking the scope of νa gives

$$mf(\nu a.P) := \nu\tilde{a}^{\mathcal{C}}.\left(\nu a.(\Pi_{i\in I_a}F_i) \mid \Pi_{i\in I\setminus I_a}F_i\right).$$

Example 4. We observed that $\nu val.P \notin \mathcal{P}_{mf}$. An application of the function gives $mf(\nu val.P) = \nu in_0^{\mathcal{C}}.\nu out_0^{\mathcal{C}}.(\nu val.\overline{out_0^{\mathcal{C}}}\langle val\rangle \mid FILL\lfloor in_0^{\mathcal{C}}\rfloor \mid BAG\lfloor in_0^{\mathcal{C}}, out_0^{\mathcal{C}}\rfloor)$, which is a process in mixed normal form.

Lemma 3. *For every tagged process $P \in \mathcal{P}$ function mf yields $mf(P) \in \mathcal{P}_{mf}$ with $mf(P) \equiv P$. For $P^{mf} \in \mathcal{P}_{mf}$, even $mf(P^{mf}) = P^{mf}$ holds.*

4.2 Mixed Semantics

Like the concurrency semantics, the definition of the mixed semantics relies on a name-aware transition system that organises the creation of fresh tagged active restrictions. With the mixed normal form $\nu\tilde{a}^{\mathcal{C}}.P^{rf}$ in mind, we define the new *name-aware processes* to be pairs $(P^{rf}, \tilde{a}^{\mathcal{C}})$, where the active restrictions in P^{rf} are untagged. For these name-aware processes, the *name-aware reaction relation* $\rightarrow^{na}$ is adapted accordingly. The only difference to Definition (♣) is the use of the mixed normal form instead of the standard form:

$$(P^{rf}, \tilde{a}^{\mathcal{C}}) \rightarrow^{na} (Q^{rf}, \tilde{a}^{\mathcal{C}} \uplus \tilde{b}^{\mathcal{C}}) :\Leftrightarrow \begin{array}{l} P^{rf} \rightarrow \nu\tilde{b}^{\mathcal{C}}.Q^{rf} \text{ in mixed normal form and} \\ \forall b_k^{\mathcal{C}} \in \tilde{b}^{\mathcal{C}} : k-1 = max\{i \mid b_i^{\mathcal{C}} \in \tilde{a}^{\mathcal{C}}\}. \end{array}$$

Without change of notation, let the resulting new name-aware transition system be $\mathcal{T}_{na}(P^{rf}, \tilde{a}^{\mathcal{C}})$. It is straightforward to modify the proof of Lemma 1 in order to show bisimilarity for the adjusted definition.

Lemma 4. *For every tagged process $P \in \mathcal{P}$ with $mf(P) = \nu\tilde{a}^{\mathcal{C}}.P^{rf}$ the bisimilarity $\mathcal{T}_{na}(P^{rf}, \tilde{a}^{\mathcal{C}}) \approx \mathcal{T}(P)$ holds.*

The mixed semantics is a variant of the concurrency semantics, so again we have two sets of places. While the tagged active restrictions $\tilde{b}^{\mathcal{C}}$ in name-aware processes $([Q^{rf}], \tilde{b}^{\mathcal{C}})$ yield *name places*, the process places known from the concurrency semantics are replaced by *fragment places* $fg(Q^{rf})/_{\equiv}$ in the mixed semantics. They represent the fragments generated by untagged active restrictions. Also the transitions are changed to the form

$$t = ([F], \tilde{b}^{\mathcal{C}}, [Q^{rf}]) \quad \text{and} \quad t = ([F_1 \mid F_2], \tilde{b}^{\mathcal{C}}, [Q^{rf}]),$$

with the condition that F (and $F_1 \mid F_2$) has a name-aware reaction to the process $\nu\tilde{b}^{\mathcal{C}}.Q^{rf}$ in mixed normal form, i.e., $F \rightarrow^{na} \nu\tilde{b}^{\mathcal{C}}.Q^{rf}$ and $\nu\tilde{b}^{\mathcal{C}}.Q^{rf} \in \mathcal{P}_{mf}$. Intuitively, the transition models a communication of processes (or a τ-action or a call to an identifier) within F, which results in process Q^{rf} and generates the new tagged active restrictions $\tilde{b}^{\mathcal{C}}$. The modification of the weight function is immediate. We denote the *mixed semantics* of a process $P \in \mathcal{P}$ by $\mathcal{N}_{\mathcal{M}}[\![P]\!]$. Also the proof of bisimilarity in Lemma 2 still holds for the mixed semantics.

Lemma 5. *Consider the tagged process $P \in \mathcal{P}$ with $mf(P) = \nu\tilde{a}^{\mathcal{C}}.P^{rf}$. We have the bisimilarity $\mathcal{T}(\mathcal{N}_{\mathcal{M}}[\![P]\!]) \approx \mathcal{T}_{na}(P^{rf}, \tilde{a}^{\mathcal{C}})$.*

Combining the bisimilarities in Lemma 4 and 5, we obtain bisimilarity for the mixed semantics. Moreover, the bisimulation relation used in the proof allows one to reconstruct the reachable process terms from the markings.

Theorem 3 (Retrievability). *For every tagged process $P \in \mathcal{P}$ the bisimilarity $\mathcal{T}(\mathcal{N}_{\mathcal{M}}[\![P]\!]) \approx \mathcal{T}(P)$ holds.*

For processes *without tagged names* the mixed semantics degenerates to the structural semantics. The absence of tagged names leads to absence of name places in the semantics. Hence, transitions do not generate names and have the form $([F], \emptyset, [Q^{rf}])$ or $([F_1 \mid F_2], \emptyset, [Q^{rf}])$. They can be identified with the transitions $([F], [Q])$ and $([F_1 \mid F_2], [Q])$ in the structural semantics. In case *all names are tagged* in the process under consideration, the mixed semantics corresponds to the concurrency semantics. This follows from the fact that the mixed normal form coincides with the standard form for these processes. Hence, the fragment places in the mixed semantics are in fact sequential processes.

Proposition 1 (Conservative Extension). *If the process $P \in \mathcal{P}$ does not use tagged names, the mixed semantics coincides with the structural semantics, i.e., $\mathcal{N}_{\mathcal{M}}[\![P]\!] = \mathcal{N}_{\mathcal{S}}[\![P]\!]$. If the process only uses tagged names, the mixed semantics coincides with the concurrency semantics, i.e., $\mathcal{N}_{\mathcal{M}}[\![P]\!] = \mathcal{N}_{\mathcal{C}}[\![P]\!]$.*

According to Theorem 2 the concurrency semantics is finite if and only if the translated process is restriction bounded. The structural semantics is finite precisely if there is a finite set of fragments every reachable process consists of (structural stationarity [Mey09]). We prove the mixed semantics to be finite if and only if (a) finitely many tagged names are generated and (b) the untagged names form finitely many fragments.

Definition 2. *A tagged process $P \in \mathcal{P}$ is* mixed bounded *if there are finite sets of names $\tilde{m}^{\mathcal{C}}$ and fragments $\{F_1, \ldots, F_n\}$ so that for every reachable process $(Q^{rf}, \tilde{b}^{\mathcal{C}})$ we have $\tilde{b}^{\mathcal{C}} \subseteq \tilde{m}^{\mathcal{C}}$ and for every $F \in fg(Q^{rf})$ there is F_i with $F \equiv F_i$.*

To see that $\mathcal{N}_{\mathcal{M}}[\![P]\!]$ is finite iff P is mixed bounded, observe that finiteness of the set of places implies finiteness of the mixed semantics. Finiteness of the set of name places is equivalent to Condition (a), finiteness of the set of fragment places equivalent to Condition (b) in the definition of mixed boundedness.

Theorem 4 (Finiteness Characterisation). *For every tagged process $P \in \mathcal{P}$ the mixed semantics $\mathcal{N}_{\mathcal{M}}[\![P]\!]$ is finite if and only if P is mixed bounded.*

Structurally stationary and restriction-bounded processes are mixed bounded, hence the mixed semantics finitely represents all their *syntactic* subclasses. In *finite control processes* parallel compositions are forbidden within recursions, but an unbounded number of restricted names may be generated [Dam96]. Incomparable, *restriction-free processes* allow for unbounded parallelism but forbid the use of restrictions [AM02]. They are generalised by *finite handler processes* designed for modelling client-server systems [Mey09]. All these classes are structurally stationary [Mey09]. Restriction-free processes are also generalised by *finite-net processes*, which forbid the use of restrictions within recursions but allow for unbounded parallelism (dual to finite control processes) [BG09]. They form a subclass of restriction-bounded processes.

We conclude the section with a remark that mixed-bounded processes are a subclass of the processes of bounded depth.

Proposition 2. *If $P \in \mathcal{P}$ is mixed bounded, then it is bounded in depth.*

Theorem 4 shows that if a process is mixed bounded then *there is* a faithful representation as a finite p/t net. We now consider the reverse direction.

5 Borderline to Finite P/T Petri Nets

We argue that if we have a superclass of mixed-bounded processes, there will be no reachability-preserving translation into finite p/t Petri nets. This means mixed-bounded processes form the borderline between π-calculus and finite p/t nets. Since it is always possible to handle particular classes of processes by specialised translations, we make our argument precise. We show that in a *slight extension* of mixed-bounded processes reachability becomes undecidable. Since the problem is decidable for finite p/t nets [May84], there is no reachability-preserving translation for the extended process class.

The processes we consider are *bounded in depth by one*. To establish undecidability of reachability, we reduce the corresponding problem for 2-counter machines [Min67]. Due to the limitations of the process class, the counter machine encoding differs drastically from those in the literature [Mil89, AM02, BGZ03] modelling counters as stacks. The only related encoding that does not rely on stacks is given in [BGZ04] to prove weak bisimilarity undecidable for $\mathrm{CCS}_!$.

We imitate a construction in [DFS98] which shows undecidability of reachability for transfer nets. The idea of Dufourd, Finkel, and Schnoebelen is to represent a counter c_1 by two places c_1 and c_1'. The test for zero

$$l : \text{if } c_1 = 0 \text{ then goto } l';\ \text{else } c_1 := c_1 - 1;\ \text{goto } l''; \qquad (\spadesuit)$$

is modelled by the transfer net in Figure 3. To test counter c_1 for being zero, transition t transfers the content of c_1' to a trash place s_t. Since the transition is enabled although c_1' is empty, the content of c_1 and c_1' coincides as long as the net properly simulates the counter machine. If a transfer operation is executed when c_1' is not empty, the amount of tokens in c_1 and c_1' becomes different, Figure 3 (a) and (b). The difference is preserved throughout the computation, because increment operations add the same amount of tokens to c_1 and c_1'. Hence, a state $(c_1 = v_1, c_2 = v_2, l)$ with $v_1, v_2 \in \mathbb{N}$ is reachable in the counter machine if and only if a marking is reachable in the transfer net where place l is marked, counter c_1 and its copy c_1' carry v_1 tokens, and similar for c_2 and c_2'.

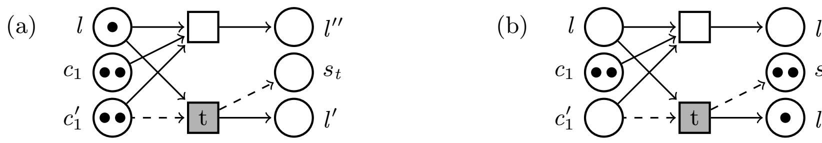


Fig. 3. A Petri net with transfer modelling a test for zero in a counter machine. Dashed lines represent transfer arcs of t that move all tokens in c_1' to the trash place s_t.

We represent a counter value by a parallel composition of processes, e.g. $c_1' = 2$ by $\overline{a} \mid \overline{a}$. The transfer operation requires us to change arbitrarily many processes with one communication. To achieve this, we attach the processes $\overline{a}$ to a so-called *process bunch* $PB\lfloor a, i_{c_1'}, d_{c_1'}, t_{c_1'}\rfloor$ by restricting the name a. The result is a process $\nu a.(PB\lfloor a, i_{c_1'}, d_{c_1'}, t_{c_1'}\rfloor \mid \overline{a} \mid \overline{a})$. Since the name a is restricted, the process bunch has exclusive access to its processes $\overline{a}$. It offers three operations to modify their numbers. A communication on $i_{c_1'}$ stands for *increment* and creates a new process $\overline{a}$. Similarly, a message on $d_{c_1'}$ *decrements* the process number by consuming a process $\overline{a}$. A *test for zero* on $t_{c_1'}$ creates a new and empty process bunch for counter c_1'. The old process bunch terminates. A process $\nu a.(\overline{a} \mid \overline{a})$ without process bunch is considered to belong to the trash place. Abbreviating d_x, i_x, t_x by $\tilde{c}_x$, a process bunch is defined by

$$PB(a, \tilde{c}_x) := i_x.(PB\lfloor a, \tilde{c}_x\rfloor \mid \overline{a}) + d_x.a.PB\lfloor a, \tilde{c}_x\rfloor + t_x.\nu b.PB\lfloor b, \tilde{c}_x\rfloor.$$

Labelled instructions $l : inst$ of the counter machine are translated to process identifiers K_l where $inst$ determines the defining process. The increment operation $l : c_1 := c_1 + 1$ goto l' yields $K_l(\tilde{c}) := \overline{i_{c_1}}.\overline{i_{c_1'}}.K_{l'}\lfloor\tilde{c}\rfloor$. Here, $\tilde{c}$ abbreviates the channels of all four process bunches $d_{c_1}, i_{c_1}, t_{c_1}, d_{c_1'}, \ldots, t_{c_2'}$. Note that both, c_1 and c_1', are incremented. Like in transfer nets, the test for zero (♠) only changes the value of counter c_1'. Decrement acts on both counters:

$$K_l(\tilde{c}) := \overline{t_{c_1'}}.K_{l'}\lfloor\tilde{c}\rfloor + \overline{d_{c_1}}.\overline{d_{c_1'}}.K_{l''}\lfloor\tilde{c}\rfloor.$$

If an empty process bunch accepts a decrement, the system deadlocks (requires synchronisation actions that we omitted here to ease presentation) and reachability is preserved. Finally, a halt instruction $l : halt$ is translated into $K_l(\tilde{c}) := \overline{halt}$. The full translation of a counter machine CM yields the process

$$\mathcal{P}[\![CM]\!] := \mathop{\Pi}_{x \in \{c_1, \ldots, c_2'\}} \nu a_x . PB\lfloor a_x, \tilde{c}_x \rfloor \mid K_{l_0}\lfloor \tilde{c} \rfloor.$$

The counter machine CM reaches the state $(c_1 = v_1, c_2 = v_2, l)$ if and only if its encoding $\mathcal{P}[\![CM]\!]$ reaches the process

$$\mathop{\Pi}_{x \in \{c_1, c_1'\}} \nu a_x . (PB\lfloor a_x, \tilde{c}_x \rfloor \mid \Pi^{v_1} \overline{a_x}) \mid \mathop{\Pi}_{x \in \{c_2, c_2'\}} \nu a_x . (PB\lfloor a_x, \tilde{c}_x \rfloor \mid \Pi^{v_2} \overline{a_x}) \mid K_l\lfloor \tilde{c} \rfloor.$$

The leftmost parallel composition ensures that the process bunches for c_1 and c_1' contain v_1 processes $\overline{a_{c_1}}$ and $\overline{a_{c_1'}}$, respectively. The construction for c_2 and c_2' is similar. Combined with the observation that the process $\mathcal{P}[\![CM]\!]$ is bounded in depth by one, we arrive at the desired undecidability result.

Theorem 5 (Undecidability in Depth One). *Consider $P, Q \in \mathcal{P}$ where the depth is bounded by one. The problem whether $Q \in Reach(P)$ is undecidable.*

Since reachability is decidable for finite p/t nets [May84], there does not exist a *reachability-preserving* translation into finite p/t nets for any class of processes subsuming those of depth one.

We argue that any reasonable extension of mixed-bounded processes will already subsume those of depth one. Reconsider the counter machine encoding, it exploits two features that mixed-bounded processes are forbidden to combine. First, in a process bunch $\nu a.(PB\lfloor a, \tilde{c}_x \rfloor \mid \overline{a} \mid \overline{a})$ the number of processes $\overline{a}$ under the restriction may be unbounded. Second, arbitrarily many instances of νa may be generated. If either of the conditions is dropped, the resulting process is mixed bounded. In case νa is shared by a bounded number of processes $\overline{a}$, we translate the name by the structural semantics. If finitely many instances of νa are generated, we use the concurrency semantics (cf. Examples 1 and 2). Hence, mixed-bounded processes form the borderline to finite p/t nets.

6 Discussion

Combining the structural semantics in [Mey09] and a new concurrency semantics yields a mixed semantics that finitely represents the mixed-bounded processes (Theorem 4). They generalise (Proposition 1) structurally stationary and restriction-bounded processes, the latter are finitely represented by the new concurrency semantics (Theorem 2). As it is not possible to extend mixed-bounded processes without losing reachability (Theorem 5), the class defines the borderline to finite p/t nets. Since mixed-bounded processes are bounded in depth (Proposition 2), also this class is more expressive than finite p/t nets, Figure 1.

We use a π-calculus with guarded choice and step-unwinding recursion. The former permits an elegant definition of fragments and the latter gives us decidability of structural congruence. However, both restrictions do not delimit the computational expressiveness of the π-calculus, which is the focus of the paper, but are made for technical convenience.

The definition of the mixed semantics relies on a typing mechanism for restricted names. In our tool Petruchio [Pet08], an approximate algorithm infers the types automatically. They need not be given by the user.

Finally, our implementation does not rely on the often infinite name-aware transition system, but computes the mixed semantics as a least fixed point on the set of Petri nets. Starting with the initially marked places it adds transitions and places where appropriate. A coverability graph allows us to compute the simultaneously markable places, and we currently experiment with more efficient algorithms. The compilation terminates iff the process is mixed bounded.

References

[AM02] Amadio, R., Meyssonnier, C.: On decidability of the control reachability problem in the asynchronous π-calculus. Nord. J. Comp. 9(1), 70–101 (2002)

[BG95] Busi, N., Gorrieri, R.: A Petri net semantics for π-calculus. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 145–159. Springer, Heidelberg (1995)

[BG09] Busi, N., Gorrieri, R.: Distributed semantics for the π-calculus based on Petri nets with inhibitor arcs. J. Log. Alg. Prog. 78(1), 138–162 (2009)

[BGZ03] Busi, N., Gabbrielli, M., Zavattaro, G.: Replication vs. recursive definitions in channel based calculi. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 133–144. Springer, Heidelberg (2003)

[BGZ04] Busi, N., Gabbrielli, M., Zavattaro, G.: Comparing recursion, replication, and iteration in process calculi. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 307–319. Springer, Heidelberg (2004)

[Dam96] Dam, M.: Model checking mobile processes. Inf. Comp. 129(1), 35–51 (1996)

[DFS98] Dufourd, C., Finkel, A., Schnoebelen, P.: Reset nets between decidability and undecidability. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998)

[DKK08] Devillers, R., Klaudel, H., Koutny, M.: A compositional Petri net translation of general π-calculus terms. For. Asp. Comp. 20(4–5), 429–450 (2008)

[EG99] Engelfriet, J., Gelsema, T.: Multisets and structural congruence of the pi-calculus with replication. Theor. Comp. Sci. 211(1-2), 311–337 (1999)

[EG04] Engelfriet, J., Gelsema, T.: A new natural structural congruence in the pi-calculus with replication. Acta Inf. 40(6), 385–430 (2004)

[Eng96] Engelfriet, J.: A multiset semantics for the pi-calculus with replication. Theor. Comp. Sci. 153(1-2), 65–94 (1996)

[FGMP03] Ferrari, G.-L., Gnesi, S., Montanari, U., Pistore, M.: A model-checking verification environment for mobile processes. ACM Trans. Softw. Eng. Methodol. 12(4), 440–473 (2003)

[Fok07] Fokkink, W.: Modelling Distributed Systems. Springer, Heidelberg (2007)

[KKN06] Khomenko, V., Koutny, M., Niaouris, A.: Applying Petri net unfoldings for verification of mobile systems. In: Proc. of MOCA, Bericht FBI-HH-B-267/06, pp. 161–178. University of Hamburg (2006)

[May84] Mayr, E.W.: An algorithm for the general Petri net reachability problem. SIAM J. Comp. 13(3), 441–460 (1984)

[Mey08a] Meyer, R.: On boundedness in depth in the π-calculus. In: Proc. of IFIP TCS. IFIP, vol. 273, pp. 477–489. Springer, Heidelberg (2008)

[Mey08b] Meyer, R.: Structural Stationarity in the π-calculus. PhD thesis, Department of Computing Science, University of Oldenburg (2008)

[Mey09] Meyer, R.: A theory of structural stationarity in the π-calculus. Acta Inf. 46(2), 87–137 (2009)

[Mil89] Milner, R.: Communication and concurrency. Prentice Hall, Englewood Cliffs (1989)

[Mil99] Milner, R.: Communicating and Mobile Systems: the π-Calculus. CUP, Cambridge (1999)

[Min67] Minsky, M.: Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs (1967)

[MKS09] Meyer, R., Khomenko, V., Strazny, T.: A practical approach to verification of mobile systems using net unfoldings. Fund. Inf. (to appear 2009)

[MP95] Montanari, U., Pistore, M.: Checking bisimilarity for finitary π-calculus. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 42–56. Springer, Heidelberg (1995)

[Pet08] Petruchio (2008), `http://petruchio.informatik.uni-oldenburg.de`

[Pis99] Pistore, M.: History Dependent Automata. PhD thesis, Dipartimento di Informatica, Università di Pisa (1999)

[SW01] Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. CUP, Cambridge (2001)

Modeling Radio Networks*

Calvin Newport and Nancy Lynch

MIT CSAIL, Cambridge, MA
{cnewport,lynch}@csail.mit.edu

Abstract. We describe a modeling framework and collection of foundational composition results for the study of probabilistic distributed algorithms in synchronous radio networks. Existing results in this setting rely on informal descriptions of the channel behavior and therefore lack easy comparability and are prone to error caused by definition subtleties. Our framework rectifies these issues by providing: (1) a method to precisely describe a radio channel as a probabilistic automaton; (2) a mathematical notion of implementing one channel using another channel, allowing for direct comparisons of channel strengths and a natural decomposition of problems into implementing a more powerful channel and solving the problem on the powerful channel; (3) a mathematical definition of a problem and solving a problem; (4) a pair of composition results that simplify the tasks of proving properties about channel implementation algorithms and combining problems with channel implementations. Our goal is to produce a model streamlined for the needs of the radio network algorithms community.

1 Introduction

In this paper we describe a modeling framework, including a collection of foundational composition results, for the study and comparison of distributed algorithms in synchronous radio networks. In the two decades that followed the deployment of *AlohaNet* [1]—the first radio data network—theoreticians invested serious effort in the study of distributed algorithms in this setting; c.f., [2,3,4]. This early research focused on the stability of ALOHA-style MAC layers under varying packet arrival rates. In a seminal 1992 paper, Bar-Yehuda, Goldreich, and Itai (BGI) [5] ushered in the modern era of radio network analysis by introducing a synchronous multihop model and a more general class of problems, such as reliable broadcast. Variants of this model have been studied extensively in the intervening years; c.f., [6,7,8]. Numerous workshops and conferences are now dedicated exclusively to radio network algorithms–e.g., POMC, ADHOCNETS—and all major distributed algorithms conference have sessions dedicated to the topic. In short, distributed algorithms for radio networks is an important and well-established field.

* This work has been support in part by Cisco-Lehman CUNY A New MAC-Layer Paradigm for Mobile Ad-Hoc Networks, AFOSR Award Number FA9550-08-1-0159, NSF Award Number CCF-0726514, and NSF Award Number CNS-0715397.

The vast majority of existing theory concerning radio networks, however, relies on informal English descriptions of the communication model (e.g., "If two or more processes broadcast at the same time then..."). This lack of formal rigor can generate subtle errors. For example, the original BGI paper [5] claimed a $\Omega(n)$ lower bound for multihop broadcast in small diameter graphs. It was subsequently discovered that due to a small ambiguity in how they described the collision behavior (whether or not a message *might* be received from among several that collide at a receiver), the bound is actually logarithmic [9]. In our work on consensus [10], for another example, subtleties in how the model treated transmitters receiving their own messages—a detail often omitted in informal model descriptions—induced a non-trivial impact on the achievable lower bounds. And so on. We also note that informal model descriptions prevent comparability between different results. Given two such descriptions, it is often difficult to infer whether one model is strictly stronger than the other or if the pair is incomparable. And without an agreed definition of what it means to implement one channel with another, algorithm designers are denied the ability to build upon existing results to avoid having to resolve problems in every model variant.

In this paper we describe a modeling framework that addresses these issues. Specifically, we use probabilistic automata to describe executions of distributed algorithms in a *synchronous* radio network.[1] (We were faced with the decision of whether to build a custom framework or use an existing formalism for modeling probabilistic distributed algorithms, such as [11,12,13]. We opted for the custom approach as we focus on the restricted case of synchronous executions of a fixed set of components. We do not the need the full power of general models which, among other things, must reconcile the nondeterminism of asynchrony with the probabilistic behavior of the system components.)

In our framework: The radio network is described by a channel automaton; the algorithm is described by a collection of n process automata; and the environment—which interacts with the processes through input and output ports—is described by its own automaton. In addition to the basic system model, we present a rigorous definition of a problem and solving a problem, and cast the task of implementing one channel with another as a special case of solving a problem. We then describe two foundational composition results. The first shows how to compose an algorithm that solves a problem P using channel C_1 with an algorithm that implements $\mathcal{C}_1$ using channel $\mathcal{C}_2$. We prove the resulting composition solves P using $\mathcal{C}_2$. (The result is also generalized to work with a chain of channel implementation algorithms.) The second composition result shows how to compose a channel implementation algorithm $\mathcal{A}$ with a channel $\mathcal{C}$ to generate a new channel $\mathcal{C}'$. We prove that $\mathcal{A}$ using $\mathcal{C}$ implements $\mathcal{C}'$. This result is useful for proving properties about a channel implementation algorithm such as $\mathcal{A}$. We conclude with a case study that demonstrates the framework and the composition theorems in action.

[1] We restrict our attention to synchronous settings as the vast majority of existing theoretical results for radio networks share this assumption. A more general *asynchronous* model remains important future work.

2 Model

We model n processes that operate in synchronized time slots and communicate on a radio network comprised of $\mathcal{F}$ independent communication frequencies. The processes can also receive inputs from and send outputs to an environment. We formalize this setting with automata definitions. Specifically, we use a probabilistic automaton for each of the n processes (which combine to form an *algorithm*), another to model the *environment*, and another to model the communication *channel*. A system is described by an algorithm, environment, and channel.

For any positive integer $x > 1$ we use the notation $[x]$ to refer to the integer set $\{1, ..., x\}$, and use S^x, for some set S, to describe all x-vectors with elements from S. Let $\mathcal{M}$, $\mathcal{R}$, $\mathcal{I}$, and $\mathcal{O}$ be four non-empty value sets that do not include the special placeholder value $\perp$. We use the notation $\mathcal{M}_\perp$, $\mathcal{R}_\perp$, $\mathcal{I}_\perp$, and $\mathcal{O}_\perp$ to describe the union of each of these sets with $\{\perp\}$. Finally, we fix n and $\mathcal{F}$ to be positive integers. They describe the number of processes and frequencies, respectively.

2.1 Systems

The primary object in our model is the *system*, which consists of an *environment* automaton, a *channel* automaton, and n process automata that combine to define an *algorithm*. We define each component below:

Definition 1 (Channel). *A* channel *is an automaton $\mathcal{C}$ consisting of the following components:*

- *$cstates_\mathcal{C}$, a potentially infinite set of* states*.*
- *$cstart_\mathcal{C}$, a state from $states_\mathcal{C}$ known as the* start state*.*
- *$crand_\mathcal{C}$, for each state $s \in cstates_\mathcal{C}$, a probability distribution over $cstates_\mathcal{C}$. (This distribution captures the probabilistic nature of the automaton. Both the environment and process definitions include similar distributions.)*
- *$crecv_\mathcal{C}$, a message set generation function that maps $cstates_\mathcal{C} \times \mathcal{M}_\perp^n \times [\mathcal{F}]^n$ to $\mathcal{R}_\perp^n$.*
- *$ctrans_\mathcal{C}$, a transition function that maps $cstates_\mathcal{C} \times \mathcal{M}_\perp^n \times [\mathcal{F}]^n$ to $cstates_\mathcal{C}$.*

Because we model a channel as an arbitrary automaton, we can capture a wide variety of possible channel behavior—from simple deterministic receive rules to complex, probabilistic multihop propagation.

We continue by the defining the elements of a system: an environment, process, and algorithm. We then define a system execution.

Definition 2 (Environment). *A environment is some automaton $\mathcal{E}$ consisting of the following components:*

- *$estates_\mathcal{E}$, a potentially infinite set of* states*.*
- *$estart_\mathcal{E}$, a state from $estates_\mathcal{E}$ known as the* start state*.*
- *$erand_\mathcal{E}$, for each state $s \in estates_\mathcal{E}$, a probability distribution over $estates_\mathcal{E}$.*
- *$ein_\mathcal{E}$, an* input generation function *that maps $estates_\mathcal{E}$ to $\mathcal{I}_\perp^n$.*
- *$etrans_\mathcal{E}$, a transition function that maps $estates_\mathcal{E} \times \mathcal{O}_\perp$ to $estates_\mathcal{E}$.*

Definition 3 (Process). *A* process *is some automaton $\mathcal{P}$ consisting of the following components:*

- *$states_{\mathcal{P}}$, a potentially infinite set of* states.
- *$rand_{\mathcal{P}}$, for each state $s \in states_{\mathcal{P}}$, is a probability distribution over $states_{\mathcal{P}}$.*
- *$start_{\mathcal{P}}$, a state from $states_{\mathcal{P}}$ known as the* start state.
- *$msg_{\mathcal{P}}$, a* message generation function *that maps $states_{\mathcal{P}} \times \mathcal{I}_{\perp}$ to $\mathcal{M}_{\perp}$.*
- *$out_{\mathcal{P}}$, an* output generation function *that maps $states_{\mathcal{P}} \times \mathcal{I}_{\perp} \times \mathcal{R}_{\perp}$ to $\mathcal{O}_{\perp}$.*
- *$freq_{\mathcal{P}}$, a* frequency selection function *that maps $states_{\mathcal{P}} \times \mathcal{I}_{\perp}$ to $[\mathcal{F}]$.*
- *$trans_{\mathcal{P}}$, a state transition function mapping $states_{\mathcal{P}} \times \mathcal{R}_{\perp} \times \mathcal{I}_{\perp}$ to $states_{\mathcal{P}}$.*

Definition 4 (Algorithm). *An algorithm $\mathcal{A}$ is a mapping from $[n]$ to processes.*

Definition 5 (System). *A system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$, consists of an environment $\mathcal{E}$, an algorithm $\mathcal{A}$, and a channel $\mathcal{C}$.*

Definition 6 (Execution). *An* execution *of a system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$ is an infinite sequence*

$$S_0, C_0, E_0, R_1^S, R_1^C, R_1^E, I_1, M_1, F_1, N_1, O_1, S_1, C_1, E_1, ...$$

where for all $r \geq 0$, S_r and R_r^S map each $i \in [n]$ to a process state from $\mathcal{A}(i)$, C_r and R_r^C are in $cstates_{\mathcal{C}}$, E_r and R_r^E are in $estates_{\mathcal{E}}$, M_r is in $\mathcal{M}_{\perp}^n$, F_r is in $[\mathcal{F}]^n$, N_r is in $\mathcal{R}_{\perp}^n$, I_r is in $\mathcal{I}_{\perp}^n$, and O_r is in $\mathcal{O}_{\perp}^n$. We assume the following constraints:

1. *$C_0 = cstart_{\mathcal{C}}$, $E_0 = estart_{\mathcal{E}}$, and $\forall i \in [n] : S_0[i] = start_{\mathcal{A}(i)}$.*
2. *For every round $r > 0$:*
   - *(a) $\forall i \in [n] : R_r^S[i]$ is selected according to distribution $rand_{\mathcal{A}(i)}(S_{r-1}[i])$, R_r^C is selected according to $crand_{\mathcal{C}}(C_{r-1})$, and R_r^E is selected according to $erand_{\mathcal{E}}(E_{r-1})$.*
   - *(b) $I_r = ein_{\mathcal{E}}(R_r^E)$.*
   - *(c) $\forall i \in [n] : M_r[i] = msg_{\mathcal{A}(i)}(R_r^S[i], I_r[i])$ and $F_r[i] = freq_{\mathcal{A}(i)}(R_r^S[i], I_r[i])$.*
   - *(d) $N_r = crecv_{\mathcal{C}}(R_r^C, M_r, F_r)$.*
   - *(e) $\forall i \in [n] : O_r[i] = out_{\mathcal{A}(i)}(R_r^S[i], I_r[i], N_r[i])$.*
   - *(f) $\forall i \in [n] : S_r[i] = trans_{\mathcal{A}(i)}(R_r^S[i], N_r[i], I_r[i])$, $C_r = ctrans_{\mathcal{C}}(R_r^C, M_r, F_r)$, and $E_r = etrans_{\mathcal{E}}(R_r^E, O_r)$.*

In each round: first the processes, environment, and channel transform their states (probabilistically); then the environment generates inputs to pass to the processes; then the processes each generate a message to send (or $\perp$ if they plan on receiving) and a frequency to use; then the channel returns the received messages to the processes; then the processes generate output values to pass back to the environment; and finally all automata transition to a new state.

Definition 7 (Execution Prefix). *An execution prefix of a system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$, is a finite prefix of some execution of the system. The prefix is either empty or ends with an environment state assignment E_r, $r \geq 0$. That is, it contains no partial rounds.*

2.2 Trace Probabilities

To capture the probability of various system behaviors we start by defining the function Q:

Definition 8 (Q). *For every system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$, and every execution prefix α of this system, $Q(\mathcal{E}, \mathcal{A}, \mathcal{C}, \alpha)$ describes the probability that $(\mathcal{E}, \mathcal{A}, \mathcal{C})$ generates α. That is, the product of the probabilities of state transitions in α as described by $rand_{\mathcal{A}}$, $crand_{\mathcal{C}}$, and $erand_{\mathcal{E}}$.*

Next, we define a *trace* to be a finite sequence of vectors from $\mathcal{I}_{\perp}^{n} \cup \mathcal{O}_{\perp}^{n}$; i.e., a sequences of inputs and outputs passed between an algorithm and an environment. And we use T to describe the set of all traces Using Q, we can define functions that return the probability that a system generates a given trace. First, however, we need a collection of helper definitions to extract traces from prefixes. Specifically, the function *io* maps an execution prefix to the subsequence consisting only of the $\mathcal{I}_{\perp}^{n}$ and $\mathcal{O}_{\perp}^{n}$ vectors. The function *cio*, by contrast, maps an execution prefix α to $io(\alpha)$ with all $\perp^n$ vectors removed. Finally, the predicate *term* returns *true* for an execution prefix α if and only if the output vector in the final round of α is not $\perp^n$.

Definition 9 (D & D_{tf}). *For every system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$, and every trace β : $D(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) = \sum_{\alpha | io(\alpha)=\beta} Q(\mathcal{E}, \mathcal{A}, \mathcal{C}, \alpha)$ and*
$D_{tf}(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) = \sum_{\alpha | term(\alpha) \wedge cio(\alpha)=\beta} Q(\mathcal{E}, \mathcal{A}, \mathcal{C}, \alpha)$.

Both D and D_{tf} return the probability of a given system generating a given trace. The difference between D and D_{tf} is that the latter ignores *empty vectors*—that is, input or output vectors consisting only of $\perp$. (The tf indicates it is *time-free*; e.g., it ignores the time required between the generation of trace elements.)

2.3 Problems

We define a problem and provide two definitions of solving a problem—one that considers empty rounds (those with $\perp^n$) and one that does not. In the following, let E be the set of all possible environments.

Definition 10 (Problem). *A problem P is a function from environments to a set of functions from traces to probabilities.*

Definition 11 (Solves & Time-Free Solves). *We say algorithm $\mathcal{A}$ solves problem P using channel $\mathcal{C}$ if and only if $\forall \mathcal{E} \in E, \exists F \in P(\mathcal{E}), \forall \beta \in T$: $D(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) = F(\beta)$. We say $\mathcal{A}$* time-free *solves P using $\mathcal{C}$ if and only if: $\forall \mathcal{E} \in E, \exists F \in P(\mathcal{E}), \forall \beta \in T : D_{tf}(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) = F(\beta)$.*

For some of the proofs that follow, we need to restrict our attention to environments that are indifferent to delays.

Definition 12. *We say an environment $\mathcal{E}$ is delay tolerant if and only if for every state $s \in estates_{\mathcal{E}}$ and $\hat{s} = etrans_{\mathcal{E}}(s, \perp^n)$, the following conditions hold:*

1. *$ein_{\mathcal{E}}(\hat{s}) = \perp^n$.*
2. *$erand_{\mathcal{E}}(\hat{s})(\hat{s}) = 1$.*
3. *$etrans_{\mathcal{E}}(\hat{s}, \perp^n) = \hat{s}$.*
4. *for every non-empty output assignment O, $etrans_{\mathcal{E}}(\hat{s}, O) = etrans_{\mathcal{E}}(s, O)$.*

When a delay tolerant environment receives output $\perp^n$ in some state s, it transitions to a special *marked* version of the current state, denoted $\hat{s}$, and cycles on this state until it next receives a non-$\perp^n$ output. In other words, it behaves the same regardless of how many consecutive $\perp^n$ outputs it receives. We use this definition of a delay tolerant environment to define a delay tolerant problem.

Definition 13 (Delay Tolerant Problem). *We say a problem P is delay tolerant if and only if for every environment $\mathcal{E}$ that is not delay tolerant, $P(E)$ returns the set containing every trace probability function.*

3 Implementing Channels

Here we construct a precise notion of implementing a channel with another channel as a special case of a problem. In the following, we say an input value is *send enabled* if it is from $(send \times \mathcal{M}_\perp \times \mathcal{F})$. We say an *input assignment* (i.e., vector from $\mathcal{I}_\perp^n$) is send enabled if all inputs values in the assignment are send enabled. Similarly, we say an output value is *receive enabled* if it is from $(recv \times \mathcal{R}_\perp)$, and an *output assignment* (i.e., vector from $\mathcal{O}_\perp^n$) is receive enabled if all output values in the assignment are receive enabled. Finally, we say an input or output assignment is *empty* if it equals $\perp^n$

Definition 14 (Channel Environment). *An environment $\mathcal{E}$ is a channel environment if and only if it satisfies the following criteria: (1) It is delay tolerant; (2) it generates only send enabled and empty input assignments; and (3) it generates a send enabled input assignment in the first round and in every round $r > 1$ such that it received a receive enabled output vector in $r - 1$. In every other round it generates an empty input assignment.*

These constraints require the environment to pass down messages to send as inputs and then wait for the corresponding received messages, encoded as algorithm outputs, before continuing with the next batch messages to send. This formalism is used below in our definition of a channel problem. The natural pair to a channel environment is a channel algorithm, which behaves symmetrically.

Definition 15 (Channel Algorithm). *We say an algorithm $\mathcal{A}$ is a channel algorithm if and only if: (1) it only generates receive enabled and empty output assignments; (2) it never generates two consecutive received enabled output assignments without a send enabled input in between; and (3) given a send enabled input it eventually generates a receive enabled output.*

Definition 16 ($\mathcal{A}^I$). *Each process $\mathcal{P}$ of the channel identity algorithm $\mathcal{A}^I$ behaves as follows. If $\mathcal{P}$ receives a send enabled input $(send, m, f)$, it sends message m on frequency f during that round and generates output $(revc, m')$, where m' is the message it receives in this same round. Otherwise it sends $\perp$ on frequency 1 and generates output $\perp$.*

Definition 17 (Channel Problem). *For a given channel $\mathcal{C}$ we define the corresponding (channel) problem $P_{\mathcal{C}}$ as follows: $\forall \mathcal{E} \in E$, if $\mathcal{E}$ is a channel environment, then $P_{\mathcal{C}}(\mathcal{E}) = \{F\}$, where, $\forall \beta \in T : F(\beta) = D_{tf}(\mathcal{E}, \mathcal{A}^I, \mathcal{C}, \beta)$. If $\mathcal{E}$ is not a channel environment, then $P_{\mathcal{C}}(\mathcal{E})$ returns the set containing every trace probability function.*

The effect of combining $\mathcal{E}$ with $\mathcal{A}^I$ and $\mathcal{C}$ is to *connect* $\mathcal{E}$ directly with $\mathcal{C}$. With the channel problem defined, we can conclude with what it means for an algorithm to implement a channel.

Definition 18 (Implements). *We say an algorithm $\mathcal{A}$ implements a channel $\mathcal{C}$ using channel $\mathcal{C}'$ only if $\mathcal{A}$ time-free solves $P_{\mathcal{C}}$ using $\mathcal{C}'$.*

4 Composition

We prove two useful composition results. The first simplifies the task of solving a complex problem on a weak channel into implementing a strong channel using a weak channel, then solving the problem on the strong channel. The second result simplifies proofs that require us to show that the channel implemented by a channel algorithm satisfies given automaton constraints.

4.1 The Composition Algorithm

Assume we have an algorithm $\mathcal{A}_P$ that time-free solves a delay tolerant problem P using channel $\mathcal{C}$, and an algorithm $\mathcal{A}_{\mathcal{C}}$ that implements channel $\mathcal{C}$ using some other channel $\mathcal{C}'$. In this section we describe how to construct algorithm $\mathcal{A}(\mathcal{A}_P, \mathcal{A}_{\mathcal{C}})$ that combines $\mathcal{A}_P$ and $\mathcal{A}_{\mathcal{C}}$. We then prove that this *composition algorithm* solves P using $\mathcal{C}'$. We conclude with a corollary that generalizes this argument to a sequence of channel implementation arguments that start with $\mathcal{C}'$ and end with $\mathcal{C}$. Such compositions are key for a layered approach to radio network algorithm design.

Composition Algorithm Overview. At a high-level, the composition algorithm $\mathcal{A}(\mathcal{A}_P, \mathcal{A}_C)$ calculates the messages generated by $\mathcal{A}_P$ for the current round of $\mathcal{A}_P$ being emulated. It then *pauses* $\mathcal{A}_P$ and executes $\mathcal{A}_{\mathcal{C}}$ to emulate the messages being sent on $\mathcal{C}$. This may require many rounds (during which the environment is receiving only $\perp^n$ from the composed algorithm—necessitating its delay tolerance property). When $\mathcal{A}_{\mathcal{C}}$ finishes computing the received messages, we *unpause* $\mathcal{A}_P$ and finish the emulated round using these messages. The only tricky point in this construction is that when we pause $\mathcal{A}_P$ we need to store a copy of its input, as we will need this later to complete the simulated round once we unpause. The formal definition follows:

Definition 19 (The Composition Algorithm: $\mathcal{A}(\mathcal{A}, \mathcal{A}_{\mathcal{C}})$). *Let $\mathcal{A}_P$ be an algorithm and $\mathcal{A}_{\mathcal{C}}$ be a channel algorithm that implements channel $\mathcal{C}$ using channel $\mathcal{C}'$. Fix any $i \in [n]$. To simplify notation, let $A = \mathcal{A}(\mathcal{A}_P, \mathcal{A}_{\mathcal{C}})(i)$, $B = \mathcal{A}_P(i)$, and $C = \mathcal{A}_{\mathcal{C}}(i)$. We define process A as follows:*

- $\mathbf{states_A} \in states_B \times states_C \times \{active, paused\} \times \mathcal{I}_\perp$.
  Given such a state $s \in states_A$, we use the notation s.prob to refer to the $states_B$ component, s.chan to refer to the $states_C$ component, s.status to refer to the $\{active, paused\}$ component, and s.input to refer to the $\mathcal{I}_\perp$ component. The following two helper function simplify the remaining definitions of process components:
  - *$siminput(s \in states_A, in \in \mathcal{I}_\perp)$: the function evaluates to $\perp$ if $s.status = paused$, and otherwise evaluates to input: $(send, msg_B(s.prob, in), freq_B(s.prob, in))$.*
  - *$simrec(s \in states_A, in \in \mathcal{I}_\perp, m \in \mathcal{R}_\perp)$: the function evaluates to $\perp$ if $out_C(s.chan, siminput(s, in), m) = \perp$, otherwise if $out_C(s.chan, siminput(s, in), m) = (recv, m')$ for some $m' \in \mathcal{R}_\perp$, it returns m'.*
- $\mathbf{start_A} = (start_B, start_C, active, \perp)$.
- $\mathbf{msg_A(s, in)} = msg_C(s.chan, siminput(s, in))$.
- $\mathbf{freq_A(s, in)} = freq_C(s.chan, siminput(s, in))$.
- $\mathbf{out_A(s, in, m)}$ *: let $m' = simrec(s.chan, siminput(s, in), m)$. The out_A function evaluates to $\perp$ if $m' = \perp$, or $out_B(s.prob, s.input, m')$ if $m' \neq \perp$ and $s.state = passive$, or $out_B(s.prob, in, m')$ if $m' \neq \perp$ and $s.state = active$.*
- $\mathbf{rand_A(s)(s')}$ *: the distribution evaluates to $rand_C(s.chan)(s'.chan)$ if $s.status = s'.status = paused$, $s.input = s'.input$, and $s.prob = s'.prob$, or evaluates to $rand_B(s.prob)(s'.prob) \cdot rand_C(s.chan)(s'.chan)$ if $s.status = s'.status = active$ and $s.input = s'.input$, or evaluates to 0 if neither of the above two cases hold.*
- $\mathbf{trans_A(s, m, in)} = s'$ *where we define s' as follows. As in our definition of out_A, we let $m' = simrec(s.chan, siminput(s, in), m)$:*
  - *$s'.prob = trans_B(s.prob, m', s.input)$ if $m' \neq \perp$ and $s.status = paused$, or $trans_B(s.prob, m', in)$ if $m' \neq \perp$ and $s.status = active$, or s.prob if neither of the above two cases hold.*
  - *$s'.chan = trans_C(s.chan, m, siminput(s, in))$.*
  - *$s'.input = in$ if $in \neq \perp$, otherwise it equals s.input.*
  - *$s'.status = active$ if $m' \neq \perp$, otherwise it equals paused.*

We now prove that this composition works (i.e., solves P on $\mathcal{C}'$). Our strategy uses channel-free prefixes: execution prefixes with the channel states removed. We define two functions for extracting these prefixes. The first, *simpleReduce*, removes the channel states from an execution prefix. The second, *compReduce*, extracts the channel-free prefix that describes the emulated execution prefix of $\mathcal{A}_P$ captured in an execution prefix of a (*complex*) system that includes a composition algorithm consisting of $\mathcal{A}_P$ and a channel implementation algorithm.

Definition 20 (Channel-Free Prefix). *We define a sequence α to be a channel-free prefix of an environment $\mathcal{E}$ and algorithm $\mathcal{A}$ if and only if there exists an execution prefix α' of a system including $\mathcal{E}$ and $\mathcal{A}$, such that α describes α' with the channel state assignments removed.*

Definition 21 (simpleReduce). *Let $\mathcal{E}$ be a delay tolerant environment, $\mathcal{A}_P$ be an algorithm, and $\mathcal{C}$ a channel. Let α be an execution prefix of the system $(\mathcal{E}, \mathcal{A}_P, \mathcal{C})$. We define simpleReduce$(\alpha)$ to be the channel-free prefix of $\mathcal{E}$ and $\mathcal{A}_P$ that results when remove the channel state assignments from α.*

Definition 22 (compReduce). *Let $\mathcal{E}$ be a delay tolerant environment, $\mathcal{A}_P$ be an algorithm, $\mathcal{A}_\mathcal{C}$ be a channel algorithm, and $\mathcal{C}'$ a channel. Let α' be an execution prefix of the system $(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_\mathcal{C}), \mathcal{C}')$. We define compReduce$(\alpha')$ to return a special marker "null" if α' includes a partial emulated round of $\mathcal{A}_P$ (i.e., ends in a paused state of the composition algorithm). This captures our desire that compReduce should be undefined for such partial round emulations. Otherwise, it returns the emulated execution of $\mathcal{A}_P$ encoded in the composition algorithm state. Roughly speaking, this involves projecting the algorithm state onto the prob component, removing all but the first and last round of each emulated round, combining, for each emulated round, the initial state of the algorithm and environment of the first round with the final states from the last round, and replacing the messages and frequencies with those described by the emulation. (A formal definition can be found in the full version of this paper [14].)*

We continue with a helper lemma that proves that the execution of $\mathcal{A}_P$ emulated in an execution of a composition algorithm that includes $\mathcal{A}_P$, unfolds the same as $\mathcal{A}_P$ running by itself.

Lemma 1. *Let $\mathcal{E}$ be a delay tolerant environment, $\mathcal{A}_P$ be an algorithm, and $\mathcal{A}_\mathcal{C}$ be a channel algorithm that implements $\mathcal{C}$ with $\mathcal{C}'$. Let α be a channel-free prefix of $\mathcal{E}$ and $\mathcal{A}_P$. It follows:*

$$\sum_{\alpha' | simpleReduce(\alpha')=\alpha} Q(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \alpha') = \sum_{\alpha'' | compReduce(\alpha'')=\alpha} Q(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \alpha'')$$

Proof. The proof can be found in the full version of the paper [14].

We can now prove our main theorem and then a corollary that generalizes the result to a chain of implementation algorithms.

Theorem 1 (Algorithm Composition). *Let $\mathcal{A}_P$ be an algorithm that time-free solves delay tolerant problem P using channel $\mathcal{C}$. Let $\mathcal{A}_C$ be an algorithm that implements channel $\mathcal{C}$ using channel $\mathcal{C}'$. It follows that the composition algorithm $\mathcal{A}(\mathcal{A}_P, \mathcal{A}_C)$ time-free solves P using $\mathcal{C}'$.*

Proof. By unwinding the definition of *time-free solves*, we rewrite our task as follows:

$$\forall \mathcal{E} \in E, \exists F \in P(\mathcal{E}), \forall \beta \in T : D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta) = F(\beta).$$

Fix some $\mathcal{E}$. Assume $\mathcal{E}$ is delay tolerant (if it is not, then $P(\mathcal{E})$ describes every trace probability function, and we are done). Define trace probability function

F such that $\forall\beta \in T : F(\beta) = D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta)$. By assumption $F \in P(\mathcal{E})$. It is sufficient, therefore, to show that $\forall\beta \in T : D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta) = F(\beta) = D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta)$. Fix some β. Below we prove the equivalence. We begin, however, with the following helper definitions:

– Let $ccp(\beta)$ be the set of every channel-free prefix α of $\mathcal{E}$ and $\mathcal{A}_P$ such that $term(\alpha) = true$ and $cio(\alpha) = \beta$.[2]
– Let $S_s(\beta)$, for trace β, describe the set of prefixes included in the sum that defines $D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta)$, and $S_c(\beta)$ describe the set of prefixes included in the sum that defines $D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_{\mathcal{C}}), \mathcal{C}', \beta)$. (The s and c subscripts denote *simple* and *complex*, respectively.) Notice, for a prefix to be included in S_c it cannot end in the middle of an emulated round, as this prefix would not satisfy *term*.
– Let $S'_s(\alpha)$, for channel-free prefix α of $\mathcal{E}$ and $\mathcal{A}_P$, be the set of every prefix α' of $(\mathcal{E}, \mathcal{A}_P, \mathcal{C})$ such that $simpleReduce(\alpha') = \alpha$. Let $S'_c(\alpha)$ be the set of every prefix α'' of $(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_{\mathcal{C}}), \mathcal{C}')$ such that $compReduce(\alpha'') = \alpha$. Notice, for a prefix α'' to be included in S'_c, it cannot end in the middle of an emulated round, as this prefix would cause *compReduce* to return *null*.

We continue with a series of 4 claims that establish that $\{S'_s(\alpha) : \alpha \in ccp(\beta)\}$ and $\{S'_c(\alpha) : \alpha \in ccp(\beta)\}$ partition $S_s(\beta)$ and $S_c(\beta)$, respectively.

Claim 1: $\bigcup_{\alpha \in ccp(\beta)} S'_s(\alpha) = S_s(\beta)$.
We must show two directions of inclusion. First, given some $\alpha' \in S_s(\beta)$, we know $\alpha = simpleReduce(\alpha') \in ccp(\beta)$, thus $\alpha' \in S'_s(\alpha)$. To show the other direction, we note that given some $\alpha' \in S'_s(\alpha)$, for some $\alpha \in ccp(\beta)$, $simpleReduce(\alpha') = \alpha$. Because α generates β by *cio* and satisfies *term*, the same holds for α', so $\alpha' \in S_s(\beta)$.

Claim 2: $\bigcup_{\alpha \in ccp(\beta)} S'_c(\alpha) = S_c(\beta)$.
As above, we must show two directions of inclusion. First, given some $\alpha'' \in S_c(\beta)$, we know $\alpha = compReduce(\alpha'') \in ccp(\beta)$, thus $\alpha'' \in S'_c(\alpha)$. To show the other direction, we note that given some $\alpha'' \in S'_c(\alpha)$, for some $\alpha \in ccp(\beta)$, $compReduce(\alpha'') = \alpha$. We know α generates β by *cio* and satisfies *term*. It follows that α'' ends with the same final non-empty output as α, so it satisfies *term*. We also know that *compReduce* removes only empty inputs and outputs, so α'' also maps to β by *cio*. Therefore, $\alpha'' \in S_c(\beta)$.

Claim 3: $\forall \alpha_1, \alpha_2 \in ccp(\beta), \alpha_1 \neq \alpha_2 : S'_s(\alpha_1) \cap S'_s(\alpha_2) = \emptyset$.
Assume for contradiction that some α' is in the intersection. It follows that $simpleReduce(\alpha')$ equals both α_1 and α_2. Because *simpleReduce* returns a single channel-free prefix, and $\alpha_1 \neq \alpha_2$, this is impossible.

[2] This requires some abuse of notation as *cio* and *term* are defined for prefixes, not channel-free prefixes. These extensions, however, follow naturally, as both *cio* and *term* are defined only in terms of the input and output assignments of the prefixes, and these assignments are present in channel-free prefixes as well as in standard execution prefixes.

Claim 4: $\forall \alpha_1, \alpha_2 \in ccp(\beta), \alpha_1 \neq \alpha_2 : S'_c(\alpha_1) \cap S'_c(\alpha_2) = \emptyset$.
Follows from the same argument as claim 3 with *compReduce* substituted for *simpleReduce*.

The following two claims are a direct consequence of the partitioning proved above and the definition of D_{tf}:

Claim 5: $\sum_{\alpha \in ccp(\beta)} \sum_{\alpha' \in S'_s(\alpha)} Q(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \alpha') = D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta)$.
Claim 6: $\sum_{\alpha \in ccp(\beta)} \sum_{\alpha' \in S'_c(\alpha)} Q(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \alpha') = D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta)$.

We conclude by combining claims 5 and 6 with Lemma 1 to prove that:

$$D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta) = D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta),$$

as needed. □

Corollary 1 (Generalized Algorithm Composition). *Let $\mathcal{A}_{1,2}$, ..., $\mathcal{A}_{j-1,j}$, $j > 2$, be a sequence of algorithms such that each $\mathcal{A}_{i-1,i}$, $1 < i \leq j$, implements channel $\mathcal{C}_{i-1}$ using channel $\mathcal{C}_i$. Let $\mathcal{A}_{P,1}$ be an algorithm that time-free solves delay tolerant problem P using channel $\mathcal{C}_1$. It follows that there exists an algorithm that time-free solves P using $\mathcal{C}_j$.*

Proof. Given an algorithm $\mathcal{A}_{P,i}$ that time-free solves P with channel C_i, $1 \leq i < j$, we can apply Theorem 1 to prove that $\mathcal{A}_{P,i+1} = \mathcal{A}(\mathcal{A}_{P,i}, A_{i,i+1})$ time-free solves P with channel $\mathcal{C}_{i+1}$. We begin with $\mathcal{A}_{P,1}$, and apply Theorem 1 $j-1$ times to arrive at algorithm $\mathcal{A}_{P,j}$ that time-free solves P using $\mathcal{C}_j$.

4.2 The Composition Channel

Given a channel implementation algorithm $\mathcal{A}$ and a channel $\mathcal{C}'$, we define the channel $\mathcal{C}(\mathcal{A}, \mathcal{C}')$. This *composition channel* encodes a local emulation of $\mathcal{A}$ and $\mathcal{C}'$ into its probabilistic state transitions. We formalize this notion by proving that $\mathcal{A}$ implements $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ using $\mathcal{C}'$. To understand the utility of this result, assume you have a channel implementation algorithm $\mathcal{A}$ and you want to prove that $\mathcal{A}$ using $\mathcal{C}'$ implements a channel that satisfies some useful automaton property. (As shown in Sect. 5, it is often easier to talk about all channels that satisfy a property than to talk about a specific channel.) You can apply our composition channel result to establish that $\mathcal{A}$ implements $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ using $\mathcal{C}'$. This reduces the task to showing that $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ satisfies the relevant automaton properties.

Composition Channel Overview. At a high-level, the composition channel $\mathcal{C}(\mathcal{A}, \mathcal{C}')$, when passed a message and frequency assignment, *emulates* $\mathcal{A}$ using $\mathcal{C}'$ being passed these messages and frequencies as input and then returning the emulated output from $\mathcal{A}$ as the received messages. This emulation is encoded into the *crand* probabilistic state transition of $\mathcal{C}(\mathcal{A}, \mathcal{C}')$. To accomplish this feat, we have define two types of states: *simple* and *complex*. The composition channel starts in a simple state. The *crand* distribution always returns complex states, and the *ctrans* transition function always returns simple states, so we alternate between the two. The simple state contains a component *pre* that encodes the history of the emulation of $\mathcal{A}$ and $\mathcal{C}'$ used by $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ so far. The complex state also encodes this history in *pre*, in addition it encodes the next randomized state transitions of $\mathcal{A}$ and $\mathcal{C}'$ in a component named *ext*, and it stores a table, encoded in a component named *oext*, that stores for each possible pair of message and frequency assignments, an emulated execution prefix that extends *ext* with those messages and frequencies arriving as input and ending when $\mathcal{A}$ generates the corresponding received messages. The *crecv* function, given a message and frequency assignment and complex state, can look up the appropriate row in *oext* and return the received messages described in the final output of this extension. This approach of simulating prefixes for all possible messages in advance is necessitated by the fact that the randomized state transition occurs before the channel receives the messages being sent in that round.

The formal definition of the composition channel, and the proof for the theorem below, can be found in the full version of the paper [14].

Theorem 2 (The Composition Implementation Theorem). *Let $\mathcal{A}$ be a channel algorithm and $\mathcal{C}'$ be a channel. It follows that $\mathcal{A}$ implements $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ using $\mathcal{C}'$.*

5 Case Study

We highlight the power and flexibility of our framework with a simple example. We begin by defining two types of channels: p-reliable and t-disrupted. The former is an idealized single-hop single-frequency radio channel with a probabilistic guarantee of successful delivery (e.g., as considered in [15]). The latter is a realistic single-hop radio channel, comprised of multiple independent frequencies, up to t of which might be permentantly disrupted by outside sources of interference (e.g., as considered in [16]). We then describe a simple algorithm $\mathcal{A}_{rel}$ and sketch a proof that it implements the reliable channel using the disrupted channel. Before defining the two channel types, however, we begin with this basic property used by both:

Definition 23 (Basic Broadcast Property). *We say a channel $\mathcal{C}$ satisfies the basic broadcast property if and only if for every state s, message assignment M, and frequency assignments F, $N = crecv_{\mathcal{C}}(s, M, F)$ satisfies the following:*

1. *If $M[i] \neq \bot$ for some $i \in [n]$: $N[i] = M[i]$. (Broadcasters receive their own messages.)*
2. *If $N[i] \neq \bot$, for some $i \in [n]$, then there exists a $j \in [n] : M[j] = N[i] \wedge F[j] = F[i]$. (If i receives a message then some process sent that message on the same frequency as i.)*
3. *If there exists some $i, j, k \in [n], i \neq j \neq k$, such that $F[i] = F[j] = F[k]$, $M[i] \neq \bot$, $M[j] \neq \bot$, and $M[k] = \bot$, it follows that $N[k] = \bot$. (Two or more broadcasters on the same frequency cause a collision at receivers on this frequency.)*

Definition 24 (p-Reliable Channel). *We say a channel $\mathcal{C}$ satisfies the p-reliable channel property, $p \in [0,1]$, if and only if $\mathcal{C}$ satisfies the basic broadcast property, and there exists a subset S of the states, such that for every state s, message assignment M, and frequency assignments F, $N = crecv_{\mathcal{C}}(s, M, F)$ satisfies the following:*

1. *If $F[i] > 1 \wedge M[i] = \perp$, for some $i \in [n]$, then $N[i] = \perp$.*
   (Receivers on frequencies other than 1 receive nothing.)
2. *If $s \in S$ and $|\{i \in [n] : F[i] = 1, M[i] \neq \perp\}| = 1$, then for all $j \in [n]$ such that $F[j] = 1$ and $M[j] = \perp$: $N[j] = M[i]$.*
   (If there is a single broadcaster on frequency 1, and the channel is in a state from S, then all receivers on frequency 1 receive its message.)
3. *For any state s', $\sum_{s \in S} crand_{\mathcal{C}}(s')(s) \geq p$.*
   (The probability that we transition into a state in S—i.e., a state that guarantees reliable message delivery—is at least p.)

Definition 25 (t-Disrupted Channel). *We say a channel $\mathcal{C}$ satisfies the t-disrupted property, $0 \leq t < \mathcal{F}$, if and only if $\mathcal{C}$ satisfies the basic broadcast channel property, and there exists a set $B_t \subset [\mathcal{F}]$, $|B_t| \leq t$, such that for every state s, message assignment M, and frequency assignment F, $N = crecv_{\mathcal{C}}(s, M, F)$ satisfies the following:*

1. *If $M[i] = \perp$ and $F[i] \in B_t$, for some $i \in [n]$: $N[i] = \perp$.*
   (Receivers receive nothing if they receive on a disrupted frequency.)
2. *If for some $f \in [\mathcal{F}], f \notin B_t$, $|\{i \in [n] : F[i] = f, M[i] \neq \perp\}| = 1$, then for all $j \in [n]$ such that $F[j] = f$ and $M[j] = \perp$, $N[j] = M[i]$, where i is the single process from the above set of broadcasters on f.*
   (If there is a single broadcaster on a non-disrupted frequency then all receivers on that frequency receive the message.)

Consider the channel algorithm, $\mathcal{A}_{rel}$, that works as follows: The randomized transition $rand_{\mathcal{A}_{rel}(i)}$ encodes a random frequency f_i for each process i in the resulting state. This choice is made independently and at random for each process. If a process $\mathcal{A}_{rel}(i)$ receives an input from $(send, m \in \mathcal{M}, 1)$, it broadcasts m on frequency f_i and outputs $(recv, m)$. If the process receives input $(send, \perp, 1)$ it receives on f_i, and then outputs $(recv, m')$, where m' is the message it receives. Otherwise, it outputs $(recv, \perp)$. We now prove that $\mathcal{A}_{rel}$ implements a reliable channel using a disrupted channel.

Theorem 3. *Fix some t, $0 \leq t < \mathcal{F}$. Given any channel $\mathcal{C}$ that satisfies the t-disrupted channel property, the algorithm $\mathcal{A}_{rel}$ implements a channel that satisfies the $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$-reliable channel property using $\mathcal{C}$.*

Proof (Sketch). By Theorem 2 we know $\mathcal{A}_{rel}$ implements $\mathcal{C}(\mathcal{A}_{rel}, \mathcal{C})$ using $\mathcal{C}$. We are left to show that $\mathcal{C}(\mathcal{A}_{rel}, \mathcal{C})$ satisfies the $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$-reliable channel property. Condition 1 of this property follows from the definition of $\mathcal{A}_{rel}$. More interesting is the combination of 2 and 3. Let B_t be the set of disrupted frequencies associated with $\mathcal{C}$. A state s returned by $crand_{\mathcal{C}(\mathcal{A}_{rel},\mathcal{C})}$ is in S if the final state of

$\mathcal{A}_{rel}$ in $s.ext$ encodes the same f value for all processes, and this value is not in B_t. Because each process chooses this f value independently and at random, this occurs with probability at least $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$. □

Next, imagine that we have some algorithm $\mathcal{A}_P$ that solves a delay tolerant problem P (such as randomized consensus, which is easily defined in a delay tolerant manner) on a $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$-reliable channel. We can apply Theorem 1 to directly derive that $\mathcal{A}(\mathcal{A}_P, \mathcal{A}_{rel})$ solves P on any t-disrupted channel $\mathcal{C}'$. In a similar spirit, imagine we have an algorithm $\mathcal{A}^+_{rel}$ that implements a (1/2)-reliable channel using a $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$-reliable channel, and we have an algorithm $\mathcal{A}_{P'}$ that solves delay tolerant problem P' on a (1/2)-reliable channel. We could apply Corollary 1 to $\mathcal{A}_{P'}$, $\mathcal{A}^+_{rel}$, and $\mathcal{A}_{rel}$, to identify an algorithm that solves P' on our t-disrupted channel. And so on.

6 Conclusion

In this paper we present a modeling framework for synchronous probabilistic radio networks. The framework allows for the precise definition of radio channels and includes a pair of composition results that simplify a layered approach to network design (e.g., implementing stronger channels with weaker channels). We argue that this framework can help algorithm designers sidestep problems due to informal model definitions and more easily build new results using existing results. Much future work remains regarding this research direction, including the formalization of well-known results, exploration of more advanced channel definitions (e.g., multihop networks or adversarial sources of error), and the construction of implementation algorithms to link existing channel definitions.

References

1. Abramson, N.: The Aloha system - Another approach for computer communications. In: The Proceedings of the Fall Joint Computer Conference, vol. 37, pp. 281–285 (1970)
2. Roberts, L.G.: Aloha packet system with and without slots and capture. In: ASS Note 8. Advanced Research Projects Agency, Network Information Center, Stanford Research Institute (1972)
3. Kleinrock, L., Tobagi, F.: Packet switching in radio channels. IEEE Transactions on Communications COM-23, 1400–1416 (1975)
4. Hajek, B., van Loon, T.: Decentralized dynamic control of a multiaccess broadcast channel. IEEE Transactions on Automation and Control AC-27, 559–569 (1979)
5. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. Journal of Computer and System Sciences 45(1), 104–126 (1992)
6. Chlamtac, I., Weinstein, O.: The wave expansion approach to broakodcasting in multihop radio networks. IEEE Transactions on Communications 39, 426–433 (1991)

7. Clementi, A., Monti, A., Silvestri, R.: Round robin is optimal for fault-tolerant broadcasting on wireless networks. Journal of Parallel and Distributed Computing 64(1), 89–96 (2004)
8. Kowalski, D., Pelc, A.: Broadcasting in undirected ad hoc radio networks. In: The Proceedings of the International Symposium on Principles of Distributed Computing, pp. 73–82 (2003)
9. Kowalski, D., Pelc, A.: Time of deterministic broadcasting in radio networks with local knowledge. SIAM Journal on Computing 33(4), 870–891 (2004)
10. Chockler, G., Demirbas, M., Gilbert, S., Lynch, N., Newport, C., Nolte, T.: Consensus and collision detectors in radio networks. Distributed Computing 21, 55–84 (2008)
11. Wu, S.H., Smolka, S.A., Stark, E.W.: Composition and behaviors of probabilistic I/O automata. In: The Proceedings of the International Conference on Concurrency Theory (1994)
12. Segala, R.: Modeling and verification of randomized distributed real-time systems. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (June 1995)
13. Cheung, L.: Reconciling Nondeterministic and Probabilistic Choices. PhD thesis, Radboud University Nijmege (2006)
14. Newport, C., Lynch, N.: Modeling radio networks. Technical report, MIT CSAIL (2009)
15. Bar-Yehuda, R., Goldreich, O., Itai, A.: Efficient emulation of single-hop radio network with collision detection on multi-hop radio network with no collision detection. Distributed Computing 5, 67–71 (1991)
16. Gilbert, S., Guerraoui, R., Kowalski, D., Newport, C.: Interference-resilient information exchange. In: The Proceedings of the Conference on Computer Communication (2009)

Time-Bounded Verification*

Joël Ouaknine[1], Alexander Rabinovich[2], and James Worrell[1]

[1] Oxford University Computing Laboratory, UK
{joel,jbw}@comlab.ox.ac.uk

[2] School of Computer Science, Tel Aviv University, Israel
rabinoa@post.tau.ac.il

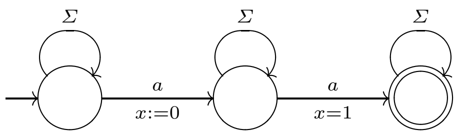
Abstract. We study the decidability and complexity of verification problems for timed automata over time intervals of fixed, bounded length. One of our main results is that time-bounded language inclusion for timed automata is 2EXPSPACE-Complete. We also investigate the satisfiability and model-checking problems for Metric Temporal Logic (MTL), as well as monadic first- and second-order logics over the reals with order and the +1 function ($\mathsf{FO}(<, +1)$ and $\mathsf{MSO}(<, +1)$ respectively). We show that, over bounded time intervals, MTL satisfiability and model checking are EXPSPACE-Complete, whereas these problems are decidable but non-elementary for the predicate logics. Nevertheless, we show that MTL and $\mathsf{FO}(<, +1)$ are equally expressive over bounded intervals, which can be viewed as an extension of Kamp's well-known theorem to metric logics.

It is worth recalling that, over unbounded time intervals, the satisfiability and model-checking problems listed above are all well-known to be undecidable.

1 Introduction

Timed automata were introduced by Alur and Dill in [2] as a natural and versatile model for real-time systems. They have been widely studied ever since, both by practitioners and theoreticians. A celebrated result concerning timed automata, which originally appeared in [1] in a slightly different context, is the PSPACE decidability of the *language emptiness* (or *reachability*) problem.

Unfortunately, the *language inclusion* problem—given two timed automata $\mathcal{A}$ and $\mathcal{B}$, is every timed word accepted by $\mathcal{A}$ also accepted by $\mathcal{B}$?—is known to be undecidable. A related phenomenon is the fact that timed automata are not closed under complementation. For example, the automaton below accepts every timed word in which there are two a-events separated by exactly one time unit.



* This work was partially supported by the ESF Research Networking Programme Games and the UK's EPSRC.

The complement language consists of all timed words in which no two a-events are separated by precisely one time unit. Intuitively, this language is not expressible by a timed automaton, since such an automaton would need an unbounded number of clocks to keep track of the time delay from each a-event. (We refer the reader to [17] for a formal treatment of these considerations.)

The undecidability of language inclusion severely restricts the algorithmic analysis of timed automata, both from a practical and theoretical perspective, as many interesting questions can be phrased in terms of language inclusion. Over the past decade, several researchers have therefore attempted to circumvent this state of affairs by investigating language inclusion, or closely related concepts, under various assumptions and restrictions. Among others, we note the use of (i) topological restrictions and digitisation techniques: [14, 10, 31, 28]; (ii) fuzzy semantics: [13, 15, 30, 7]; (iii) determinisable subclasses of timed automata: [4, 37]; (iv) timed simulation relations and homomorphisms: [42, 26, 23]; and (v) restrictions on the number of clocks: [32, 11].

The undecidability of language inclusion, first established in [2], derives from the undecidability of an even more fundamental problem, that of *universality*: does a given timed automaton accept every timed word? The proof of undecidability of universality in [2] uses in a crucial way the unboundedness of the time domain. Roughly speaking, this allows one to encode arbitrarily long computations of a Turing machine. On the other hand, many verification questions are naturally stated over bounded time domains. For example, a run of a communication protocol might normally be expected to have an *a priori* time bound, even if the total number of messages exchanged is potentially unbounded. Thus numerous researchers have considered the problem of time-bounded verification in the context of real-time systems [39, 8, 22]. This leads us to the question of the decidability of the time-bounded version of the language inclusion problem for timed automata. This problem asks, given timed automata $\mathcal{A}$ and $\mathcal{B}$ and a time bound N, whether all finite timed words of duration at most N that are accepted by $\mathcal{A}$ are also accepted by $\mathcal{B}$. One of the main results of this paper is that the time-bounded language inclusion problem is 2EXPSPACE-Complete. It is worth noting that, since we are working with a dense model of time, time-bounded runs of a given automaton may contain arbitrarily many events. Moreover the restriction to time boundedness does not alter the fact that timed automata are not closed under complement, and hence classical techniques for language inclusion do not trivially apply.

A second line of investigation in this paper concerns the relative expressiveness of monadic second-order and first-order metric logics over the reals. This direction is motivated by the celebrated result of Kamp [21] that Linear Temporal Logic (LTL) has the same expressiveness over the structure $(\mathbb{N}, <)$ as monadic first-order logic ($\mathsf{FO}(<)$). An influential consequence of Kamp's result is that LTL has emerged as the canonical temporal logic over the naturals. While a version of Kamp's result holds over the structure $(\mathbb{R}_{\geq 0}, <)$, the correspondence between predicate logics and temporal logics becomes considerably more complicated with the introduction of *metric* specifications. In practice this has led to a veritable babel of metric temporal logics over the reals [5].

A natural predicate logic in which to formalise metric specifications over the reals is the first-order monadic logic over the structure $(\mathbb{R}_{\geq 0}, <, +1)$. Given a set of uninterpreted monadic predicates $\mathbf{P}$, a model over $(\mathbb{R}_{\geq 0}, <, +1)$ is nothing but a function $f : \mathbb{R}_{\geq 0} \to 2^{\mathbf{P}}$ mapping each $x \in \mathbb{R}_{\geq 0}$ to the set of predicates that hold at x. Such a model is called a *flow* or *signal*, and naturally corresponds to the trajectory of a real-time system.

On the side of temporal logics, an appealing extension of LTL, called Metric Temporal Logic (MTL), was proposed by Koymans [24] almost twenty years ago. While MTL has been widely studied, it is well-known that there are first-order formulas over $(\mathbb{R}_{\geq 0}, <, +1)$ that cannot be expressed in MTL [20].

Our second main result is that, over bounded time domains, MTL has the same expressiveness as monadic first-order logic. Specifically, we show that MTL is as expressive as first-order logic over the structure $([0, N), <, +1)$, for any fixed integer N. Thus, as with language inclusion for timed automata, the restriction to time-boundedness leads to a better-behaved theory.

Finally, we relate automata and logics by showing decidability of the model-checking problem for timed automata against specifications expressed in MTL, first-order, and second-order monadic logics over $([0, N), <, +1)$. We also show decidability of the satisfiability problems for first-order and second-order logics over $([0, N), <, +1)$. In contrast to the case of language inclusion between timed automata, the model-checking and satisfiability problems for monadic predicate logics all have non-elementary complexity, whereas these problems are EXPSPACE-Complete in the case of MTL.

2 Timed Automata

Let X be a finite set of clocks, denoted x, y, z, etc. We define the set Φ_X of clock constraints over X via the following grammar, where $k \in \mathbb{N}$ stands for any non-negative integer, and $\bowtie \in \{=, \neq, <, >, \leq, \geq\}$ is a comparison operator:

$$\phi ::= \mathbf{true} \mid x \bowtie k \mid x - y \bowtie k \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \,.$$

A ***timed automaton.*** $\mathcal{A}$ is a six-tuple $(\Sigma, S, S_0, S_F, X, \Delta)$, where

- Σ is a finite set (alphabet) of events,
- S is a finite set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $S_F \subseteq S$ is a set of accepting states,
- X is a finite set of clocks, and
- $\Delta \subseteq S \times S \times \Sigma \times \Phi_X \times 2^X$ is a finite set of transitions. A transition (s, s', a, ϕ, R) allows a jump from state s to s', consuming event $a \in \Sigma$ in the process, provided the constraint ϕ on clocks is met. Afterwards, the clocks in R are reset to zero, while all other clocks remain unchanged.

Given a timed automaton $\mathcal{A}$ as above, a *clock valuation* is a function $\nu : X \to \mathbb{R}_{\geq 0}$. If $t \in \mathbb{R}_{\geq 0}$, we let $\nu + t$ be the clock valuation such that $(\nu+t)(x) = \nu(x) + t$ for all $x \in X$.

A *configuration* of $\mathcal{A}$ is a pair (s, ν), where $s \in S$ is a state and ν is a clock valuation.

An *accepting run* of $\mathcal{A}$ is a finite alternating sequence of configurations and delayed transitions $\pi = (s_0, \nu_0) \xrightarrow{d_1, \theta_1} (s_1, \nu_1) \xrightarrow{d_2, \theta_2} \ldots \xrightarrow{d_n, \theta_n} (s_n, \nu_n)$, where each $d_i \in \mathbb{R}_{>0}$ and each $\theta_i = (s_{i-1}, s_i, a_i, \phi_i, R_i) \in \Delta$, subject to the following conditions:

1. $s_0 \in S_0$, and for all $x \in X$, $\nu_0(x) = 0$.
2. For all $0 \leq i \leq n-1$, $\nu_i + d_{i+1}$ satisfies ϕ_{i+1}.
3. For all $0 \leq i \leq n-1$, $\nu_{i+1}(x) = \nu_i(x) + d_{i+1}$ for all $x \in X \setminus R_{i+1}$, and $\nu_{i+1}(x) = 0$ for all $x \in R_{i+1}$.
4. $s_n \in S_F$.

Each d_i is interpreted as the (strictly positive[1]) time delay between the firing of transitions, and each configuration (s_i, ν_i), for $i \geq 1$, records the data immediately following transition θ_i. Abusing notation, we also write runs in the form $(s_0, \nu_0) \xrightarrow{d_1, a_1} (s_1, \nu_1) \xrightarrow{d_2, a_2} \ldots \xrightarrow{d_n, a_n} (s_n, \nu_n)$ to highlight the run's events.

A ***timed word*** is a pair (σ, τ), where $\sigma = \langle a_1 a_2 \ldots a_n \rangle \in \Sigma^*$ is a word and $\tau = \langle t_1 t_2 \ldots t_n \rangle \in (\mathbb{R}_{>0})^*$ is a strictly increasing sequence of real-valued timestamps of the same length.

Such a timed word is *accepted* by $\mathcal{A}$ if $\mathcal{A}$ has some accepting run of the form $\pi = (s_0, \nu_0) \xrightarrow{d_1, a_1} (s_1, \nu_1) \xrightarrow{d_2, a_2} \ldots \xrightarrow{d_n, a_n} (s_n, \nu_n)$ where, for each $1 \leq i \leq n$, $t_i = d_1 + d_2 + \ldots + d_i$.

In this paper, we are mainly concerned with behaviours over time domains of the form $[0, N)$, where $N \in \mathbb{N}$ is a positive integer. Let us in general write $\mathbb{T}$ to denote either $[0, N)$ or $\mathbb{R}_{\geq 0}$. We then define $L_{\mathbb{T}}(\mathcal{A})$ to be the set of timed words accepted by $\mathcal{A}$ all of whose timestamps belong to $\mathbb{T}$.

Remark 1. Our timed automata have transitions that are labelled with instantaneous events; this is by far the most common model found in the literature. Alternatives include automata in which states are labelled with atomic propositions [3], or even mixed models in which states carry atomic propositions and transitions carry events. Other variants allow for silent transitions (invisible events), invariants on states, and combinations thereof. All the results presented in this paper carry over without difficulty to these more expressive models.

Note that we are focussing on *finite* words. Timed automata can be defined to accept infinite words (for example, by using Büchi acceptance conditions [2]), although over bounded time infinite words are automatically *Zeno* (and *ipso facto* ruled out from the accepted language by most researchers). Theorem 4 could nonetheless be extended to such infinite words, if desired.

[1] This gives rise to the *strongly monotonic* semantics for timed automata; in contrast, the *weakly monotonic* semantics allows multiple events to happen 'simultaneously' (or, more precisely, with null-duration delays between them). The main results of this paper remain substantively the same under either semantics, although the weakly monotonic semantics causes some slight complications.

3 Metric Logics

3.1 Syntax

Let **Var** be a set of *first-order variables*, denoted x, y, z, etc., ranging over non-negative real numbers. Let **MP** be a set of *monadic predicates*, denoted P, Q, R, etc. Monadic predicates will alternately be viewed as second-order variables over $\mathbb{R}_{\geq 0}$, i.e., ranging over sets of non-negative real numbers, and also as atomic propositions holding at various points in time.

Second-order monadic formulas are obtained from the following grammar:

$$\varphi ::= \mathbf{true} \mid x < y \mid +1(x, y) \mid P(x) \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \forall x\, \varphi \mid \forall P\, \varphi\,,$$

where $+1$ is a binary relation symbol, with the intuitive interpretation of $+1(x, y)$ as '$x+1 = y$'.[2] We refer to $\forall x$ and $\forall P$ as *first-order* and *second-order* quantifiers respectively.

The ***monadic second-order metric logic of order***, written $\mathsf{MSO}(<, +1)$, comprises all second-order monadic formulas. Its first-order fragment, the ***(monadic) first-order metric logic of order***, written $\mathsf{FO}(<, +1)$, comprises all $\mathsf{MSO}(<, +1)$ formulas that do not contain any second-order quantifier; note that these formulas are however allowed free monadic predicates.

We also define two further purely order-theoretic sublogics, which are peripheral to our main concerns but necessary to express some key related results. The *monadic second-order logic of order*, $\mathsf{MSO}(<)$, comprises all second-order monadic formulas that do not make use of the $+1$ relation. Likewise, the *(monadic) first-order logic of order*, $\mathsf{FO}(<)$, comprises those $\mathsf{MSO}(<)$ formulas that eschew second-order quantification.

Metric Temporal Logic, abbreviated MTL, comprises the following *temporal formulas*:

$$\theta ::= \mathbf{true} \mid P \mid \theta_1 \wedge \theta_2 \mid \theta_1 \vee \theta_2 \mid \neg\theta \mid \Diamond_I \theta \mid \Box_I \theta \mid \theta_1 \,\mathcal{U}_I\, \theta_2\,,$$

where $P \in \mathbf{MP}$ is a monadic predicate, and $I \subseteq \mathbb{R}_{\geq 0}$ is an open, closed, or half-open interval with endpoints in $\mathbb{N} \cup \{\infty\}$. If $I = [0, \infty)$, then we omit the annotation I in the corresponding temporal operator. We also use pseudo-arithmetic expressions to denote intervals. For example, the expression '≥ 1' denotes $[1, \infty)$ and '$=1$' denotes the singleton $\{1\}$.

Note that our version of MTL includes only *forwards* temporal operators, in keeping with the most common definition found in the literature. All our results extend straightforwardly to variants of MTL that make use of both forwards and backwards operators. It is also worth pointing out that the $\Diamond_I$ and $\Box_I$ operators are derivable from $\mathcal{U}_I$.

[2] The usual approach is of course to define $+1$ as a unary function symbol; this however necessitates an awkward treatment over bounded domains, as considered in this paper. We shall nonetheless abuse notation later on and invoke $+1$ as if it were a function, in the interest of clarity.

Finally, *Linear Temporal Logic*, written LTL, consists of those MTL formulas in which every indexing interval I on temporal operators is $[0, \infty)$ (and hence omitted).

Figure 1 pictorially summarises the syntactic inclusions and relative expressive powers of these various logics.

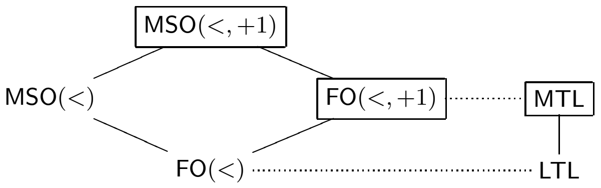


Fig. 1. Relative expressiveness among the various logics. Metric logics are enclosed in boxes. Straight lines denote syntactical inclusion, whereas dotted lines express semantic equivalence over bounded time domains (cf. Section 5).

3.2 Semantics

Let $\mathbf{P} \subseteq \mathbf{MP}$ be a finite set of monadic predicates, and let us again write $\mathbb{T}$ to denote either $[0, N)$ (for some fixed $N \in \mathbb{N}$) or $\mathbb{R}_{\geq 0}$. A ***flow*** (or *signal*) over $\mathbf{P}$ is a function $f : \mathbb{T} \to 2^{\mathbf{P}}$ that is *finitely variable*. Finite variability means that the restriction of f to any finite subinterval of $\mathbb{T}$ has only finitely many discontinuities.[3] Note that we do not place any bound on the variability, other than requiring that it be finite.

Observe that a timed word $(\langle a_1 \dots a_n \rangle, \langle t_1 \dots t_n \rangle)$ over alphabet Σ can be viewed as a (particular type of) flow, as follows. Let $\mathbb{T}$ be either $[0, N)$ (for some $N > t_n$) or $\mathbb{R}_{\geq 0}$, and let $\mathbf{P} = \Sigma$. Set $f(t_i) = \{a_i\}$, for $1 \leq i \leq n$, and $f(t) = \emptyset$ for all other values of $t \in \mathbb{T}$.

Fix a time domain $\mathbb{T}$, equipped with the standard order relation $<$ and the obvious binary relation $+1$, i.e., $+1(a, b)$ iff $a, b \in \mathbb{T}$ and $a+1 = b$. Given a formula φ of MSO$(<, +1)$ or one of its sublogics, let $\mathbf{P}$ and $\{x_1, \dots, x_n\}$ respectively be the sets of free monadic predicates and free first-order variables appearing in φ. For any flow $f : \mathbb{T} \to 2^{\mathbf{P}}$ and real numbers $a_1, \dots, a_n \in \mathbb{T}$, the satisfaction relation $(f, a_1, \dots, a_n) \models \varphi$ is defined inductively on the structure of φ in the standard way. We shall particularly be interested in the special case in which φ is a *sentence*, i.e., a formula with no free first-order variable. In such instances, we simply write the satisfaction relation as $f \models \varphi$.

For θ an MTL or LTL formula, let $\mathbf{P}$ be the set of monadic predicates appearing in θ. Given a flow $f : \mathbb{T} \to 2^{\mathbf{P}}$ and $t \in \mathbb{T}$, the satisfaction relation $(f, t) \models \theta$ is defined inductively on the structure of θ, as follows:

[3] The restriction to finitely-variable flows can be partially lifted, as we discuss in the full version of this paper [29]. Note however that from a computer science perspective, infinitely-variable flows do not correspond to feasible computations, hence the widespread adoption of the present restriction in the literature.

– $(f,t) \models \mathbf{true}$.
– $(f,t) \models P$ iff $P \in f(t)$.
– $(f,t) \models \theta_1 \wedge \theta_2$ iff $(f,t) \models \theta_1$ and $(f,t) \models \theta_2$.
– $(f,t) \models \theta_1 \vee \theta_2$ iff $(f,t) \models \theta_1$ or $(f,t) \models \theta_2$.
– $(f,t) \models \neg\theta$ iff $(f,t) \not\models \theta$.
– $(f,t) \models \Diamond_I \theta$ iff there exists $u \in \mathbb{T}$ with $u > t$, $u - t \in I$, and $(f,u) \models \theta$.
– $(f,t) \models \Box_I \theta$ iff for all $u \in \mathbb{T}$ with $u > t$ and $u - t \in I$, $(f,u) \models \theta$.
– $(f,t) \models \theta_1 \, \mathcal{U}_I \, \theta_2$ iff there exists $u \in \mathbb{T}$ with $u > t$, $u - t \in I$, $(f,u) \models \theta_2$, and for all $v \in (t,u)$, $(f,v) \models \theta_1$.

Finally, we write $f \models \theta$ iff $(f,0) \models \theta$. This is sometimes referred to as the *initial semantics.*

Note that we have adopted a *strict* semantics, in which the present time t has no influence on the truth values of future temporal subformulas. Strictness is required for Theorem 2, but our other results hold under both the strict and non-strict semantics.

An important point concerning our semantics is that it is *continuous*, rather than *pointwise*: more precisely, the temporal operators quantify over all time points of the domain, as opposed to merely those time points at which a discontinuity occurs. Positive decidability results for satisfiability and model checking of MTL over unbounded time intervals have been obtained in the pointwise semantics [33, 34, 35]; it is worth noting that none of these results hold in the continuous semantics.

4 Satisfiability

The canonical time domain for interpreting the metric logics $\mathsf{MSO}(<,+1)$, $\mathsf{FO}(<,+1)$, and MTL is the non-negative real line $\mathbb{R}_{\geq 0}$. Unfortunately, none of these logics are decidable over $\mathbb{R}_{\geq 0}$ [5, 6, 19].

Our main focus in this paper is therefore on satisfiability over bounded time domains of the form $[0,N)$, for $N \in \mathbb{N}$. For each of the logics introduced in Section 3.1, one can consider the corresponding ***time-bounded satisfiability problem***: given a sentence φ over a set $\mathbf{P}$ of free monadic predicates, together with a time bound $N \in \mathbb{N}$, does there exist a flow $f : [0,N) \to 2^{\mathbf{P}}$ such that $f \models \varphi$?

One of our main results is the following:

Theorem 1. *The time-bounded satisfiability problems for the metric logics* $\mathsf{MSO}(<,+1)$, $\mathsf{FO}(<,+1)$, *and* MTL *are all decidable, with the following complexities.*[4]

$\mathsf{MSO}(<,+1)$	Non-elementary
$\mathsf{FO}(<,+1)$	Non-elementary
MTL	EXPSPACE-Complete

[4] All the complexity results in this paper assume that the time bound N is provided in binary.

Remark 2. It is worth noting that if one allows second-order quantification over flows of *arbitrary* variability, then $\mathsf{MSO}(<)$ is undecidable over $\mathbb{R}_{\geq 0}$ [40]. Since any non-trivial interval of the form $[0, N)$ is order-isomorphic to $\mathbb{R}_{\geq 0}$, the same result holds over bounded time domains, and also clearly carries over to $\mathsf{MSO}(<, +1)$. $\mathsf{FO}(<, +1)$ and MTL however remain decidable over bounded time domains regardless of the variability of flows—see [29].

The proofs of all theorems in this paper are given in Appendix A.

5 Expressiveness

Fix a time domain $\mathbb{T}$ to be either $[0, N)$ (for some $N \in \mathbb{N}$) or $\mathbb{R}_{\geq 0}$. Let $\mathcal{L}$ and $\mathcal{J}$ be two logics. We say that $\mathcal{L}$ is ***semantically at least as expressive as*** $\mathcal{J}$ *(with respect to the initial semantics of* $\mathbb{T}$*)* if, for any sentence θ of $\mathcal{J}$, there exists a sentence φ of $\mathcal{L}$ such that θ and φ are satisfied by precisely the same set of flows over $\mathbb{T}$.

Two logics are then said to be ***semantically equally expressive*** *(with respect to the initial semantics of* $\mathbb{T}$*)* if each is at least as expressive as the other.

The following result can be viewed as an extension of Kamp's celebrated theorem [21, 12] to metric logics over bounded time domains:

Theorem 2. *For any fixed bounded time domain of the form* $[0, N)$*, with* $N \in \mathbb{N}$*, the metric logics* $\mathsf{FO}(<, +1)$ *and* MTL *are semantically equally expressive. Moreover, this equivalence is effective.*

Remark 3. Note that semantic expressiveness here is relative to a *single* structure $\mathbb{T}$, rather than to a *class* of structures. In particular, although $\mathsf{FO}(<, +1)$ and MTL are equally expressive over any bounded time domain of the form $[0, N)$, the correspondence and witnessing formulas may very well vary according to the time domain.

It is interesting to note that $\mathsf{FO}(<, +1)$ is strictly more expressive than MTL over $\mathbb{R}_{\geq 0}$ [20]. For example, MTL is incapable of expressing the following formula (in slightly abusive but readable notation)

$$\exists x\, \exists y\, \exists z\, (x < y < z < x + 1 \wedge P(x) \wedge P(y) \wedge P(z))$$

over the non-negative reals. This formula asserts that, sometime in the future, P will hold at three distinct time points within a single time unit.

It is also worth noting that $\mathsf{MSO}(<, +1)$ is strictly more expressive than $\mathsf{FO}(<, +1)$—and hence MTL—over any time domain; see [29].

Finally, we point out that, in contrast to Kamp's theorem [21], but similarly to [12], Theorem 2 does not require backwards temporal operators for MTL (although adding these would be harmless).

6 Model Checking and Language Inclusion

We now turn to questions concerning the time-bounded behaviours of timed automata. Recall from Section 3.2 that timed words over an alphabet Σ can be viewed as flows from a sufficiently large time domain over the set of monadic predicates $\mathbf{P} = \Sigma$. The ***model checking problem*** takes as inputs a timed automaton $\mathcal{A}$ with alphabet Σ, a sentence φ with set of free monadic predicates $\mathbf{P} = \Sigma$, and a time domain $\mathbb{T}$ (taken to be either $[0, N)$, for some $N \in \mathbb{N}$, or $\mathbb{R}_{\geq 0}$). The question is then whether every timed word (flow) in $L_{\mathbb{T}}(\mathcal{A})$ satisfies φ.

Unfortunately, the model checking problem for timed automata and any of the metric logics introduced in this paper is undecidable over the non-negative real line $\mathbb{R}_{\geq 0}$ [5, 6]; this in fact also follows easily from the undecidability of the satisfiability problem for these logics over flows. We therefore focus on the ***time-bounded model checking problem***, in which the time domain is required to be bounded. We have:

Theorem 3. *The time-bounded model-checking problems for timed automata against the metric logics* $\mathsf{MSO}(<, +1)$*,* $\mathsf{FO}(<, +1)$*, and* MTL *are all decidable, with the same complexities as the corresponding time-bounded satisfiability problems (cf. Theorem 1): non-elementary for* $\mathsf{MSO}(<, +1)$ *and* $\mathsf{FO}(<, +1)$*, and EXPSPACE-Complete for* MTL*.*

The ***language inclusion problem*** takes as inputs two timed automata, $\mathcal{A}$ and $\mathcal{B}$, sharing a common alphabet, together with a time domain $\mathbb{T}$ (of the form $[0, N)$, for some $N \in \mathbb{N}$, or $\mathbb{R}_{\geq 0}$). The question is then whether every timed word accepted by $\mathcal{A}$ over $\mathbb{T}$ is also accepted by $\mathcal{B}$.

As for model checking, language inclusion is unfortunately undecidable over $\mathbb{R}_{\geq 0}$ [2]. The ***time-bounded language inclusion problem*** circumvents this by restricting to bounded time domains. This leads us to our final main result, as follows:

Theorem 4. *The time-bounded language inclusion problem for timed automata is decidable and 2EXPSPACE-Complete.*

A Proofs of Theorems

Theorem 1. *The time-bounded satisfiability problems for* $\mathsf{MSO}(<, +1)$ *and* $\mathsf{FO}(<, +1)$ *are decidable and non-elementary, whereas the time-bounded satisfiability problem for* MTL *is EXPSPACE-Complete.*

Proof. Throughout this proof, let $N \in \mathbb{N}$ be fixed.

An easy first observation is that any MTL formula can be translated into an equivalent $\mathsf{FO}(<, +1)$ formula over $[0, N)$. For decidability, it therefore suffices to handle the case of $\mathsf{MSO}(<, +1)$.

Let $\mathbf{P} \subseteq \mathbf{MP}$ be a finite set of monadic predicates. With each $P \in \mathbf{P}$, we associate a collection $P_0, \ldots, P_{N-1}$ of N fresh monadic predicates. We then let $\overline{\mathbf{P}} = \{P_i \mid P \in \mathbf{P}, 0 \leq i \leq N - 1\}$.

Intuitively, each monadic predicate P_i represents P over the subinterval $[i, i+1)$. Indeed, there is an obvious bijection (indicated by overlining) between the set of flows $\{f : [0, N) \to 2^{\mathbf{P}}\}$ and the set of flows $\{\overline{f} : [0, 1) \to 2^{\overline{\mathbf{P}}}\}$.

Let φ be an $\mathsf{MSO}(<,+1)$ sentence with set of free monadic predicates $\mathbf{P}$. We will define an $\mathsf{MSO}(<)$ sentence $\overline{\varphi}$ such that, for any flow $f : [0,N) \to 2^{\mathbf{P}}$, $f \models \varphi$ iff $\overline{f} \models \overline{\varphi}$.

We can assume that φ does not contain any (first- or second-order) existential quantifiers, by replacing the latter with combinations of universal quantifiers and negations if need be. It is also convenient to rewrite φ into a formula that makes use of the integer constants $0, 1, \ldots, N$ as well as a family of unary functions $+k$ (for $k \in \mathbb{N}$) instead of the $+1$ relation. To this end, replace every occurrence of $+1(x, y)$ in φ by $(x < N - 1 \wedge x + 1 = y)$.

Next, replace every instance of $\forall x\, \psi$ in φ by the formula

$$\forall x\, (\psi[x/x] \wedge \psi[x + 1/x] \wedge \ldots \wedge \psi[x + (N - 1)/x])\,,$$

where $\psi[t/x]$ denotes the formula resulting from substituting every free occurrence of the variable x in ψ by the term t. Intuitively, this transformation is legitimate since first-order variables in our target formula will range over $[0, 1)$ rather than $[0, N)$.

Having carried out these substitutions, rewrite every term in φ in the form $x + k$, where x is a variable and $k \in \mathbb{N}$ is a non-negative integer constant.

Every inequality occurring in φ is now of the form $x + k_1 < y + k_2$. Replace every such inequality by (i) $x < y$, if $k_1 = k_2$; (ii) **true**, if $k_1 < k_2$; and (iii) $\neg$**true** otherwise.

Every occurrence of a monadic predicate in φ now has the form $P(x + k)$. Replace every such predicate by (i) $P_k(x)$, if $k \le N-1$, and (ii) $\neg$**true** otherwise.

Finally, replace every occurrence of $\forall P\, \psi$ in φ by $\forall P_0\, \forall P_1 \ldots \forall P_{N-1}\, \psi$. The resulting formula is the desired $\overline{\varphi}$.

It is now straightforward to prove by induction that the set of $[0, N)$-flows satisfying the original φ are indeed in one-to-one correspondence with the set of $[0, 1)$-flows satisfying $\overline{\varphi}$.

Note that $\overline{\varphi}$ does not use any $+1$ or $+k$ functions or relations, and is therefore indeed a non-metric (i.e., purely order-theoretic) sentence in $\mathsf{MSO}(<)$. Moreover, the satisfiability problem for $\mathsf{MSO}(<)$ by finitely-variable flows over $\mathbb{R}_{\ge 0}$ is decidable [36]. However, although $\mathbb{R}_{\ge 0}$ and $[0, 1)$ are order-isomorphic, finitely-variable flows over $\mathbb{R}_{\ge 0}$ do not necessarily translate to finitely-variable flows over $[0, 1)$. In fact, it is easily seen that the finitely-variable flows over $\mathbb{R}_{\ge 0}$ whose counterparts in $[0, 1)$ are finitely-variable are precisely those flows that are ultimately constant: every monadic predicate P is such that there is a time point beyond which P is either always true or always false.

For P a monadic predicate, let us therefore write σ_P to denote the first-order sentence

$$\exists x\, (\forall y\, (x < y \to P(y)) \vee \forall y\, (x < y \to \neg P(y)))\,.$$

We now perform one last transformation on $\overline{\varphi}$, by replacing every instance of $\forall P_i\, \psi$ in $\overline{\varphi}$ by $\forall P_i(\sigma_{P_i} \to \psi)$, yielding a new sentence $\overline{\varphi}'$. Recall that $\overline{\mathbf{P}}$ is the set of monadic predicates appearing freely in $\overline{\varphi}$ (and hence also $\overline{\varphi}'$). We now have that our original sentence φ is satisfiable by finitely-variable flows over $[0, N)$ iff the $\mathsf{MSO}(<)$ sentence

$$\overline{\varphi}' \wedge \bigwedge\{\sigma_{P_i} \mid P_i \in \overline{\mathbf{P}}\}$$

is satisfiable by finitely-variable flows over $\mathbb{R}_{\geq 0}$, which is decidable.

We now tackle the complexity. Satisfiability of $\mathsf{FO}(<)$ over $\mathbb{R}_{\geq 0}$ is known to be non-elementary [41, 27]. An examination of the proof shows that this result in fact also holds over (finitely-variable) flows that are ultimately constant. Since $\mathbb{R}_{\geq 0}$ is order-isomorphic to $[0, N)$, $\mathsf{FO}(<)$, and *a fortiori* $\mathsf{FO}(<, +1)$ and $\mathsf{MSO}(<, +1)$, also have non-elementary time-bounded satisfiability problems.

Let θ now be an MTL formula. It would be possible to show, following an approach somewhat similar to the reduction above of $\mathsf{MSO}(<, +1)$ over $[0, N)$ to $\mathsf{MSO}(<)$, that one can manufacture an equi-satisfiable but exponentially larger formula of *Linear Temporal Logic with past operators* ($\mathsf{LTL}+\mathsf{Past}$). Since $\mathsf{LTL}+\mathsf{Past}$ satisfiability is PSPACE-Complete, satisfiability for θ over $[0, N)$ can be decided in exponential space.

In the interest of brevity, we shall however proceed differently, and translate instead the satisfiability problem for θ over $[0, N)$ to that of a related *Flat Metric Temporal Logic* ($\mathsf{Flat\text{-}MTL}$) formula over $\mathbb{R}_{\geq 0}$. The result will follow by the known EXPSPACE-Completeness of satisfiability for $\mathsf{Flat\text{-}MTL}$ [9].

Note first that we can assume that all intervals I appearing as subscripts to temporal operators in θ are subsets of $[0, N)$—indeed, if this is not the case, replace any offending interval I by $I \cap [0, N)$ to obtain a semantically equivalent formula over $[0, N)$. θ can therefore be taken to be a *Bounded Metric Temporal Logic* ($\mathsf{Bounded\text{-}MTL}$) formula.

Next, let us postulate a fresh predicate T which will be required to hold precisely within the time domain $[0, N)$. We now modify θ by relativising all temporal operators to quantify over the 'absolute' time domain $[0, N)$, as follows: (i) replace every subformula in θ of the form $\Diamond_I \rho$ by $\Diamond_I (T \wedge \rho)$; (ii) replace every subformula in θ of the form $\Box_I \rho$ by $\Box_I (T \rightarrow \rho)$; and (iii) replace every subformula in θ of the form $\rho_1 \, \mathcal{U}_I \, \rho_2$ by $\rho_1 \, \mathcal{U}_I \, (T \wedge \rho_2)$. Let us call the resulting formula θ'.

Consider now the formula

$$\theta' \wedge \Box_{[0,N)} T \wedge \Box_{[N,\infty)} \neg T$$

which belongs to $\mathsf{Flat\text{-}MTL}$ (cf. [9]) since θ' is in $\mathsf{Bounded\text{-}MTL}$. This formula is clearly satisfiable by finitely-variable flows over $\mathbb{R}_{\geq 0}$ iff θ is satisfiable by finitely-variable flows over $[0, N)$. Since the former can be decided in EXPSPACE, then so can the latter.

Finally, the EXPSPACE-Hardness proof of $\mathsf{Bounded\text{-}MTL}$ satisfiability in [9] easily carries over to MTL satisfiability over $[0, N)$. This therefore establishes EXPSPACE-Completeness of time-bounded satisfiability for MTL. $\Box$

Before proving Theorem 2, we first state a preliminary lemma, easily proven by induction.

Lemma 1. *Let φ be an $\mathsf{FO}(<)$ or LTL formula with set of monadic predicates $\mathbf{P}$. Assume that φ is only satisfiable by ultimately-constant flows. Let β be any bijection from $[0, 1)$ to $\mathbb{R}_{\geq 0}$. Then β extends to a bijection between $[0, 1)$-flows* *and $\mathbb{R}_{\geq 0}$-flows that are ultimately constant. Moreover, this bijection preserves and reflect the flows that satisfy φ; in other words, for any flow $f : [0, 1) \to 2^{\mathbf{P}}$, $f \models \varphi$ iff $\beta(f) \models \varphi$.*

Theorem 2. *For any fixed bounded time domain of the form $[0, N)$, with $N \in \mathbb{N}$, the metric logics $\mathsf{FO}(<, +1)$ and MTL are semantically equally expressive. Moreover, this equivalence is effective.*

Proof. Throughout this proof, let $N \in \mathbb{N}$ be fixed.

The reduction from MTL to $\mathsf{FO}(<, +1)$ is straightforward, and therefore omitted.

For the other direction, let φ be an $\mathsf{FO}(<, +1)$ sentence with set of free monadic predicates $\mathbf{P} \subseteq \mathbf{MP}$. As in the proof of Theorem 1, let $\overline{\mathbf{P}} = \{P_i \mid P \in \mathbf{P}, 0 \leq i \leq N - 1\}$ be a set of fresh monadic predicates. The construction used in Theorem 1 yields an $\mathsf{FO}(<)$ sentence $\overline{\varphi}$ with set of free monadic predicates $\overline{\mathbf{P}}$, such that there is a bijection (indicated by overlining) from the set of $[0, N)$-flows over $\mathbf{P}$ satisfying φ to the set of $[0, 1)$-flows over $\overline{\mathbf{P}}$ satisfying $\overline{\varphi}$. Moreover, we can ensure that, when interpreted over $\mathbb{R}_{\geq 0}$, $\overline{\varphi}$ is only satisfied by flows that are ultimately constant.

According to [18], one can now construct an LTL formula ψ, with set of monadic predicates $\overline{\mathbf{P}}$, that defines precisely the same set of finitely-variable $\mathbb{R}_{\geq 0}$-flows as $\overline{\varphi}$. By Lemma 1, $\overline{\varphi}$ and ψ therefore also define precisely the same set of finitely-variable $[0, 1)$-flows over $\overline{\mathbf{P}}$.

It therefore suffices to exhibit an MTL formula θ, over set of monadic predicates $\mathbf{P}$, such that, for any flow $f : [0, N) \to 2^{\mathbf{P}}$, $f \models \theta$ iff $\overline{f} \models \psi$.

To this end, write ι to denote the MTL formula $\Diamond_{=(N-1)}\mathbf{true}$. Note that, when interpreted within the time domain $[0, N)$, ι holds precisely over the time interval $[0, 1)$. Perform the following substitutions on ψ to obtain the desired θ: (i) for each $P \in \mathbf{P}$, replace every occurrence of P_0 in ψ by P, and every occurrence of P_i in ψ (for $i \geq 1$) by $\Diamond_{=i}P$; (ii) replace every occurrence of $\Diamond\gamma$ in ψ by $\Diamond(\iota \wedge \gamma)$; (iii) replace every occurrence of $\Box\gamma$ in ψ by $\Box(\iota \to \gamma)$; (iv) replace every occurrence of $\gamma_1 \,\mathcal{U}\, \gamma_2$ in ψ by $\gamma_1 \,\mathcal{U}\, (\iota \wedge \gamma_2)$.

Finally, show by induction on ψ that, for any flow $f : [0, N) \to 2^{\mathbf{P}}$ and any $t \in [0, 1)$, one has $(f, t) \models \theta$ iff $(\overline{f}, t) \models \psi$. The desired result follows by setting $t = 0$. □

Theorem 3. *The time-bounded model-checking problems for timed automata against the metric logics $\mathsf{MSO}(<, +1)$, $\mathsf{FO}(<, +1)$, and MTL are all decidable, with the same complexities as the corresponding time-bounded satisfiability problems: non-elementary for $\mathsf{MSO}(<, +1)$ and $\mathsf{FO}(<, +1)$, and EXPSPACE-Complete for MTL.*

Proof. Fix $N \in \mathbb{N}$, and let $\mathcal{A}$ be a timed automaton over alphabet Σ. In [16], it is shown how to construct (in polynomial time) an MTL formula $\theta_{\mathcal{A}}$, over a potentially larger set of monadic predicates $\mathbf{P} \supseteq \Sigma$, such that, for any flow $f : [0, N) \to 2^{\Sigma}$, $f \in L_{[0,N)}(\mathcal{A})$ iff there exists a flow $g : [0, N) \to 2^{\mathbf{P}}$ such that $g \models \theta_{\mathcal{A}}$ and $g\restriction_{\Sigma} = f$. Intuitively, the extra monadic predicates of $\theta_{\mathcal{A}}$ keep track of the (otherwise invisible) identity of transitions and clock resets that occur during runs of $\mathcal{A}$.

Of course, $\theta_{\mathcal{A}}$ can clearly instead be taken to be an $\mathsf{FO}(<,+1)$ or $\mathsf{MSO}(<,+1)$ formula, if desired. In all cases, given a metric formula φ, the model-checking problem for $\mathcal{A}$ and φ over $[0,N)$ boils down to whether $\theta_{\mathcal{A}} \wedge \neg\varphi$ is unsatisfiable over $[0,N)$ or not.

This shows that time-bounded model checking reduces to time-bounded satisfiability. For the converse, simply pick an automaton $\mathcal{A}$ that accepts every flow. $\square$

Theorem 4. *The time-bounded language inclusion problem for timed automata is decidable and 2EXPSPACE-Complete.*

Proof. Fix $N \in \mathbb{N}$, and let $\mathcal{A}$ and $\mathcal{B}$ be timed automata over alphabet Σ. We give a procedure for deciding whether $L_{[0,N)}(\mathcal{A}) \subseteq L_{[0,N)}(\mathcal{B})$.

As in the proof of Theorem 3, let $\theta_{\mathcal{A}}$ be an MTL formula over set of monadic predicates $\mathbf{P} = \Sigma \cup \mathbf{U}$, with the property that each $[0,N)$-timed word over Σ accepted by $\mathcal{A}$ can be extended to a $[0,N)$-flow over $\mathbf{P}$ satisfying $\theta_{\mathcal{A}}$, and vice-versa. Likewise, let $\theta_{\mathcal{B}}$ be a similar MTL formula over set of monadic predicates $\mathbf{Q} = \Sigma \cup \mathbf{V}$ for the timed automaton $\mathcal{B}$. We assume that Σ, $\mathbf{U}$, and $\mathbf{V}$ are all pairwise disjoint.

Abusing notation, we see that $L_{[0,N)}(\mathcal{A}) \subseteq L_{[0,N)}(\mathcal{B})$ iff the following formula holds over $[0,N)$:

$$\forall \Sigma\, \forall \mathbf{U}\, \exists \mathbf{V}\, (\neg\theta_{\mathcal{A}}(\Sigma, \mathbf{U}) \vee \theta_{\mathcal{B}}(\Sigma, \mathbf{V}))\,. \tag{1}$$

Observe, as argued in the proof of Theorem 1, that $\theta_{\mathcal{A}}$ and $\theta_{\mathcal{B}}$ can in fact be taken to be $\mathsf{Bounded\text{-}MTL}$ formulas. We can therefore invoke a result of [9] and transform $\neg\theta_{\mathcal{A}} \vee \theta_{\mathcal{B}}$ into an equivalent but exponentially larger formula ψ of $\mathsf{LTL{+}Past}$. More precisely, ψ has a different (and exponentially larger) set of monadic predicates $\mathbf{R} = \overline{\Sigma} \cup \overline{\mathbf{U}} \cup \overline{\mathbf{V}} \cup \mathbf{W}$, yet there is a one-to-one correspondence between the flows satisfying $\neg\theta_{\mathcal{A}} \vee \theta_{\mathcal{B}}$ and those satisfying ψ. We also adjust the outside quantifiers accordingly to transform Formula (1) into the equivalent formula

$$\forall \overline{\Sigma}\, \forall \overline{\mathbf{U}}\, \exists \overline{\mathbf{V}}\, \exists \mathbf{W} \psi(\overline{\Sigma}, \overline{\mathbf{U}}, \overline{\mathbf{V}}, \mathbf{W})\,.$$

Next, following [25, 38], we transform ψ into an equivalent untimed finite-state automaton $\mathcal{C}$ whose transitions are labelled by subsets of $\overline{\Sigma} \cup \overline{\mathbf{U}} \cup \overline{\mathbf{V}} \cup \mathbf{W}$. This incurs a second exponential blowup.

Note that the existential quantifications $\exists \mathbf{W}$ and $\exists \overline{\mathbf{V}}$ simply correspond to relabelling all $\mathbf{W}$- and $\overline{\mathbf{V}}$-labelled transitions in $\mathcal{C}$; this can be carried out in polynomial time. We are therefore asking whether the resulting automaton is universal over $\overline{\Sigma} \cup \overline{\mathbf{U}}$, i.e., accepts any string over this alphabet. Since universality is decidable in PSPACE, the overall procedure can be carried out in doubly-exponential space.

The proof of 2EXPSPACE-Hardness is fairly intricate and will appear in the full version of this paper [29]. $\square$

References

[1] Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Proceedings of LICS. IEEE Computer Society Press, Los Alamitos (1990)
[2] Alur, R., Dill, D.: A theory of timed automata. Theor. Comput. Sci. 126 (1994)
[3] Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM 43(1) (1996)
[4] Alur, R., Fix, L., Henzinger, T.A.: Event-clock automata: A determinizable class of timed automata. Theor. Comput. Sci. 211 (1999)
[5] Alur, R., Henzinger, T.A.: Logics and models of real time: A survey. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 74–106. Springer, Heidelberg (1992)
[6] Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. Inf. and Comput. 104(1) (1993)
[7] Alur, R., La Torre, S., Madhusudan, P.: Perturbed timed automata. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 70–85. Springer, Heidelberg (2005)
[8] Baier, C., Hermanns, H., Katoen, J.-P., Haverkort, B.R.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. Theor. Comput. Sci. 345(1) (2005)
[9] Bouyer, P., Markey, N., Ouaknine, J., Worrell, J.: On expressiveness and complexity in real-time model checking. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 124–135. Springer, Heidelberg (2008)
[10] Bošnački, D.: Digitization of timed automata. In: Proceedings of FMICS (1999)
[11] Emmi, M., Majumdar, R.: Decision problems for the verification of real-time software. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 200–211. Springer, Heidelberg (2006)
[12] Gabbay, D.M., Pnueli, A., Shelah, S., Stavi, J.: On the temporal basis of fairness. In: Proceedings of POPL. ACM Press, New York (1980)
[13] Gupta, V., Henzinger, T.A., Jagadeesan, R.: Robust timed automata. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 331–345. Springer, Heidelberg (1997)
[14] Henzinger, T.A., Manna, Z., Pnueli, A.: What good are digital clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
[15] Henzinger, T.A., Raskin, J.-F.: Robust undecidability of timed and hybrid systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, p. 145. Springer, Heidelberg (2000)
[16] Henzinger, T.A., Raskin, J.-F., Schobbens, P.-Y.: The regular real-time languages. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, p. 580. Springer, Heidelberg (1998)
[17] Herrmann, P.: Timed automata and recognizability. Inf. Process. Lett. 65 (1998)
[18] Hirshfeld, Y., Rabinovich, A.: Future temporal logic needs infinitely many modalities. Inf. Comput. 187(2) (2003)
[19] Hirshfeld, Y., Rabinovich, A.: Logics for real time: Decidability and complexity. Fundam. Inform. 62(1) (2004)
[20] Hirshfeld, Y., Rabinovich, A.: Expressiveness of metric modalities for continuous time. Logical Methods in Computer Science 3(1) (2007)
[21] Kamp, H.: Tense logic and the theory of linear order. Ph.D. Thesis (1968)
[22] Katoen, J.-P., Zapreev, I.S.: Safe on-the-fly steady-state detection for time-bounded reachability. In: Proceedings of QEST. IEEE Computer Society, Los Alamitos (2006)

[23] Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: Timed I/O Automata: A mathematical framework for modeling and analyzing real-time systems. In: Proceedings of RTSS. IEEE Computer Society Press, Los Alamitos (2003)
[24] Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2(4) (1990)
[25] Lutz, C., Walther, D., Wolter, F.: Quantitative temporal logics over the reals: PSpace and below. Inf. and Comput. 205 (2007)
[26] Lynch, N.A., Attiya, H.: Using mappings to prove timing properties. Distributed Computing 6(2) (1992)
[27] Meyer, A.R.: Weak monadic second-order theory of successor is not elementary-recursive. In: Logic colloquium. LNM, vol. 453, pp. 72–73. Springer, Heidelberg (1975)
[28] Ouaknine, J.: Digitisation and full abstraction for dense-time model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, p. 37. Springer, Heidelberg (2002)
[29] Ouaknine, J., Rabinovich, A., Worrell, J.: Time-bounded verification (full version) (in preparation, 2009)
[30] Ouaknine, J., Worrell, J.: Revisiting digitization, robustness, and decidability for timed automata. In: Proceedings of LICS. IEEE Computer Society Press, Los Alamitos (2003)
[31] Ouaknine, J., Worrell, J.: Universality and language inclusion for open and closed timed automata. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 375–388. Springer, Heidelberg (2003)
[32] Ouaknine, J., Worrell, J.: On the language inclusion problem for timed automata: Closing a decidability gap. In: Proceedings of LICS. IEEE Computer Society Press, Los Alamitos (2004)
[33] Ouaknine, J., Worrell, J.: On the decidability of Metric Temporal Logic. In: Proceedings of LICS. IEEE Computer Society Press, Los Alamitos (2005)
[34] Ouaknine, J., Worrell, J.: Safety Metric Temporal Logic is fully decidable. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 411–425. Springer, Heidelberg (2006)
[35] Ouaknine, J., Worrell, J.: On the decidability and complexity of Metric Temporal Logic over finite words. Logical Methods in Computer Science 3(1) (2007)
[36] Rabinovich, A.: Finite variability interpretation of monadic logic of order. Theor. Comput. Sci. 275(1-2) (2002)
[37] Raskin, J.-F.: Logics, Automata and Classical Theories for Deciding Real Time. PhD thesis, University of Namur (1999)
[38] Reynolds, M.: The complexity of temporal logic over the reals (2004) (submitted)
[39] Roux, O., Rusu, V.: Verifying time-bounded properties for ELECTRE reactive programs with stopwatch automata. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1994. LNCS, vol. 999, pp. 105–116. Springer, Heidelberg (1995)
[40] Shelah, S.: The monadic theory of order. Ann. Math. 102 (1975)
[41] Stockmeyer, L.J.: The complexity of decision problems in automata theory and logic. PhD thesis, MIT (1974)
[42] Taşiran, S., Alur, R., Kurshan, R.P., Brayton, R.K.: Verifying abstractions of timed systems. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 546–562. Springer, Heidelberg (1996)

Secure Enforcement for Global Process Specifications

Jérémy Planul[1], Ricardo Corin[1], and Cédric Fournet[1,2]

[1] MSR-INRIA
[2] Microsoft Research

Abstract. Distributed applications may be specified as parallel compositions of processes that summarize their global interactions and hide local implementation details. These processes define a fixed protocol (also known as a contract, or a session) which may be used for instance to compile or verify code for these applications.

Security is a concern when the runtime environment for these applications is not fully trusted. Then, parts of their implementation may run on remote, corrupted machines, which do not comply with the global process specification. To mitigate this concern, one may write defensive implementations that monitor the application run and perform cryptographic checks. However, hand crafting such implementations is ad hoc and error-prone.

We develop a theory of secure implementations for process specifications. We propose a generic defensive implementation scheme, relying on history-tracking mechanisms, and we identify sufficient conditions on processes, expressed as a new type system, that ensure that our implementation is secure for all integrity properties. We illustrate our approach on a series of examples and special cases, including an existing implementation for sequential multiparty sessions.

1 Introduction

Distributed applications may be specified using concurrent processes that capture their global interactions, or *protocol*, and otherwise ignore their local implementation details. Hence, each machine that takes part in the application is assigned a *fixed initial process* (often called a *role* of the protocol) and may run any local code that implements its process, under the global assumption that all other participating machines will also comply with their respective assigned processes. This approach yields strong static guarantees, cuts the number of cases to consider at runtime, and thus simplifies distributed programming. It has been explored using (binary) sessions [6,5] and, more recently, *multiparty sessions* [3,7]. Within sessions, machines exchange messages according to fixed, pre-agreed patterns, for instance a sequence of inputs and outputs between a client and a server; these patterns may be captured by types, expressed as declarative contracts (also known as workflows), or more generally specified as processes.

Global process specifications also provide an adequate level of abstraction to address distributed security concerns. Each machine is then interpreted as a unit of trust, under the control of a principal authorized to run a particular role in the application. Conversely, these principals do not necessarily trust one another. Indeed, many applications commonly involve unknown parties on open networks, and a machine may participate in a session run that also involves untrustworthy machines (either malicious, or compromised, or poorly programmed). In this situation, we still expect *session integrity* to hold: an "honest" machine should accept a message as genuine only if it is enabled by the global specification. To this end, their implementation may perform cryptographic checks, using for example message signatures, as well as checks of causal dependencies between messages.

Cryptographic implementation mechanisms need not appear in the global specification. They can sometimes be automatically generated from this specification. Corin *et al.* perform an initial step in this direction, by securing implementations of n-ary sequential sessions [3,1]. Their sessions specify sequences of communications between n parties as paths in directed graphs. A global session graph is compiled down to secure local implementations for each role, using a custom cryptographic protocol that protects messages and monitors their dependencies. Their main security result is that session integrity is guaranteed for all session runs—even those that involve compromised participants. They also report on prototype implementations, showing that the protocol overhead is small. Although their sequential sessions are simple and intuitive, they also lack flexibility, and are sometimes too restrictive. For instance, concurrent specifications must be decomposed into series of smaller, sequential sessions, and the programmer is left to infer any security properties that span several sub-sessions. More generally, they leave the applicability of their approach to more expressive specification languages as an open problem.

In this paper, we consider a much larger class of session specifications that enable concurrency and synchronization within session runs; and we systematically explore their secure implementation mechanisms, thereby enforcing more global properties in a distributed setting. We aim at supporting arbitrary processes, and as such we depart significantly from prior work: our specification language is closer to a generic process algebraic setting, such as CCS [8]. On the other hand, we leave the cryptography implicit and do not attempt to generalize their concrete protocol design.

Example 1. To illustrate our approach, consider a simple model of an election, with three voting machines (V_1, V_2, and V_3) and one election officer machine (E) in charge of counting the votes for two possible candidates (c_1 and c_2) and sending the result (r_1 or r_2) to a receiver machine (R). We specify the possible actions of each machine using CCS-like processes (defined in Section 2), as follows:

$$V_1 = V_2 = V_3 = \overline{c_1} + \overline{c_2} \qquad E = c_1.c_1.\overline{r_1} \mid c_2.c_2.\overline{r_2} \qquad R = r_1 + r_2$$

Machines running process V_i for $i = 1, 2, 3$ may vote for one of the two candidates by emitting one of the two messages c_1 or c_2; this is expressed as a choice between two output actions. The machine running process E waits for two votes for the same candidate and then fires the result; this is expressed as a parallel composition of two inputs followed by an output action. Finally, the machine running process R waits for one of the two outcomes; this is expressed as a choice between two input actions.

Protocols and applications. We now give an overview of the protocols and the attacker model. Our processes represent protocol specifications, rather than the application code that would run on top of the protocol. Accordingly, our protocol implementation monitors and protects high-level actions driven by an abstract application. The implementation immediately delivers to the application any input action enabled by the specification, and sends to other machines any action output enabled by the specification and selected by the application. The application is in charge of resolving internal choices between different message outputs enabled by the specification, and to provide the message payloads.

We intend that our implementations meet two design constraints: (1) they rely only on the global specification—not on the knowledge of which machines have been compromised, or the mediation of a trusted third party; and (2) they do not introduce any extra message: each high-level communication is mapped to an implementation message. This *message transparency* constraint excludes unrealistic implementations, such as a global, synchronized implementation that reaches a distributed consensus on each action.

Attacker Model. We are interested in the security of any subset of machines that run our implementation, under the assumption that the other machines may be corrupted. In Example 1, for instance, the receiver machine should never accept a result from the election officer if no voter has cast its vote.

Our implementations provide protection against active adversaries that controls parts of the session run: We assume an unsafe network, so the adversary may intercept, reorder, and inject messages—this is in line with classic symbolic models of cryptography pioneered by Dolev and Yao [4]. We also assume partial compromise, so the adversary may control some of the machines supposed to run our implementation, and run instead arbitrary code; hence, those machines need not follow their prescribed roles in the session specification and may instead collude to forge messages in an attempt to confuse the remaining compliant, honest machines.

We focus on global control flow integrity, rather than cryptographic wire formats; thus, we assume that the message components produced by the compliant machines cannot be forged by our adversary. (This can be achieved by standard cryptographic integrity mechanisms, such as digital signatures.) Conversely, the adversary controls the network and may forge any message component from compromised machines. Also, we do not address other security properties of interest, such as payload confidentiality or anonymity.

Contributions. Relying on process calculi, we define an expressive language for specifications and formally describe its implementation. We construct a secure, generic implementation scheme: we propose a general implementability condition, expressed as a type system, and show that it suffices to ensure that any set of compliant machines remain in a globally-consistent state, despite any coordinated attack by an adversary in control of the remaining, corrupted machines.

Contents. Section 2 defines a global semantics for specifications. Section 3 describes their generic implementations, and states their soundness and completeness properties. Section 4 considers binary specifications. Section 5 defines our type system. Section 6 presents our history-tracking implementation and establishes its correctness for well-typed specifications. Section 7 considers sequential n-ary sessions. Section 8 concludes. Additional details and proofs appear in an online paper at `http://msr-inria.inria.fr/projects/sec/sessions`.

2 Global Process Specifications

We consider specifications that consist of distributed parallel compositions of local processes, each process running on its own machine. In the remainder of the paper, we let n be a fixed number of machines, and let $\widetilde{P}$ range over global process specifications, that is, n-tuples of local processes $(P_0, \ldots, P_i, \ldots, P_{n-1})$.

Syntax and informal semantics. Our local processes use a CCS syntax, given below. Their outputs are asynchronous, since we are in a distributed setting.

$P ::=$	Local processes
$\mathbf{0}$	inert process
$\overline{a}$	asynchronous send
$a.P$	asynchronous receive
$P + P'$	choice
$P \mid P'$	parallel fork
$!P$	replication

The specification $\widetilde{P} = (P_0, \ldots, P_{n-1})$ sets a global "contract" between n machines; it dictates that each machine i behaves as specified by P_i. For instance, in Example 1 we have $n = 5$ and $\widetilde{P} = (V_1, V_2, V_3, E, R)$.

Operational semantics ($\rightarrow_{\mathsf{P}}$ and $\rightarrow_{\mathsf{S}}$). We define standard labelled transitions for local processes, with the rules given below. We write $P \xrightarrow{\beta}_{\mathsf{P}} P'$ when process P evolves to P' with action β ranging over a and $\overline{a}$. We omit the symmetric rules for sum and parallel composition.

$$a.P \xrightarrow{a}_{\mathsf{P}} P \qquad \overline{a} \xrightarrow{\overline{a}}_{\mathsf{P}} 0 \qquad \frac{P_0 \xrightarrow{\beta}_{\mathsf{P}} P_0'}{P_0 + P_1 \xrightarrow{\beta}_{\mathsf{P}} P_0'} \qquad \frac{P_0 \xrightarrow{\beta}_{\mathsf{P}} P_0'}{P_0 | P_1 \xrightarrow{\beta}_{\mathsf{P}} P_0' | P_1} \qquad \frac{P \xrightarrow{\beta}_{\mathsf{P}} P'}{!P \xrightarrow{\beta}_{\mathsf{P}} !P | P'}$$

We also define labelled transitions for global configurations, with the communication rule below. We write $\widetilde{P} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P}'$ when $\widetilde{P}$ evolves to $\widetilde{P}'$ by action a with sender P_i and receiver P_j. (The case $i = j$ is for local, but still observable communications.) We let α range over global communication labels $i\ a\ j$, and let φ range over sequences of these labels (written for instance $i\ a\ j.\alpha$) representing high-level traces. We write $\alpha \in \varphi$ when α occurs in φ.

$$\frac{P_i \xrightarrow{\overline{a}}_{\mathsf{P}} P_i^\circ \quad (P_k = P_k^\circ)^{k \neq i} \quad P_j^\circ \xrightarrow{a}_{\mathsf{P}} P_j' \quad (P_k^\circ = P_k')^{k \neq j}}{\widetilde{P} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P}'}$$

3 Distributed Process Implementations

We describe distributed implementations of the specifications of Section 2, each process being mapped to one machine. We separate compliant (honest) machines from compromised (dishonest) machines, then we define their implementations, their semantics, and their properties (soundness and completeness).

We let $\mathcal{C}$ range over subsets of $\{0, \ldots, n-1\}$, representing the indexes of $\widetilde{P}$ whose machines are compliant. We let $\widehat{P}$ range over tuples of processes indexed by $\mathcal{C}$. Intuitively, these machines follow our implementation semantics, whereas the other machines are assumed to be compromised, and may jointly attempt to make the compliant machines deviate from the specification $\widetilde{P}$. For instance, if the election officer of Example 1 is compromised, it may immediately issue a result $\overline{r}_1$ without waiting for two votes c_1. Similarly, a compromised voter may attempt to vote twice. We are interested in protecting (the implementations of) compliant machines from such actions.

We give a generic definition of process implementations. In the following sections, we show how to instantiate this definition for a given specification. Informally, an implementation is a set of programs, one for each specification process to implement, plus a definition of the messages that the adversary may be able to construct, in the spirit of symbolic models for cryptography.

Definition 1 (Distributed implementation). *A* distributed implementation *is a triple* $\langle (Q_i)_{i<n}, (\xrightarrow{\gamma}_i)_{i<n}, (\vdash_{\mathcal{C}})_{\mathcal{C}\subseteq 0..n-1} \rangle$ *(abbreviated* $\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$*) where, for each* $i \in 0..n-1$ *and* $\mathcal{C} \subseteq 0..n-1$*,*

- Q_i *is an implementation process;*
- $\xrightarrow{\gamma}_i$ *is a labelled transition relation between implementation processes;*
- $\vdash_{\mathcal{C}}$ *is a relation between message sets and messages.*

In the definition, each Q_i is an initial implementation process, and each $\xrightarrow{\gamma}_i$ is a specialized transition relation between implementation processes for machine i, labelled with either an input (M) or an output ($\overline{M}$). We let γ range over M and $\overline{M}$, and let ψ range over sequences of γ, representing low-level traces. For each local implementation process, these transitions define the messages that may be sent, and the messages that may be received and accepted as genuine. We assume that every message M implements a single high-level communication α. (Intuitively, M is a wire format for α, carrying additional information.) We write $\rho(\gamma)$ for the corresponding high-level communication α, obtained by parsing γ.

The relations $\vdash_{\mathcal{C}}$ model the capabilities of an adversary that controls all machines outside $\mathcal{C}$ to produce (or forge) a message M after receiving (or eavesdropping) the set of messages $\mathcal{M}$. For instance, if the implementation relies on digital signatures, the definition of $\vdash_{\mathcal{C}}$ may reflect the capability of signing arbitrary messages on behalf of the non-compliant machines.

Example 2. We may implement the request-response protocol with processes $A_0 = \overline{a} \,|\, b$, $A_1 = a.\overline{b}$ by re-using the syntax of specification processes and labels (i.e. M is just α), with initial implementation processes (A_0, A_1), implementation transitions

$$A_0 \xrightarrow{\overline{0\ a\ 1}}_0 b \xrightarrow{1\ b\ 0}_0 \mathbf{0} \qquad\qquad A_1 \xrightarrow{0\ a\ 1}_1 \overline{b} \xrightarrow{\overline{1\ b\ 0}}_1 \mathbf{0}$$

and an adversary that can replay intercepted messages and produce any messages from compromised principals, modelled with the deduction rules

$$\frac{}{\mathcal{M}, M \vdash_{\mathcal{C}} M} \qquad\qquad \frac{i \notin \mathcal{C} \qquad i,j \in \{0,1\} \qquad x \in \{a,b\}}{\mathcal{M} \vdash_{\mathcal{C}} i\ x\ j}$$

Distributed semantics. For a given distributed implementation $\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ and set of compliant machines $\mathcal{C}$, we define a global implementation semantics that collects all message exchanges between compliant machines on the network. In our model, compromised processes do not have implementations; instead, the environment represents an abstract adversary with the capabilities specified by $\vdash_{\mathcal{C}}$. Thus, the global implementation transitions $\xrightarrow{\gamma}_{\mathrm{I}}$ relate tuples of compliant implementation processes $\widehat{Q}$ and also collects all compliant messages $\mathcal{M}$ exchanged on the network (starting with an empty set), with the following two rules:

$$\text{(SendI)}\quad \frac{Q_i \xrightarrow{\overline{M}}_i Q'_i \qquad (Q_k = Q'_k)^{k \neq i} \qquad i \in \mathcal{C} \qquad \rho(\overline{M}) = i\ a\ j}{\mathcal{M}, \widehat{Q} \xrightarrow{\overline{M}}_{\mathrm{I}} \mathcal{M} \cup M, \widehat{Q'}} \qquad \text{(ReceiveI)}\quad \frac{Q_i \xrightarrow{M}_i Q'_i \qquad (Q_k = Q'_k)^{k \neq i} \qquad i \in \mathcal{C} \qquad \mathcal{M} \vdash_{\mathcal{C}} M \qquad \rho(M) = j\ a\ i}{\mathcal{M}, \widehat{Q} \xrightarrow{M}_{\mathrm{I}} \mathcal{M}, \widehat{Q'}}$$

Rule (SendI) simply records the message sent by a compliant participant, after ensuring that its index i matches the sender recorded in the message interpretation $i\ a\ j$ (so that no compliant participant may send a message on behalf of another). Rule (ReceiveI) enables a compliant participant to input any message M that the adversary may produce from $\mathcal{M}$ (including, for instance, any intercepted message) if its index matches the receiver i in its interpretation $j\ a\ i$; this does not affect $\mathcal{M}$, so the adversary may a priori replay M several times. The wire format M may include additional information, such as a nonce to detect message replays, or some evidence of a previously-received message. The example below illustrates the need for both mechanisms.

Example 3. In the specification $A_0 = \overline{a} \,|\, \overline{a}, A_1 = a \,|\, a$, machine A_1 must discriminate between a replay of A_0's first message and A_0's genuine second message, so these two messages must have different wire formats. For instance, an implementation may include a sequence number, or a fresh nonce.

In the specification $A_0 = \overline{a}, A_1 = a.\overline{b}, A_2 = b$, the implementation of A_2 that receives b from A_1 must check that A_1 previously received a from A_0. For instance, the two messages may be signed by their senders, and A_1's message may also include A_0's message, as evidence that its message on b is legitimate.

Concrete threat model and adequacy. The distributed implementations of Definition 1 are expressed in terms of processes and transitions, rather than message handlers for a concrete, cryptographic wire format. (We refer to prior work for sample message formats and protocols for sequential specifications [3,1]).

In the implementation, the use of cryptographic primitives such as digital signatures can prevent the adversary to forge arbitrary messages. On the other hand, since the adversary controls the network, he can resend any intercepted messages and, when it controls a participant, he can at least send any message that this participant could send if it were honest. Accordingly, for any realistic implementation, we intend that the two rules displayed below follow from the definition of $\vdash_{\mathcal{C}}$. We say that an implementation is *adequate* when these two rules are valid.

$$\mathcal{M}, M \vdash_{\mathcal{C}} M \qquad \frac{i \notin \mathcal{C} \qquad Q_i \xrightarrow{\psi}_i Q_i' \xrightarrow{\overline{M_0}}_i Q_i'' \qquad \mathcal{M} \vdash_{\mathcal{C}} M \text{ for each } M \in \psi}{\mathcal{M} \vdash_{\mathcal{C}} M_0}$$

In the second rule, if (1) the adversary controls machine i; (2) a compliant implementation Q_i can send the message M_0 after a trace ψ has been taken; and (3) every message received in ψ can be constructed from $\mathcal{M}$ by the adversary; then he can also construct the message M_0 from $\mathcal{M}$.

Soundness and completeness. We first formally relate high-level traces of the specification, ranged over by φ, which include the communications of all processes, to low-level implementation traces, ranged over by ψ, which record only the inputs and outputs of the compliant processes. We then define our property of soundness, stating that every implementation run corresponds to a run of the specification; and our property of completeness, stating that every specification run corresponds to a run of the implementation. These properties depend only on the implementation and specification traces: trace-equivalent specifications would accept the same sound and complete implementations. The correspondence between high-level and low-level traces is captured by the following definition of *valid explanations.*

Definition 2. *A high-level trace $\varphi = \alpha_0 \dots \alpha_{p-1}$ is a* valid explanation *of a low-level trace $\psi = \gamma_0 \dots \gamma_{q-1}$ for a given set $\mathcal{C}$ when there is a partial function ι from (indexes of) low-level messages γ_k of ψ to (indexes of) high-level communications $\alpha_{\iota(k)}$ of φ such that $\rho(\gamma_k) = \alpha_{\iota(k)}$ for $k \in 0..q-1$ and*

1. *the restriction of ι on the indexes of the low-level inputs of ψ is a one-to-one, increasing function to the indexes of the high-level communications of φ with honest receivers ($i \ a \ j$ with $j \in \mathcal{C}$);*
2. *the restriction of ι on the indexes of the low-level outputs of ψ is a partial one-to-one function to the indexes of the high-level communications of φ with honest senders ($i \ a \ j$ with $i \in \mathcal{C}$); and*
3. *whenever a low-level input precedes a low-level output, their images by ι are in the same order when defined.*

The definition relates every trace of messages sent and received by honest implementations to a global trace of communications between specification processes (including compromised processes). In particular, the specification trace may have additional communications between compromised processes. The relation guarantees that the implementation messages are received in the same order as the communications in the specification trace. Conversely, since the adversary controls the network, the relation does not guarantee that all low-level outputs are received, or that they are received in order.

For example, consider an implementation that uses specification messages as wire format (that is, M is just α) and let $\mathcal{C} = \{0; 2\}$. We may reflect the trace

$$\psi = (\overline{0\ b\ 2}).(\overline{0\ c\ 1}).(1\ d\ 2).(\overline{0\ a\ 1}).(0\ b\ 2)$$

using, for instance, the valid explanation $\varphi_1 = (0\ c\ 1).(1\ d\ 2).(0\ b\ 2)$ and the index function ι from ψ to φ_1 defined by $\{0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 1, 4 \mapsto 2\}$. Following Definition 2, we check that

1. the restriction of ι to the indexes of the low-level inputs is $\{2 \mapsto 1; 4 \mapsto 2\}$ and relates the inputs of compliant participants in ψ and in φ;
2. the restriction of ι to the indexes of the low-level outputs is $\{0 \mapsto 2; 1 \mapsto 0\}$ and relates two of the three outputs in ψ to the outputs of compliant participants in φ, out of order; and
3. $(1\ d\ 2)$ precedes $(\overline{0\ a\ 1})$ in ψ, but ι is undefined on that low-level output.

Another valid explanation is $\varphi_2 = (1\ d\ 2).(0\ a\ 1).(0\ c\ 1).(0\ b\ 2)$.

We are now ready to define our main security property, which states that an implementation is *sound* when the compliant machines cannot be driven into an execution disallowed by the global specification.

Definition 3 (Soundness). *$\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a sound implementation of $\widetilde{P}$ when, for every $\mathcal{C} \subseteq 0..n-1$ and every implementation trace $\emptyset, \widehat{Q} \xrightarrow{\psi}_{\mathrm{I}} \mathcal{M}, \widehat{Q}'$, there exists a source trace $\widetilde{P} \xrightarrow{\varphi}_{\mathrm{S}} \widetilde{P}'$ where φ is a valid explanation of ψ.*

Also, an implementation is *complete* if, when all machines comply, every trace of the global specification can be simulated by a trace of the implementation.

Definition 4 (Completeness). *$\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a complete implementation of $\widetilde{P}$ when, for $\mathcal{C} = 0..n-1$ and for every source traces $\widetilde{P} \xrightarrow{\varphi}_{\mathrm{S}} \widetilde{P}'$, there exists an implementation trace $\emptyset, \widetilde{Q} \xrightarrow{\psi}_{\mathrm{I}} \mathcal{M}, \widetilde{Q}'$ where φ is a valid explanation of ψ.*

We easily check that, with our definition of valid explanations, an implementation that is both sound and complete also satisfies the message transparency property discussed in Section 1.

4 Implementing Two-Party Specifications (Application)

We instantiate our definitions to specifications with only two participants, such as a client and a server. Such specifications have been much studied using session types. For this section, we set $n = 2$ and implement specifications of the form $\widetilde{P} = (P_0, P_1)$. For simplicity, we also exclude local communications: for any action a, each P_i for $i = 0, 1$ may include either a or $\overline{a}$, but not both.

A simple (insecure) implementation. An implementation that re-uses the syntax of specification processes and labels with initial implementation processes (P_0, P_1) (that is, Q_i is just P_i, and M is just α) is generally not sound. Consider, for instance, the specification $A_0 = a.\overline{e} \mid b.(\overline{e} + \overline{c})$ and $A_1 = \overline{a} \mid \overline{b} \mid e \mid c$. After the communications a, e, and b, the implementation of machine 1 would be in state c, while that of machine 0 would be in state $\overline{e} + \overline{c}$: machine 1 should accept a message c. However, after the communications b, e, and a, the implementation of machine 1 would still be in state c, but that of machine 0 would be in state $\overline{e}$, unable to send c: machine 1 should not accept a message c. In state c machine 1 does not know whether a message c from machine 0 is legitimate or not. Therefore, an implementation accepting the message c is unsound, and an implementation refusing it is incomplete.

History-tracking Implementations. Our implementation relies on a refinement of the specification syntax and semantics to keep track of past communications: local processes are of the form $P : \psi$ where ψ is a sequence of global communications, each tagged with a fresh nonce ℓ used to prevent message replays (any received message whose tag already occurs in ψ is ignored).

- We use $P_0 : \varepsilon$ and $P_1 : \varepsilon$ as initial processes (where ε is the empty sequence).
- We define local implementation transitions $\rightarrow_i$ from the initial specification $\widetilde{P}$ and the specifications traces:

$$\frac{\widetilde{P} \xrightarrow{\rho(\psi)}_{\mathsf{S}} \widetilde{P'} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P''} \qquad i\ a\ j\ \ell \notin \psi}{P'_i : \psi \xrightarrow{\overline{i\ a\ j\ \ell}}_i P''_i : \psi.(i\ a\ j\ \ell)} \quad \frac{\widetilde{P} \xrightarrow{\rho(\psi)}_{\mathsf{S}} \widetilde{P'} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P''} \qquad i\ a\ j\ \ell \notin \psi}{P'_j : \psi \xrightarrow{i\ a\ j\ \ell}_j P''_j : \psi.(i\ a\ j\ \ell)}$$

  where i, j is either $0, 1$ or $1, 0$ and where ρ yields a specification trace by erasing all nonces ℓ in ψ.
- We define the adversary knowledge $\vdash_{\mathcal{C}}$ by $\dfrac{}{\mathcal{M}, M \vdash_{\mathcal{C}} M}$ and $\dfrac{i \notin \mathcal{C}}{\mathcal{M} \vdash_{\mathcal{C}} i\ a\ j\ \ell}$.

Hence, an action is locally enabled only when it extends the specification trace recorded so far and the nonce ℓ is fresh. The adversary may send a message either by eavesdropping it or by constructing it with a compromised sender. (Pragmatically, a concrete implementation may generate ℓ at random, or increment a message sequence number, and may use a more compact representation of ψ.)

Theorem 1. *$\langle (P_0 : \varepsilon, P_1 : \varepsilon), \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a sound and complete implementation of (P_0, P_1).*

The soundness of our implementation above relies on every machine recording every communication (since it is either sending or receiving every message); this approach does not extend to specifications with more than two machines, inasmuch as these machines do not directly observe actions between two remote machines.

5 Implementability by Typing

In the preceding section, we presented a complete and sound implementation for binary sessions. We now illustrate some difficulties in the general n-ary case.

Example 4. Consider a variant of Example 1 with the same V_1, V_2, V_3 and R but with the election officer E split into $E_1 = c_1.c_1.\overline{r_1}$ and $E_2 = c_2.c_2.\overline{r_2}$. One of the voters (say, V_1) may cheat, and send *both* c_1 to E_1 and c_2 to E_2. To prevent this attack, E_1 and E_2 would need to communicate with one another, thereby breaking message transparency. Therefore no adequate implementation of this example can be both sound and complete. To prevent this pattern, we will demand that both sides of a sum affect the same participants in the same order.

Example 5. Consider now the specification

$$A = (\overline{b_1} \mid \overline{c_1}) + (\overline{b_2} \mid \overline{c_2}) \qquad B = b_1 \mid b_2 \qquad C = c_1 \mid c_2$$

The process A can send either b_1 to B and c_1 to C, or b_2 to B and c_2 to C. A dishonest machines in charge of running A can send b_1 to B and c_2 to C. Therefore no adequate implementation of this example can be both sound and complete. To prevent this pattern, we will also demand that both sides of some parallel composition (e.g.under a sum) affect the same participants in the same order.

To prevent these situations, we develop a type system with two kinds of types, for sequential processes and (possibly) parallel processes:

$\sigma ::=$	sequential types	$\pi ::=$	parallel types
$\mathbf{0}$	completion	σ	sequential type
$i.\sigma$	sequence	$i.\pi$	sequence
		$\pi \mid \pi$	parallel

Intuitively, our types indicate (by their indexes) which other participants may be affected by each action, and in what order. In Example 1, action c_1 would be of type 3.4 since it is received by process 3 and this reception may contribute to the emission of r_1 to process 4.

We define subtyping with three base rules and two context rules:

$$\mathbf{0} \leq \sigma \qquad \pi \mid \pi \leq \pi \qquad \pi \leq \pi \mid \pi \qquad \frac{\pi \leq \pi'}{i.\pi \leq i.\pi'} \qquad \frac{\pi_1 \leq \pi_1' \quad \pi_2 \leq \pi_2'}{\pi_1 \mid \pi_2 \leq \pi_1' \mid \pi_2'}$$

Thus, for a sequential type, we can "forget" potential future actions and obtain a less precise type and, for parallel types, we can duplicate or merge parallel copies carrying the same information.

We type local processes at each machine $i \in 0..n-1$, in a given environments Γ that map channels to parallel types. The typing judgment $\Gamma \vdash_i P : \pi$ indicates that P can be given type π at machine i in environment Γ, with the rules below:

$$\text{(Send)}\quad \Gamma, a:\pi \vdash_i \overline{a} : i.(\pi\backslash i) \qquad \text{(Receive)}\quad \frac{\Gamma, a:\pi \vdash_i P:\pi' \quad \pi' \le \pi}{\Gamma, a:\pi \vdash_i a.P:\pi'} \qquad \text{(Sub)}\quad \frac{\Gamma \vdash_i P:\pi \quad \pi \le \pi'}{\Gamma \vdash_i P:\pi'}$$

$$\text{(Nil)}\quad \Gamma \vdash_i \mathbf{0} : i \qquad \text{(Plus)}\quad \frac{\Gamma \vdash_i P_0:\sigma \quad \Gamma \vdash_i P_1:\sigma}{\Gamma \vdash_i P_0 + P_1:\sigma} \qquad \text{(Par)}\quad \frac{\Gamma \vdash_i P_0:\pi \quad \Gamma \vdash_i P_1:\pi'}{\Gamma \vdash_i P_0 \,|\, P_1:\pi \,|\, \pi'} \qquad \text{(Repl)}\quad \frac{\Gamma \vdash_i P:\pi}{\Gamma \vdash_i !P:\pi}$$

where $\pi\backslash i$ is π after erasure of every occurrence of i.

Rule (Send) gives to the output $\overline{a}$ the type of action a (minus i) preceded by i. This records that $\overline{a}$ at host i affects any process that receives on a. Conversely, rule (Receive) gives to $a.P$ the type of the continuation process P, and checks that it is at least as precise as the type of action a. Rule (Sub) enables subtyping. Rule (Nil) gives type i to an empty process, since it has no impact outside i. Rule (Plus) ensures that the two branches of a choice have the same effect, a sequential type, excluding e.g.the typing of the specifications in Example 4. Rules (Par) and (Repl) deal with parallel compositions.

For instance, in the environment $\Gamma = r_1 : 4, r_2 : 4, c_1 : 3.4, c_2 : 3.4$, the processes of Example 1 have types

$$\Gamma \vdash_0 V_1 : 0.3.4 \quad \Gamma \vdash_1 V_2 : 1.3.4 \quad \Gamma \vdash_2 V_3 : 2.3.4 \quad \Gamma \vdash_3 E : 3.4 \quad \Gamma \vdash_4 R : 4$$

Conversely, the processes $V_1 = V_2 = V_3 = \overline{c_1} + \overline{c_2}$ are not typable within the unsafe specification of Example 4, because $\overline{c_1}$ and $\overline{c_2}$ necessarily have incompatible types.

We end this section by defining typability for global specifications, with a shared environment for all machines and a technical condition to ensure consistency on channels with parallel types.

Definition 5. *A global specification $\widetilde{P}$ is* well-typed *when, for some environment Γ and each $i \in 0..n-1$, we have $\Gamma \vdash_i P_i : \pi_i$ and, for each $(a : \pi) \in \Gamma$, either π is (a subtype of) a sequential type, or $\widetilde{P}$ has at most one reception on a.*

6 History-Tracking Implementations

In this section we present an implementation for session specifications. We prove that the implementation is complete, and that it is sound when the specification is well-typed (Definition 5). The resulting family of implementations subsumes those presented in the special cases of binary sessions (Section 4) and sequential sessions (Section 7).

Multiparty specifications and history-tracking implementations. As seen in Example 1, in a multiparty system, a local action at one machine may causally depend on communications between other machines. To avoid cheating, we embed evidence of past execution history in our implementation messages. Thus, to implement Example 1, the code for the election officer E explicitly forwards evidence of receiving c_1 twice in order to convince R that it can send the result r_1.

As a preliminary step, we enrich processes with histories of prior communications. Then, we equip these processes with a refined semantics, with rules that define how histories are collected and communicated. Finally, the presence of histories allows us to constrain each local implementation by prescribing what messages may be sent and received at runtime. (Our history-tracking implementation is related to locality semantics for CCS; for instance Boudol and Castellani [2] use *proved labelled transitions* that keep track of causality by recording where each action occurs in a process.)

Histories are lists of messages, defined by the following grammar:

$H ::=$	Histories
$\quad\varepsilon$	empty history
$\quad H.M$	recorded receive
$M ::=$	Messages
$\quad(H\ i\ a\ j\ \ell)$	

Each message $H\ i\ a\ j\ \ell$ records an action a between sender i and receiver j (where i and j are indexes of processes in the global specification), with a history H that provides evidence that action a is indeed enabled. In addition, ℓ denotes a unique nonce, freshly generated for this message, used to avoid replays.

The syntax of processes extended with histories is as follows:

$T ::=$	Threads
$\quad\mathbf{0}$	inert thread
$\quad\overline{a}$	asynchronous send
$\quad a.P$	asynchronous receive
$\quad P + P'$	choice
$\quad !P$	replication
$R ::=$	History-tracking processes
$\quad(T_0 : H_0 \mid T_1 : H_1 \mid \ldots \mid T_{k-1} : H_{k-1})$	
	parallel composition of history-tracking threads
$\widetilde{R} ::=$	Global history-tracking specifications
$\quad(R_0, R_1, \ldots, R_{n-1})$	tuple of n history-tracking processes

where P ranges over the local processes of Section 2. Our specification processes are split into different parallel components, each with its own history. For example, when $P = a.\overline{b} \mid \overline{c}$ receives a, this receive enables action $\overline{b}$ (and is tracked in its history) but is independent from action $\overline{c}$. So, a *thread* T is a (specification) process without parallel composition at top-level, a *history-tracking process* R is a collection of threads in parallel, each with its history, and a *global history-tracking specification* $\widetilde{R}$ is a tuple of n history-tracking processes.

The function $\mathit{Ths}(P : H)$ normalizes the process P into a parallel composition of threads, each annotated with the same history H. It is recursively defined from $\mathit{Ths}(P_0 \mid P_1 : H) = \mathit{Ths}(P_0 : H) \mid \mathit{Ths}(P_1 : H)$. Further, the function $[\![\cdot]\!]_0$ normalizes a global specification, with an initial, empty history. Conversely, since any thread is a local process, a history-tracking process (resp. a global history specification) stripped of its histories is a local process (resp. a global specification).

Semantics of history specifications ($\rightarrow_{\mathsf{h}}$ and $\rightarrow_{\mathsf{H}}$). We define labelled transitions for history specifications. We write $R \xrightarrow{\gamma}_{\mathsf{h}} R'$ when R can evolve to R' with action γ. It corresponds to an input or an output on one of its threads.

(SendH)
$$\frac{T \xrightarrow{\overline{a}}_{\mathsf{P}} P}{T : H \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} P : H}$$

(ReceiveH)
$$\frac{T \xrightarrow{a}_{\mathsf{P}} P}{T : H' \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} P : H'.(H\ i\ a\ j\ \ell)}$$

(ParH)
$$\frac{R \xrightarrow{\gamma}_{\mathsf{h}} R'}{R \mid R'' \xrightarrow{\gamma}_{\mathsf{h}} R' \mid R''}$$

Rule (ReceiveH) records the message in the thread history. In contrast, rule (SendH) does not record the message, since our semantics is asynchronous. Rule (ParH) is a rule for parallel contexts; we omit the symmetric rule.

We write $\widetilde{R} \xrightarrow{M}_{\mathsf{H}} \widetilde{R}'$ to represent communications between history-tracking specifications, with a single global rule:

$$\frac{\begin{array}{c} R_i \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R_i^\circ \quad (R_k = R_k^\circ)^{k \neq i} \\ R_j^\circ \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} R_j' \quad (R_k^\circ = R_k')^{k \neq j} \quad H\ i\ a\ j\ \ell \notin \widetilde{R} \end{array}}{\widetilde{R} \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{H}} \widetilde{R}'}$$

A communication step consists of a send (at machine i) followed by a receive (at machine j) for local history-tracking processes (possibly with $i = j$). The condition $H\ i\ a\ j\ \ell \notin \widetilde{R}$ excludes multiple usage of the same message.

Example 6. Consider a global specification with three processes $\overline{a}$, $a.\overline{b}$ and b; the following is a trace of its global history specification.

$$\begin{array}{l} [\![(\overline{a}, a.\overline{b}, b)]\!]_0 \xrightarrow{\varepsilon\ 0\ a\ 1}_{\mathsf{H}} (\mathbf{0} : \varepsilon, \overline{b} : \varepsilon\ 0\ a\ 1, b : \varepsilon) \\ \qquad \xrightarrow{(\varepsilon\ 0\ a\ 1)1\ b\ 2}_{\mathsf{H}} (\mathbf{0} : \varepsilon, \mathbf{0} : \varepsilon\ 0\ a\ 1, \mathbf{0} : (\varepsilon\ 0\ a\ 1)1\ b\ 2) \end{array}$$

Local semantics ($\rightarrow_i$). We are now ready to define distributed implementation transitions locally, for each machine $i \in 0..n-1$, written $R_i \xrightarrow{M}_i R_i'$:

$$\frac{\begin{array}{c} [\![\widetilde{P}]\!]_0 \xrightarrow{\psi}_{\mathsf{H}} \widetilde{R'} \quad H\ i\ a\ j\ \ell \notin \widetilde{R'} \\ R_i' \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R_i^\circ \quad (R_k' = R_k^\circ)^{k \neq i} \\ R_j^\circ \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} R_j'' \quad (R_k^\circ = R_k'')^{k \neq j} \end{array}}{R_i' \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_i R_i^\circ} \qquad \frac{\begin{array}{c} [\![\widetilde{P}]\!]_0 \xrightarrow{\psi}_{\mathsf{H}} \widetilde{R'} \quad H\ i\ a\ j\ \ell \notin \widetilde{R'} \\ R_i' \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R_i^\circ \quad (R_k' = R_k^\circ)^{k \neq i} \\ R_j^\circ \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} R_j'' \quad (R_k^\circ = R_k'')^{k \neq j} \end{array}}{R_j^\circ \xrightarrow{H\ i\ a\ j\ \ell}_j R_j''}$$

These rules (with identical premises) prescribe that a distributed implementation can send or receive a message when the corresponding communication is enabled in some global history specification state that is reachable from the initial history specification process $[\![\widetilde{P}]\!]_0$.

A naive concrete implementation may enumerate all possible runs at every communication. A more efficient implementation would cache this computation and perform incremental checks, or perform this computation at compile-time. (See [3,1] for optimized implementations in the sequential case.)

Distributed implementation. We finally define our distributed implementation, with $\widetilde{Q} = [\![\widetilde{P}]\!]_0$ as initial implementation processes, with $(\xrightarrow{\gamma}_i)_i$ defined above as transition relations between implementation processes, and with capabilities $\vdash_{\mathcal{C}}$ for the adversary defined by

$$\mathcal{M}, M \vdash_{\mathcal{C}} M \qquad\qquad \frac{(\mathcal{M} \vdash_{\mathcal{C}} M_m)^{m<k} \qquad i \notin \mathcal{C}}{\mathcal{M} \vdash_{\mathcal{C}} M_0.\cdots.M_{k-1}\ i\ a\ j\ \ell}$$

The first rule states that the adversary can eavesdrop messages on the network. The second rule states that the adversary can build any message sent by a dishonest participant, with a history recursively composed of sequence of messages previously obtained. Conversely, the adversary cannot forge any message from a compliant machine (i.e.a machine $i \in \mathcal{C}$). This can be cryptographically enforced by authenticating messages and countersigning their histories. Our implementation is adequate, in particular a dishonest participant can behave as an honest participant.

(The global transition rules $\rightarrow_{\mathrm{I}}$ for our distributed implementation follow from the general definitions of Section 3.)

Soundness and completeness. Our implementation is complete, that is, it can simulate any specification trace:

Theorem 2 (Completeness). $\langle [\![\widetilde{P}]\!]_0, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ *is a complete implementation of* $\widetilde{P}$.

Our main result states that our implementation is also sound *when applied to well-typed specifications*; as explained in Section 5, many other specifications cannot be safely implemented.

Theorem 3 (Soundness by Typing). *If* $\widetilde{P}$ *is well-typed, then* $\langle [\![\widetilde{P}]\!]_0, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ *is sound.*

7 Sequential Multiparty Sessions (Application)

We consider secure implementations of sequential multiparty sessions, as defined by Corin *et al.* [3]. Their sessions are a special case of process specifications. We recall their grammar, which defines a session as a parallel composition of *role processes*, each process specifying the local actions for one role of the session.

$\tau ::=$	Payload types
$\mathrm{int} \mid \mathrm{string}$	base types
$p ::=$	Role processes
$!(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$	send
$?(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$	receive
$\mu\chi.p$	recursion declaration
χ	recursion
0	end
$S ::=$	Sequential session (with n roles)
$(r_i = p_i)_{i \in 0..n-1}$	

Their role processes must alternate between send and receive actions, and moreover only the initiator role process (p_0) begins with a send. Thus, there is always at most one role that can send the next message, as expected of a sequential session. From a more global viewpoint, a session is represented as a directed graph, whose nodes represent roles and whose arrows are indexed by unique communication labels. The paths in the graph correspond to the global execution traces for the session. Given an additional implementability property on these paths, named "no blind fork", they construct a cryptographic implementation that guarantees *session integrity*, even for sessions with dishonest participants, a property closely related to Soundness (Definition 3).

To illustrate our approach, we show that our generic implementation directly applies to every session that they implement (although with less compact message formats). We translate their role processes into our syntax as follows:

$$\begin{aligned}
[\![p]\!] = {} & \textstyle\sum_{i<k}(\overline{f_i}) \qquad \text{when } p = !(f_i : \widetilde{\tau}_i \ ; \ p_i)_{i<k} \\
& \mid \textstyle\prod_{(?(f_i:\widetilde{\tau}_i \ ; \ p_i)_{i<k})\in p,\ i<k,\ p_i=!(f'_j:\widetilde{\tau}'_j \ ; \ p'_j)_{j<l}} !f_i . \sum_{j<l}(\overline{f'_j}) \\
& \mid \textstyle\prod_{(?(f_i:\widetilde{\tau}_i \ ; \ p_i)_{i<k})\in p,\ i<k,\ p_i=\chi,\ (\mu\chi.!(f_j:\widetilde{\tau}_j \ ; \ p_j)_{j<l})\in p} !f_i . \sum_{j<l}(\overline{f'_j}) \\
[\![(r_i = p_i)_{i\in 0..n-1}]\!] = {} & ([\![p_i]\!])_{i\in 0..n-1}
\end{aligned}$$

Each node in a session graph has an input arrow and one or several output arrows, representing an internal choice between outputs. Accordingly, our translation associates to each node a replicated input (using $q \in p$ to denote any syntactic subprocess q of p) following by a choice between asynchronous outputs. In addition, the initial role for the session is an internal choice between outputs, translated to an internal choice of asynchronous outputs. By induction on paths in the graph, we can check that our translation behaves as the initial sequential session. (The sequentiality of the session follows from the presence of a single choice between outputs in every reachable state, so we can replicate all inputs, whether they occur in recursive loops or not.)

For any given sequential session, typability of the translation (Definition 5) coincides with the "no blind fork" property [3]. Hence, every sequential session supported by their implementation is typable, and can also be implemented in our general framework:

Theorem 4. *The history-tracking implementation of the translation of a sequential session that respects the "no blind fork" property is sound and complete.*

8 Conclusions

We have given an account of distributed specifications and their implementations in three steps: (1) a global specification language; (2) a distributed implementation semantics; (3) correctness and completeness results, depending on an implementability condition. In combination, this yields a general framework for designing and verifying n-ary communication abstractions with strong, guaranteed security properties. (In comparison, the work on sequential multiparty sessions [3] can now be seen as a specialized cryptographic implementation for the sequential case.)

Future work. Each of the above contributions may be improved. First, the specification language may be extended, for example by accounting for the message contents and stating additional security goals (secrecy, causality, commitment) or by adding mobile channel names. Second, the implementation semantics may be further refined. Although we believe our implementation approach is fully general, its performance can clearly be improved, for example by avoiding redundant communications of history once it is either irrelevant or common knowledge. Also, even if the cryptographic protection mechanisms are standard, their efficient implementation remains delicate. More experimentally, we have not prototyped an actual session compiler for our implementation scheme, and it remains unclear how to deal efficiently e.g.with infinite numbers of states. This leads us to our third point, possible improvement on implementability conditions: typability only provides a sufficient condition; we have built an efficient (quadratic) typability verifier, and our only examples of specifications that are sound but not typable can be easily rewritten into typable ones, but still it would be interesting to address this gap.

Acknowledgements. Karthikeyan Bhargavan, Pierre-Malo Deniélou, Gurvan le Guernic, Cosimo Laneve, James Leifer, Jean-Jacques Lévy, Eugen Zălinescu, and the anonymous reviewers provided useful comments on this work.

References

1. Bhargavan, K., Corin, R., Deniélou, P.-M., Fournet, C., Leifer, J.J.: Cryptographic protocol synthesis and verification for multiparty sessions. In: 22nd IEEE Computer Security Foundations Symposium (CSF 2009) (July 2009)
2. Boudol, G., Castellani, I.: Flow models of distributed computations: three equivalent semantics for CCS. Information and Computation 114(2), 247–314 (1994)
3. Corin, R., Deniélou, P.-M., Fournet, C., Bhargavan, K., Leifer, J.J.: A secure compiler for session abstractions. Journal of Computer Security (Special issue for CSF 2007) 16(5), 573–636 (2008)
4. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
5. Gay, S.J., Hole, M.: Types and subtypes for client-server interactions. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 74–90. Springer, Heidelberg (1999)
6. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
7. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 273–284. ACM, New York (2008)
8. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., Englewood Cliffs (1989)

On Convergence of Concurrent Systems under Regular Interactions

Pavithra Prabhakar, Sayan Mitra, and Mahesh Viswanathan

University of Illinois, Urbana-Champaign

Abstract. Convergence is often the key liveness property for distributed systems that interact with physical processes. Techniques for proving convergence (asymptotic stability) have been extensively studied by control theorists. In particular, for the asynchronous model of computation Tsitsiklis [8] provides a set of necessary and sufficient conditions for proving stability and convergence under the assumption that each asynchronous operator (state transition function) is applied infinitely often. This paper generalize these results to obtain necessary and sufficient conditions for systems where the infinite sequence of operators is a member of an arbitrary omega regular language. This enables us to apply our theory to distributed systems with changing communication topology, node failures and joins. We illustrate an application of the new set of conditions in verifying the convergence of a simple (continuous) consensus protocol.

1 Introduction

Convergence or asymptotic stability is a key requirement of many concurrent and distributed systems that interact with physical processes. Roughly, a system A *converges* to a target state x^* if the state of A along infinite executions get closer and closer to x^*, with respect to some topology on the state space X, as time goes to infinity. While termination has been the defacto liveness property of interest for software systems, the more general convergence property becomes relevant for systems with both software and physical components. Examples of such systems include algorithms for mobile robots for forming a spatial pattern, synchronization of coupled oscillators, distributed control algorithms over switching networks [7](see for e.g. [4], [1], [2] and [6]). Convergence may indeed be viewed as a liveness property quantified over a (possibly infinite) sequence of shrinking predicates containing the target state.

Necessary and sufficient conditions for proving convergence of distributed systems which broadly fall under the category of *continuous consensus* have been studied extensively by control theorists for over three decades [7]. Specifically, two types of models of distributed computation have been considered. In the synchronous model, the state of the entire system $x \in X$ evolves according to some difference equation: $x_{k+1} = f(x_k)$ or differential equation $\dot{x} = f(x)$, where $f : X \to X$. Convergence conditions in this case are derived based on the eigenvalues of f. We refer the reader to [7] for a survey of the results of this type. In the asynchronous model, the evolution of the system is specified by a collection of transition functions $\{T_k\}$, where each $T_k : X \to X$, and ans execution of the system is obtained by applying an infinite sequence σ of $T_k's$ to the starting state. In [8], Tsitsiklis has identified a general set of necessary and sufficient conditions for the convergence of executions that satisfy a particular fairness assumption.

Tsitsiklis' condition, informally, is as follows. He requires one to identify a collection of shrinking neighborhoods, indexed by a totally ordered index set, that converges to x^* and satisfies the following properties. First the neighborhoods are required to be invariant, i.e., for any neighborhood U, $T_k(x) \in U$ for every $x \in U$ and every T_k. Second, for every neighborhood U, there must be a transition T_U that takes U to a strictly smaller neighborhood. Tsitsiklis shows that when such a neighborhood "system" exists, the system can be proved to converge to x^* in every execution where each transition T_k is applied *infinitely often*. Moreover, he shows that the convergence of a system also implies the existence of such a neighborhood system.

In this paper, we generalize Tsitsiklis' observations as follows. We identify necessary and sufficient conditions for convergence under executions described by an *arbitrary* ω-regular language, instead of focusing on a particular set of executions that satisfy a specific fairness condition. While this is a philosophically natural extension of Tsitsiklis' investigations, it allows us to model a variety of asynchronous behavior, such as ordered execution of certain events, communication patterns between distributed agents over a dynamically evolving or unreliable communication network, and distributed network with nodes failing and recovering, that are not captured by Tsitsiklis' original formulation.

Our necessary and sufficient condition for convergence is remarkably similar to Tsitsiklis' condition. Let us assume that $\mathcal{A}$ is a Müller automaton that describes the set of valid executions. Once again, we require a collection of shrinking neighborhoods, indexed by a totally ordered index set, that converges to x^*. We also require these neighborhoods to be "invariant". However, since every finite sequence of operations need not be the prefix of a valid execution, our definition of invariance accounts for the state of the automaton $\mathcal{A}$. Next, like Tsitsiklis, we have a condition that ensures "progress towards" x^* is eventually made. This is captured by our insight that edges crossing "cuts" in accepting cycles of $\mathcal{A}$ are traversed infinitely often, and so for every neighborhood set U, there must be some cut that ensures progress. The proof showing that these conditions are sufficient, is very similar to Tsitsiklis' proof. To demonstrate the necessity of these condition for convergence is more challenging primarily because every finite sequence of operations need not be the prefix of a valid execution.

We conclude the paper by demonstrating the application of the new set of conditions to prove the convergence of a simple continuous consensus algorithm. We consider a variety of scenarios ranging from a dynamically evolving communication network, to a situation where nodes in the distributed system can fail and recover.

Related Work. Tsitsiklis' result for the asynchronous model have been extended in several ways. For example, in [7] sufficient conditions have been given for proving convergence of distributed algorithms in which the communication graph of the participating agents is dynamic, but never permanently partitioned. More recently, in [5] sufficient conditions for convergence have been derived for partially synchronous systems where messages may be lost or delayed by some constant but unknown time. All of these constrained executions can be modelled as ω-regular languages, and therefore the results of this paper can be seen as a generalization of these observations.

2 Motivating Example

We model the behavior of a distributed system where agents starting at arbitrary positions on a line communicate based on an underlying dynamic graph to move closer to each other. In addition they can fail and join the system a finite number of times. However when they join they start at the same position in which they originally started. We show that the agents finally converge to a common point. We describe the protocol formally below.

Let $N \in \mathbb{N}$ denote the maximum number of agents that can ever be present in the system. Each agent has a unique identifier from the set $[N] = \{1, 2, \cdots, N\}$. We denote the state variable which stores the position of agent i, $i \in [N]$, by x_i, and it takes values in $\mathbb{R} \cup \{\perp\}$. For any $i \in [N]$, agent i is said to be failed if $x_i = \perp$; otherwise i is alive. We denote the collective states of all agents by vectors x, y etc.

Let $G = (V, E)$ be an undirected graph with $V = [N]$ and $E \subseteq V \times V$. Each vertex in the graph corresponds to an agent in the system. G is the underlying graph that remains fixed throughout our discussion. At a given point in the execution of the system, the actual communication graph G' is the subgraph of G restricted to the alive nodes.

Let us concentrate on a particular initial state (G, c) where $c = (c_1, \cdots, c_N)$. Let the current configuration of the system be (H, x), where $H = (V_H, E_H)$. There are three kinds of operators which can modify the configuration, namely, *join*, *fail* and *move*, and correspond to a node joining the system, a node failing and two nodes communicating to move to their average value.

We say that a configuration has reached a fixpoint if all the unfailed nodes have the same value, that is, (H, x) is a fixpoint if for every $i, j \in [N]$, $i \neq j$, $x_i \neq \perp$ and $x_j \neq \perp$ implies $x_i = x_j$. When (H, x) is a fixpoint, for all $i, j \in [N]$, $\mathit{fail}_j((H, x)) = (H, x)$, $\mathit{join}_j((H, x)) = (H, x)$ and $\mathit{move}_{i,j}((H, x)) = (H, x)$. When (H, x) is not a fixpoint,

- $\mathit{fail}_j((H, x)) = (H', x')$ where $x'_j = \perp$ and $x'_i = x_i$ for $i \neq j$ and $H' = (V'_H, E'_H)$ where $V'_H = V_H - \{j\}$ and $E'_H = E \cap (V'_H \times V'_H)$.
- $\mathit{join}_j((H, x)) = (H', x')$ where $x'_j = c_j$ and $x'_i = x_i$ for $i \neq j$ and $H' = (V'_H, E'_H)$ where $V'_H = V_H \cup \{j\}$ and $E'_H = E \cap (V'_H \times V'_H)$.
- $\mathit{move}_{i,j}((H, x)) = (H, x')$ where $x' = x$ if either $x_i = \perp$ or $x_j = \perp$, otherwise $x'_i = x'_j = (x_i + x_j)/2$ and $x'_k = x_k$ for $k \notin \{i, j\}$.

We note that $\mathit{move}_{i,j}$ is defined only if $(i, j) \in E$, that is, communication between (i, j) is allowed. We want to show that an infinite sequence of operations converges to a point if it contains a finite number of fail_j and join_j operations and the set of edges (i, j) of E such that $\mathit{move}_{i,j}$ occurs infinitely often forms a connected graph. We will see later that this set of sequences is an ω-regular language. We will develop sufficiency conditions for proving such properties, and apply it to this example. Several generalizations of this type of consensus protocol has been presented in the literature (see for e.g. [3]).

3 Preliminaries

3.1 Directed and Undirected Graphs

A labelled directed graph (LDG) G is a triple (V, E, Σ), where V is a finite set of vertices, $E \subseteq V \times \Sigma \times V$ is a set of edges and Σ is a finite set of labels. Let $G = (V, E, \Sigma)$ be a LDG. Given $V' \subseteq V$, the restriction of G to V' is given by the LDG $G[V'] = (V', E', \Sigma)$ where $E' \subseteq E$ is the set $\{(u, a, v) \in E \mid u, v \in V'\}$. Given $E' \subseteq E$, $G - E' = (V, E - E', \Sigma)$. A *path* in G is a sequence of edges $e_1 \cdots e_n$ such that $e_i = (q_i, a, q_{i+1})$ for all i. We say that q_{n+1} is *reachable* from q_1. We say that G is *strongly connected* if for every $u, v \in V$, v is reachable from u. A set $V' \subseteq V$ is strongly connected in G if $G[V]$ is strongly connected and is *maximally strongly connected* if in addition for all V'' such that $V' \subset V''$, $G[V'']$ is not strongly connected.

An undirected graph G is a pair (V, E) where $E \subseteq V \times V$ is a symmetric relation. Whenever we refer to a set of edges of an undirected graph it is assumed to be symmetric. A path in G and reachability of a vertex is defined as before. We say that a graph G is connected if every vertex is reachable from every other vertex. A cut in a connected graph G is a non-empty set of edges E such that $G - E$ is not connected. A subgraph of $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$.

3.2 Stability and Convergence

Let $\mathcal{X}$ be a set and $T_k : \mathcal{X} \to \mathcal{X}$ for $1 \leq k \leq K$ be a finite collection of functions ("operators"). Let $X_{\mathit{fp}} \subseteq \mathcal{X}$ be a set of common fixpoints, that is, $T_k(x) = x$ for all $1 \leq k \leq K$ and $x \in X_{\mathit{fp}}$. We will denote an infinite sequence of operators by σ. Let $\sigma = a_1 a_2 a_3 \cdots$, then $\sigma(i)$ denotes the i-th element of σ namely a_i, and $\mathit{Pref}(\sigma, i)$ denotes the finite sequence consisting of the first i elements, namely, $a_1 \cdots a_i$. Given an $x \in \mathcal{X}$, we denote the element obtained by applying the first n operators of σ to x in order by $\sigma(x, n)$. Formally $\sigma(x, 0) = x$, $\sigma(x, n) = \sigma(n)(\sigma(x, n - 1))$, for $n \geq 1$.

Next we want to define the notion of convergence. We say that starting from x a sequence σ converges to some point in X_{fp} if it moves closer and closer to X_{fp} along σ. To make this notion precise we need to define a neighborhood system around X_{fp}.

Definition 1. *An $\mathcal{X}$-neighborhood system $\mathcal{U}$ around X_{fp} is a collection of subsets of $\mathcal{X}$ such that:*

Property 1. *$X_{fp} \subseteq U, \forall U \in \mathcal{U}$.*

Property 2. *For any $y \in \mathcal{X}$ such that $y \notin X_{fp}$, there exists some $U \in \mathcal{U}$ such that $y \notin U$.*

Property 3. *$\mathcal{U}$ is closed under finite intersections.*

Property 4. *$\mathcal{U}$ is closed under unions.*

We say that $\mathcal{U}$ has a *countable base* if there exists a sequence $\{U_n\}_{n=1}^{\infty}$ of elements of $\mathcal{U}$ such that for every $U \in \mathcal{U}$ there exists some n such that $U_n \subseteq U$.

We say that a sequence $\{x_n\}_{n=1}^{\infty}$ of elements of $\mathcal{X}$ *converges* to X_{fp} with respect to $\mathcal{U}$ if for every $U \in \mathcal{U}$ there exists a positive integer N such that $x_n \in U, \forall n \geq N$.

We want to converge not with respect to a single sequence but a set of sequences. Let us fix a set of operators $\Sigma = \{T_1, \cdots, T_k\}$. An infinite sequence of operators from Σ will also be called an infinite word. We will denote the set of all infinite words over Σ by Σ^ω. We will call a subset L of Σ^ω a language over Σ.

Definition 2. *Stability and convergence. Given a neighborhood system $\mathcal{U}$ and $L \subseteq \Sigma^\omega$, we say that L is* stable *with respect to $\mathcal{U}$ if $\forall U \in \mathcal{U}$, $\exists V \in \mathcal{U}$ such that $\forall x \in \mathcal{X}$, $\forall \sigma \in L$, if there exists $n_0 \in \mathbb{N}$ such that $\sigma(x, n_0) \in V$ then $\forall n \geq n_0$, $\sigma(x, n) \in U$.*

We say that L converges to X_{fp} with respect to $\mathcal{U}$ if $\{\sigma(x,n)\}_{n=1}^{\infty}$ converges to X_{fp} for all $x \in \mathcal{X}$ and $\sigma \in L$.

Remark 1. *Our definition of convergence is analogous to asymptotic stability used is control theory. However our definition of stability is slightly stronger than the classical notion of Lyapunov stability in that instead of requiring that for every U there exists a V such that any trajectory starting in V remains within U, we require that if a trajectory starting anywhere enters V, then we remain within U. This stronger condition is equivalent to the weaker condition, when the L we consider is a suffix closed language.*

Example 1. *Let us now try to formalize the convergence of the Example in Section 2. Let the agents have identifiers from $[N]$ and $G = (V, E)$ be the underlying undirected graph which changes when nodes fail and join. The state space $\mathcal{X}$ is the set of all pairs (H, x) where H is a subgraph of G restricted to the nodes i which have not failed. $\mathcal{X} = \{(H,x) \mid H = (V_H, E_H), V_H = \{i \mid x_i \neq \bot\}, E_H = E \cap (V_H \times V_H)\}$. We take X_{fp} to be the set of all fixpoints, which are configurations in which all the unfailed nodes have the same value. $X_{fp} = \{(H, x) \in \mathcal{X} \mid \forall i, j, (x_i \neq \bot, x_j \neq \bot) \Rightarrow x_i = x_j\}$.*

Next we need to define a neighborhood system which satisfies the properties 1, 2, 3 and 4. Before defining the neighborhood, we need to set some notation. Let $\beta(n) = (1 - 1/(2n^3)$ when $n > 0$ and $\beta(0) = 0$. Let alive(x) is the size of the set $\{i \mid x_i \neq \bot\}$. We define a function $f : (\mathbb{R} \cup \{\bot\})^N \rightarrow \mathbb{R}_{\geq 0}$ given by $f(x) = \sum_{j: x_j \neq \bot} (x_j - M)^2$, where $M = \frac{1}{alive(x)} \sum_{j: x_j \neq \bot} x_j$ when alive$(x) \neq 0$, *otherwise $f(x) = 0$. Let $\mathbb{I}$ be the set of all integers. We can now define the neighborhood as:*

$$\mathcal{U} = \{U_i\}_{i \in \mathbb{I}}, \text{where } U_i = \{(H, x) \in \mathcal{X} \mid f(x) \leq \beta(alive(x))^i\}.$$

$\mathcal{U}$ is a neighborhood system. To see the Property 1 is satisfied, observe that for all $(H, x) \in X_{fp}$, $f(x) = 0$ and $\beta(alive(x))^i \geq 0$. Hence $X_{fp} \subseteq U_i$ for all i. Given any $(H, x) \notin X_{fp}$, $f(x) > 0$. And $\beta(n)^i \to 0$ as $i \to \infty$ for $n \geq 0$. Therefore $(H, x) \notin U_i$ for some i. In particular we have $X_{fp} \subset \cdots \subset U_3 \subset U_2 \subset U_1 \subset U_0 \subset U_{-1} \subset U_{-2} \subset U_{-3} \subset \cdots \subset \mathcal{X}$, and $\bigcap_{i \in I} U_i = X_{fp}$. Clearly Properties 3 and 4 are satisfied.

We want to show that starting from any $(H, x) \in \mathcal{X}$, we converge to X_{fp} on any sequence of operations of join, fail and move with finite number of join and fail operations and such that the moves form a connected component of the alive nodes. Let $join = \{join_j \mid j \in [N]\}$, $fail = \{fail_j \mid j \in [N]\}$ and $move = \{move_{i,j} \mid (i, j) \in E\}$. Formally $\Sigma = join \cup fail \cup move$. We can define a function $Nodes{-}Alive : \Sigma^* \to 2^{[N]}$ which takes a finite sequence of operators and returns the set of nodes alive after applying the operators in the sequence. We will use . for concatenation of two finite sequences or for concatenation of an infinite sequence to the end of a finite sequence. $Nodes{-}Alive(\epsilon) = [N]$. $Nodes{-}Alive(\sigma.T) = Nodes{-}Alive(\sigma)$ if $T \in move$, $= Nodes{-}Alive(\sigma) - \{i\}$ if $T = fail_i$ and $= Nodes{-}Alive(\sigma) \cup \{i\}$ if $T = join_i$. $L_{converge} = \{\sigma \in \Sigma^\omega \mid \exists \sigma_1 \in \Sigma^*, \sigma_2 \in move^\omega, \sigma = \sigma_1\sigma_2, G_{\sigma_1,\sigma_2}$ is connected$\}$, where $G_{\sigma_1,\sigma_2} = (Nodes{-}Alive(\sigma_1), \{(i, j) \mid move_{i,j} \in \inf(\sigma_2)$ or $move_{j,i} \in \inf(\sigma_2)\})$.

Remark 2. *This system is not stable even in the classical sense because starting in any configuration, executing $join_j$, $j = 1, \cdots, N$, will result in the initial configuration. So given a U which is sufficiently small, for every V, there exists an $x \in V$ and σ such that σ takes x out of U. However we will prove the convergence using our sufficiency results.*

3.3 Muller Automata and ω-Regular Languages

A *Muller* automaton $\mathcal{A}$ over an alphabet Σ is a tuple $(Q, q_{init}, \delta, \{F_1, \cdots, F_k\})$ where:

- Q is a finite set of states.
- q_{init} is the initial state.
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation (or the set of edges).
- $F_i \subseteq Q$ for $1 \leq i \leq k$ are accepting sets.

An automaton $\mathcal{A}$ defines a language over Σ. Given an infinite sequence sequence $\tau = \tau_1\tau_2\cdots$, we define $\inf(\tau) = \{\tau_i \mid \{j \mid \tau_j = \tau_i\}$ is an infinite set $\}$. A run of $\mathcal{A}$ on $\sigma \in \Sigma^\omega$ is an infinite sequence of states $\rho = q_1q_2\cdots$ such that $q_1 = q_{init}$ and $(q_i, \sigma(i), q_{i+1}) \in \delta$ for $i \geq 1$. A run ρ of $\mathcal{A}$ on σ is *accepting* if $\inf(\rho) = F_i$ for some i. An infinite word $\sigma \in \Sigma^\omega$ is *accepted* by $\mathcal{A}$ if there exists a run of $\mathcal{A}$ on σ which is accepting. The language accepted by $\mathcal{A}$, denoted $Lang(\mathcal{A})$ is the set of all infinite words accepted by $\mathcal{A}$. A language $L \subseteq \Sigma^\omega$ is *ω-regular* if there exists a Muller automaton whose language is L. We associate a labelled directed graph with $\mathcal{A}$ denote $Graph(\mathcal{A})$ and is defined as $Graph(\mathcal{A}) = (Q, \delta, \Sigma)$. Henceforth when we refer to a path of an automaton or a state being reachable, we refer to the underlying graph.

We call a Muller automaton $\mathcal{A} = (Q, q_{init}, \delta, \{F_1, \cdots, F_k\})$ *simple* if:

– Every state in Q is reachable from q_{init}, and every edge $e \in \delta$ is useful, that is, there exists $\sigma \in Lang(\mathcal{A})$ and an accepting run ρ of $\mathcal{A}$ on σ such that $e = (\rho(i), \sigma(i), \rho(i+1))$ for some i.
– $Graph(\mathcal{A})[F_i]$ is maximally strongly connected in $Graph(\mathcal{A})$ for all i.
– All edges going out of F_i go into F_i for all i, that is, $(q, a, q') \in \delta$ and $q \in F_i$ implies $q' \in F_i$.

The next proposition states that the class of languages accepted by simple Muller automata is exactly the class of ω-regular languages.

Proposition 1. *For every Muller automaton $\mathcal{A}$, there exists a simple Muller automaton $\mathcal{B}$ such that $Lang(\mathcal{A}) = Lang(\mathcal{B})$. Further $\mathcal{B}$ can be constructed in time polynomial in the size of $\mathcal{A}$.*

Proposition 2. *Given a simple Muller automaton $\mathcal{A}$ and a set of edges $E \subseteq \delta$ such that $(Graph(\mathcal{A}) - E)[F_i]$ is not strongly connected for every i, an accepting run ρ of $\mathcal{A}$ on any $\sigma \in \Sigma^\omega$ has infinitely many indices i such that $(\rho(i), \sigma(i), \rho(i+1)) \in E$.*

Example 1. *The language $L_{converge}$ of the Example in section 2 is ω-regular. It is accepted by the following automaton $\mathcal{A}_{converge} = (Q, q_{init}, \delta, \mathcal{F})$. Q consists of two types of states: the first set $Q_1 = 2^{[N]}$ stores the set of alive nodes, the second set $Q_2 = \{(S, E_S, e) \mid S \subseteq [N], (S, E_S)$ is a connected subgraph of G, $e \in E_S\}$. $q_{init} = [N]$. $\mathcal{F} = \{F_{S,E_S} \mid (S, E_S, e) \in Q_2\}$, where $F_{S,E_S} = \{(S, E_S, e) \in Q\}$. All nodes in F_{S,E_S} ensure that eventually the set of alive nodes will be S and those which will communicate infinitely often will be those in E_S. δ consists of three sets of transition:*

– $\delta_1 = \{(S, T, S \cup \{j\}) \mid S \in Q_1, T \in join\} \cup \{(S, T, S - \{j\}) \mid S \in Q_1, T \in fail\} \cup \{(S, T, S) \mid S \in Q_1, T \in move\}$.
– $\delta_2 = \{((S, E_S, e), T, (S, E_S, e')) \mid e = (i, j), T = move_{i,j}, e' \in E_S - \{e\}\}$.
– $\delta_3 = \{(S, T, (S, E_S, e)) \mid T \in move\}$.

3.4 $\mathcal{A}, \mathcal{X}$-Neighborhood System

For the rest of the paper, let us fix some notation. Let $\mathcal{X}$ be a set and $\Sigma = \{T_1, \cdots, T_k\}$ a set of operators on $\mathcal{X}$. Let X_{fp} be a non-empty set of common fixpoints of $\mathcal{X}$ with respect to the operators in Σ, and $\mathcal{U}$ a neighborhood system around X_{fp}. Let $\mathcal{A}$ be a Muller automaton on Σ. Let $\mathcal{Y} = Q \times \mathcal{X}$.

We will define some concepts related to $\mathcal{Y}$. Given $Y \subseteq \mathcal{Y}$ and $q \in Q$, $Proj_q(Y) = \{x \mid (q, x) \in Y\}$ and $Proj(Y) = \cup_{q \in Q} Proj_q(Y)$. Given an edge

$e = (q, T_i, q') \in \delta$, $func_e((q,x)) = (q', T_i(x))$. Given $Y \subseteq \mathcal{Y}$, $func_e(Y) = \{q'\} \times T_i(Proj_q(Y))$. A set $Y \subseteq \mathcal{Y}$ is said to be $\mathcal{A}$-invariant if for all $e \in \delta$, $func_e(Y) \subseteq Y$. We say that a state $y \in \mathcal{Y}$ is *reachable* if there exists an $x \in \mathcal{X}$, a $\sigma \in Lang(\mathcal{A})$ and a run ρ of $\mathcal{A}$ on σ such that $y = (\rho(i), \sigma(x, i-1))$ for some i. We say that y is reached from x using $w = Pref(\sigma, i-1)$. We will denote the set of all reachable states of Y by $Reachable(Y)$.

Let $Y_{fp} = Reachable(Q \times X_{fp})$. Note that Y_{fp} is an $\mathcal{A}$-invariant set. A $\mathcal{A}, \mathcal{X}$-neighborhood system around X_{fp} is a $Q \times \mathcal{X}$-neighborhood system $\mathcal{W}$ around Y_{fp}. When $\mathcal{X}$ is clear from the context we will drop the $\mathcal{X}$ and call it an $\mathcal{A}$-neighborhood system. $\mathcal{W}$ is said to be *finer* than $\mathcal{U}$, if for every $U \in \mathcal{U}$, there exists a $W \in \mathcal{W}$ such that $Proj(W) \subseteq U$. $\mathcal{U}$ is said to be *finer* than an $\mathcal{W}$, if for every $W \in \mathcal{W}$, there exists a $U \in \mathcal{U}$ such that W contains $Reachable(Q \times U)$. $\mathcal{U}$ and $\mathcal{W}$ are said to be *equivalent*, if $\mathcal{U}$ is finer than $\mathcal{W}$ and $\mathcal{W}$ is finer than $\mathcal{U}$. An $\mathcal{A}, \mathcal{X}$-neighborhood system $\mathcal{W}$ is said to be $\mathcal{A}$-*invariant* if every $W \in \mathcal{W}$ is $\mathcal{A}$-invariant.

Proposition 3. *Let $\mathcal{W}$ be an $\mathcal{A}$-neighborhood system equivalent to $\mathcal{U}$ such that every $W \in \mathcal{W}$ is a subset of $Reachable(Q \times \mathcal{X})$. Then $\mathcal{U}$ has a countable base if and only if $\mathcal{W}$ has a countable base.*

Proof. Let $\{U_n\}_{n=1}^{\infty}$ be a countable base for $\mathcal{U}$. Let $W \in \mathcal{W}$. Then there exists $U \in \mathcal{U}$ such that $Reachable(Q \times U) \subseteq W$ since $\mathcal{U}$ is finer than $\mathcal{W}$. There exists $U_n \subseteq U$ by the definition of countable base. Also there exists W_n such that $W_n \subseteq Q \times U_n$ since $\mathcal{W}$ is finer than $\mathcal{U}$. Since $W_n \subseteq Reachable(Q \times \mathcal{X})$ by assumption, we have $W_n \subseteq Reachable(Q \times U_n)$. Therefore $W_n \subseteq W$. $\{W_n\}_{n=1}^{\infty}$ is a countable base for $\mathcal{W}$.

Similarly let $\{W_n\}_{n=1}^{\infty}$ be a countable base for $\mathcal{W}$. Let $U \in \mathcal{U}$. Then there exists $W \in \mathcal{W}$ such that $Proj(W) \subseteq U$, or there exists W_n such that $Proj(W_n) \subseteq U$. Also there exists $U_n \in \mathcal{U}$ such that $Reachable(Q \times U_n) \subseteq W_n$. Hence $Proj(Reachable(Q \times U_n)) \subseteq U$, or $U_n \subseteq U$. $\{U_n\}_{n=1}^{\infty}$ is a countable base for $\mathcal{U}$. □

4 Stability

From now on we will assume that $\mathcal{A}$ is a simple Muller automaton.

The following result generalizes the result of Tsitsiklis [8].

Theorem 1. *The following are equivalent.*

1. *$Lang(\mathcal{A})$ is stable with respect to an $\mathcal{X}$-neighborhood system $\mathcal{U}$ around X_{fp}.*
2. *There exists an $\mathcal{A}, \mathcal{X}$-invariant neighborhood system $\mathcal{W}$ around Reachable $(Q \times X_{fp})$ which is equivalent to $\mathcal{U}$.*

Proof. The proof is along the lines of that given in [8]. Let $Lang(\mathcal{A}) = L$.

($1 \Rightarrow 2$): We assume that L is stable with respect to $\mathcal{U}$ and we need to construct an $\mathcal{A}$-invariant neighborhood system $\mathcal{W}$ of subsets of $\mathcal{Y}$ which is equivalent to $\mathcal{U}$.

We do this as follows. Given $U \in \mathcal{U}$, we define W_U as the union of all $\mathcal{A}$-invariant subsets of $U' = \mathit{Reachable}(Q \times U)$. Note that $Y_{\mathit{fp}} = \mathit{Reachable}(Q \times X_{\mathit{fp}})$ is an $\mathcal{A}$-invariant subset of U', which shows that W_U is nonempty for all $U \in \mathcal{U}$. Also W_U is the largest $\mathcal{A}$-invariant subset of U'. Let $\mathcal{W}' = \{W_U : U \in \mathcal{U}\}$ and let $\mathcal{W}$ be the closure of $\mathcal{W}'$ under finite intersections and unions.

$\mathcal{W}$ is a neighborhood system. The first property is satisfied since $Q \times X_{\mathit{fp}} \subseteq W_U$ for all U implies $Q \times X_{\mathit{fp}} \subseteq W$ for all $W \in \mathcal{W}$. The sets in $\mathcal{W}$ are closed under finite intersections and arbitrary union by definition. Let $y = (q, x) \in \mathcal{Y} - Y_{\mathit{fp}}$. If y is not reachable then it does not belong to any W_U. Otherwise $x \notin X_{\mathit{fp}}$, and therefore $x \notin U$ for some $U \in \mathcal{U}$. Therefore $y \notin W_U$. Therefore for every $y \in \mathcal{Y} - Y_{\mathit{fp}}$, there exists $W \in \mathcal{W}$ such that $y \notin W$.

$\mathcal{W}$ is $\mathcal{A}$-invariant, since W_U are $\mathcal{A}$-invariant and finite intersections and arbitrary unions of $\mathcal{A}$-invariant sets are $\mathcal{A}$-invariant.

$\mathcal{W}$ is equivalent to $\mathcal{U}$. For every $U \in \mathcal{U}$, there exists $W_U \in \mathcal{W}$ such that $W_U \subseteq Q \times U$, hence $\mathcal{W}$ is finer than $\mathcal{U}$. To show that $\mathcal{U}$ is finer than $\mathcal{W}$, it is enough to show that $\mathcal{U}$ is finer than $\mathcal{W}'$, because the sets in $\mathcal{U}$ are also closed under finite intersections and arbitrary unions. Let $W \in \mathcal{W}'$, then $W_U = W$ for some $U \in \mathcal{U}$. Using the fact that L is *stable* $\exists V \in \mathcal{U}$ such that $\forall x \in \mathcal{X}$, $\forall \sigma \in L$, if there exists $n_0 \in \mathbb{N}$ such that $\sigma(x, n_0) \in V$ then $\forall n \geq n_0$, $\sigma(x, n) \in U$. Define $V' = \{y \mid y \text{ is reached from } x \in \mathcal{X} \text{ using } \mathit{Pref}(\sigma, j) \text{ for some } \sigma \in L \text{ and } \sigma(x, i) \in V \text{ for some } i \leq j\}$. In particular, V' contains $\mathit{Reachable}(Q \times V)$. Note that V' is an $\mathcal{A}$-invariant subset of U' (here we use that fact that every edge is useful). Hence $V' \subseteq W_U$. Therefore there exists $V \in \mathcal{U}$ such that $\mathit{Reachable}(Q \times V) \subseteq W$.

($2 \Rightarrow 1$): Given any $U \in \mathcal{U}$, there exists some $W \in \mathcal{W}$ such that $\mathit{Proj}(W) \subseteq U$, because $\mathcal{W}$ is finer than $\mathcal{U}$. Moreover, since $\mathcal{U}$ is finer than $\mathcal{W}$, there exists some $V \in \mathcal{U}$ such that $V' = \mathit{Reachable}(Q \times V)$ is contained in W. Consider an x, $n_0 \in \mathbb{N}$ and $\sigma \in L$ such that $\sigma(x, n_0) \in V$. Let ρ be an accepting run of $\mathcal{A}$ on σ. Then $(\rho(n_0 + 1), \sigma(x, n_0)) \in V' \subseteq W$. Since W is $\mathcal{A}$-invariant $(\rho(n + 1), \sigma(x, n)) \in V'$ for all $n \geq n_0$. Therefore $\sigma(x, n) \in \mathit{Proj}(W) \subseteq U$ for all $n \geq n_0$. Hence L is stable with respect to $\mathcal{U}$. □

Remark 3. *We note that the $\mathcal{W}$ constructed in the first part of the proof is such that every $W \in \mathcal{W}$ is a subset of $\mathit{Reachable}(Q \times \mathcal{X})$.*

5 Convergence

In this section we present necessary and sufficient conditions for convergence of a ω-regular language L. Let $\mathcal{X}$, X_{fp}, Σ, $\mathcal{U}$, $\mathcal{Y}$ and Y_{fp} be as above. Let L be an ω-regular language over Σ such that $L = \mathit{Lang}(\mathcal{A})$, where $\mathcal{A}$ is a simple Muller automaton.

First, we present a sufficient condition for convergence.

Condition 1. *There exists a totally ordered index set I and a collection $\{X_\alpha : \alpha \in I\}$ of distinct subsets of $\mathcal{Y}$ containing Y_{fp} with the following properties:*

Property 1. *$\alpha < \beta$ implies $X_\alpha \subseteq X_\beta$.*

Property 2. *For every $U \in \mathcal{U}$, there exists some $\alpha \in I$ such that $Proj(X_\alpha) \subseteq U$.*

Property 3. $\bigcup_{\alpha \in I} Proj_{q_{init}}(X_\alpha) = \mathcal{X}$.

Property 4. *X_α is $\mathcal{A}$-invariant for all $\alpha \in I$.*

Property 5. *For every $\alpha \in I$ such that $X_\alpha \neq Y_{fp}$, there exists $E \subseteq \delta$ such that for every i $(Graph(\mathcal{A}) - E)[F_i]$ is not strongly connected, and for every $e \in E$, $func_e(X_\alpha) \subseteq \bigcup_{\beta < \alpha} X_\beta$.*

Property 6. *Every non-empty subset of I which is bounded below has a smallest element.*

Following theorem states that the above condition is sufficient for convergence.

Theorem 2. *If Condition 1 holds, then L converges to X_{fp} with respect to $\mathcal{U}$.*

Proof. Let I, $\{X_\alpha : \alpha \in I\}$ have the properties in Condition 1. Suppose that we are given some $U \in \mathcal{U}$, $x_0 \in \mathcal{X}$ and $\sigma \in L$. We must show that $\sigma(x_0, n)$ eventually enters and remains in U.

Let us fix an accepting run $\rho = q_1 q_2 q_3 \cdots$ of $\mathcal{A}$ on σ. Let
$J = \{\alpha \in I : \exists n \text{ such that } (\rho(n), \sigma(x_0, n-1)) \in X_\alpha\}$.

Lemma 1. *$J = I$.*

Proof. Since from Property 3, we have $\mathcal{X} = \bigcup_{\alpha \in I} Proj_{q_{init}}(X_\alpha)$, there exists some $\alpha \in I$ such that $(q_{init}, x_0) \in X_\alpha$. Hence J is nonempty. We consider two cases: we first assume that J is not bounded below. Then, for every $\alpha \in I$, there exists a $\beta \in J$ such that $\beta < \alpha$. Hence for every $\alpha \in I$, there exists some $\beta < \alpha$ and some integer n such that $(\rho(n), \sigma(x_0, n-1)) \in X_\beta \subseteq X_\alpha$ (Property 1). So, every $\alpha \in I$ belongs to J, and $I = J$.

Let us now assume that J is bounded below. Since it is nonempty, it has a smallest element from Property 6, denoted by β. If $X_\beta = Y_{fp}$, then β is also the smallest element of I, and $I = J$ follows. So, let us assume that $X_\beta \neq Y_{fp}$. Then from Property 5 there exists $E \subseteq \delta$ such that $(Graph(\mathcal{A}) - E)[F_i]$ is not strongly connected, and for every $e \in E$, $func_e(X_\beta) \subseteq \bigcup_{\gamma < \beta} X_\gamma$. From the definition of J, there exists some n_0 such that $(\rho(n_0), \sigma(x_0, n_0 - 1)) \in X_\beta$ and by invariance of X_β (Property 4), $(\rho(n), \sigma(x_0, n-1)) \in X_\beta$ for all $n \geq n_0$. Since $\sigma \in L$ and ρ is an accepting run, we have an $m > n_0$ such that $(\rho(m), \sigma(m), \rho(m+1)) \in E$ by Proposition 2. Since $(\rho(m), \sigma(x_0, m-1)) \in X_\beta$, we have $(\rho(m+1), \sigma(x_0, m)) \in \bigcup_{\gamma < \beta} X_\gamma$, or $(\rho(m+1), \sigma(x_0, m)) \in X_\gamma$ for some $\gamma < \beta$. Hence $\gamma \in J$, which contradicts the assumption that β was the smallest element of J. This completes the proof of the lemma. $\square$

Given $U \in \mathcal{U}$, there exists some $\alpha \in I$ such that $Proj(X_\alpha) \subseteq U$ (Property 2). Since $J = I$, there exists some n_0 such that $(\rho(n_0), \sigma(x_0, n_0 - 1)) \in X_\alpha$. Since $func_e(X_\alpha) \subseteq X_\alpha$ for all e, it follows that $(\rho(n), \sigma(x_0, n-1)) \in X_\alpha$ for all $n \geq n_0$. Or $\sigma(x_0, n-1) \in Proj(X_\alpha)$ for all $n \geq n_0$. Hence $\sigma(x_0, n-1) \in U$ for all $n \geq n_0$, which completes the proof. $\square$

Next we show that Condition 1 is a necessary condition when the system satisfies some additional properties.

Theorem 3. *If L is stable and converges to X_{fp} with respect to $\mathcal{U}$ and if $\mathcal{U}$ has a countable base, then Condition 1 holds.*

Proof. Since L is stable with respect to $\mathcal{U}$, we have from Theorem 1 that there exists an $\mathcal{A}$-invariant neighborhood system $\mathcal{W}$ which is equivalent to $\mathcal{U}$. Since $\mathcal{U}$ has a countable base, from Proposition 3 and Remark 3, we have that $\mathcal{W}$ has a countable base as well $\{W_n\}_{n=1}^{\infty}$. Without loss of generality we may assume that $W_{n+1} \subseteq W_n$ for all n. (Otherwise we could define a new countable base $W'_n = \bigcap_{k=0}^{n} W_k$.) Let W_0 be $\mathit{Reachable}(Q \times \mathcal{X})$.

Our proof consists of two main steps: for each $n \geq 0$ we construct a nested collection of subsets of $\mathcal{Y}$ which lie between W_n and W_{n+1}. Then we merge these collections to get a single nested collection.

Lemma 2. *Let $W', W'' \in \mathcal{W}$ such that $W' \subset W''$. Let Inv be the set of all $\mathcal{A}$-invariant subsets of W'' containing W'. Then there exists a function $f : \mathit{Inv} \to \mathit{Inv}$ and $g : \mathit{Inv} \to 2^{\delta}$ such that:*

- *For any $W \in \mathit{Inv}$, we have $W \subseteq f(W)$ and if $\mathit{Proj}_{q_{init}}(W) \neq \mathit{Proj}_{q_{init}}(W'')$ then $f(W) \subset W$.*
- *$\mathit{func}_e(f(W)) \subseteq W$ for all $e \in g(W)$.*
- *$(\mathit{Graph}(\mathcal{A}) - g(W))[F_i]$ is not strongly connected for all i.*

For the sake of continuity we prove continue with the proof of the theorem and prove the lemma later.

Let I_n be a well-ordered set with cardinality larger than that of $\mathcal{Y}$ and let $\alpha_{0,n}$ be its smallest element. We apply Lemma 2 with $W'' = W_n$ and $W' = W_{n+1}$ to obtain a function f_n satisfying the properties of the lemma above. We define a function $h_n : I_n \to \mathit{Inv}$ using the following transfinite recursion: $h_n(\alpha_{0,n}) = W_{n+1}$, and for all $\alpha > \alpha_{0,n}$, $h_n(\alpha) = f_n(\bigcup_{\beta<\alpha} h_n(\beta))$.

Notice that $W_{n+1} \subseteq h_n(\beta) \subseteq h_n(\alpha) \subseteq W_n$, for any α, β such that $\alpha > \beta$, and that if $\mathit{Proj}_{q_{init}}(h_n(\beta)) \neq \mathit{Proj}_{q_{init}}(W_n)$, then $h_n(\beta) \subset h_n(\alpha)$. Since I_n has cardinality larger than that of $\mathcal{Y}$, there exists some $\alpha \in I_n$ such that $\mathit{Proj}_{q_{init}}(h_n(\alpha)) = \mathit{Proj}_{q_{init}}(W_n)$. Let $\bar{\alpha}_n$ be the smallest such α and let $\bar{I}_n = \{\alpha \in I_n \mid \alpha < \bar{\alpha}_n\}$.

We now define $I = \{\top\} \cup \{(n, \alpha) \mid \alpha \in \bar{I}_n, n = 0, 1, \cdots\}$ with the following total order: $(n, \alpha) < (m, \beta)$ if either $n > m$ or $n = m$ and $\alpha < \beta$, and $(n, \alpha) < \top$ for all n and α. Finally, let $X_{(n,\alpha)} = h_n(\alpha)$ and $X_{\top} = W_0$.

We claim that the collection $\{X_\alpha \mid \alpha \in I\}$ satisfies all the properties of Condition 1. Property 1 is satisfied because $h_n(\beta) \subset h_n(\alpha)$ for every $\beta < \alpha$, where $\beta, \alpha \in \bar{I}_n$. Property 2 follows from the fact that our X_αs include the countable base $\{W_n\}_{n=1}^{\infty}$ we started with and $\mathcal{W}$ is equivalent to $\mathcal{U}$. Since $\mathit{Proj}_{q_{init}}(W_0) = \mathcal{X}$, Property 3 is true. Property 4 holds since all the new sets we introduce (basically in Lemma 2) are $\mathcal{A}$-invariant. Again Property 5 follows from Lemma 2, where the function g gives the set of edges E for every invariant set. Finally, since every non-empty subset of a well-ordered set has a least element, and countable concatenations of well-ordered sets is well ordered, we satisfy 6. □

5.1 Proof of Lemma 2

Given $Y \subseteq \mathcal{Y}$ and $e = (q, T, q')$, define $\mathit{Reach}_e(Y) = \{(q', x') \mid \exists (q, x) \in Y, x' = T(x)\}$. Given a path P in $\mathit{Graph}(\mathcal{A})$, $\mathit{Reach}_P(S)$ is defined inductively as follows. If $P = e \in \delta$, then $\mathit{Reach}_P(Y) = \mathit{Reach}_e(Y)$. Otherwise if $P = P'e$, and $\mathit{Reach}_P(Y) = \mathit{Reach}_e(\mathit{Reach}_{P'}(Y))$. Given an edge $e \in \delta$, $\mathit{PathsEnd}(e) = \{P \mid P = P'e\}$, is the set of all paths ending in e.

Given a set of edges $E \subseteq \delta$ and $W \in \mathcal{W}$, define $f_E(W) = \{(q, x) \mid \forall e \in E, P \in \mathit{PathsEnd}(e), \mathit{Reach}_P(\{(q, x)\}) \subseteq W\}$. $f_E(W)$ has the following properties.

- $W \subseteq f_E(W)$: Since W is invariant, for all P $\mathit{Reach}_P(W) \subseteq W$.
- $f_E(W)$ is $\mathcal{A}$-invariant: Let $x \in f_E(W)$, and $Y = f_E(\{x\})$. If there exists $y \in Y$ such that $y \notin f_E(W)$, then there exists $e \in E$ and $P \in \mathit{PathsEnd}(e)$ such that $\mathit{Reach}_P(\{y\}) \not\subseteq W$. Then $x \notin f_E(W)$, since there exists $e \in E$ and a path $e'P \in \mathit{PathsEnd}(e)$ such that $\mathit{Reach}_P(\{x\}) \not\subseteq W$.
- $\mathit{func}_e(f_E(W)) \subseteq W$ for all $e \in E$: The argument is similar to the previous.

Let $\mathcal{E} = \{E \subseteq (\delta \cap \cup_i (F_i \times \Sigma \times F_i)) \mid (\mathit{Graph}(\mathcal{A}) - E)[F_i]$ is not strongly connected for every $i\}$. Given $W \in \mathit{Inv}$, we claim that if $\mathit{Proj}_{q_{init}}(W) \neq \mathit{Proj}_{q_{init}}(W'')$, then $W \subset f_E(W)$ for some $E \in \mathcal{E}$. Suppose not. Then there exists $(q_{init}, x_0) \in W'' - W$ and $f_E(W) = W$ for all $E \in \mathcal{E}$. Therefore (q_{init}, x_0) does not belong to any $f_E(W)$.

We will construct a $\sigma \in \mathit{Lang}(\mathcal{A})$ and an accepting run ρ of $\mathcal{A}$ on σ such that $(\rho(i), \sigma(x_0, i-1)) \notin f_E(W)$ for all $E \in \mathcal{E}$ and $i \geq 1$. This contradicts the convergence of L as follows. There exists $U \in \mathcal{U}$ such that $Z = \mathit{Reach}(Q \times U) \subseteq W'$, since $\mathcal{U}$ is finer than $\mathcal{W}$. Since $Z \subseteq W' \subseteq W = f_E(W)$, $(\rho(i), \sigma(x_0, i-1)) \notin Z$ for all i. Therefore $\sigma(x_0, i) \notin U$ for all i, contradicting the convergence of L with respect to $\mathcal{U}$. Hence $W \subset f_E(W)$ for some $E \in \mathcal{E}$. We set $f(W) = f_E(W)$ for some E for which $W \subset f_E(W)$.

It remains to construct a $\sigma \in L$ and ρ which satisfy the above condition. Given a path P starting in q and a singleton set $\{(q, x)\}$, $\mathit{Reach}_P(\{(q, x)\})$ is a singleton. Hence we will write this as just $\mathit{Reach}_P((q, x))$.

Let $E \in \mathcal{E}$ be non-empty. Let $s_0 = (q_{init}, x_0)$. Since $s_0 \notin f_E(W)$ there exists some P ending in an edge in E such that $\mathit{Reach}_P(s_0) \notin f_E(W) = W$. Let us call this P as P_1 and $\mathit{Reach}_P((q_{init}, x_0))$ as s_1. Let the last of edge of P belong to $F_{i^*} \times \Sigma \times F_{i^*}$. Since the automaton is simple any path starting from F_{i^*} will remain within F_{i^*}. We will assume $|F_{i^*}| \geq 2$, (a similar procedure can be used when $|F_{i^*}| = 1$.

The following procedure generates a sequence of P_js:

```
1. Let $P_1$ and $s_1$ be as defined above. Initialize $j$ to 1.
2. Let $Q' = \emptyset$.
3. While $Q' \neq Q$ do:
     – Add the last state of $P_j$ to $Q'$.
     – Consider $E = \delta \cap Q' \times \Sigma \times (Q - Q')$.
     – Increment $j$.
     – Set $P_j$ to be a path ending in $E$ such that $\mathit{Reach}_{P_j}(s_{j-1}) \notin f_E(W) = W$.
     – Set $s_j = \mathit{Reach}_{P_j}(s_{j-1})$.
```

Note that we maintain the invariant that $s_{j-1} \notin f_E(W)$ for some E and hence $s_{j-1} \notin W$. Therefore there always exists a path P_j ending in E such that $\mathit{Reach}_{P_j}(s_{j-1}) \notin f_E(W) = W$. Let $P' = P_1 P_2 \cdots$ be the sequence of edges $e_1 e_2 \cdots$ with $e_i = (q_i, a_i, q_{i+1})$. Define $\sigma = a_1 a_2 \cdots$ and $\rho = q_1 q_2 \cdots$. ρ is a run of $\mathcal{A}$ on σ. It is accepting because each P_j contains every state from F_{i^*} at least once, and only contains states from F_{i^*}, because the automaton is simple. We have infinitely many i such that $(\rho(i), \sigma(x_0, i-1)) \notin f_{\mathcal{E}}(W)$ for any E or equivalently $(\rho(i), \sigma(x_0, i-1)) \notin W$ (They correspond to s_js). Since W is invariant, if $(\rho(i), \sigma(x_0, i-1)) \in W$ for some i, then $(\rho(j), \sigma(x_0, j-1)) \in W$ for all $j \geq i$, which contradicts the previous statement. Therefore $(\rho(i), \sigma(x_0, i-1)) \notin W$ for all i.

6 An Application

In this section, we illustrate the application of our results to prove convergence of the Example in Section 2.

We have already defined $\mathcal{X}, X_{fp}, \mathcal{U}$ and $L_{converge}$. We will prove convergence using the simple Muller Automaton $\mathcal{A}_{converge}$. We will then point out how one can prove convergence given any simple Muller automaton for $L_{converge}$.

6.1 Properties of the Neighborhood System $\mathcal{U}$

Let us define $\mathit{move}_{i,j}(x) = x'$ as follows: If $x_i \neq \perp, x_j \neq \perp$, then $x'_i = x'_j = (x_i + x_j)/2$ and $x'_k = x_k$ for $k \notin \{i, j\}$, otherwise $x' = x$.

We recall the following two results from [5].

Proposition 4. *$f(\mathit{move}_{i,j}(x)) \leq f(x)$.*

Let $\mathit{Sorted}(x)$ from $[N] \to [N]$ be a one-one and onto function which satisfies for $i < j$, $x_{\mathit{Sorted}(x)(i)} < x_{\mathit{Sorted}(x)(j)}$, or $x_{\mathit{Sorted}(x)(i)} \leq x_{\mathit{Sorted}(x)(j)}$ and $\mathit{Sorted}(x)(i) < \mathit{Sorted}(x)(j)$. $\mathit{Sorted}(x)(i)$ will give the identifier of the agent with the i-th smallest value and when agents have same values, the value of the agent with the smaller identifier is considered smaller. Here $\perp$ is considered to have a value of ∞.

Proposition 5. *Given any x, there exists $k \in [N]$ such that $x_{\mathit{Sorted}(x)(k+1)} - x_{\mathit{Sorted}(x)(k)} > \frac{1}{\mathit{alive}(x)} \sqrt{\frac{f(x)}{\mathit{alive}(x)}}$. Given any i, j such that $1 \leq i \leq k$ and $k + 1 \leq j \leq \mathit{alive}(x)$, $f(\mathit{move}_{i',j'}(x)) \leq \beta(\mathit{alive}(x)) f(x)$, where $i' = \mathit{Sorted}(x)(i)$ and $j' = \mathit{Sorted}(x)(j)$.*

The above property says that if we start at some (H, x) in U_i, then there is partition of the nodes in H, such that for all edges (i, j) which go between the partitions, $\mathit{move}_{i,j}(H, x)$ will be in U_{i+1}. But we need more, we need one such partition which will work for all elements of U_i.

Define $\mathit{Cut}(x, k) = (A, B)$ where $A = \{\mathit{Sorted}(x)(i) \mid i \leq k\}$ and $B = \{\mathit{Sorted}(x)(j) \mid \mathit{alive}(x) \geq j \geq k + 1\}$. Define $\mathit{Gap}(x) = \{\mathit{Cut}(x, k) \mid 1 \leq k < \mathit{alive}(x), x_{\mathit{Sorted}(x)(k+1)} - x_{\mathit{Sorted}(x)(k)} > \frac{1}{\mathit{alive}(x)} \sqrt{\frac{f(x)}{\mathit{alive}(x)}}\}$.

Proposition 6. *For all $i, j \in [N]$ and x such that for all $(A, B) \in Gap(x)$, either $i, j \leq A$ or $i, j \geq B$, we have $Gap(x) \subseteq Gap(move_{i,j}(x))$.*

Let $\{Gap(x) \,|\, (H, x) \in U_i - U_{i+1}\} = \{C_1, \cdots, C_{i_n}\}$ such that if $C_i \subset C_j$ then $i < j$. Between U_i and U_{i+1} we define a finite number of sets as follows. $U_{i,0} = U_i$, $U_{i,j+1} = U_{i,j} - \{(H, x) \in \mathcal{X} \,|\, Gap(x) = C_{j+1}\}$ for $0 \leq j \leq i_n - 1$. We define the index set to be $J = \{(i, j) \,|\, 0 \leq j \leq i_n - 2\}$ with the ordering $(i, j) < (i', j')$ if $i > i'$ or $i = i'$ and $j > j'$. The required sets are $\{U_\alpha \,|\, \alpha \in J\}$. This set has the following property.

Proposition 7. *For all $\alpha \in J$, there exists $C \subseteq [N] \times [N]$ such that for all $(i, j) \in C$ and $(H, x) \in U_\alpha$, C is a cut in H and $move_{i,j}(H, x) \in U_\beta$ for some $\beta < \alpha$. Also, for all $\alpha \in J$, $i, j \in [N]$, $(H, x) \in U_\alpha$, we have $move_{i,j}((H, x)) \in U_\alpha$.*

Proof. Given α, $U_\alpha = U_{m,n}$ for some m, n. Let $Gap(x) = Z$ which is the same for any $(H, x) \in U_\alpha - U_{\alpha+1}$ where $\alpha + 1 = m, n + 1$ if $(m, n + 1) \in J$, otherwise $\alpha + 1 = m + 1, n$. The required cut $C = \{(i, j) \,|\, \exists (A, B) \in Z, i \in A, j \in B\}$. Then from Proposition 5, we have for all $(i, j) \in C$, $move_{i,j}((H, x)) \in U_{m+1,n}$ for all $(H, x) \in U_\alpha - U_{\alpha+1}$.

Given $(H, x) \in U_\alpha$, if $(i, j) \in C$, $move_{i,j}(H, x) \in U_\alpha$ from above. If $(i, j) \notin C$, then from Proposition 4 we have $move_{i,j}(H, x) \in U_{m',n'}$ where $m' \geq m$ and from 6, we have $n' \geq n$. Therefore $move_{i,j}((H, x)) \in U'_\alpha$ for some $\alpha' \leq \alpha$, hence also in U_α. □

6.2 Convergence Proof

We are now ready to define the invariant sets required to prove convergence.

Recall $\mathcal{A}_{converge} = (Q, q_{init}, \delta, \mathcal{F})$ with $Q = Q_1 \cup Q_2$ and $\delta = \delta_1 \cup \delta_2 \cup \delta_3$ and $\mathcal{F} = \{func_{S,E_S} \,|\, (S, E_S, e) \in Q_2\}$. Let $\mathcal{Y} = Q \times \mathcal{X}$, $Y_{fp} = Reachable(Q \times X_{fp})$. The index set $I = J \cup \{\top\}$, with $\top > j$ for all $j \in J$. For $\alpha \in J$, $Y_\alpha = (Q_2 \times X_\alpha) \cup Y_{fp}$, and $Y_\top = \mathcal{Y}$. The index set I with the sets $\{Y_\alpha \,|\, \alpha \in I\}$ satisfy all the properties of Condition 1. It is easy to see that Properties 1, 2, 3 and 6 are satisfied. $Y_\top$ is clearly invariant. For $\alpha \in J$ and any edge e not in δ_2, $func_e(Y_\alpha) = \emptyset \subseteq Y_\alpha$. For $\alpha \in J$ and $e = (q, a, q') \in \delta_2$, $a = move_{i,j}$ for some i, j and $func_e(Y_\alpha) \subseteq (\{q'\} \times move_{i,j}(X_\alpha)) \cup (Q \times X_{fp})$. Since $move_{i,j}(X_\alpha) \subseteq X_\alpha$ from Proposition 7, we have that Y_α is invariant. Now we show that Property 5 also holds. For $Y_\top$, we can choose E to be δ_2. $(Graph(\mathcal{A}) - E)[F_i]$ is not strongly connected for every i. We need to show that for all $(q, (H, x)) \in Y_\top$, for all $e \in E$, $func_e((q, (H, x))) \in Y_\alpha$ for some $\alpha \in J$. We need to consider only $(q, (H, x)) \in Y_\top - \bigcup_{\alpha \in J} Y_\alpha$. But then $q \in Q_1$ and hence $func_e((q, (H, x))) = \emptyset$. For any other Y_α, we can choose $E = \{(q, move_{i,j}, q') \,|\, (i, j) \text{ or } (j, i) \in C\}$, where C is the cut associated with X_α given by Proposition 7. It is easy to see that $(Graph(\mathcal{A}) - E)[F_i]$ is not strongly connected since the labels of the edges in each F_i correspond to a connected subgraph on the unfailed nodes, and C is cut in the induced subgraph of G with unfailed nodes.

Since the system is not stable as mentioned before we cannot use Theorem 2 to guarantee existence of level sets to prove convergence of the system for any arbitrary automaton accepting $L_{converge}$. However we can always find such sets because of the following structure of any simple Muller automaton $\mathcal{A} = (Q, q_{init}, \delta, \Sigma, \{F_1, \cdots, F_k\})$ such that $Lang(\mathcal{A}) = L_{converge}$.

- There is no edge labelled by *join* or *fail* in any of the F_i, that is, there is no $a \in join \cup fail$ and $q, q' \in F_i$ for some i, such that $(q, a, q') \in \delta$. Because then we would have an accepting run with infinite *join* or *fail* operations.
- Let C be a cut of $G = (V, E)$, i.e, $G - C$ is not connected. Then removing the edges in $\mathcal{A}$ labelled by elements in C renders every F_i not strongly connected, i.e., for every i $(Graph(\mathcal{A}) - E')[F_i]$ is not strongly connected, where $E' = \{(q, a, q') \mid a = move_{i,j}, (i, j) \in C, q, q' \in \bigcup_j F_j\}$.

References

1. Blondel, V.D., Hendrickx, J.M., Olshevsky, A., Tsitsiklis, J.N.: Convergence in multiagent coordination consensus and flocking. In: Proceedings of the Joint forty-fourth IEEE Conference on Decision and Control and European Control Conference, pp. 2996–3000 (2005)
2. Borkar, V., Varaiya, P.: Asymptotic Agreement in Distributed Estimation. IEEE Trans. on Automatic Control 27(3), 650–655 (1982)
3. Chandy, K.M., Mitra, S., Pilotto, C.: Convergence verification: From shared memory to partially synchronous systems. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 218–232. Springer, Heidelberg (2008)
4. Jadbabaie, A., Lin, J., Morse, A.S.: Coordination of groups of mobile autonomous agents using nearest neighbor rules. IEEE Transactions on Automatic Control 48(6), 988–1001 (2003)
5. Mitra, S., Chandy, K.M.: A formalized theory for verifying stability and convergence of automata in PVS. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 230–245. Springer, Heidelberg (2008)
6. Olfati-Saber, R.: Flocking for multi-agent dynamic systems: algorithms and theory. IEEE Transactions on Automatic Control 51(3), 401–420 (2006)
7. Olfati-saber, R., Alex Fax, J., Murray, R.M.: Consensus and cooperation in networked multi-agent systems. Proceedings of the IEEE, 2007 (2007)
8. Tsitsiklis, J.N.: On the stability of asynchronous iterative processes. Mathematical Systems Theory 20(2-3), 137–153 (1987)

Computing Stuttering Simulations

Francesco Ranzato and Francesco Tapparo

Dipartimento di Matematica Pura ed Applicata
Università di Padova, Italy

Abstract. Stuttering bisimulation is a well-known behavioural equivalence that preserves CTL-X, namely CTL without the next-time operator X. Correspondingly, the stuttering simulation preorder induces a coarser behavioural equivalence that preserves the existential fragment ECTL-$\{\mathrm{X}, \mathrm{G}\}$, namely ECTL without the next-time X and globally G operators. While stuttering bisimulation equivalence can be computed by the well-known Groote and Vaandrager's algorithm, to the best of our knowledge, no algorithm for computing the stuttering simulation preorder and equivalence is available. This paper presents such an algorithm for finite state systems.

1 Introduction

The Problem. Lamport's criticism [8] of the next-time operator X in CTL/CTL* arouse the interest in studying temporal logics like CTL-X/CTL*-X, obtained from CTL/CTL* by removing the next-time operator, and related notions of behavioural *stuttering*-based equivalences [1,4,6]. We are interested here in *divergence blind stuttering* simulation and bisimulation, that we call, respectively, stuttering simulation and bisimulation for short. We focus here on systems specified as Kripke structures (KSs), but analogous considerations hold for labeled transition systems (LTSs). Let $\mathcal{K} = \langle \Sigma, \rightarrow, \ell \rangle$ be a KS where $\langle \Sigma, \rightarrow \rangle$ is a transition system and ℓ is a state labeling function. A relation $R \subseteq \Sigma \times \Sigma$ is a stuttering simulation on $\mathcal{K}$ when for any $s, t \in \Sigma$ such that $(s, t) \in R$: (1) s and t have the same labeling by ℓ and (2) if $s \rightarrow s'$ then $t \rightarrow^* t'$ for some t' in such a way that the following diagram holds:

$$\begin{tikzcd}
 & & & s \arrow[r] \arrow[dlll, dotted, no head] \arrow[dll, dotted, no head] \arrow[d, dotted, no head] \arrow[dl, phantom, "\cdots"] & s' \arrow[d, dotted, no head] \\
t \arrow[r] & t_1 \arrow[r] & \cdots \arrow[r] & t_k \arrow[r] & t'
\end{tikzcd}$$

where a dotted line between two states means that they are related by R. The intuition is that t is allowed to simulate a transition $s \rightarrow s'$ possibly through some initial "stuttering" transitions (τ-transitions in case of LTSs). R is called a stuttering bisimulation when it is symmetric. It turns out that the largest stuttering simulation R_{stsim} and bisimulation R_{stbis} relations exist: R_{stsim} is a preorder called the *stuttering simulation preorder* while R_{stbis} is an equivalence relation called the stuttering bisimulation equivalence. Moreover, the preorder R_{stsim} induces by symmetric reduction the *stuttering simulation equivalence* $R_{\mathrm{stsimeq}} = R_{\mathrm{stsim}} \cap R_{\mathrm{stsim}}^{-1}$. The partition of Σ corresponding to the equivalence R_{stsimeq} is denoted by P_{stsim}.

De Nicola and Vaandrager [4] showed that for finite KSs and for an interpretation of universal/existential path quantifiers over all the (possibly non-maximal and finite) paths, the stuttering bisimulation equivalence coincides with the state equivalence induced by the language CTL-X (this also holds for CTL*-X). This is not true with the standard interpretation of path quantifiers over maximal (possibly infinite) paths, since this requires a divergence sensitive notion of stuttering (see the details in [4]). Groote and Vaandrager [6] designed a well-known algorithm that computes the stuttering bisimulation equivalence R_{stbis} in $O(|\Sigma||{\to}|)$-time and $O(|{\to}|)$-space.

Clearly, stuttering simulation equivalence is coarser than stuttering bisimulation equivalence, i.e. $R_{\mathrm{stbis}} \subseteq R_{\mathrm{stsimeq}}$. As far as language preservation is concerned, it turns out that stuttering simulation equivalence coincides with the state equivalence induced by the language ECTL-$\{\mathrm{X}, \mathrm{G}\}$, namely the existential fragment of CTL without next-time X and globally G operators. Thus, on the one hand, stuttering simulation equivalence still preserves a significantly expressive fragment of CTL and, on the other hand, it may provide a significantly better state space reduction than simulation equivalence (and, in turn, bisimulation equivalence), and this has been shown to be useful in abstract model checking [9,10].

State of the Art. To the best of our knowledge, there exists no algorithm for computing stuttering simulation equivalence or, more in general, the stuttering simulation preorder. There is instead an algorithm by Bulychev et al. [2] for *checking* stuttering simulation, namely, this procedure checks whether a given relation $R \subseteq \Sigma \times \Sigma$ is a stuttering simulation or not. This algorithm formalizes the problem of checking stuttering simulation as a two players game in a straightforward way and then exploits Etessami et al.'s [5] algorithm for solving such a game. The authors claim that this provides an algorithm for checking stuttering simulation on finite KSs that runs in $O(|{\to}|^2)$ time and space.

Main Contributions. In this paper we present an algorithm for computing simultaneously both the simulation preorder R_{stsim} and stuttering simulation equivalence R_{stsimeq} for finite KSs. This procedure is incrementally designed in two steps. We first put forward a basic procedure for computing the stuttering simulation preorder that relies directly on the notion of stuttering simulation. For any state $x \in \Sigma$, $\mathrm{StSim}(x) \subseteq \Sigma$ represents the set of states that are candidate to stuttering simulate x so that a family of sets $\{\mathrm{StSim}(x)\}_{x\in\Sigma}$ is maintained. A pair of states $(x, y) \in \Sigma \times \Sigma$ is called a refiner for StSim when $x{\to}y$ and there exists $z \in \mathrm{StSim}(x)$ that cannot stuttering simulate x w.r.t. y, i.e., $z \notin \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ where $\mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ is the set of all the states in $\mathrm{StSim}(x)$ that may reach a state in $\mathrm{StSim}(y)$ through a path of states in $\mathrm{StSim}(x)$. Hence, any such z can be correctly removed from $\mathrm{StSim}(x)$. It turns out that one such refiner (x, y) allows to refine StSim to StSim$'$ as follows: if $S = \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ then

$$\mathrm{StSim}'(w) := \begin{cases} \mathrm{StSim}(w) \cap S & \text{if } w \in S \\ \mathrm{StSim}(w) & \text{if } w \notin S \end{cases}$$

Thus, our basic algorithm consists in initializing $\{\mathrm{StSim}(x)\}_{x\in\Sigma}$ as $\{y \in \Sigma \mid \ell(y) = \ell(x)\}_{x\in\Sigma}$ and then iteratively refining StSim as long as a refiner exists. This provides an *explicit* stuttering simulation algorithm, meaning that this procedure requires that for any explicit state $x \in \Sigma$, $\mathrm{StSim}(x)$ is explicitly represented as a set of states.

Inspired by techniques used in algorithms that compute standard simulation preorders and equivalences (cf. Henzinger et al. [7] and Ranzato and Tapparo [11]) and in abstract interpretation-based algorithms for computing strongly preserving abstract models [12], our stuttering simulation algorithm, called SSA, is obtained by the above basic procedure by exploiting these two main ideas.

(1) The above explicit algorithm is made "symbolic" by representing the family of sets of states $\{\mathrm{StSim}(x)\}_{x\in\Sigma}$ as a family of sets of blocks of a partition P of the state space Σ. More precisely, we maintain a partition P of Σ together with a binary relation $\trianglelefteq \subseteq P \times P$ — a so-called *partition-relation pair* — so that: (i) two states x and y in the same block of P are candidate to be stuttering simulation equivalent and (ii) if B and C are two blocks of P and $B \trianglelefteq C$ then any state in C is candidate to stuttering simulate each state in B. Therefore, here, for any $x \in \Sigma$, if $B_x \in P$ is the block of P that contains x then $\mathrm{StSim}(x) = \mathrm{StSim}(B_x) = \cup\{C \in P \mid B_x \trianglelefteq C\}$.

(2) In this setting, a refiner of the current partition-relation $\langle P, \trianglelefteq\rangle$ is a pair of blocks $(B, C) \in P \times P$ such that $B{\to}^{\exists}C$ and $\mathrm{StSim}(B) \not\subseteq \mathbf{pos}(\mathrm{StSim}(B), \mathrm{StSim}(C))$, where $\to^{\exists}$ is the existential transition relation between blocks of P, i.e., $B{\to}^{\exists}C$ iff there exist $x \in B$ and $y \in C$ such that $x{\to}y$. We devise an efficient way for finding a refiner of the current partition-relation pair that allows us to check whether a given preorder R is a stuttering simulation in $O(|P||{\to}|)$ time and $O(|\Sigma||P|\log|\Sigma|)$ space, where P is the partition corresponding to the equivalence $R \cap R^{-1}$. Hence, this algorithm for checking stuttering simulation already significantly improves both in time and space Bulychev et al.'s [2] procedure.

Our algorithm SSA iteratively refines the current partition-relation pair $\langle P, \trianglelefteq\rangle$ by first splitting the partition P and then by pruning the relation $\trianglelefteq$ until a fixpoint is reached. Hence, SSA outputs a partition-relation pair $\langle P, \trianglelefteq\rangle$ where $P = P_{\mathrm{stsim}}$ and y stuttering simulates x iff $P(x) \trianglelefteq P(y)$, where $P(x)$ and $P(y)$ are the blocks of P that contain, respectively, x and y. As far as complexity is concerned, it turns out that SSA runs in $O(|P_{\mathrm{stsim}}|^2(|{\to}| + |P_{\mathrm{stsim}}||{\to}^{\exists}|))$ time and $O(|\Sigma||P_{\mathrm{stsim}}|\log|\Sigma|)$ space. It is worth remarking that stuttering simulation yields a rather coarse equivalence so that $|P_{\mathrm{stsim}}|$ should be in general much less than the size $|\Sigma|$ of the concrete state space.

2 Background

Notation. If $R \subseteq \Sigma \times \Sigma$ is any relation and $x \in \Sigma$ then $R(x) \triangleq \{x' \in \Sigma \mid (x, x') \in R\}$. Let us recall that R is called a preorder when it is reflexive and transitive. If f is a function defined on $\wp(\Sigma)$ and $x \in \Sigma$ then we often write $f(x)$ to mean $f(\{x\})$. A partition P of a set Σ is a set of nonempty subsets of Σ, called blocks, that are pairwise disjoint and whose union gives Σ. $\mathrm{Part}(\Sigma)$ denotes the set of partitions of Σ. If $P \in \mathrm{Part}(\Sigma)$ and $s \in \Sigma$ then $P(s)$ denotes the block of P that contains s. $\mathrm{Part}(\Sigma)$ is endowed with the following standard partial order $\preceq$: $P_1 \preceq P_2$, i.e. P_2 is coarser than P_1, iff $\forall B \in P_1. \exists B' \in P_2.\ B \subseteq B'$. For a given nonempty subset $S \subseteq \Sigma$ called splitter, we denote by $\mathit{Split}(P, S)$ the partition obtained from P by replacing each block $B \in P$ with the nonempty sets $B \cap S$ and $B \smallsetminus S$, where we also allow no splitting, namely $\mathit{Split}(P, S) = P$ (this happens exactly when S is a union of some blocks of P). If $B \in P' = \mathit{Split}(P, S)$ then we denote by $\mathrm{parent}_P(B)$ (or simply by $\mathrm{parent}(B)$) the unique block in P that contains B (this may possibly be B itself).

A transition system $(\Sigma, \to)$ consists of a set Σ of states and a transition relation $\to \subseteq \Sigma \times \Sigma$. The predecessor transformer $\mathrm{pre} : \wp(\Sigma) \to \wp(\Sigma)$ is defined as usual: $\mathrm{pre}(Y) \triangleq \{s \in \Sigma \mid \exists t \in Y.\ s \to t\}$. If $S_1, S_2 \subseteq \Sigma$ then $S_1 \to^{\exists} S_2$ iff there exist $s_1 \in S_1$ and $s_2 \in S_2$ such that $s_1 \to s_2$. Given a set AP of atomic propositions (of some specification language), a Kripke structure (KS) $\mathcal{K} = (\Sigma, \to, \ell)$ over AP consists of a transition system $(\Sigma, \to)$ together with a state labeling function $\ell : \Sigma \to \wp(AP)$. $P_\ell \in \mathrm{Part}(\Sigma)$ denotes the state partition induced by ℓ, namely, $P_\ell \triangleq \{\{s' \in \Sigma \mid \ell(s) = \ell(s')\}\}_{s \in \Sigma}$.

Stuttering Simulation. Let $\mathcal{K} = (\Sigma, \to, \ell)$ be a KS. A relation $R \subseteq \Sigma \times \Sigma$ is a *divergence blind stuttering simulation* on $\mathcal{K}$ if for any $s, t \in \Sigma$ such that $(s, t) \in R$:

(1) $\ell(s) = \ell(t)$;
(2) If $s \to s'$ then there exist $t_0, ..., t_k \in \Sigma$, with $k \geq 0$, such that: (i) $t_0 = t$; (ii) for all $i \in [0, k)$, $t_i \to t_{i+1}$ and $(s, t_i) \in R$; (iii) $(s', t_k) \in R$.

Observe that condition (2) allows the case $k = 0$ and this boils down to requiring that $(s', t) \in R$. With a slight abuse of terminology, R is simply called a *stuttering simulation*. If $(s, t) \in R$ for some stuttering simulation R then we say that t stuttering simulates s and we denote this by $s \leq t$. If R is a symmetric relation then it is called a stuttering bisimulation. The empty relation is a stuttering simulation and stuttering simulations are closed under union so that the largest stuttering simulation relation exists. It turns out that the largest simulation is a preorder relation called *stuttering simulation preorder* (on $\mathcal{K}$) and denoted by R_{stsim}. Thus, for any $s, t \in \Sigma$, $s \leq t$ iff $(s, t) \in R_{\mathrm{stsim}}$. *Stuttering simulation equivalence* R_{stsimeq} is the symmetric reduction of R_{stsim}, namely $R_{\mathrm{stsimeq}} \triangleq R_{\mathrm{stsim}} \cap R_{\mathrm{stsim}}^{-1}$, so that $(s, t) \in R_{\mathrm{stsimeq}}$ iff $s \leq t$ and $t \leq s$. $P_{\mathrm{stsim}} \in \mathrm{Part}(\Sigma)$ denotes the partition corresponding to the equivalence R_{stsimeq} and is called stuttering simulation partition.

Following Groote and Vaandrager [6], $\mathbf{pos} : \wp(\Sigma) \times \wp(\Sigma) \to \wp(\Sigma)$ is defined as:

$$\mathbf{pos}(S, T) \triangleq$$
$$\{s \in S \mid \exists k \geq 0. \exists s_0, ..., s_k.\ s_0 = s \ \&\ \forall i \in [0, k).\ s_i \in S,\ s_i \to s_{i+1} \ \&\ s_k \in T\}$$

so that a relation $R \subseteq \Sigma \times \Sigma$ is a stuttering simulation iff for any $x, y \in \Sigma$, $R(x) \subseteq P_\ell(x)$ and if $x \to y$ then $R(x) \subseteq \mathbf{pos}(R(x), R(y))$.

It turns out [4] that P_{stsim} is the coarsest partition preserved by the temporal language ECTL-{X, G}. More precisely, ECTL-{X, G} is inductively defined as follows:

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \mathrm{EU}(\phi_1, \phi_2)$$

where $p \in AP$ and its semantics is standard, namely $[\![p]\!] \triangleq \{s \in \Sigma \mid p \in \ell(s)\}$ and $[\![\mathrm{EU}(\varphi_1, \varphi_2)]\!] \triangleq [\![\varphi_2]\!] \cup \mathbf{pos}([\![\varphi_1]\!], [\![\varphi_2]\!])$. The coarsest partition preserved by the

```
$BasicSSA$(Partition $P_\ell$) {
  forall $x \in \Sigma$ do $\mathrm{StSim}(x) := P_\ell(x)$;
  while ($\exists x, y \in \Sigma$ such that $x \rightarrow y$ & $\mathrm{StSim}(x) \not\subseteq \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$) do
    $S := \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$;
    forall $w \in S$ do $\mathrm{StSim}(w) := \mathrm{StSim}(w) \cap S$;
}
```

Fig. 1. Basic Stuttering Simulation Algorithm *BasicSSA*

language ECTL-$\{\mathrm{X}, \mathrm{G}\}$ is the state partition corresponding to the following equivalence $\sim$ between states: for any $s, t \in \Sigma$, $s \sim t$ iff $\forall \phi \in$ ECTL-$\{\mathrm{X}, \mathrm{G}\}$. $s \in [\![\phi]\!] \Leftrightarrow t \in [\![\phi]\!]$.

3 Basic Algorithm

For each state $x \in \Sigma$, the algorithm *BasicSSA* in Figure 1 computes the stuttering simulator set $\mathrm{StSim}(x) \subseteq \Sigma$, i.e., the set of states that stuttering simulate x. The basic idea is that $\mathrm{StSim}(x)$ contains states that are candidate for stuttering simulating x. Thus, the input partition of *BasicSSA* is taken as the partition P_ℓ determined by the labeling ℓ so that $\mathrm{StSim}(x)$ is initialized with $P_\ell(x)$, i.e., with all the states that have the same labeling of x. Following the definition of stuttering simulation, a refiner is a pair of states (x, y) such that $x \rightarrow y$ and $\mathrm{StSim}(x) \not\subseteq \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$. In fact, if $z \in \mathrm{StSim}(x) \smallsetminus \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ then z cannot stuttering simulate x and therefore can be correctly removed from $\mathrm{StSim}(x)$. On the other hand, if no such refiner exists then for any $x, y \in \Sigma$ such that $x \rightarrow y$ we have that $\mathrm{StSim}(x) \subseteq \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ so that any $z \in \mathrm{StSim}(x)$ actually stuttering simulates x. Hence, *BasicSSA* consists in iteratively refining $\{\mathrm{StSim}(x)\}_{x \in \Sigma}$ as long as a refiner exists, where, given a refiner (x, y), the refinement of StSim by means of $S = \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ is as follows:

$$\mathrm{StSim}(w) := \begin{cases} \mathrm{StSim}(w) \cap S & \text{if } w \in S \\ \mathrm{StSim}(w) & \text{if } w \notin S \end{cases}$$

Theorem 3.1 (Termination and Correctness). *For finite KSs, BasicSSA terminates and is correct, i.e., if* StSim *is the output of BasicSSA on input* P_ℓ *then for any* $x, y \in \Sigma$, $y \in \mathrm{StSim}(x) \Leftrightarrow x \leq y$.

4 Partition-Relation Pairs

A *partition-relation pair* $\langle P, \trianglelefteq \rangle$, PR for short, is given by a partition $P \in \mathrm{Part}(\Sigma)$ together with a binary relation $\trianglelefteq \subseteq P \times P$ between blocks of P. We write $B \lhd C$ when $B \trianglelefteq C$ and $B \neq C$ and $(B', C') \trianglelefteq (B, C)$ when $B' \trianglelefteq B$ and $C' \trianglelefteq C$. Our stuttering simulation algorithm relies on the idea of symbolizing the *BasicSSA* procedure in order to maintain a PR $\langle P, \trianglelefteq \rangle$ in place of the family of explicit sets of states

$\{\mathrm{StSim}(s)\}_{s\in\Sigma}$. As a first step, $\mathcal{S} = \{\mathrm{StSim}(s)\}_{s\in\Sigma}$ induces a partition P that corresponds to the following equivalence $\sim_{\mathcal{S}}$: $s_1 \sim_{\mathcal{S}} s_2 \;\Leftrightarrow\; \mathrm{StSim}(s_1) = \mathrm{StSim}(s_2)$. Hence, the intuition is that if $P(s_1) = P(s_2)$ then s_1 and s_2 are "currently" candidates to be stuttering simulation equivalent. Accordingly, a relation $\trianglelefteq$ on P encodes stuttering simulation as follows: if $s \in \Sigma$ then $\mathrm{StSim}(s) = \{t \in \Sigma \mid P(s) \trianglelefteq P(t)\}$. Here, the intuition is that if $B \trianglelefteq C$ then any state $t \in C$ is "currently" candidate to stuttering simulate any state $s \in B$. Equivalently, the following invariant property is maintained: if $s \leq t$ then $P(s) \trianglelefteq P(t)$. Thus, a PR $\langle P, \trianglelefteq\rangle$ will represent the current approximation of the stuttering simulation preorder and in particular P will represent the current approximation of stuttering simulation equivalence.

More in detail, a PR $\mathcal{P} = \langle P, \trianglelefteq\rangle$ induces the following map $\mu_{\mathcal{P}} : \wp(\Sigma) \to \wp(\Sigma)$: for any $X \in \wp(\Sigma)$,

$$\mu_{\mathcal{P}}(X) \triangleq \cup\{C \in P \mid \exists B \in P.\, B \cap X \neq \varnothing,\ B \trianglelefteq C\}.$$

Note that, for any $s \in \Sigma$, $\mu_{\mathcal{P}}(s) = \mu_{\mathcal{P}}(P(s)) = \{t \in \Sigma \mid P(s) \trianglelefteq P(t)\}$, that is, $\mu_{\mathcal{P}}(s)$ represents the set of states that are currently candidates to stuttering simulate s. A PR $\mathcal{P}$ is therefore defined to be a stuttering simulation for a KS $\mathcal{K}$ when the relation $\{(s,t) \in \Sigma \times \Sigma \mid s \in \Sigma,\ t \in \mu_{\mathcal{P}}(s)\}$ is a stuttering simulation on $\mathcal{K}$.

Recall that in *BasicSSA* a pair of states $(s,t) \in \Sigma \times \Sigma$ is a refiner for StSim when $s \to t$ and $\mathrm{StSim}(s) \not\subseteq \mathbf{pos}(\mathrm{StSim}(s), \mathrm{StSim}(t))$. Accordingly, a pair of blocks $(B,C) \in P \times P$ is a refiner for $\mathcal{P}$ when $B \to^{\exists} C$ and $\mu_{\mathcal{P}}(B) \not\subseteq \mathbf{pos}(\mu_{\mathcal{P}}(B), \mu_{\mathcal{P}}(C))$. Thus, by defining

$$\mathrm{Refiner}(\mathcal{P}) \triangleq \{(B,C) \in P \times P \mid B \to^{\exists} C,\ \mu_{\mathcal{P}}(B) \not\subseteq \mathbf{pos}(\mu_{\mathcal{P}}(B), \mu_{\mathcal{P}}(C))\}$$

the following characterization holds:

Theorem 4.1. $\mathcal{P} = (P, \trianglelefteq)$ *is a stuttering simulation iff for any* $s \in \Sigma$, $\mu_{\mathcal{P}}(s) \subseteq P_\ell(s)$ *and* $\mathrm{Refiner}(\mathcal{P}) = \varnothing$.

4.1 A Symbolic Algorithm

The algorithm *BasicSSA* is therefore made symbolic as follows:

(1) $\langle P_\ell, \mathrm{id}\rangle$ is the input PR, where $(B,C) \in \mathrm{id} \Leftrightarrow B = C$;

(2) Find $(B,C) \in \mathrm{Refiner}(\mathcal{P})$; if $\mathrm{Refiner}(\mathcal{P}) = \varnothing$ then exit;

(3) Compute $S = \mathbf{pos}(\mu_{\mathcal{P}}(B), \mu_{\mathcal{P}}(C))$;

(4) $\mathcal{P}' := \langle P', \trianglelefteq'\rangle$, where $P' = \mathit{Split}(P, S)$ and $\trianglelefteq'$ is modified in such a way that for any $s \in \Sigma$, $\mu_{\mathcal{P}'}(P'(s)) = \mu_{\mathcal{P}}(P(s))$;

(5) $\mathcal{P}'' := \langle P', \trianglelefteq''\rangle$, where $\trianglelefteq'$ is modified to $\trianglelefteq''$ in such a way that for any $B \in P'$:

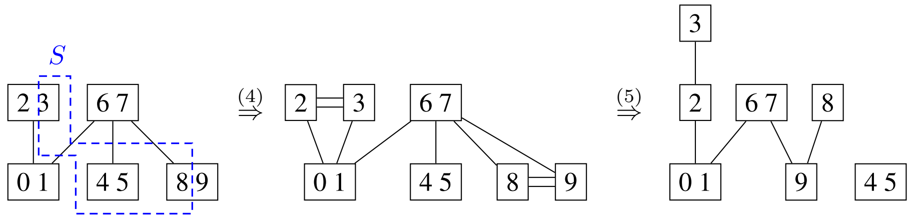
$$\mu_{\mathcal{P}''}(B) = \begin{cases} \mu_{\mathcal{P}'}(B) \cap S & \text{if } B \subseteq S \\ \mu_{\mathcal{P}'}(B) & \text{if } B \cap S = \varnothing \end{cases}$$

(6) $\mathcal{P} := \mathcal{P}''$ and go to (2).

```
1 $SSA(\mathbf{PR}\ \langle P, Rel\rangle)$ {
2     $Initialize()$;
3     while $((B, C) := FindRefiner()) \neq (\mathbf{null}, \mathbf{null})$ do
4         list⟨State⟩ $X := Image(\langle P, Rel\rangle, B)$, $Y := Image(\langle P, Rel\rangle, C)$;
5         list⟨State⟩ $S := \mathbf{pos}(X, Y)$;
6         $SplittingProcedure(\langle P, Rel\rangle, S)$;
7         $Refine(\langle P, Rel\rangle, S)$;
8 }
```

Fig. 2. Stuttering Simulation Algorithm SSA

This leads to the symbolic algorithm SSA described in Figure 2, where: the input PR $\langle P, Rel\rangle$ at line 1 is $\langle P_\ell, \mathrm{id}\rangle$ of point (1); point (2) corresponds to the call $FindRefiner()$ at line 3; point (3) corresponds to lines 4-5; point (4) corresponds to the call $SplittingProcedure(\langle P, Rel\rangle, S)$ at line 6; point (5) corresponds to the call $Refine(\langle P, Rel\rangle, S)$ at line 7. The following graphical example shows how points (4) and (5) refine a PR $\langle\{[0,1],[2,3],[4,5],[6,7],[8,9]\}, \trianglelefteq\rangle$ w.r.t. the set $S = \{3,4,5,8\}$, where if $B \lhd C$ then B is drawed below C while if $B \lhd C$ and $C \lhd B$ then B and C are at same height and connected by a double line.



Theorem 4.2 (Correctness). *SSA is a correct implementation of BasicSSA, i.e., if* StSim *is the output function of BasicSSA on input partition P_ℓ then SSA on input PR $\langle P_\ell, \mathrm{id}\rangle$ terminates with an output PR $\mathcal{P}$ such that for any $x \in \Sigma$,* $\mathrm{StSim}(x) = \mu_{\mathcal{P}}(x)$.

5 Bottom States

While it is not too hard to devise an efficient implementation of lines 2 and 4-7 of the SSA algorithm, it is instead not straightforward to find a refiner in an efficient way. In Groote and Vaandrager's [6] algorithm for computing stuttering bisimulations the key point for efficiently finding a refiner in their setting is the notion of *bottom state*. Given a set of states $S \subseteq \Sigma$, a bottom state of S is a state $s \in S$ that cannot go inside S, i.e., s can only go outside S (note that s may also have no outgoing transition). For any $S \subseteq \Sigma$, we therefore define:

$$\mathrm{Bottom}(S) \triangleq S \smallsetminus \mathrm{pre}(S).$$

Bottom states allow to efficiently find refiners in KSs that do not contain cycles of states all having the same labeling. Following Groote and Vaandrager [6], a transition

$s \rightarrow t$ is called *inert* for a partition $P \in \mathrm{Part}(\Sigma)$ when $P(s) = P(t)$. Clearly, if a set of states S in a KS is strongly connected via inert transitions for the labeling partition P_ℓ then all the states in S are stuttering simulation equivalent, i.e., if $s, s' \in S$ then $P_{\mathrm{stsim}}(s) = P_{\mathrm{stsim}}(s')$. Thus, each strongly connected component (s.c.c.) S with respect to inert transitions for P_ℓ, called *inert s.c.c.*, can be collapsed to one single "symbolic state". In particular, if $\{s\}$ is one such inert s.c.c., i.e. if $s \rightarrow s$, then this collapse simply removes the transition $s \rightarrow s$. It is important to remark that a standard depth-first search algorithm by Tarjan [3], running in $O(|\Sigma| + |{\rightarrow}|)$ time, allows us to find and then collapse all the inert s.c.c.'s in the input KS. We can thus assume w.l.o.g. that the input KS $\mathcal{K}$ does not contain inert s.c.c.'s. The following characterization of refiners therefore holds.

Lemma 5.1. *Assume that $\mathcal{K}$ does not contain inert s.c.c.'s. Let $\mathcal{P} = \langle P, \trianglelefteq \rangle$ be a PR such that for any $B \in P$, $\mu_{\mathcal{P}}(B) \subseteq P_\ell(B)$. Consider $(B, C) \in P \times P$ such that $B \rightarrow^{\exists} C$. Then, $(B, C) \in \mathrm{Refiner}(\mathcal{P})$ iff $\mathrm{Bottom}(\mu_{\mathcal{P}}(B)) \not\subseteq \mu_{\mathcal{P}}(C) \cup \mathrm{pre}(\mu_{\mathcal{P}}(C))$.*

If $B \in P$ is any block then we define as *local bottom states* of B all the bottom states of $\mu_{\mathcal{P}}(B)$ that belong to B, namely

$$\mathrm{localBottom}(B) \triangleq \mathrm{Bottom}(\mu_{\mathcal{P}}(B)) \cap B.$$

Also, we define $C \in P$ as a *bottom block* for B when $B \triangleleft C$ and C contains at least a bottom state of $\mu_{\mathcal{P}}(B)$, that is:

$$\mathrm{bottomBlock}(B) \triangleq \{C \in P \mid B \triangleleft C,\ C \cap \mathrm{Bottom}(\mu_{\mathcal{P}}(B)) \neq \varnothing\}.$$

Local bottoms and bottom blocks characterize refiners for stuttering simulation as follows:

Theorem 5.2. *Assume that $\mathcal{K}$ does not contain inert s.c.c.'s. Let $\mathcal{P} = \langle P, \trianglelefteq \rangle$ be a PR such that $\trianglelefteq$ is a preorder and for any $B \in P$, $\mu_{\mathcal{P}}(B) \subseteq P_\ell(B)$. Consider $(B, C) \in P \times P$ such that $B \rightarrow^{\exists} C$ and for any (D, E) such that $D \rightarrow^{\exists} E$ and $(B, C) \triangleleft (D, E)$, $(D, E) \notin \mathrm{Refiner}(\mathcal{P})$. Then, $(B, C) \in \mathrm{Refiner}(\mathcal{P})$ iff at least one of the following two conditions holds:*

(i) *$C \ntrianglelefteq B$ and $\mathrm{localBottom}(B) \not\subseteq \mathrm{pre}(\mu_{\mathcal{P}}(C))$;*
(ii) *There exists $D \in \mathrm{bottomBlock}(B)$ such that $C \ntrianglelefteq D$ and $D \not\rightarrow^{\exists} \mu_{\mathcal{P}}(C)$.*

We will show that this characterization provides the basis for an algorithm that efficiently finds refiners. Hence, this procedure also checks whether a given preorder R is a stuttering simulation or not. This can be done in $O(|P||{\rightarrow}|)$ time and $O(|\Sigma||P| \log |\Sigma|)$ space, where P is the partition corresponding to the equivalence $R \cap R^{-1}$. Thus, this algorithm for checking stuttering simulation already significantly improves Bulychev et al.'s [2] procedure that runs in $O(|{\rightarrow}|^2)$ time and space.

6 Implementation

6.1 Data Structures

SSA is implemented by exploiting the following data structures.

(i) A state s is represented by a record that contains the list $\mathrm{pre}(s)$ of its predecessors and a pointer s.block to the block $P(s)$ that contains s. The state space Σ is represented as a doubly linked list of states.
(ii) The states of any block B of the current partition P are consecutive in the list Σ, so that B is represented by two pointers begin and end: the first state of B in Σ and the successor of the last state of B in Σ, i.e., $B = [B.\mathrm{begin}, B.\mathrm{end}[$. Moreover, B contains a pointer B.intersection to a block whose meaning is as follows: after a call to $\mathit{Split}(P, S)$ for splitting P w.r.t. a set of states S, if $\varnothing \neq B \cap S \subsetneq B$ then B.intersection points to a block that represents $B \cap S$, otherwise B.intersection $=$ **null**. Finally, the fields localBottoms and bottomBlocks for a block B represent, respectively, the local bottom states of B and the bottom blocks of B. The current partition P is stored as a doubly linked list of blocks.
(iii) The current relation $\trianglelefteq$ on P is stored as a resizable $|P| \times |P|$ boolean matrix Rel: $\mathit{Rel}(B, C) =$ **tt** iff $B \trianglelefteq C$. Recall [3, Section 17.4] that insert operations in a resizable array (whose capacity is doubled as needed) take amortized constant time, and a resizable matrix (or table) can be implemented as a resizable array of resizable arrays. The boolean matrix Rel is resized by adding a new entry to Rel, namely a new row and a new column, for any block B that is split into two new blocks $B \smallsetminus S$ and $B \cap S$.
(iv) SSA additionally stores and maintains a resizable integer table Count and a resizable integer matrix BCount. Count is indexed over Σ and P and has the following meaning: $\mathrm{Count}(s, C) \triangleq |\{(s,t) \mid C \trianglelefteq D, t \in D, s{\rightarrow}t\}|$. BCount is indexed over $P \times P$ and has the following meaning: $\mathrm{BCount}(B, C) \triangleq \sum_{s \in B} \mathrm{Count}(s, C)$. The table Count allows to implement the test $s \notin \mathrm{pre}(\mu_{\mathcal{P}}(C))$ in constant time as $\mathrm{Count}(s, C) = 0$, while BCount allows to implement the test $B \not\rightarrow^{\exists} \mu_{\mathcal{P}}(C)$ in constant time as $\mathrm{BCount}(B, C) = 0$.

6.2 *FindRefiner* Algorithm

The algorithm $\mathit{FindRefiner}()$ in Figure 3 is an implementation of the characterization of refiners provided by Theorem 5.2. In particular, lines 8-10 implement condition (i) of Theorem 5.2 and lines 11-12 implement condition (ii). The correctness of this implementation depends on the following key point. Given a pair of blocks $(B, C) \in P \times P$ such that $B {\rightarrow}^{\exists} C$, in order to ensure the equivalence: $(B, C) \in \mathrm{Refiner}(\mathcal{P})$ iff (i) $\vee$ (ii), Theorem 5.2 requires as hypothesis the following condition:

$$\forall (D, E) \in P \times P.\ D {\rightarrow}^{\exists} E \ \&\ (B, C) \vartriangleleft (D, E) \ \Rightarrow (D, E) \notin \mathrm{Refiner}(\mathcal{P}) \qquad (*)$$

In order to ensure this condition $(*)$, we guarantee throughout the execution of SSA that the list P of blocks is stored in reverse topological ordering w.r.t. $\trianglelefteq$, so that if $B \vartriangleleft B'$ then B' precedes B in the list P. The reverse topological ordering of P initially holds because the input PR is the DAG $\langle P_\ell, \mathrm{id} \rangle$ which is trivially topologically ordered (whatever the ordering of P_ℓ is). More in general, for a generic input PR $\langle P, \mathit{Rel} \rangle$ to SSA the function $\mathit{Initialize}()$ achieves this reverse topological ordering by a standard algorithm [3, Section 22.4] that runs in $O(|P|^2)$ time. Then, the reverse topological ordering of P is always maintained throughout the execution of SSA. In fact, if the

```
1  Precondition: The list $P$ is stored in reverse topological ordering wrt $Rel$
2  ⟨Block, Block⟩ $FindRefiner()$ {
3    matrix⟨bool⟩ Refiner;
4    forall $B \in P$ do  forall $C \in P$ do Refiner($B$,$C$) := maybe;
5    forall $C \in P$ do
6      forall $B \in P$ such that $B \to^{\exists} C$ do
7        if (Refiner($B$,$C$) = maybe) then
8          if ($Rel(C, B)$ = ff) then
9            forall $s \in B$.localBottoms do
10             if (Count($s, C$) = 0) then  return $(B, C)$;
11         forall $D \in B$.bottomBlocks do
12           if ($Rel(C, D)$ = ff & BCount($D, C$) = 0) then return $(B, C)$;
13         forall $E \in P$ do
14           if ($Rel(E, C)$ = tt) then Refiner($B$,$E$) := ff;
15   return (null,null);
16 }
```

Fig. 3. $FindRefiner()$ algorithm

partition P is split w.r.t. a set S and a block B generates two new descendant blocks $B \cap S$ and $B \smallsetminus S$ then our $SplittingProcedure$ in Figure 5 modifies the ordering of the list P as follows: B is replaced in P by inserting $B \cap S$ immediately followed by $B \smallsetminus S$. This guarantees that at the exit of $Refine(\langle P, Rel\rangle, S)$ at line 7 of SSA the list P is still in reverse topological ordering w.r.t. Rel. This is a consequence of the fact that at the exit of $Refine(\langle P, Rel\rangle, S)$, by point (5) in Section 4.1, we have that $\mu_{\langle P,Rel\rangle}(B \cap S) = \mu_{\langle P,Rel\rangle}(B) \cap S$, i.e., $\mu_{\langle P,Rel\rangle}(B \cap S) \cap (B \smallsetminus S) = \varnothing$ so that $B \cap S \not\trianglelefteq B \smallsetminus S$. The reverse topological ordering of P w.r.t. $\trianglelefteq$ ensures that if $(B, C) \triangleleft (B', C')$ then the pair (B, C) is scanned by $FindRefiner$ after the pair (B', C'). Since $FindRefiner()$ exits as soon as a refiner is found, we have that (B', C') cannot be a refiner, so that condition $(*)$ holds for (B, C).

When $FindRefiner()$ determines that a pair of blocks (B, C), with $B \to^{\exists} C$, is not a refiner, it stores this information in a local boolean matrix Refiner that is indexed over $P \times P$ and initialized to **maybe**. Thus, the meaning of the matrix Refiner is as follows: if Refiner(B, C) = **ff** then $(B, C) \notin$ Refiner$(\mathcal{P})$. If $(B, C) \notin$ Refiner$(\mathcal{P})$ then both (i) and (ii) do not hold, therefore $FindRefiner()$ executes the for-loop at lines 13-14 so that any (B, E) with $E \trianglelefteq C$ is marked as Refiner(B, E) = **ff**. This is correct because if $(B, C) \notin$ Refiner$(\mathcal{P})$ and $(B, E) \trianglelefteq (B, C)$ then $(B, E) \notin$ Refiner$(\mathcal{P})$: in fact, by Lemma 5.1, Bottom$(\mu_{\mathcal{P}}(B)) \subseteq \mu_{\mathcal{P}}(C) \cup \text{pre}(\mu_{\mathcal{P}}(C))$, and since $E \trianglelefteq C$ implies, because $\trianglelefteq$ is transitive, $\mu_{\mathcal{P}}(C) \subseteq \mu_{\mathcal{P}}(E)$, we have that Bottom$(\mu_{\mathcal{P}}(B)) \subseteq \mu_{\mathcal{P}}(E) \cup \text{pre}(\mu_{\mathcal{P}}(E))$, so that, by Lemma 5.1, $(B, E) \notin$ Refiner$(\mathcal{P})$. The for-loop at lines 13-14 is therefore an optimization of Theorem 5.2 since it determines that some pairs of blocks are not a refiner without resorting to the condition ¬(i) ∧ ¬(ii) of Theorem 5.2. This optimization and the related matrix Refiner turn out to be crucial for obtaining the overall time complexity of SSA.

```
1  Precondition: $\mathrm{TS}(S, \to, P_\ell)$ & $\forall x, y \in S.\ P_\ell(x) = P_\ell(y)$
2  list⟨State⟩ pos(list⟨State⟩ $S$, list⟨State⟩ $T$) {
3      list⟨State⟩ $R := \varnothing$;
4      forall $s \in S$ do mark1($s$);
5      forall $t \in T$ do
6          forall $s \in \mathrm{pre}(t)$ such that marked1($s$) do
7              mark2($s$); $R$.append($s$);
8      forall $y \in S$ backward such that marked2($y$) do
9          forall $x \in \mathrm{pre}(y)$ such that marked1($x$) & unmarked2($x$) do
10             mark2($x$); $R$.append($x$);
11     forall $x \in S$ do unmark1($x$); forall $x \in R$ do unmark2($x$);
12     return $R$;
13 }
```

Fig. 4. Computation of **pos**

6.3 Computing pos

Given two lists of states S and T, we want to compute the set of states that belong to $\mathbf{pos}(S, T)$. This can be done by traversing once the edges of the transition relation $\to$ provided that the list Σ of states satisfies the following property:

For all $x, y \in \Sigma$, if x precedes y in the list Σ and $\ell(x) = \ell(y)$ then $y \not\to x$.

We denote this property by $\mathrm{TS}(\Sigma, \to, P_\ell)$. Hence, this is a topological ordering of Σ w.r.t. the transition relation $\to$ that is local to each block of the labeling partition P_ℓ. As described in Section 5, as an initial pre-processing step of SSA, we find and collapse inert s.s.c.'s. After this pre-processing step, Σ is successively topologically ordered w.r.t. $\to$ locally to each block of P_ℓ in $O(|\Sigma| + |{\to}|)$ time in order to initially establish $\mathrm{TS}(\Sigma, \to, P_\ell)$. We will see in Section 6.4 that while the ordering of the list Σ of states changes across the execution of SSA, the property $\mathrm{TS}(\Sigma, \to, P_\ell)$ is always maintained invariant.

The computation of $\mathbf{pos}(S, T)$ is done by the algorithm in Figure 4. The result R consists of all the states in S that are marked2. We assume that all the states in S have the same labeling by ℓ: this is clearly true when the function **pos** is called from the algorithm SSA. The for-loop at lines 5-7 makes the states in $S \cap \mathrm{pre}(T)$ marked2. Then, the for-loop at lines 8-10 scans backward the list of states S and when a marked2 state y is encountered then all the states in $S \cap \mathrm{pre}(y)$ are marked2. It is clear that the property $\mathrm{TS}(\Sigma, \to, P_\ell)$ guarantees that this procedure does not miss states that are in $\mathbf{pos}(S, T)$.

6.4 *SplittingProcedure*

SSA calls $SplittingProcedure(\langle P, Rel\rangle, S)$ at line 6 with the precondition $\mathrm{TS}(\Sigma, \to, P_\ell)$ and needs to maintain this invariant property at the exit (as discussed in Section 6.3 this

```
list⟨Block⟩ Split(list⟨Block⟩ P, list⟨State⟩ S) {
    list⟨Block⟩ split;
    forall $x \in S$ do
        if ($x$.block.intersection = null) then
            Block $B$ := new Block;
            $x$.block.intersection := $B$;
            split.append($x$.block);
        move $x$ in the list $\Sigma$ from $x$.block at the end of $B$;
        if ($x$.block = $\varnothing$) then $x$.block := copy($B$); $x$.block.intersection := null;
    forall $B \in$ split do
        if ($B$.intersection = null) then split.remove($B$); delete $B$;
        else insert $B$.intersection in $P$ in front of $B$;
    return split;
}

void SplittingProcedure(PR ⟨P, Rel⟩, list⟨State⟩ S) {
    list⟨Block⟩ split := $Split(P, S)$;
    if (split $\neq \varnothing$) then
        resize $Rel$; // update $Rel$
        forall $B \in P$ do forall $C \in$ split do $Rel(C$.intersection$, B) := Rel(C, B)$;
        forall $B \in$ split do forall $C \in P$ do $Rel(C, B$.intersection$) := Rel(C, B)$;
        $Update()$; // update Count, BCount, localBottoms, bottomBlocks
        forall $B \in P$ do $B$.intersection := null;
}
```

Fig. 5. Splitting Procedure

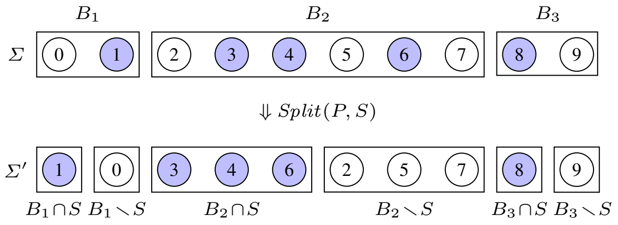
is crucial for computing **pos**). This function must modify the current PR $\mathcal{P} = \langle P, Rel\rangle$ to $\mathcal{P}' = \langle P', Rel'\rangle$ as follows:

(A) P' is the partition obtained by splitting P w.r.t. the splitter S;
(B) Rel is modified to Rel' in such a way that for any $x \in \Sigma$, $\mu_{\mathcal{P}'}(P'(x)) = \mu_{\mathcal{P}}(P(x))$.

Recall that the states of a block B of P are consecutive in the list Σ, so that B is represented as $B = [B.\text{begin}, B.\text{end}[$. An implementation of the splitting operation $Split(P, S)$ that only scans the states in S, i.e. that takes $O(|S|)$ time, is quite easy and standard (see e.g. [6,11]). However, this operation affects the ordering of the states in the list Σ because states are moved from old blocks to newly generated blocks. It turns out that this splitting operation can be implemented in a careful way that preserves the invariant property $\text{TS}(\Sigma, \to, P_\ell)$. The idea is rather simple. Observe that the list of states $S = \textbf{pos}(\mu_{\mathcal{P}}(X), \mu_{\mathcal{P}}(Y))$ can be (and actually is) built as a sublist of Σ so that the following property holds: If x precedes y in S and $P_\ell(x) = P_\ell(y)$ then $y \not\to x$. The following picture shows the idea of our implementation of $Split(P, S)$, where states within filled circles determine the splitter set S.

```
void $Refine$(PR $\langle P, Rel\rangle$, list$\langle$State$\rangle$ $S$) {
   list$\langle$Block$\rangle$ $L := \varnothing$;
   forall $s \in S$ such that unmarked($s$.block) do mark($s$.block); $L$.append($s$.block);
   forall $B \in L$ do
      forall $C \in P$ do
         if ($Rel(B, C) =$ tt & unmarked($C$)) then
            $Rel(B, C) :=$ ff;
            forall $y \in C$ do
               forall $x \in$ pre($y$) do Count($x, B$)−−; BCount($x$.block, $B$)−−;
            if ($C \in B$.bottomBlocks) then $B$.bottomBlocks.erase($C$);
            forall $y \in C$ do
               forall $x \in$ pre($y$) do
                  if ($x$.block $\neq B$ & $Rel(B, x$.block$) =$ tt & Count($x, B$) $= 0$)
                  then
                     mark2($x$.block);
                     if unmarked2($x$.block) then
                        $B$.bottomBlocks.append($x$.block);
                  else if ($x$.block $= B$ & Count($x, B$) $= 0$) then
                     $B$.localBottoms.append($x$);
   forall $B \in P$ do unmark($B$); unmark2($B$);
}
```

Fig. 6. *Refine* function



The property $\mathrm{TS}(\Sigma', \to, P_\ell)$ still holds for the modified list of states Σ'. In fact, from the above picture observe that it is enough to check that: if B has been split into $B \cap S$ and $B \smallsetminus S$ by preserving the relative orders of the states in Σ then if $x \in B \cap S$ and $y \in B \smallsetminus S$ then $y \not\to x$. This is true because if $y \to x$ and $x \in S = \mathbf{pos}(\mu_{\mathcal{P}}(X), \mu_{\mathcal{P}}(Y))$ then, since x and y are in the same block of P and $\mu_{\mathcal{P}}(X)$ is a union of some blocks of P, by definition of **pos** we would also have that $y \in S$, which is a contradiction.

The functions in Figure 5 sketch a pseudo-code that implements the above described splitting operation (the $Update()$ function that updates data structures is omitted). The above point (B), i.e., the modification of Rel to Rel' so that for any $x \in \Sigma$, $\mu_{\mathcal{P}'}(P'(x)) = \mu_{\mathcal{P}}(P(x))$ is straightforward and is implemented at lines 18-20 of $SplittingProcedure()$.

6.5 *Refine* Function

SSA calls $\mathit{Refine}(\langle P, \mathit{Rel}\rangle, S)$ at line 7 with the precondition that S is a union of blocks of the current partition P. The function $\mathit{Refine}(\langle P, \mathit{Rel}\rangle, S)$ in Figure 6 implements the point (5) of Section 4.1. This function must modify the current PR $\mathcal{P} = \langle P, \mathit{Rel}\rangle$ to $\mathcal{P}' = \langle P, \mathit{Rel}'\rangle$ by pruning the relation Rel in such a way that for any $B \in P$:

$$\mu_{\mathcal{P}'}(B) = \begin{cases} \mu_{\mathcal{P}}(B) \cap S & \text{if } B \subseteq S \\ \mu_{\mathcal{P}}(B) & \text{if } B \cap S = \varnothing \end{cases}$$

This is done by the $\mathit{Refine}()$ function at lines 5-7 by reducing the relation Rel to Rel' as follows: if $B, C \in P$ and $\mathit{Rel}(B,C) = \mathbf{tt}$ then $\mathit{Rel}'(B,C) = \mathbf{ff}$ iff $B \subseteq S$ and $C \cap S = \varnothing$, while the rest of the code updates the data structures Count, BCount, localBottoms and bottomBlocks accordingly.

6.6 Auxiliary Functions

The implementation of the remaining functions $\mathit{Initialize}()$ and $\mathit{Image}()$ is easy and is omitted. It is just worth remarking that $\mathit{Initialize}()$ in particular initially establishes the property $\mathrm{TS}(\Sigma, \to, P_\ell)$ and provides an initial reverse topological order of P w.r.t. Rel when the input partial order Rel is not the identity relation id.

6.7 Complexity

Time and space bounds for SSA are as follows. In the following statement we assume, as usual in model checking, that the transition relation $\to$ is total, i.e., for any $s \in \Sigma$ there exists $t \in \Sigma$ such that $s{\to}t$, so that the inequalities $|\Sigma| \leq |{\to}|$ and $|P_{\mathrm{stsim}}| \leq |{\to^{\exists}}|$ hold, where $\to^{\exists}$ is the existential transition relation between blocks of P_{stsim}, and this allows us to simplify the expression of the time bound.

Theorem 6.1 (Complexity). *SSA runs in $O(|P_{\mathrm{stsim}}|^2(|{\to}| + |P_{\mathrm{stsim}}||{\to^{\exists}}|))$-time and $O(|\Sigma||P_{\mathrm{stsim}}| \log |\Sigma|)$-space.*

6.8 Adapting SSA for LTSs

The algorithms SSA computes the stuttering simulation preorder on KSs, but it can be modified to work over LTSs by following the adaptation to LTSs of Groote and Vaandrager's algorithm [6] for KSs. Due to lack of space the details are here omitted. We just mention that we have a parametric $\mathbf{pos}_a$ operator for any action $a \in \mathit{Act}$ so that the notions of splitting and refinement of the current PR are parameterized w.r.t. the action a.

7 Conclusion

We presented an algorithm, called SSA, for computing the stuttering simulation preorder and equivalence on Kripke structures (or labeled transition systems). To the best of our knowledge, this is the first algorithm for computing this behavioural preorder. The only available algorithm related to stuttering simulation is a procedure by Bulychev et al. [2] that checks whether a given relation is a stuttering simulation. Our procedure SSA includes an algorithm for checking whether a given relation is a stuttering simulation that significantly improves Bulychev et al.'s one both in time and in space.

Acknowledgements. We are grateful to Silvia Crafa for numerous helpful discussions. This work was partially supported by the PRIN 2007 Project "*AIDA2007: Abstract Interpretation Design and Applications*" and by the University of Padova under the Projects "*Formal methods for specifying and verifying behavioural properties of software systems*" and "*Analysis, verification and abstract interpretation of models for concurrency*".

References

1. Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theor. Comp. Sci. 59, 115–131 (1988)
2. Bulychev, P.E., Konnov, I.V., Zakharov, V.A.: Computing (bi)simulation relations preserving CTL*-X for ordinary and fair Kripke structures. In: Mathematical Methods and Algorithms, Institute for System Programming, Russian Academy of Sciences, vol. 12 (2007), http://lvk.cs.msu.su/~peterbul
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press and McGraw-Hill, Cambridge (2001)
4. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation. J. ACM 42(2), 458–487 (1995)
5. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for Buchi automata. SIAM J. Comput. 34(5), 1159–1175 (2001)
6. Groote, J.F., Vaandrager, F.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 626–638. Springer, Heidelberg (1990)
7. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. 36^{th} FOCS, pp. 453–462 (1995)
8. Lamport, L.: What good is temporal logic? In: Information Processing 1983. IFIP, pp. 657–668 (1983)
9. Manolios, P.: Mechanical Verification of Reactive Systems. PhD thesis, University of Texas at Austin (2001)
10. Nejati, S., Gurfinkel, A., Chechik, M.: Stuttering abstraction for model checking. In: 3rd IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM 2005), pp. 311–320 (2005)
11. Ranzato, F., Tapparo, F.: A new efficient simulation equivalence algorithm. In: Proc. 22nd IEEE Symp. on Logic in Computer Science (LICS 2007), pp. 171–180. IEEE Press, Los Alamitos (2007)
12. Ranzato, F., Tapparo, F.: Generalizing the Paige-Tarjan algorithm by abstract interpretation. Information and Computation 206(5), 620–651 (2008)

Weak Time Petri Nets Strike Back!

Pierre-Alain Reynier[1,⋆] and Arnaud Sangnier[2,⋆⋆]

[1] LIF, Université Aix-Marseille & CNRS, France
pierre-alain.reynier@lif.univ-mrs.fr
[2] Dipartimento di Informatica, Università di Torino, Italy
sangnier@di.unito.it

Abstract. We consider the model of Time Petri Nets where time is associated with transitions. Two semantics for time elapsing can be considered: the strong one, for which all transitions are urgent, and the weak one, for which time can elapse arbitrarily. It is well known that many verification problems such as the marking reachability are undecidable with the strong semantics. In this paper, we focus on Time Petri Nets with weak semantics equipped with three different memory policies for the firing of transitions. We prove that the reachability problem is decidable for the most common memory policy (intermediate) and becomes undecidable otherwise. Moreover, we study the relative expressiveness of these memory policies and obtain partial results.

1 Introduction

For verification purpose, *e.g.* in the development of embedded platforms, there is an obvious need for considering time features and the study of timed models has thus become increasingly important. For distributed systems, different timed extensions of Petri nets have been proposed which allow the combination of an unbounded discrete structure with dense-time variables.

There are several ways to express urgency in timed systems, as discussed in [17]. In timed extensions of Petri nets, two types of semantics are considered for time elapsing. In the *weak* semantics, all time delays are allowed whereas in the *strong* one, all transitions are urgent, *i.e.* time delays cannot disable transitions. While for models with finite discrete structure (such as timed extensions of bounded Petri nets or timed automata [3]), standard verification problems are decidable for both semantics, for models with infinite discrete structure, the choice of the semantics has a deep influence on decidability issues. In this work, we consider the model of Time Petri Nets [14] (TPN) where clocks are associated with transitions, and which is commonly considered under a strong semantics. In this model, all the standard verification problems are known to be undecidable [10]. On the other hand, in the model of timed-arc Petri nets [5], where clocks are associated with tokens and which is equipped with a weak semantics, many verification problems are decidable (coverability, boundedness...). Indeed, this

⋆ Partly supported by the French project DOTS (ANR-06-SETI-003) and by the European project QUASIMODO (FP7-ICT-STREP-214755).

⋆⋆ Supported by a post-doctoral scolarship from DGA/ENS Cachan and by the MIUR-PRIN project PaCo - Performability-Aware Computing: Logics, Models and Languages.

semantics entails for this model monotonicity properties which allow the application of well-quasi-ordering techniques, see [8,2,1]. Note however that the reachability of a discrete marking is undecidable, as proven in [18]. A natural question, which had surprisingly no answer until now, as mentioned in a recent survey on the topic [7], is thus to study TPN under a weak semantics of time elapsing.

The time-elapsing policy states which delays are allowed in a configuration. The memory policy is concerned with the resets of clocks, and intuitively specifies, when firing a transition, which timing informations are preserved. The original model of Merlin [14] is equipped with an *intermediate* semantics which considers the intermediary marking bewteen consumption and production. Two others memory policies have been considered in [4] (the *atomic* and the *persistent atomic*) in which the firings of transitions are performed atomically. While these policies can be thought as cosmetic for the model of TPN, the results we obtain show this is not the case.

We are interested in the impact of the weak semantics on TPN, distinguishing between the different memory policies. We first study the decidability issues, and prove that for TPN with weak intermediate semantics, a discrete marking is reachable if and only if it is reachable in the underlying untimed Petri net. As a corollary, the problem of the marking reachability (and also coverability, boundedness) is decidable for this model. More surprisingly, we also prove that when changing the memory policy this result does not hold anymore and the verification problems become undecidable. In this work, we only consider untimed verification problems and we plan to study timed versions in future work. We then compare w.r.t. weak time bisimilarity (weak stands here for silent transitions) the expressive power of weak TPN looking at the different memory policies. We first prove that the persistent atomic semantics is strictly more expressive that the atomic semantics. Then, concerning atomic and intermediate memory policies, we provide a TPN which shows that the atomic semantics is not included in the intermediate one.

Related works. As mentioned above, there are, up to our knowledge, only very few works considering TPN under a weak semantics. In [7] the authors have proven that the weak intermediate semantics and the strong intermediate semantics are uncomparable. In another line of work, [9] considers TPN under a semantics which is a kind of compromise between the standard strong and weak semantics. They provide translations between this model and timed state machines.

Due to lack of space omitted proofs can be found in [16].

2 Definitions

Let Σ be a finite alphabet, Σ^* is the set of finite words over Σ. We note $\Sigma_\tau = \Sigma \cup \{\tau\}$ where $\tau \notin \Sigma$ represents internal actions. ϵ will represent the empty word. The sets $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{Q}_{\geq 0}$, $\mathbb{R}$ and $\mathbb{R}_{\geq 0}$ are respectively the sets of natural, rational, non-negative rational, real and non-negative real numbers. A *valuation* v over a finite set X is a mapping in $\mathbb{R}_{\geq 0}^X$. For $v \in \mathbb{R}_{\geq 0}^X$ and $d \in \mathbb{R}_{\geq 0}$, $v+d$ denotes the valuation defined by $(v+d)(x) = v(x)+d$. We note $\mathbf{0}$ the valuation which assigns to every $x \in X$ the value 0.

As commonly in use for Time Petri Nets, we will associate rational intervals with transitions. Note that we could handle intervals with bounds given as real numbers if we abstract the problem of comparison of real numbers. We consider the set $\mathcal{I}(\mathbb{Q}_{\geq 0})$ of non-empty intervals (a, b) with non-negative rational bounds $a, b \in \mathbb{Q}_{\geq 0}$. We consider both open and closed bounds, and also allow a right open infinite bound as in $[2, +\infty[$.

2.1 Petri Nets

Definition 1 (Labeled Petri Net (PN)). *A Labeled Petri Net over the alphabet Σ_τ is a tuple $(P, T, \Sigma_\tau, {}^\bullet(.), (.)^\bullet, M_0, \Lambda)$ where:*

- *P is a finite set of* places,
- *T is a finite set of* transitions *with $P \cap T = \emptyset$,*
- *${}^\bullet(.) \in (\mathbb{N}^P)^T$ is the* backward *incidence mapping,*
- *$(.)^\bullet \in (\mathbb{N}^P)^T$ is the* forward *incidence mapping,*
- *$M_0 \in \mathbb{N}^P$ is the* initial *marking,*
- *$\Lambda : T \to \Sigma_\tau$ is the* labeling function

As commonly in use in the literature, the vector ${}^\bullet(t)$ (resp. $(t)^\bullet$) in $\mathbb{N}^P$ is noted ${}^\bullet t$ (resp. $t^\bullet$). The semantics of a PN $\mathcal{N} = (P, T, \Sigma_\tau, {}^\bullet(.), (.)^\bullet, M_0, \Lambda)$ is given by its associated labeled transition system $[\![\mathcal{N}]\!] = (\mathbb{N}^P, M_0, \Sigma_\tau, \Rightarrow)$ where $\Rightarrow \subseteq \mathbb{N}^P \times \Sigma_\tau \times \mathbb{N}^P$ is the transition relation defined by $M \overset{a}{\Longrightarrow} M'$ iff $\exists t \in T$ s.t. $\Lambda(t) = a \wedge M \geq {}^\bullet t \wedge M' = M - {}^\bullet t + t^\bullet$. For convenience we will sometimes also write, for $t \in T$, $M \overset{t}{\Longrightarrow} M'$ if $M \geq {}^\bullet t$ and $M' = M - {}^\bullet t + t^\bullet$. We also write $M \Rightarrow M'$ if there exists $a \in \Sigma_\tau$ such that $M \overset{a}{\Longrightarrow} M'$. The relation $\Rightarrow^*$ represents the reflexive and transitive closure of $\Rightarrow$. We denote by $\texttt{Reach}(\mathcal{N})$ the set of reachable markings defined by $\{M \in \mathbb{N}^P \mid M_0 \Rightarrow^* M\}$.

It is well known that for PN the reachability problem which consists in determining whether a given marking M belongs to $\texttt{Reach}(\mathcal{N})$ is decidable; it has in fact been proved independently in [13] and [12].

We introduce a last notation concerning Labeled Petri Nets. Given a PN $\mathcal{N}$, a marking M of $\mathcal{N}$ and a multi-set $\Delta = \langle t_1, \ldots, t_n \rangle$ of transitions of $\mathcal{N}$, we write $M \overset{\Delta}{\Longmapsto} M'$ if and only if the multi-set Δ can be fired from M, meaning that there exists an ordering of transitions in Δ, represented as a permutation φ of $\{1, \ldots, n\}$, such that the sequence of firings $M \overset{t_{\varphi(1)}}{\Longrightarrow} M_1 \overset{t_{\varphi(2)}}{\Longrightarrow} M_2 \ldots \overset{t_{\varphi(n)}}{\Longrightarrow} M'$ exists in $[\![\mathcal{N}]\!]$.

2.2 Timed Transition Systems

Timed transition systems describe systems which combine discrete and continuous evolutions. They are used to define the behavior of timed systems such as Time Petri Nets [14] or Timed Automata [3].

Definition 2 (Timed Transition System (TTS)). *A* timed transition system *over the alphabet Σ_τ is a transition system $S = (Q, q_0, \Sigma_\tau, \to)$, where the transition relation $\to \subseteq Q \times (\Sigma_\tau \cup \mathbb{R}_{\geq 0}) \times Q$ consists of discrete transitions $q \xrightarrow{a} q'$ (with $a \in \Sigma_\tau$) representing an instantaneous action, and continuous transitions $q \xrightarrow{d} q'$ (with $d \in \mathbb{R}_{\geq 0}$) representing the passage of d units of time.*

Moreover, we require the following standard properties for TTS :

- TIME-DETERMINISM : if $q \xrightarrow{d} q'$ and $q \xrightarrow{d} q''$ with $d \in \mathbb{R}_{\geq 0}$, then $q' = q''$,
- 0-DELAY : $q \xrightarrow{0} q$,
- ADDITIVITY : if $q \xrightarrow{d} q'$ and $q' \xrightarrow{d'} q''$ with $d,\ d' \in \mathbb{R}_{\geq 0}$, then $q \xrightarrow{d+d'} q''$,
- CONTINUITY : if $q \xrightarrow{d} q'$, then for every d' and d'' in $\mathbb{R}_{\geq 0}$ such that $d = d' + d''$, there exists q'' such that $q \xrightarrow{d'} q'' \xrightarrow{d''} q'$.

With these properties, a *run* of S can be defined as a finite sequence of moves $\rho = q_0 \xrightarrow{d_0} q_0' \xrightarrow{a_0} q_1 \xrightarrow{d_1} q_1' \xrightarrow{a_1} q_2 \ldots \xrightarrow{a_n} q_{n+1}$ where discrete and continuous transitions alternate. To such a run corresponds the timed word $w = (a_i, \eta_i)_{0 \leq i \leq n}$ over Σ_τ where $\eta_i = \sum_{j=0}^{i} d_j$ is the time at which a_i happens. We then denote by *Untimed*(w) the projection of the word $a_0 a_1 \ldots a_n$ over the alphabet Σ and by *Duration*(w) the duration η_n. Note that in the word *Untimed*(w) the symbol τ does not appear. We will sometimes apply, without possible ambiguities, these notations to runs writing *Untimed*(ρ) and *Duration*(ρ). We might also describe the run writing directly $q_0 \xrightarrow{w} q_{n+1}$.

2.3 Time Petri Nets

Syntax. Introduced in [14], Time Petri Nets associate a time interval with each transition of a Petri net.

Definition 3 (Labeled Time Petri Net (TPN)). *A Labeled Time Petri Net over the alphabet Σ_τ is a tuple $(P, T, \Sigma_\tau, {}^\bullet(.), (.)^\bullet, M_0, \Lambda, I)$ where:*

- $(P, T, \Sigma_\tau, {}^\bullet(.), (.)^\bullet, M_0, \Lambda)$ *is a PN,*
- $I : T \mapsto \mathcal{I}(\mathbb{Q}_{\geq 0})$ *associates with each transition a* firing interval.

In the sequel, we associate with an interval its left bound and its right bound. More generally, given a transition t of a TPN, we will denote by *eft*(t) (resp. *lft*(t)) the left bound of $I(t)$ (resp. the right bound of $I(t)$), standing for earliest firing time (resp. latest firing time). We have hence $I(t) = (\textit{eft}(t), \textit{lft}(t))$.

Semantics. A *configuration* of a TPN is a pair (M, ν), where M is a *marking* over P, *i.e.* a mapping in $\mathbb{N}^P$, with $M(p)$ the number of tokens in place p. A transition t is *enabled* in a marking M if $M \geq {}^\bullet t$. We denote by *En*(M) the set of enabled transitions in M. The second component of the pair (M, ν) is a valuation over *En*(M), *i.e.* a mapping in $\mathbb{R}_{\geq 0}^{\textit{En}(M)}$. Intuitively, for each enabled transition t in M, $\nu(t)$ represents the amount of time that has elapsed since t is enabled. An enabled transition t can be fired if $\nu(t)$ belongs to the interval $I(t)$. The marking obtained after this firing is as usual the new marking $M' = M - {}^\bullet t + t^\bullet$. Moreover, some valuations are reset and we say that the corresponding transitions are newly enabled.

Different semantics can be chosen in order to realize these resets. This choice depends of what is called the *memory policy*. For $M \in \mathbb{N}^P$ and $t, t' \in T$ such that $t \in \textit{En}(M)$ we define in different matters a predicate $\uparrow \textit{enabled}_s(t', M, t)$ with $s \in \{I, A, PA\}$ which is true if t' is *newly enabled* by the firing of transition t from marking M, and false otherwise. This predicate indicates whether we need to reset the clock of t' after firing the transition t at the marking M.

I: The *intermediate semantics* considers that the firing of a transition is performed in two steps: consuming the tokens in ${}^\bullet t$, and then producing the tokens in $t^\bullet$. Intuitively, it resets the clocks of t and of the transitions that could not be fired in parallel with t from the marking M. Formally, the predicate $\uparrow\!\mathit{enabled}_I(t', M, t)$ is defined by:

$$\uparrow\!\mathit{enabled}_I(t', M, t) = \big(t' \in \mathit{En}(M - {}^\bullet t + t^\bullet) \wedge (t' \notin \mathit{En}(M - {}^\bullet t) \vee t = t')\big)$$

A: The *atomic semantics* considers that the firing of a transition is obtained by an atomic step. It resets the clocks of t and of the transitions t' which are not enabled at M. The corresponding predicate $\uparrow\!\mathit{enabled}_A(t', M, t)$ is defined by:

$$\uparrow\!\mathit{enabled}_A(t', M, t) = \big(t' \in \mathit{En}(M - {}^\bullet t + t^\bullet) \wedge (t' \notin \mathit{En}(M) \vee t = t')\big)$$

PA: The *persistent atomic semantics* behaves as the atomic semantics except that it does not reset the clock of t.

$$\uparrow\!\mathit{enabled}_{PA}(t', M, t) = \big(t' \in \mathit{En}(M - {}^\bullet t + t^\bullet) \wedge t' \notin \mathit{En}(M)\big)$$

Finally, as recalled in the introduction, there are two ways of letting the time elapse in TPN. The first way, known as the *strong semantics*, is defined in such a matter that time elapsing cannot disable a transition. Hence, when the upper bound of a firing interval is reached then the transition must be fired or disabled. In contrast to that the *weak semantics* does not make any restriction on the elapsing of time. In this work, we focus on the weak semantics of TPN.

Definition 4 (Weak semantics of a TPN). *Let $s \in \{I, A, PA\}$. The weak s-semantics of a TPN $\mathcal{N} = (P, T, \Sigma_\tau, {}^\bullet(.), (.)^\bullet, M_0, \Lambda, I)$ is a timed transition system $[\![\mathcal{N}]\!]_s = (Q, q_0, \Sigma_\tau, \rightarrow_s)$ where $Q = \mathbb{N}^P \times \mathbb{R}_{\geq 0}^{\mathit{En}(M)}$, $q_0 = (M_0, \mathbf{0})$ and $\rightarrow_s$ consists of discrete and continuous moves:*

– *the discrete transition relation is defined $\forall a \in \Sigma_\tau$ by:*

$$(M, \nu) \xrightarrow{a}_s (M', \nu') \text{ iff } \exists t \in T \text{ s.t. } \begin{cases} \Lambda(t) = a, \text{ and,} \\ t \in \mathit{En}(M) \wedge M' = M - {}^\bullet t + t^\bullet, \text{ and,} \\ \nu(t) \in I(t), \text{ and,} \\ \forall t' \in \mathit{En}(M'), \\ \quad \nu'(t') = \begin{cases} 0 & \text{if } \uparrow\!\mathit{enabled}_s(t', M, t) \\ \nu(t') & \text{otherwise} \end{cases} \end{cases}$$

– *the continuous transition relation is defined $\forall d \in \mathbb{R}_{\geq 0}$ by:*

$$(M, \nu) \xrightarrow{d}_s (M, \nu') \text{ iff } \nu' = \nu + d$$

We also write a discrete transition $(M, \nu) \xrightarrow{t}_s (M', \nu')$ to characterize the transition $t \in T$ which allows the firing $(M, \nu) \xrightarrow{\Lambda(t)}_s (M', \nu')$. We extend this notation to words $\theta \in (T \cup \mathbb{R}_{\geq 0})^*$, which correspond to sequences of transitions and delays and lead to a unique (if it exists) run ρ. We may write this run $\rho : (M, \nu) \xrightarrow{\theta}_s (M', \nu')$ and use *Untimed*(θ) (resp. *Duration*(θ)) to denote the word *Untimed*(ρ) (resp. to represent the delay *Duration*(ρ)). Finally, for $s \in \{I, A, PA\}$, we write $(M, \nu) \rightarrow_s (M', \nu')$ if there exists $a \in \Sigma_\tau \cup \mathbb{R}_{\geq 0}$ such that $(M, \nu) \xrightarrow{a}_s (M', \nu')$. The relation $\rightarrow_s^*$ denotes the reflexive and transitive closure of $\rightarrow_s$. For a TPN $\mathcal{N}$ with an initial marking M_0 we define the following reachability sets according to the considered semantics: $\texttt{Reach}(\mathcal{N})_s = \{(M, v) \mid (M_0, \mathbf{0}) \rightarrow_s^* (M, v)\}$.

Example 1. We illustrate the impact of the three memory policies in weak semantics. Consider the net depicted on Figure 1, and the execution $(M, \mathbf{0}) \xrightarrow{1}_s (M, \mathbf{1}) \xrightarrow{a}_s (M, v)$ where $M(p) = 1$. With the intermediate semantics, both clocks are reset as in the intermediate marking, the place p is empty. With the atomic semantics, the clock associated with $t2$ is not reset and the clock associated with $t1$ is reset because it corresponds to the fired transition. Finally, with the persistent atomic semantics no clock is reset.

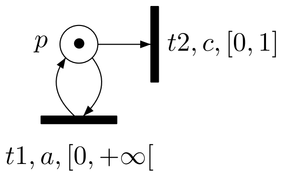


Fig. 1. The TPN $\mathcal{N}_1$

3 Decidability

3.1 Considered Problems and Known Results

Assume $\mathcal{N} = (P, T, \Sigma_\tau, {}^\bullet(.), (.)^\bullet, M_0, \Lambda, I)$ is a TPN. In this section, we will consider the following problems for $s \in \{I, A, PA\}$:

(1) The *marking reachability problem* : given $M \in \mathbb{N}^P$, does there exist $\nu \in \mathbb{R}_{\geq 0}^{En(M)}$ such that $(M, \nu) \in \texttt{Reach}(\mathcal{N})_s$?

(2) The *marking coverability problem* : given $M \in \mathbb{N}^P$, does there exist $M' \in \mathbb{N}^P$ and $\nu \in \mathbb{R}_{\geq 0}^{En(M')}$ such that $M' \geq M$ and $(M', \nu) \in \texttt{Reach}(\mathcal{N})_s$?

(3) The *boundedness problem* : does there exist $b \in \mathbb{N}$ such that for all $(M, \nu) \in \texttt{Reach}(\mathcal{N})_s$ and for all $p \in P$, $M(p) \leq b$?

It is well known that the "untimed" versions of these problems are decidable in the case of Petri nets. In fact, as mentioned before the marking reachability problem is decidable for Petri nets [12,13] and the two other problems can be solved using the Karp and Miller tree whose construction is given in [11].

From [10], we know that these problems are all undecidable when considering TPN with strong semantics no matter whether the semantics is intermediate, atomic or persistent atomic. In fact a TPN with strong semantics can simulate a Minsky machine. A Minsky machine manipulates two integer variables c_1 and c_2 and is composed of a finite number of instructions, each of these instructions being either an incrementation ($q : c_i := c_i + 1$) or a decrementation with a test to zero (q : if $c_i = 0$ goto q' else $c_i := c_i - 1$; goto q''), where $i \in \{1, 2\}$ and q, q', q'' are some labels preceding each instruction. There is also a special label q_f from which the machine cannot do anything. In [15], Minsky proved that the halting problem, which consists in determining if the instruction labeled with q_f is reachable, is undecidable.

It is easy to encode an incrementation using a TPN (or even a PN), with a transition consuming a token in a place characterizing the current control state and producing a token in the next control state and in a place representing the incremented counter.

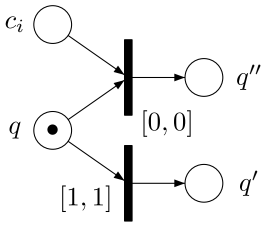


Fig. 2. Encoding decrementation with strong semantics

When encoding the decrementation with the test to zero, the strong semantics plays a crucial role. This encoding is represented on Figure 2. If there is a token in the place c_i, there is no way for the TPN to produce a token in the place q' because time cannot elapse since the transition labeled with the interval $[0, 0]$ is firable. The example of the Figure 2 shows that the strong time semantics allows to encode priorities (between transitions in conflict) and thus to encode inhibitor arcs. This construction obviously fails with the weak semantics.

3.2 The Peculiar Case of TPN with Weak Intermediate Semantics

We prove here that the undecidability results we had before in the case of TPN with strong semantics do not hold anymore when considering the weak intermediate semantics. Before proving this we introduce some notations. For a TPN $\mathcal{N} = (P, T, \Sigma_\tau, {}^\bullet(.), (.)^\bullet, M_0, \Lambda, I)$, we denote by $\mathcal{N}^U$ the untimed PN obtained by removing from $\mathcal{N}$ the component I. Furthermore given a set of configurations $C \subseteq \mathbb{N}^P \times \mathbb{R}^T_{\geq 0}$ of $\mathcal{N}$, we denote by $\texttt{Untime}(C)$ the projection of C over the set $\mathbb{N}^P$. For $s \in \{I, A, PA\}$, we have by definition of the different semantics that $\texttt{Untime}(\texttt{Reach}(\mathcal{N})_s) \subseteq \texttt{Reach}(\mathcal{N}^U)$ and as shown by the example given in Figure 2 this inclusion is strict in the case of the strong semantics. When considering the **weak intermediate** semantics, we prove that from any sequence of transitions Δ firable in $[\![\mathcal{N}^U]\!]$, we can effectively compute a reordering of Δ, and associated timestamps, leading to a correct run of $[\![\mathcal{N}]\!]_I$.

Theorem 5. *For all TPN $\mathcal{N}$,* $\texttt{Untime}(\texttt{Reach}(\mathcal{N})_I) = \texttt{Reach}(\mathcal{N}^U)$.

According to the previous remark, we only have to prove that $\texttt{Reach}(\mathcal{N}^U) \subseteq \texttt{Untime}(\texttt{Reach}(\mathcal{N})_I)$. Therefore, we first state the following property expressing that if we reduce the intervals associated with transitions, this restricts the set of reachable configurations:

Lemma 6. *Let $\mathcal{N}$, $\mathcal{N}'$ be two TPN identical except on their last component associating intervals to transitions, say respectively I and I'. If we have $I'(t) \subseteq I(t)$ for any $t \in T$, then* $\texttt{Reach}(\mathcal{N}')_I \subseteq \texttt{Reach}(\mathcal{N})_I$.

In the sequel, we will consider TPN in which intervals are reduced to singletons. That is we have $I(t) = [\mathit{eft}(t), \mathit{lft}(t)]$ with $\mathit{eft}(t) = \mathit{lft}(t)$ for all transitions $t \in T$. The proof of the result in this particular case thus entails the result in the general case. Before to proceed we introduce additional definitions for TPN.

Given a TPN $\mathcal{N}$, a marking M of $\mathcal{N}$ and Δ a multiset of transitions of $\mathcal{N}$, we define the set $\mathit{Candidate}(M, \Delta) = \{t \in \Delta \mid M \overset{t}{\Rightarrow} M' \overset{\Delta\setminus t}{\Longmapsto}\}$. We will then say that a configuration (M, ν) is *compatible* with a multiset Δ iff:

$$M \overset{\Delta}{\Longmapsto} \text{ and } \forall t \in \mathit{Candidate}(M, \Delta), \nu(t) \leq \mathit{lft}(t).$$

We now prove the following proposition, which intuitively states how to turn a sequence of transitions in the untimed Petri net into a timed execution of the TPN.

Proposition 7. *Let $\mathcal{N}$ be a TPN with singleton intervals and (M, ν) be a configuration of $\mathcal{N}$ compatible with some multiset of transitions Δ. Then, for any transition $t \in$ Candidate(M, Δ) such that $\delta(t) = lft(t) - \nu(t)$ is minimal (among the transitions of Candidate(M, Δ)), we have:*

(i) $(M, \nu) \xrightarrow{\delta(t)}_I (M, \nu + \delta(t)) \xrightarrow{t}_I (M', \nu')$,
(ii) (M', ν') *is compatible with* $\Delta' = \Delta \setminus t$,

Proof. Let $t \in$ *Candidate*(M, Δ) be such that for all $t' \in$ *Candidate*(M, Δ), we have $lft(t) - \nu(t) = \delta(t) \leq \delta(t') = lft(t') - \nu(t')$.

(i) First the time elpasing transition $(M, \nu) \xrightarrow{\delta(t)}_I (M, \nu + \delta(t))$ is possible as we consider the weak semantics. Second, the discrete transition $(M, \nu + \delta(t)) \xrightarrow{t}_I (M', \nu')$ is also possible as $\nu(t) + \delta(t) = lft(t)$ by definition of $\delta(t)$, and since the intervals associated with transitions are all singletons.

(ii) To prove compatibility, first note that $M' \overset{\Delta'}{\models\!\!\Longrightarrow}$ because $t \in$ *Candidate*(M, Δ). Second, let $t' \in$ *Candidate*(M', Δ'). We distinguish two cases according to the value of the predicate $\uparrow$*enabled*$_I(t, M, t')$:
- If $\uparrow$*enabled*$_I(t, M, t')$ is true, then we have $\nu'(t') = 0$ and the result follows.
- Otherwise, the definition of $\uparrow$*enabled*$_I(t, M, t')$ implies that $M \geq {}^\bullet t + {}^\bullet t'$. As a consequence, we have $M \overset{t'}{\Rightarrow}\overset{t}{\Rightarrow}$. Then as $t' \in$ *Candidate*$(M', \Delta \setminus t)$ we get that $t' \in$ *Candidate*(M, Δ). Due to the minimality of $\delta(t)$ among the set *Candidate*(M, Δ), we obtain $\nu'(t') = \nu(t') + \delta(t) \leq \nu(t') + \delta(t') = lft(t')$ as desired.

This concludes the proof. $\square$

The inclusion $\texttt{Reach}(\mathcal{N}^U) \subseteq \texttt{Untime}(\texttt{Reach}(\mathcal{N})_I)$ in the case of TPN with singleton intervals easily follows from this result. Indeed, consider some reachable marking M in $\texttt{Reach}(\mathcal{N}^U)$. There exists a sequence of transitions that leads to M from M_0, we represent it through some multiset Δ. As initially all clock valuations are null in $[\![\mathcal{N}]\!]_I$, the configuration $(M_0, \mathbf{0})$ is thus compatible with Δ. An induction on the size of Δ, together with Proposition 7, thus gives the result. Note that Proposition 7 describes an effective procedure to compute a timed execution of $[\![\mathcal{N}]\!]_I$: simply consider the transitions that are candidates, and choose one with the earliest deadline.

Using the decidability results in the case of PN, we obtain the following corollary:

Corollary 8. *The marking reachability, marking coverability and boundedness problems are decidable in the case of TPN with weak intermediate semantics.*

3.3 Undecidability for Weak Atomic and Weak Persistent Atomic Semantics

We consider now the case of the weak atomic and weak persistent atomic semantics. As for the strong semantics, but with a more involved construction, we will show that it is possible to encode the behavior of a Minsky machine into a TPN with weak (persistent) atomic semantics from which we will deduce the undecidability results. The TPN we build contains a place for each counter c_i with $i \in \{1, 2\}$ and a place for each label q of the considered Minsky machine. Furthermore, when executing the net, we will preserve the invariant that there is a single place corresponding to a label q which is marked.

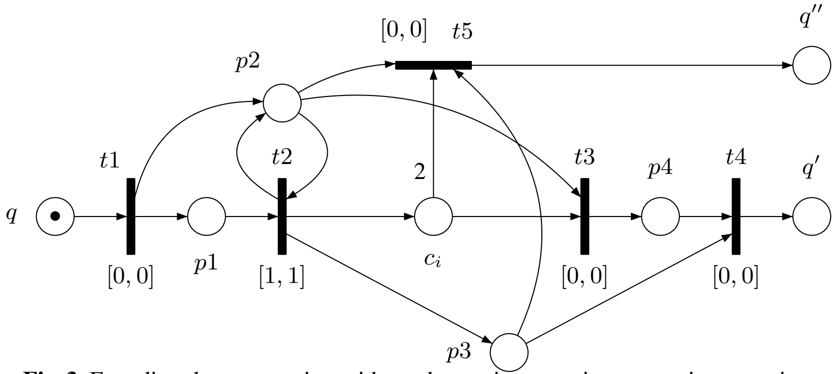


Fig. 3. Encoding decrementation with weak atomic or persistent atomic semantics

Encoding an incrementation can be done as in the strong semantics. Figure 3 shows how to encode the instruction (q : if $c_i = 0$ goto q' else $c_i := c_i - 1$; goto q'') using a TPN with weak atomic or persistent atomic semantics. We now explain the idea of this encoding. We consider the two following cases for the net shown in Figure 3:

1. Assume that the only place which contains a token is the place q, which means we are in the case where the value of c_i is equal to 0 (no token in place c_i). The net then can only fire the sequence of transitions $t1$, $t2$, $t3$ and then $t4$ and finally it reaches a configuration where the only marked place is q'.
2. Assume now that there is a token in place q and that there is at least one token in place c_i. We are in the case where the value of c_i is different of 0. We have the following sequence of transitions:
   – only the transition $t1$ is firable, so the net fires it;
   – afterwards the transition $t2$ and the transition $t3$ are firable. In fact, since we are considering weak semantics the deadline of $t3$ can be ignored thus making time passage in order to fire $t2$. Note that if the net chooses to fire $t3$, it will reach a deadlock state where no more transitions can be fired without having put a token in the place q' or q'', therefore we assume that the transition $t2$ is first fired;
   – after having waiting one time unit and firing $t2$, the only transition which can be fired is $t5$. In fact since we are considering atomic (or persistent atomic) semantics, firing $t2$ does not make $t3$ newly enabled, whereas the weak intermediate semantics would have reset the clock associated to $t3$. So the net fires $t5$ consuming the token in $p2$, $p3$ and two tokens in c_i (at least one was present from the initial configuration and the first firing of $t2$ added another one);
   – finally the net ends in a configuration with one token in q'' and the place c_i contains one token less than in the initial configuration.

The above construction allows to reduce the halting problem for Minsky machine to the marking coverability problem for weak (persistent) atomic semantics. From this we can also deduce the undecidability for the marking reachability and boundedness problems. Hence:

Theorem 9. *The marking reachability, marking coverability and boundedness problems are undecidable for TPN with weak atomic or weak persistent atomic semantics.*

In comparison with what occurs in the case of the strong semantics, this result is surprising, and it reveals the important role played by the memory policy when considering the weak semantics. Recall that as we have seen earlier, with the strong semantics, these problems are undecidable no matter which memory policy is chosen.

Finally, in the above construction, we can replace the edges between $p2$ and $t2$ by a read arc. Consequently, the considered problems are also undecidable for weak intermediate TPN with read arcs, unlike what happens for timed-arc Petri nets [6].

4 Expressiveness

4.1 Preliminaries

Let $S = (Q, q_0, \Sigma_\tau, \rightarrow)$ be a TTS. We define the relation $\hookrightarrow \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$ by:

- for $d \in \mathbb{R}_{\geq 0}$, $q \xhookrightarrow{d} q'$ iff there is a run ρ in S such that $\rho = q \xrightarrow{w} q'$ and $Untimed(w) = \varepsilon$ and $Duration(w) = d$,
- for $a \in \Sigma$, $q \xhookrightarrow{a} q'$ iff there is a run ρ in S such that $\rho = q \xrightarrow{w} q'$ and $Untimed(w) = a$ and $Duration(w) = 0$.

This allows us to define the following notion:

Definition 10 (Weak Timed Bisimulation). *Let $S_1 = (Q_1, q_0^1, \Sigma_\tau, \rightarrow_1)$ and $S_2 = (Q_2, q_0^2, \Sigma_\tau, \rightarrow_2)$ be two TTS and $\sim$ be a binary relation over $Q_1 \times Q_2$. $\sim$ is a weak timed bisimulation between S_1 and S_2 if and only if:*

- *$q_0^1 \sim q_0^2$, and,*
- *for $a \in \Sigma \cup \mathbb{R}_{\geq 0}$, if $q_1 \xhookrightarrow{a}_1 q_1'$ and if $q_1 \sim q_2$ then there exists $q_2' \in Q_2$ such that $q_2 \xhookrightarrow{a}_2 q_2'$ and $q_1' \sim q_2'$; conversely if $q_2 \xhookrightarrow{a}_2 q_2'$ and if $q_1 \sim q_2$ then there exists $q_1' \in Q_1$ such that $q_1 \xhookrightarrow{a}_1 q_1'$ and $q_1' \sim q_2'$.*

Two TTS S_1 and S_2 are *weak timed bisimilar* if there exists a weak timed bisimulation between S_1 and S_2. We then write $S_1 \approx S_2$.

Definition 11 (Expressiveness w.r.t. Weak Timed Bisimilarity). *The class $\mathcal{C}$ of TTS is less expressive than $\mathcal{C}'$ w.r.t. weak timed bisimilarity if for all TTS $S \in \mathcal{C}$ there is a TTS $S' \in \mathcal{C}'$ such that $S \approx S'$. We write $\mathcal{C} \sqsubseteq \mathcal{C}'$. If moreover there is a $S' \in \mathcal{C}'$ such that there is no $S \in \mathcal{C}$ with $S \approx S'$, then $\mathcal{C}$ is strictly less expressive than $\mathcal{C}'$. We then write $\mathcal{C} \sqsubset \mathcal{C}'$.*

For $s \in \{I, A, PA\}$, we will denote by $\mathcal{TPN}_s$ the class of TTS induced by TPN with s-semantics.

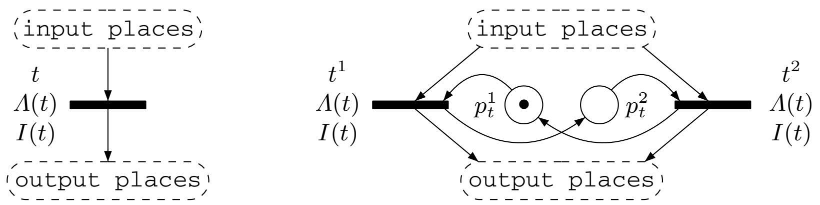


Fig. 4. From atomic to persistent atomic semantics

4.2 Atomic versus Persistent Atomic semantics

In [4], the authors prove that for TPN with strong semantics, the persistent atomic semantics is more expressive than the atomic semantics. We prove here that this result still holds in the case of the weak semantics. Intuitively, as it is shown on Figure 4, from a TPN with atomic semantics, we build another TPN in which we duplicate each transition. During an execution of this last TPN, at most one of the transitions obtained after duplication is enabled, and when it is fired it cannot be enabled again at the next step whereas the other one can. This trick allows us to simulate the atomic semantics with the persistent atomic one.

Proposition 12. *For all TPN $\mathcal{N}$, we can build a TPN $\mathcal{N}'$ such that $[\![\mathcal{N}]\!]_A \approx [\![\mathcal{N}']\!]_{PA}$.*

Proof. Let $\mathcal{N} = (P, T, \Sigma_\tau, {}^\bullet(.), (.)^\bullet, M_0, \Lambda, I)$ be a TPN over Σ_τ. Figure 4 represents the construction of the TPN $\mathcal{N}'$. Formally, its set of places P' is equal to $P \cup \{p_t^1, p_t^2 \mid t \in T\}$ and its set of transitions T' contains two copies t^1 and t^2 of each transition $t \in T$. These copies are connected in the same way as the transition t is in $\mathcal{N}$, plus additional edges to the places p_t^1 and p_t^2, as depicted on Figure 4. Finally the initial marking of $\mathcal{N}'$ is M_0' such that for all $p \in P$, $M_0'(p) = M_0(p)$ and for all $t \in T$, $M_0'(p_t^1) = 1$ and $M_0'(p_t^2) = 0$.

We now consider the relation $\sim \subseteq (\mathbb{N}^P \times \mathbb{R}^T_{\geq 0}) \times (\mathbb{N}^{P'} \times \mathbb{R}^{T'}_{\geq 0})$ between the configurations of $[\![\mathcal{N}]\!]_A$ and the ones of $[\![\mathcal{N}']\!]_{PA}$ defined by $(M, \nu) \sim (M', \nu')$ iff:

- for all $p \in P$, $M(p) = M'(p)$ and for all $t \in T$, $M'(p_t^1) + M'(p_t^2) = 1$,
- for all $t \in T$, for all $i \in \{1, 2\}$ if $t \in \mathit{En}(M)$ and $t^i \in \mathit{En}(M')$ then $\nu(t) = \nu'(t^i)$.

It is then easy to verify that the relation $\sim$ is a weak timed bisimulation. □

We will now prove that the inclusion we obtain in the above proposition is strict. But before, we address a technical point which we will use to delay some sequences of transitions in weak TPN.

Lemma 13. *Let $s \in \{I, A, PA\}$ and consider a TPN $\mathcal{N}$ such that b is the smallest positive upper bound of the intervals of $\mathcal{N}$. Let ρ be a run in $[\![\mathcal{N}]\!]_s$ of the form ρ : $(M, \nu) \xrightarrow{\delta>0}_s (M, \nu + \delta) \xrightarrow{t_1}_s \cdots \xrightarrow{t_n}_s$, such that there exists a value $\tau \geq 0$ verifying:*

(i) $\forall i \in \{1, \ldots, n\}$, $t_i \in \mathit{En}(M) \Rightarrow \nu(t_i) \leq \eta$,
(ii) $\eta + \delta < \frac{b}{2}$

Then the sequence $\rho' : (M, \nu) \xrightarrow{\delta + \frac{b}{2}}_s (M, \nu') \xrightarrow{t_1}_s \cdots \xrightarrow{t_n}_s$ is firable in $[\![\mathcal{N}]\!]_s$.

We now consider the TPN $\mathcal{N}_2$ represented on Figure 5. Equipped with persistent atomic semantics, it accepts the set of timed words composed of letters a occurring before time 1. We will prove that this timed language cannot be accepted by any TPN equipped with the weak atomic semantics.

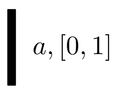


Fig. 5. The TPN $\mathcal{N}_2$

Proposition 14. *There exists no TPN $\mathcal{N}$ (even unbounded) s.t. $[\![\mathcal{N}]\!]_A \approx [\![\mathcal{N}_2]\!]_{PA}$.*

Proof. Assume there exists a TPN $\mathcal{N}$ such that $[\![\mathcal{N}]\!]_A \approx [\![\mathcal{N}_2]\!]_{PA}$. Denote by N the number of transitions of $\mathcal{N}$, by b the smallest positive upper bound of the intervals of $\mathcal{N}$, and consider a timed word $w = (a, \eta_1)(a, \eta_2) \dots (a, \eta_k)$ such that $\forall i, 1 - \frac{b}{2} < \eta_i < \eta_{i+1} < 1$, and $k \geq N + 1$.

This timed word w is recognized by $[\![\mathcal{N}_1]\!]_{PA}$ and there exists thus a run of $[\![\mathcal{N}]\!]_A$ along w. We denote it by ρ and decompose it as follows :

$$\rho : \xrightarrow{\theta_0}_A \xrightarrow{d_1}_A \xrightarrow{\theta_1}_A \xrightarrow{t_a^1}_A \xrightarrow{\theta_1'}_A \cdots \xrightarrow{d_i}_A \xrightarrow{\theta_i}_A \xrightarrow{t_a^i}_A \xrightarrow{\theta_i'}_A \cdots \xrightarrow{d_k}_A \xrightarrow{\theta_k}_A \xrightarrow{t_a^k}_A$$

To obtain this decomposition we proceed as follows. We denote by t_a^i the i-th transition labelled by a. Then for each position i, we isolate the last delay step occuring before the transition t_a^i (it exists since $\eta_i > \eta_{i-1}$) and denote it by d_i. Then we gather all the internal transitions occuring between this delay step and the transition t_a^i, and denote this sequence by θ_i. The transitions between t_a^{i-1} and the delay step constitute the sequence θ_{i-1}'. In particular, the following properties hold for any position i : $\Lambda(t_a^i) = a$, *Untimed*$(\theta_i) =$ *Untimed*$(\theta_i') = \varepsilon$, $d_i > 0$, *Duration*$(\theta_i) = 0$, and t_a^i occurs at time η_i.

We claim there exists an index $i \in \{1, \dots, k\}$ such that each transition t appearing in $\theta_i \; t_a^i$ has already been fired since θ_0, i.e. t also appears in $\theta_1 \; t_a^1 \; \theta_1' \cdots \theta_{i-1} \; t_a^{i-1} \; \theta_{i-1}'$. By contradiction, if it is not the case, then we can find, for each index $i \in \{1, \dots, k\}$, a transition, denoted t_i, that never appears before. The choice of k verifying $k \geq N + 1$ then implies that there exist two positions $i \neq j$ such that $t_i = t_j$, thus yielding a contradiction. We can now fix an index i verifying the above described property.

We now show that Lemma 13 can be applied to the part of ρ associated with the sequence $d_i \; \theta_i \; t_a^i$. More precisely, (M, ν) is the configuration reached after firing $\theta_0 \cdots t_a^{i-1} \; \theta_{i-1}'$, the delay δ is equal to d_i, the sequence $t_1 \cdots t_n$ corresponds to $\theta_i \; t_a^i$, and η is defined as $(\eta_i - d_i) - (1 - \frac{b}{2})$. In the atomic semantics, when a transition is fired, its clock is reset if it is still enabled. This property allows, together with timing constraints on the word w, to verify hypotheses (i) and (ii) of the Lemma 13. Indeed, since each transition in $\theta_i \; t_a^i$ has been reset along $\theta_1 \; t_a^1 \; \theta_1' \cdots \theta_{i-1}'$, it has been reset since time η_1. Since the global time associated with (M, ν) is equal to $\eta_i - d_i$, these valuations are bounded by above by the value $(\eta_i - d_i) - \tau_1 \leq (\eta_i - d_i) - (1 - \frac{b}{2}) = \eta$. Second, we have $\eta + \delta = \eta_i - (1 - \frac{b}{2}) < \frac{b}{2}$, as desired (this follows from the inequalities $1 - \frac{b}{2} < \eta_i < 1$).

Finally, Lemma 13 thus allows to delay of $\frac{b}{2}$ the firing of the sequence $\theta_i \; t_a^i$. In particular, this will produce a letter a at time $\eta_i + \frac{b}{2} > 1$. The TTS $[\![\mathcal{N}]\!]_A$ thus accepts a timed word not recognized by $[\![\mathcal{N}_2]\!]_{PA}$, providing a contradiction. □

Using the results of Propositions 12 and 14, we deduce that:

Theorem 15. $\mathcal{TPN}_A \sqsubset \mathcal{TPN}_{PA}$.

4.3 About Atomic and Intermediate Policies in Weak and Strong Semantics

In this subsection, we discuss the comparison of the intermediate and atomic policies. As we will see, the situation is more complex than in the previous comparison.

On the inclusion of $\mathcal{TPN}_I$ into $\mathcal{TPN}_A$. For the **strong semantics**, a construction has been proposed in [4] to transform any TPN with intermediate policy into an equivalent (w.r.t. weak timed bisimilarity) TPN with atomic semantics. A first attempt was thus to adapt this construction for the weak semantics. But studying this construction, we noticed that it is erroneous (even for the strong semantics). We present below an example exhibiting the error.

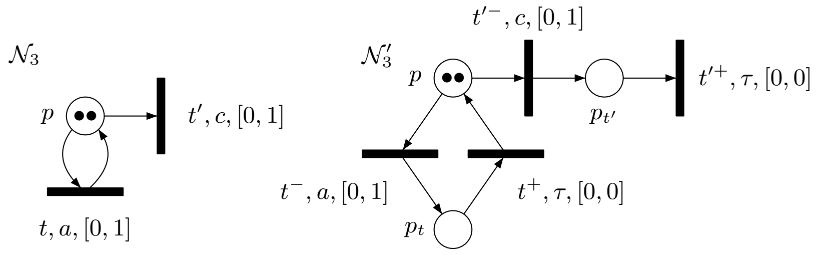


Fig. 6. A counter example to the construction of [4].

Example 2. Consider the net $\mathcal{N}_3$ depicted on the left of Figure 6. The application of the construction proposed in [4] leads to the net $\mathcal{N}_3'$ depicted on the right of Figure 6. According to [4], we should have, under the **strong semantics**, the relation $[\![\mathcal{N}_3]\!]_I \approx [\![\mathcal{N}_3']\!]_A$. However, it is easy to verify that in the TTS $[\![\mathcal{N}_3']\!]_A$ the letter c can be read after 2 times units (with the timed word $(a,1)(a,1)(\tau,1)(\tau,1)(c,2)$) whereas it is not possible in $[\![\mathcal{N}_3]\!]_I$, thus proving that the construction proposed in [4] is erroneous.

This example leaves open the question of the inclusion of $\mathcal{TPN}_I$ into $\mathcal{TPN}_A$ for the strong semantics, and then for this semantics both inclusions are left open. For weak semantics, this inclusion is also open, but we show below that the converse inclusion is false.

Non inclusion of $\mathcal{TPN}_A$ into $\mathcal{TPN}_I$. We exhibit a TPN with atomic semantics which cannot be expressed in an equivalent way by any TPN with intermediate semantics (with weak elapsing of time). This is formally stated in the Proposition below. We consider the TPN $\mathcal{N}_1$ represented on Figure 1. Interpreted in weak atomic semantics, the firing of the a-labelled transition does not newly enable transition labelled by c. This transition thus shares a token with transition a while preserving a time reference to the origin of global time, what is impossible in intermediate semantics.

Proposition 16. *There exists no TPN $\mathcal{N}$ (even unbounded) such that $[\![\mathcal{N}]\!]_I \approx [\![\mathcal{N}_1]\!]_A$.*

Proof. We only present a sketch of the proof (details can be found in the [16]). We proceed by contradiction and assume there exists such a TPN $\mathcal{N}$, and denote by N its number of transitions, and b the smallest positive upper bound of its intervals. As in the proof of Proposition 14, we first exhibit a particular execution ρ of $[\![\mathcal{N}]\!]_I$:

Lemma 17. *Let $(\eta_i)_{1 \leq i \leq k}$ be a set of timestamps such that for any $1 \leq i \leq k$, $1 - \frac{b}{2} < \eta_i < \eta_{i+1} < 1$ and $k \geq N + 1$. There exists a run ρ in $[\![\mathcal{N}]\!]_I$ of the following form:*

$$\rho : \xrightarrow{1-\frac{b}{2}}_I \xrightarrow{\theta_1}_I \xrightarrow{d_1}_I \xrightarrow{\theta'_1}_I \xrightarrow{t^1_a}_I \xrightarrow{\theta''_1}_I \cdots \xrightarrow{\theta_i}_I \xrightarrow{d_i}_I \xrightarrow{\theta'_i}_I \xrightarrow{t^i_a}_I \xrightarrow{\theta''_i}_I \cdots \xrightarrow{\theta_n}_I \xrightarrow{d_n}_I \xrightarrow{\theta'_n}_I \xrightarrow{t^n_a}_I \xrightarrow{\theta''_n}_I$$

such that for any position i, $\Lambda(t^i_a) = a$, the transition t^i_a occurs at time η_i, $d_i > 0$, Untimed(θ_i) = Untimed(θ'_i) = Untimed$(\theta''_i) = \varepsilon$, Duration$(\theta'_i)$ = Duration$(\theta''_i) = 0$, and there exists a transition t^i_c, labelled by c, newly enabled by the last transition of $t^i_a\ \theta''_i$ and (immediately) firable from the configuration reached after θ''_i.

To conclude we use a reasonning very similar to the one done in the proof of Proposition 14. In fact, applying Lemma 13, it is possible to delay by $\frac{b}{2}$ time units the firing of a subsequence $d_i\ \theta'_i\ t^i_a\ \theta''_i$ and since t^i_c is newly enabled by the last transition of $t^i_a\ \theta''_i$, we thus obtain a run in $[\![\mathcal{N}]\!]_I$ with a c action following an a action occuring after time 1, which is impossible in $[\![\mathcal{N}_1]\!]_A$, thus yielding a contradiction. □

5 Conclusion

We have studied in this paper the model of Time Petri Nets under a weak semantics of time elapsing, allowing any delay transition. We have first proven that for the intermediate memory policy, the set of reachable markings coincides with the reachability set of the underlying untimed Petri net. As a consequence, many verification problems are decidable for weak intermediate TPN. On the other hand, we have proven that the two other memory policies, namely atomic and persistent atomic, allow to simulate Minsky machines and thus are undecidable. Finally, we have studied expressiveness and have proven that (i) the atomic semantics is strictly less expressive than the persistent atomic one and (ii) the atomic semantics is not included in the intermediate one.

In further work, we plan to investigate properties concerning executions of weak intermediate TPN; such as time-optimal reachability, or LTL model checking. Indeed, while discrete markings are the same, the executions are different from those accepted by the underlying Petri net. Concerning expressiveness, we conjecture that intermediate and atomic semantics are uncomparable in general, and that bounded weak TPN are strictly less expressive than timed automata (without invariants).

Aknowledgments. We would like to thank Fabrice Chevalier for fruitful discussions and the anonymous reviewers for their insightful comments.

References

1. Abdulla, P.A., Mahata, P., Mayr, R.: Dense-timed Petri nets: Checking zenoness, token liveness and boundedness. Logical Methods in Computer Science 3(1), 1–61 (2007)
2. Abdulla, P.A., Nylén, A.: Timed Petri nets and bqos. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 53–70. Springer, Heidelberg (2001)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)

4. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: Comparison of different semantics for time Petri nets. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 293–307. Springer, Heidelberg (2005)
5. Bolognesi, T., Lucidi, F., Trigila, S.: From timed Petri nets to timed LOTOS. In: PSTV 1990, pp. 395–408. North-Holland, Amsterdam (1990)
6. Bouyer, P., Haddad, S., Reynier, P.-A.: Timed Petri nets and timed automata: On the discriminating power of zeno sequences. Information and Computation 206(1), 73–107 (2008)
7. Boyer, M., Roux, O.H.: On the compared expressiveness of arc, place and transition time Petri nets. Fundamenta Informaticae 88(3), 225–249 (2008)
8. de Frutos-Escrig, D., Valero Ruiz, V., Alonso, O.M.: Decidability of properties of timed-arc Petri nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 187–206. Springer, Heidelberg (2000)
9. Haar, S., Simonot-Lion, F., Kaiser, L., Toussaint, J.: Equivalence of timed state machines and safe time Petri nets. In: WoDES 2002, pp. 119–126 (2002)
10. Jones, N.D., Landweber, L.H., Lien, Y.E.: Complexity of some problems in Petri nets. Theoretical Computer Science 4(3), 277–299 (1977)
11. Karp, R.M., Miller, R.E.: Parallel program schemata. Journal of Computer System Sciences 3(2), 147–195 (1969)
12. Kosaraju, S.R.: Decidability of reachability in vector addition systems (preliminary version). In: STOC 1982, pp. 267–281. ACM, New York (1982)
13. Mayr, E.W.: An algorithm for the general Petri net reachability problem. SIAM Journal on Computing 13(3), 441–460 (1984)
14. Merlin, P.M.: A Study of the Recoverability of Computing Systems. PhD thesis, University of California, Irvine, CA, USA (1974)
15. Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall, Inc., Upper Saddle River (1967)
16. Reynier, P.-A., Sangnier, A.: Weak Time Petri Nets strike back! Research Report HAL-00374482, HAL, CNRS, France (2009)
17. Sifakis, J., Yovine, S.: Compositional specification of timed systems. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 347–359. Springer, Heidelberg (1996)
18. Valero Ruiz, V., de Frutos-Escrig, D., Cuartero, F.: On non-decidability of reachability for timed-arc Petri nets. In: PNPM 1999, pp. 188–196. IEEE Computer Society Press, Los Alamitos (1999)

A General Testability Theory*

Ismael Rodríguez

Universidad Complutense de Madrid, 28040 Madrid, Spain
isrodrig@sip.ucm.es

Abstract. We present a general framework allowing to classify testing problems into five *testability* classes. Classes differ in the number of tests we must apply to precisely determine whether the system is correct or not. The conditions that enable/disable finite testability are analyzed. A general method to reduce a testing problem into another is presented. The complexity of finding complete test suites and measuring the suitability of incomplete suites is analyzed.

1 Introduction

Testing consists in checking the correctness of a system by interacting with it. Typically, the goal of this interaction is checking whether an implementation fulfills a given property or specification. If the specification is formally defined then procedures for deriving tests, applying tests, and assessing the outputs collected by tests can be formal and systematic [10,13,2,15]. There exist myriads of formal testing methodologies, each one focusing on checking the correctness of a different kind of system (e.g., *labeled transition systems* [18,4], *temporal systems* [16,12], *probabilistic systems* [17,11], *Java programs* [3], etc). Some methods focus on testing a *part* of the behavior considered critical. Other methods aim at constructing *complete* test suites, that is, sets of tests such that, after applying them to the implementation, the results allow to precisely determine whether the implementation is correct or not with respect to the specification. Let us note that constructing and applying complete test suites is often unfeasible. For example, checking the correctness of a non-deterministic machine could be impossible regardless of how many tests one applies, because a given behavior could remain hidden for any arbitrarily long time. Even if a machine is deterministic, we could need to apply infinite tests if the number of available ways to interact with the implementation is infinite. In some cases where infinite tests are required, it could be the case that we can achieve any arbitrarily high degree of *partial* completeness with some finite test suite, thus enabling a kind of *unboundedly-approachable* completeness, rather than completeness. Alternatively, if the behavior of the system depends on temporal conditions and the time is assumed to be continuous, then we could need to check what happens when a given input is produced at *all* possible times, thus requiring an uncountable infinite set of tests.

* Work supported by projects TIN2006-15578-C02-01, CCG08-UCM/TIC-4124, and the UCM-BSCH programme (GR58/08 - group number 910606).

Since the feasibility of a testing method depends on our capability to test systems in finite time, investigating the conditions that enable/disable the existence of finite complete test suites in each case is a major issue in testing. In this regard, several works have studied the effect of assuming some hypotheses about the implementation [7,8,15]. By assuming hypotheses, the number of systems that could actually *be* the implementation is reduced (provided that assumed hypotheses actually hold), so less tests must be applied to seek for undesirable behaviors. Actually, this may allow to reduce the number of necessary tests from infinite to finite [1,5]. However, the specific conditions that make the *difference* between requiring infinite or finite sets of tests (regardless of the use of hypotheses) must be studied. In this paper, by *testability* we will understand the difficulty to test systems, measured in terms of the size required by test suites to be complete. We think that the conditions leading to the testability cases commented before (that is, complete test suites are (a) finite; (b) infinite, but any arbitrarily high degree of partial completeness can be finitely achieved; (c) countable infinite, but not as in (b); (d) uncountable infinite; or (e) there does not exist any complete test suite at all) must be analyzed. To the best of our knowledge, the previous five general testability scenarios have never been defined or investigated. In particular, the border between them has never been studied as a general issue, that is, by means of a framework where any kind of formalism for defining specifications and implementations fits (only some particular hypotheses enabling finite complete test suites for some particular models have been reported). This contrasts with other mature fields of Computer Science like Computability or Complexity, where a well established theory allows to relate different known problems with each other, as well as to classify them into a known hierarchy. Unfortunately, the lack of such formal roots makes the field of Formal Testing Techniques a bit *disorganized* because techniques are not easily inherited from a problem to another – even if they are quite similar after abstracting factors not directly affecting testability.

We propose a first step towards the construction of a general testability theory. We present some formal criteria to classify testing problems according to their testability. Providing a complete testability hierarchy is a huge task and is out of the goals of the paper. Instead, a hierarchy including only five main classes of testability (*finitely testable*, *unboundedly-approachable finitely testable*, *infinite countable testable*, *infinite uncountable testable*, and *untestable*) is presented. Some formal properties of the *finitely testable* class are presented, such as conditions required for finite testability, alternative characterizations, transformations keeping the testability, the effect of adding testing hypotheses, methods to *reduce* a testing problem into another, the complexity of finding a minimum complete test suite, and the complexity of measuring the completeness degree of *incomplete* test suites. We apply these properties to study the testability of some examples, both well-known and ad hoc. A deeper study of the properties of the remaining four testability classes is left as future work.

The contribution of this paper is twofold. From a theoretical point of view, the proposed techniques allow to classify and relate testing problems with each other, thus making a first step towards constructing a complete testability hierarchy and improving our understanding about testing. From a practical point of view, being able to reason about the most *ideal* testing scenario (that is, the case where completeness is feasible) allows to deal with more practical scenarios. Though test suites are incomplete in most of practical cases, identifying the additional conditions required by an incomplete test suite to *become* complete allows to compare and select good (incomplete) test suites: We prefer those test suites such that the conditions required by them to be complete are *weaker* or *more feasible*. Besides, if testing problems can be related then strategies allowing to find good test suites for a given kind of systems can be exported to find good test suites in other testing scenarios. Moreover, testability issues could affect the design of systems: If there are several suitable alternatives for a design and, for one of them, the problem of testing the corresponding system would belong to a better class (or incomplete test suites would be *closer* to be complete), then this alternative is preferable. The structure of this paper is direct. All proofs, as well as some auxiliary results and secondary notions, are presented in [14].

2 Testability Concepts

In this section we introduce some preliminary concepts and we define testability classes. First, we present a general notion to denote implementations and specifications in our framework. Since testing consists in studying systems in terms of their observable behavior, the behavior of a system can be defined by a function relating inputs with their possible outputs. Next we assume that 2^S denotes the powerset of the set S.

Definition 1. Let I be a set of *input symbols* and O be a set of *output symbols*. A *computation formalism* C for I and O is a set of functions $f : I \rightarrow 2^O$ where for all $i \in I$ we have $f(i) \neq \emptyset$. □

Given a function $f \in C$, $f(i)$ represents the set of outputs we can obtain after applying input $i \in I$ to the computation artifact represented by f. Since $f(i)$ is a set, f may represent a non-deterministic behavior. Besides, C, I, and O can be infinite sets. Representing the behavior of systems by means of functions avoids to implicitly impose a *specific* structure to system models (e.g., states, transitions). Still, elaborated behaviors can be represented. We illustrate this.

Example 1. Let C be a computation formalism representing the set of all (possibly non-deterministic) Mealy machines (also known as *finite state machines*, FSMs). Let M be an FSM and I' and O' be the set of inputs and the set of outputs of M, respectively. M is represented in C by a function $f \in C$ such that we have $f(\sigma) = \{\sigma'_1, \ldots, \sigma'_n\}$ if and only if $\{\sigma'_1, \ldots, \sigma'_n\}$ is the set of sequences of O' outputs that can be answered by M when it receives the sequence σ of I' inputs. For instance, if $\sigma = a \cdot b$, $\sigma' = x \cdot y$, and $\sigma'' = w \cdot z$, then $f(\sigma) = \{\sigma', \sigma''\}$ means that f represents an FSM M where, in particular, if $a \cdot b$ is given then the machine can answer either $x \cdot y$ or $w \cdot z$. Hence, the set I considered in Definition 1 (respectively, the set O) is the set of all *sequences* of symbols belonging to I' (resp. to O'), that is, $I = I'^*$ and $O = O'^*$.[1] Thus, even if I' and O' are finite, I and O are infinite sets. Alternatively, if systems were assumed to be *deterministic* FSMs, then all functions representing a non-deterministic FSM would be removed from C. Other restrictions over the set of systems that are represented by C could be considered (e.g. considering only FSMs with less than n states, etc).

Now, let C represent the set of all programs in a Turing-complete language, e.g. Java. We codify the interaction of a user with a program by using an appropriate notation. For example, the behavior of a given program P could be denoted by a function f such that, in particular, we have

$$f(\langle(0.5, but_1), (1.25, but_2)\rangle) = \left\{\begin{array}{l}\langle(0.75, mes_1), (1.5, mes_2), (2, stop)\rangle,\\ \langle(0.75, mes_1), (1.5, mes_3), (2, mes_4), (2.5, stop)\rangle,\\ \langle(0.75, mes_1), \perp\rangle\end{array}\right\}$$

meaning that if the user presses button 1 at time 0.5 and button 2 at time 1.25, then she will receive message 1 at time 0.75 for sure, and next the program non-deterministically chooses one of the following choices: (a) answering message 2 at time 1.5 and stopping at time 2; (b) answering message 3 at time 1.5, giving message 4 at time 2, and then finishing at time 2.5; or (c) not terminating (denoted by the $\perp$ symbol; the effect of non-terminating behaviors on testing will be discussed later). For example, we have $\langle(0.5, but_1), (1.25, but_2)\rangle \in I$ and $\langle(0.75, mes_1), (1.5, mes_2), (2, stop)\rangle \in O$. Alternatively, if temporal issues were not considered relevant for assessing the correctness of behaviors, then time stamps could be removed from the proposed representation, or they could be replaced by *ordering* stamps denoting the relative order of each O' output with respect to I' inputs. Additional details could denote other factors.

If two FSMs (resp., two Java programs) produce the same sets of outputs for all inputs then both machines are represented by the same function $f \in C$, since a function belonging to C represents a *relation* between inputs and outputs (but not the internal structure of the machine leading to this behavior). □

Computation formalisms will be used to represent the set of implementations we are considering in a given testing scenario. Implicitly, a computation formalism C represents a *fault model* (i.e. the definition of what can be wrong in the IUT) as well as the *hypotheses* about the IUT the tester is assuming. For instance, if the IUT is assumed to be a deterministic FSM and to differ from a given correct FSM in at most one transition, then only functions denoting the behaviors of such FSMs (including the correct one) are in C; alternatively, if all we assume is that the IUT is represented by a deterministic FSM, then C will represent all deterministic FSMs. Computation formalisms will also be used to represent the subset of specification-compliant implementations. Let C represent the set of possible implementations and $E \subseteq C$ represent the set of implementations fulfilling the specification. The goal of testing is interacting with the *implementation under test* (IUT) so that, according to the collected responses, we can

[1] When dealing with FSMs we will assume the same meaning for I', O', I, and O.

decide whether the IUT actually belongs to E or not. Typically, we apply some tests (i.e., some inputs $i_1, i_2, \ldots \in I$) to the IUT one after each other so that observed results $o_1 \in f(i_1)$, $o_2 \in f(i_2)$, $\ldots$ allow us to provide a verdict. Since all outputs are returned by the same function f, we are implicitly assuming that all input applications are *independent*, i.e. the result after applying i_1 does not affect the output observed next, when we apply i_2.[2] Alternatively, we may choose not to assume this independence. In this case, an interaction where we apply i_1 and next i_2 is *not* assumed to return o_1 and next o_2, for some $o_1 \in f(i_1)$ and $o_2 \in f(i_2)$. Instead, it returns o for some $o \in f(i')$, where i' is a *single* input representing an interaction where we apply i_1 and next we apply i_2. Hence, scenarios where the independence of input applications is not guaranteed can also be represented in the proposed formalism.

Definition 2. A *specification* of a computation formalism C is a set $E \subseteq C$. □

If $f \in E$ then f denotes a *correct* behavior, while $f \in C \backslash E$ denotes that f is incorrect. Thus, a specification implicitly denotes a correctness criterion. For example, let $f, f' \in C$ be such that for all i we have $f(i) = \{a\}$ and $f'(i) = \{b\}$. Then, $E = \{f, f'\}$ denotes that only machines producing *always* a or *always* b are considered correct. We can also construct E in such a way that a given semantic relation is considered (e.g., bisimulation, testing preorder, traces inclusion, conformance testing, etc). For instance, given some $f \in C$, let us consider that f is correct and we wish to be consistent with respect to a given semantic relation $\preceq$, where $A \preceq B$ means that B is correct with respect to A. Then we could define E as follows: $E = \{f' | f \preceq f' \ \wedge \ f' \in C\}$.

Testing provides verdicts in terms of the outputs collected by interacting with the IUT. In some situations, there might exist two different outputs that are not *distinguishable* in practice by means of observation. This is specially relevant if one of them is produced when the machine fulfills the specification and the other one is given when the machine is incorrect, because then the observed output does not allow to assess the correctness of the observed IUT.

Definition 3. A *distinguishing relation* for a set of outputs O is a non-reflexive symmetric binary relation $\mathcal{D}$ over O. Its complementary is denoted by $\not{\mathcal{D}}$. □

Example 2. We revisit the example where C represents FSMs. If after receiving some $i \in I$ the IUT can answer either $o_1 \in O$ or $o_2 \in O$ then, by observing the actual response of the IUT, we can decide which of these sequences is produced. Hence, we may consider a trivial distinguishing relation $\mathcal{D}$ where two outputs $o_1, o_2 \in O$ are distinguishable iff they represent different sequences of O' outputs, i.e. $o_1 \ \mathcal{D} \ o_2$ iff $o_1 \neq o_2$. This strong distinguishing capability may not be feasible in other frameworks. We revisit the example where C represents Java programs. Let us suppose that the time at which messages are produced is *not* represented, that is, outputs only represent sequences of messages. Let us consider outputs $o_1 = \langle mes_1, mes_2, stop \rangle$ and $o_2 = \langle mes_1, \perp \rangle$. In practice,

[2] In practice this is equivalent to assuming a *reliable reset*, a typical testing assumption.

o_1 is not distinguishable from o_2 via observation because, if mes_1 is produced, then we cannot guarantee that mes_2 will *not* be observed later, no matter how much time passes. Consequently, we may consider the following distinguishing relation: We have $o_1 \; \mathcal{D} \; o_2$ iff $o_1 \neq o_2$ *and* neither $o_1 = w \cdot \bot$ nor $o_2 = w \cdot \bot$, where w is the longest common prefix of o_1 and o_2.[3] It is worth to point out that, if outputs are given as in Example 1 (that is, a time stamp is attached to each O' output), then non-termination issues do not affect the distinguishability of outputs. Let us consider $o = \langle (t_1, o_1), \ldots, (t_n, o_n) \rangle \in O$. If time t_i is reached and o_i is not observed then we know for sure that we are not observing o. Thus, in this case we may use the following criterion: We have $o_1 \; \mathcal{D} \; o_2$ iff $o_1 \neq o_2$. □

Next we identify *complete test suites*, i.e. sets of inputs such that, if they are applied to the IUT, then collected outputs allow to precisely determine if the IUT fulfills the considered specification or not. More precisely, a complete test suite is a set of inputs such that, for all correct function f and incorrect function f', there is at least one input i in the set such that the sets of outputs that can be produced by f and f' in response to i are *pairwise distinguishable* (i.e. all outputs in one set are distinguishable from all outputs belonging to the other set). We also introduce three of the five classes of our testability hierarchy: `Class I`, `Class III`, and `Class V` (`Class II` will be defined later in this section, while `Class IV` is defined in [14]). Next we consider that if $\mathcal{I}$ is a set of inputs then `pairs` $(f,\mathcal{I})$ denotes the set of all pairs $(i, f(i))$ such that $i \in \mathcal{I}$.

Definition 4. Let C be a computation formalism for I and O, $E \subseteq C$ be a specification, $\mathcal{D}$ be a distinguishing relation, and $\mathcal{I} \subseteq I$ be a set of inputs.

We say that $f \in E$ and $f' \in C \backslash E$ are *distinguished* by $\mathcal{I}$, denoted by `di` $(f, f', \mathcal{I})$, if there exist $i \in \mathcal{I}$, $(i, outs) \in$ `pairs` $(f, \mathcal{I})$, and $(i, outs') \in$ `pairs` $(f', \mathcal{I})$ such that for all $o \in outs$ and $o' \in outs'$ we have $o \; \mathcal{D} \; o'$.

We say that $\mathcal{I}$ is a *complete test suite* for C, E, and $\mathcal{D}$ if for all $f \in E$ and $f' \in C \backslash E$ we have `di` $(f, f', \mathcal{I})$.

A triple $(C, E, \mathcal{D})$ is *finitely testable* if there exists a finite complete test suite for C, E, and $\mathcal{D}$. `Class I` denotes the set of all finitely testable triples $(C, E, \mathcal{D})$.

$(C, E, \mathcal{D})$ is *countable testable* if there exists a countable complete test suite for C, E, and $\mathcal{D}$. `Class III` is the set of all countable testable triples $(C, E, \mathcal{D})$.

`Class V` denotes the set of all triples $(C, E, \mathcal{D})$. □

This classification induces the relation `Class I` $\subseteq$ `Class III` $\subseteq$ `Class V`. Next we show some examples fitting into each of these testability classes. They show that all the inclusions considered in the previous relation are proper indeed.

Example 3. Let C_1 represent the set of all deterministic completely-specified FSMs with at most n states where the finite sets of inputs and outputs are I' and O', respectively. Let $E_1 \subseteq C_1$, $\mathcal{D}$ be the trivial distinguishing relation (i.e., $o_1 \; \mathcal{D} \; o_2$ iff $o_1 \neq o_2$), and $\mathcal{I}_1$ be the set of all sequences of I' symbols whose length

[3] Alternatively, a tester could assume a kind of *non-termination observability*, which is common for practical reasons. In that case, we could consider $o_1 \; \mathcal{D} \; o_2$ iff $o_1 \neq o_2$.

is at most $2n+1$. It is known that, if two FSMs M_1 and M_2 represented by C_1 produce different responses for some input sequence, then sequences answered by M_1 and M_2 are different for at least one sequence belonging to $\mathcal{I}_1$ [10]. Hence, for all $f \in E_1$ and $f' \in C_1 \backslash E_1$ there exists a sequence $i \in \mathcal{I}_1$ allowing to distinguish f and f'. Thus, $(C_1, E_1, \mathcal{D}) \in$ `Class I`. As we can see in Example 7 (given in [14]) this result can also be proved by applying Lemma 2 (c) (this lemma, given right before Example 7 in [14], makes this result straightforward and avoids the necessity of identifying any specific complete test suite $\mathcal{I}_1$).

Let us remove the restriction that FSMs have at most n states. Let C_2 be the resulting computation formalism, and let $E_2 \subseteq C_2$. It is known that, in general, given two (possible infinite) sets of deterministic FSMs, there is no finite set of sequences $\mathcal{I}_2$ allowing to distinguish each member of the first set from each FSM in the other set. Hence, in general we have $(C_2, E_2, \mathcal{D}) \notin$ `Class I` (we also prove this property in Example 7 of [14], this time by using Lemma 2 (b)). However, the set of all sequences of I' inputs, that is I'^*, distinguishes all pairs of non-equivalent deterministic FSMs. In fact, I'^* can be numbered (e.g., lexicographically). Thus, we have $(C_2, E_2, \mathcal{D}) \in$ `Class III`.

Let C_4 be a computation formalism representing all non-deterministic FSMs, and let $E_4 \subseteq C_4$. We consider an FSM M_1 that answers b when a is received, and another FSM M_2 with the same behavior as M_1 for all inputs but a: If M_2 receives a, then M_2 can answer either b or c. Let us suppose that M_1 is *correct* and M_2 is not, and let they be represented in C_4 by $f \in E_4$ and $f' \in C_4 \backslash E_4$, respectively. Despite of the fact that we *could* obtain c after applying a to f', input a does not necessarily distinguish f and f' because both of them could produce b in response to a. In fact, M_2 could hide output c for any arbitrarily long time, no matter how many times we apply a. Thus, no input allows the tester to necessarily distinguish f and f', so we have $(C_4, E_4, \mathcal{D}) \in$ `Class V` but $(C_4, E_4, \mathcal{D}) \notin$ `Class III`.[4]

Non-determinism does *not* imply that finite testability is not possible. Let $C_5 = \{f, f'\}$ and $E_5 = \{f\}$ be such that $f(a) = \{x\}$, $f'(a) = \{x, y\}$, $f(b) = \{x, y\}$, and $f'(b) = \{z, w\}$. Then, $\{b\}$ is a complete test suite for C_5, E_5, and $\mathcal{D}$. Thus, $(C_5, E_5, \mathcal{D}) \in$ `Class I`. Finally let us note that, given C, E, and $\mathcal{D}$, $(C, E, \mathcal{D}) \in$ `Class I` does not imply that C or E are finite. Let C represent all deterministic terminating Java-written functions from integers to integers that are either strictly increasing or strictly decreasing. Let $E \subseteq C$ consist of all strictly increasing functions in C. Though C and E are infinite sets, $\{3, 5\}$ is a complete test suite for $(C, E, \mathcal{D})$: For all $f \in C$, $f(3) < f(5)$ iff $f \in E$. So, $(C, E, \mathcal{D}) \in$ `Class I`. This trivial example shows that the finite testability does not lie in the finiteness of C or E but in their relation. □

Let $\mathcal{D}$ be defined in such a way that $o_1\ \mathcal{D}\ o_2$ only if o_1 and o_2 can be effectively distinguished via observation (see Example 2). If $(C, E, \mathcal{D}) \in$ `Class I` then we can precisely decide if the IUT is correct or not (i.e., if it belongs to the

[4] If *fairness* were assumed, we would be considering a *different* computation formalism $C'_4 \neq C_4$. In C'_4, for all input denoting a long repetition of the same experiment, the output must denote that all possible reactions are observed at least once.

specification set) by considering the answers collected by a complete test suite. If the problem belongs to `Class III` but not to `Class I`, then applying a complete test suite to the IUT is unfeasible because the suite is infinite. In particular, if the problem belongs to `Class III` then we could rather speak about testability *in the limit*, i.e. as we iteratively apply more tests, we could tend to complete testing coverage (this idea will be further elaborated below). Some issues concerning the relation between complete test suites and *non-termination* are presented in [14]. Besides, a result showing that observations collected by complete test suites univocally determine whether the IUT is correct is also presented in [14].

Next we elaborate on the idea of finite testability *in the limit* or, more precisely, *unboundedly-approachable* finite testability. In some cases where finite testability is not possible, it may still be possible to test the IUT up to any arbitrarily high *confidence degree* with a *finite* test suite. By confidence degree, here we mean the ratio between the number of pairs of correct/incorrect functions that are distinguished by the test suite and the total number of correct/incorrect pairs. If, for all real value $0 \leq \varepsilon < 1$, we can find a *finite* test suite providing a ratio higher than ε, then we say that the IUT is unboundedly-approachable by finite testing. Care must be taken to measure this ratio when (countable) *infinite* computation formalisms are considered. For instance, even if a test suite distinguishes one out of two pairs of correct/incorrect functions, and thus we expect to reach a 0.5 ratio, the set of all pairs and the set of distinguished pairs have the same cardinality (the same as $\mathbb{N}$). Instead of considering the cardinality of these infinite sets, the ratio will be measured for some *finite* subsets of the computation formalism. These subsets will be defined in such a way that they tend to cover the whole computation formalism as they increase.

Definition 5. Let C be a countable computation formalism for I and O, $E \subseteq C$ be a specification, $\mathcal{D}$ be a distinguishing relation, and $\mathcal{I} \subseteq I$ be a set of inputs. Let $h : \mathbb{N} \longrightarrow C$ be a surjective function and $k \in \mathbb{N}$. The *restriction* of C to k in h, denoted by $C \downarrow_k h$, is defined as $\{h(j) | 0 \leq j \leq k\}$. The *restriction* of E to k in h, denoted by $E \downarrow_k h$, is defined as $(C \downarrow_k h) \cap E$.

If C is finite then we define the *distinguishing rate* of $\mathcal{I}$ for $(C, E, \mathcal{D})$, denoted by $\texttt{d-rate}\,(\mathcal{I}, C, E, \mathcal{D})$, as $\dfrac{|\{(f, f') | f \in E, f' \in C \backslash E, \texttt{di}\,(f, f', \mathcal{I})\}|}{|E| \cdot |C \backslash E|}$.

We say that $(C, E, \mathcal{D})$ is *unboundedly-approachable by finite testing through* h if for all $\varepsilon \in \mathbb{R}$, with $0 \leq \varepsilon < 1$, there exists a finite set of inputs $\mathcal{I} \subseteq I$ such that, for all $k \in \mathbb{N}$, we have $\texttt{d-rate}\,(\mathcal{I}, C \downarrow_{k_1} h, E \downarrow_{k_1} h, \mathcal{D}) \geq \varepsilon$ for some $k_1 \geq k$. $\square$

Clearly, the property of being unboundedly-approachable by finite testing strongly depends on the function h considered to sort the computation formalism. In particular, given a triple $(C, E, \mathcal{D})$, it may be unboundedly-approachable for some sorting functions but not for others. Still, we can define a class of unboundedly-approachable triples $(C, E, \mathcal{D})$ as follows: We say that $(C, E, \mathcal{D})$ is *unboundedly-approachable* if there exists h such that $(C, E, \mathcal{D})$ is unboundedly-approachable by finite testing through h, and we denote by `Class II` the set of all unboundedly-approachable triples $(C, E, \mathcal{D})$. This class is related with the others as expected: We have Class I $\subseteq$ Class II $\subseteq$ Class III. In fact, as the next example shows, both inclusions are proper.

Example 4. We revisit $(C_2, E_2, \mathcal{D})$ as defined before in Example 3. In particular, let us assume that the sets of FSM inputs and outputs are $I' = \{a, b\}$ and $O' = \{0, 1\}$, respectively, and $E_2 = \{f_*\}$ consists of a single function f_*.

We have $(C_2, E_2, \mathcal{D}) \in$ Class II, i.e. $(C_2, E_2, \mathcal{D})$ is unboundedly-approachable by finite testing through some surjective function $h : \mathbb{N} \longrightarrow C_2$. The proof of this result is presented in [14], next we present an intuition. For all $l \in \mathbb{N}$, let $\mathcal{I}^l$ consist of all input sequences of length l. We define a function h sorting C_2 in such a way that, for all $l \in \mathbb{N}$, there exists $k \in \mathbb{N}$ such that all possible ways to react to $\mathcal{I}^l$ appear in the *same* proportion in $C_2 \downarrow_k h$ (i.e., the number of functions providing each combination of responses to $\mathcal{I}^l$ is the same for all combinations). Only one of these ways to react to $\mathcal{I}^l$ conforms to f_*. Thus, if there are n ways to react to $\mathcal{I}^l$, then $\mathcal{I}^l$ reaches a distinguishing rate $1 - \frac{1}{n}$. In longer restrictions $C_2 \downarrow_{k_1} h$ with $k_1 > k$, the proportion of functions providing each possible response to $\mathcal{I}^l$ is preserved, so $\mathcal{I}^l$ also reaches a $1 - \frac{1}{n}$ rate in these restrictions. Let $0 \leq \varepsilon < 1$. In order to find a finite test suite providing a rate higher than ε for *all* restrictions, we just have to consider $l' \in \mathbb{N}$ such that $\mathcal{I}^{l'}$ allows m ways to react, where $\frac{1}{m} < 1 - \varepsilon$.

Let $C_2' \subset C_2$ consist of f_* as well as all functions g_k, with $k \in \mathbb{N}$, such that g_k behaves as f_* for all input sequences but the input sequence $a^k b$ (and its extensions $a^k b\sigma$, for all $\sigma \in \{a, b\}^*$). In particular, the output produced by g_k when input b is given after a^k is the opposite as the one given by f_* (i.e. 0 if it is 1 in f_*; 1 otherwise). For all input sequence $a^k b\sigma$, the outputs produced by g_k for the rest of inputs after $a^k b$ are the same as the ones produced by f_*. There does not exist any surjective function $h : \mathbb{N} \longrightarrow C_2'$ such that $(C_2', E_2, \mathcal{D})$ is unboundedly-approachable by finite testing through h. This is because, for all finite test suite $\mathcal{I}$, the number of pairs of $\{(f_*, f) | f \in C_2' \backslash \{f_*\}\}$ distinguished by $\mathcal{I}$ is finite (in particular, this number is lower than or equal to $|\mathcal{I}|$). Thus, for all $\mathcal{I}$, h, and $0 < \varepsilon < 1$ there exists $k \in \mathbb{N}$ such that, for all $k_1 > k$, we have d-rate$(\mathcal{I}, C_2' \downarrow_{k_1} h, E_2 \downarrow_{k_1} h, \mathcal{D}) < \varepsilon$. We conclude $(C_2', E_2, \mathcal{D}) \notin$ Class II. $\square$

The two cases considered in the previous example illustrate an interesting fact: By *reducing* the computation formalism from C_2 to $C_2' \subset C_2$ (and thus reducing the number of possible *wrong* implementations) the unboundedly-approachable finite testability is *lost*. Intuitively, the reason is that C_2' only includes functions with a single fault, while *all* FSMs are in C_2 (including FSMs strongly deviating from the correct behavior). Thus, distinguishing correct implementations from incorrect ones is easier in C_2 than in C_2'. This shows that the difficulty of testing does not lie in the number of possible wrong implementations to be discarded, but in the *narrowness* of the border between correct and incorrect potential implementations. Due to this narrowness, if the computation formalism is C_2' then testing is not very productive in terms of distinguishability: After applying n tests, no more than n pairs of correct/incorrect functions will be distinguished – out of an infinite number of pairs of correct/incorrect functions to be distinguished.

3 Studying Properties of Class I

In this section we study the properties of Class I. For the sake of readability, in results presented from now on we will assume that C denotes a computation formalism for a set of inputs I and a set of outputs O, $E \subseteq C$ is a specification, and $\mathcal{D}$ is a distinguishing relation. In [14], some conditions enabling/disabling the testability, as well as some conditions allowing to preserve the testability when C and E are modified, are studied. These results allow to reason about the testability of problems from an abstract point, in such a way that most of details concerning the structure of a computation formalism (states, instructions, etc) can be ignored. This is illustrated in [14] with several examples. Next, we provide a result allowing to find a *lower bound* of the number of inputs that must be included in a set to achieve a complete test suite.

Proposition 1. Let $\mathcal{A} \subseteq 2^I$ be a set such that, for all $A \in \mathcal{A}$, we have that

(1) for all $A' \in \mathcal{A}$ with $A' \neq A$ we have $A' \cap A = \emptyset$, and
(2) there exist $f \in E$, $f' \in C \backslash E$ with $\{i \mid i \in I \wedge \forall\, o \in f(i), o' \in f'(i) : o\, \mathcal{D}\, o'\} \subseteq A$.

If $\mathcal{A}$ is finite then either $(C, E, \mathcal{D}) \notin$ `Class I` or for all finite complete test suite $\mathcal{I}$ for C, E, and $\mathcal{D}$ we have $|\mathcal{I}| \geq |\mathcal{A}|$. If $\mathcal{A}$ is infinite then $(C, E, \mathcal{D}) \notin$ `Class I`. □

An example illustrating the use of Proposition 1 is given in [14]. Next we consider an alternative way to find complete test suites by manipulating the sets distinguishing each pair of functions.

Proposition 2. Let G be the set of all *sets of inputs* allowing to distinguish each correct function from each incorrect function, that is,

$$G = \left\{ \left\{ i \,\middle|\, \begin{array}{l} i \in I \wedge \\ \forall\, o \in f(i), o' \in f'(i) : o\, \mathcal{D}\, o' \end{array} \right\} \,\middle|\, \begin{array}{l} f \in E, \\ f' \in C \backslash E \end{array} \right\}$$

We have $(C, E, \mathcal{D}) \in$ `Class I` iff there exist $n \in \mathbb{N}$ subsets of G, denoted by $A_1, \ldots, A_n \subseteq G$, such that $\bigcup_{1 \leq j \leq n} A_j = G$ and for all $1 \leq j \leq n$ we have $\bigcap_{B \in A_j} B \neq \emptyset$. □

Next we consider the problem of finding the *minimum* complete test suite in a particularly basic case: The case where (a) the computation formalism $C = \{f_1, \ldots, f_n\}$ is finite, (b) the sets $I = \{i_1, \ldots, i_k\}$ and $O = \{o_1, \ldots, o_l\}$ of inputs and outputs are finite as well, and (c) for each function $f \in C$, a tuple $(f(i_1), \ldots, f(i_k))$ denoting the possible outputs of f for all inputs is explicitly provided. We will denote this problem as the *Minimum Complete Suite* problem, and we prove its NP-completeness (in particular, a polynomial reduction from the *Minimum Set Cover* problem to this problem is constructed in [14]).

Definition 6. Let C be a finite computation formalism for the finite sets of inputs and outputs $I = \{i_1, \ldots, i_k\}$ and O, respectively, $E \subseteq C$ be a finite specification, and $\mathcal{D}$ be a finite distinguishing relation. Let C' and $E' \subseteq C'$ be sets of tuples representing the behavior of functions of C and E respectively; formally, for all $f \in C$ we have $(f(i_1), \ldots, f(i_k)) \in C'$ and viceversa, and for all $g \in E$ we have $(g(i_1), \ldots, g(i_k)) \in E'$ and viceversa.

Given C', E', I, O, $\mathcal{D}$, and some $K \in \mathbb{N}$, the *Minimum Complete Suite* problem (`MCS`) is defined as follows: Is there any complete test suite $\mathcal{I}$ for C, E, and $\mathcal{D}$ such that $|\mathcal{I}| \leq K$? $\square$

Theorem 1. `MCS` $\in$ `NP-complete`. $\square$

We present some notions allowing to reason about *testing hypotheses*, which play a key role in formal testing methodologies. As we have seen, the testability is affected by assuming some restrictions about the IUT (e.g., deterministic FSMs move from `Class III` to `Class I` if a given limit of the number of states is assumed). Auxiliary notions and properties related with hypotheses are presented in [14]. In the previous section we proposed to use the *distinguishing rate* (see Definition 5) to measure the coverage of an incomplete finite test suite. Alternatively, next we consider measuring the coverage of an incomplete test suite $\mathcal{I}$ in terms of the amount of potential functions that should be *removed* from C to make $\mathcal{I}$ *complete*, that is, in terms of the number of potential implementations we have to *assume* not to be the actual IUT to make $\mathcal{I}$ complete. We consider that an incomplete test suite is *better* than another incomplete suite if making the former suite complete requires removing less functions. That is, the suite requiring a kind of *weaker* function removal assumption to be complete is better. Let us note that *testing hypotheses* are typically assumed to make this effect indeed, that is, to reduce the number of potential implementations so that finite test suites can be complete – provided that the hypotheses hold. Thus, if a test suite requires less or weaker hypotheses to be assumed (i.e. less potential implementations have to be removed) to get completeness than another, then the former test suite is better because it is *closer* to be complete indeed. Next we consider the difficulty to measure these alternative coverage measures in the same context as when we defined the `MCS` problem in Definition 6. That is, C, E, I, O are finite and functions in C and E are explicitly denoted by some sets of tuples C' and E'. In the next definition, the set $\mathcal{H} = \{H_1, \ldots, H_n\}$ represents the hypotheses the tester may or may not assume: If the hypothesis $H_i \subseteq C$ is assumed then all functions in H_i are assumed not to be in the actual computation formalism (e.g., if we are assuming that the IUT is deterministic then we assume some $H \subseteq C$, where H consists of all non-deterministic functions in C).

Definition 7. Given the same preamble as in Definition 6, let $\mathcal{I} \subseteq I$ be a set of inputs and $\mathcal{H} = \{H_1, \ldots, H_n\}$, where for all $1 \leq i \leq n$ we have $H_i \subseteq C'$.

Given C', E', I, O, $\mathcal{D}$, $\mathcal{I}$ and some $K \in \mathbb{N}$, the *Minimum Function Removal* problem (`MFR`) is defined as follows: Is there any set $R \subseteq C$ with $|R| \leq K$ such that $\mathcal{I}$ is a complete test suite for $(C \backslash R, E \backslash R, \mathcal{D})$?

Given C', E', I, O, $\mathcal{D}$, $\mathcal{I}$, $\mathcal{H}$, and some $K \in \mathbb{N}$, the *Minimum Function removal via Hypotheses* problem (`MFH`) is defined as follows: Is there any set of hypotheses $R \subseteq \mathcal{H}$ with $|\bigcup_{H \in \mathcal{H}} H| \leq K$ such that $\mathcal{I}$ is a complete test suite for $(C \backslash (\bigcup_{H \in R} H), E \backslash (\bigcup_{H \in R} H), \mathcal{D})$?

Given C', E', I, O, $\mathcal{D}$, $\mathcal{I}$, $\mathcal{H}$, and some $K \in \mathbb{N}$, the *Minimum Hypotheses Assumption* problem (MHA) is defined as follows: Is there any set $R \subseteq \mathcal{H}$ with $|R| \leq K$ such that $\mathcal{I}$ is a complete test suite for $(C \backslash (\bigcup_{H \in R} H), E \backslash (\bigcup_{H \in R} H), \mathcal{D})$? □

The differences among the previous problems (and the corresponding coverage measures they allow to calculate) is the following: In MFR, the coverage is measured in terms of the minimal *number* of functions that must be removed, where *any* subset of C is allowed to be removed. In MFH, the number of functions is considered again, though this time the set of functions to be removed must be the union of some *hypotheses* (sets of functions) from a given hypotheses repertory $\mathcal{H}$. Finally, MHA considers the coverage in terms of the number of assumed *hypotheses*, rather than on the number of removed functions. At a first glance, one might think that all MFR, MFH, and MHA are NP-hard problems. Interestingly, MFR is not, as it can be reduced to the *Minimum Vertex Cover* problem in *bipartite graphs*, which in turn is equivalent to the *Maximum Matching* problem. This problem can be solved in polynomial time by the Hopcroft-Karp algorithm [9]. Thus, measuring the coverage of an incomplete test suite in terms of the minimum number of functions that must be removed to achieve completeness is a tractable problem indeed. The proof of the next result, given in [14], introduces the proposed reduction and exploits it to actually find the *minimum* function removal in time $\mathcal{O}(|C'|^{5/4} + |C'|^2 \cdot |\mathcal{I}| \cdot |\mathcal{O}|^2)$, thus solving MFR polynomially. On the other hand, the NP-completeness of MFH and MHA is proved by constructing polynomial reductions from *3-SAT* and *Minimum Set Cover*, respectively.

Theorem 2. We have the following properties:

(a) MFR $\in$ P
(b) MFH $\in$ NP-complete
(c) MHA $\in$ NP-complete □

Given a computation formalism, we consider the problem of finding a complete test suite for *any* specification belonging to a given set of specifications.[5] Moreover, rather than imposing a fix computation formalism for all considered specifications, a (possibly infinite) set of triples $(C, E, \mathcal{D})$ will denote the cases to be considered.[6] We call this set *testing problem*. Ideally, our knowledge about how to find suitable tests for a given testing problem (i.e. for a given kind of target formalisms and specifications) could help us to face other testing problems. In order to enable this, we introduce a general notion of *testability reduction*. If a testing problem can be solved by *transforming* it into another testing problem and solving the latter, then we will say that the former problem can be *reduced* to the latter. This provides a criterion to classify problems inside each class.

[5] This is the typical goal of testing methodologies: For any specification fitting into a given kind of specifications (for us, a *computation formalism*), find a way to derive tests from the specification in such a way that the test suite is complete.

[6] This improves the generality of the problem. For instance, if the tester assumes that the IUT includes at most n faults, a particular C should be considered for each E.

Definition 8. A *testing problem* for a set of inputs I is a set $\mathcal{T}$ of triples $(C, E, \mathcal{D})$ with the usual meanings of $C, E, \mathcal{D}$, where I is the input set for all C.

Let $\mathcal{T} \subseteq$ `Class I`. A computable function $f : \mathcal{T} \longrightarrow 2^I$ is a *finite suite derivation* for $\mathcal{T}$ if, for all $p \in \mathcal{T}$, $f(p)$ is a finite complete test suite for p.

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be testing problems for I_1 and I_2. $\mathcal{T}_1$ can be *finitely reduced* to $\mathcal{T}_2$, denoted by $\mathcal{T}_1 \leq_F \mathcal{T}_2$, if there exist some computable functions $e : \mathcal{T}_1 \longrightarrow \mathcal{T}_2$ and $t : 2^{I_2} \longrightarrow 2^{I_1}$ such that, for all $p_1 \in \mathcal{T}_1$ and $\mathcal{I} \subseteq I_2$, if $\mathcal{I}$ is a finite complete test suite for $e(p_1)$ then $t(\mathcal{I})$ is a finite complete test suite for p_1. □

Theorem 3. *(Testing reduction Theorem)* We have the following properties:

(a) $\leq_F$ is a preorder.
(b) Let $\mathcal{T}_1 \leq_F \mathcal{T}_2$. If there exists a finite suite derivation for $\mathcal{T}_2$ then there exists a finite suite derivation for $\mathcal{T}_1$.
(c) Let $\mathcal{T}_1 \leq_F \mathcal{T}_2$. If $\mathcal{T}_2 \in$ `Class I` then $\mathcal{T}_1 \in$ `Class I`. □

We should not confuse $\leq_F$ with the classical *testing preorder* relation given in [6]. In particular, the $\leq_F$ relation does not compare processes but *testing problems*. In this sense, it reminds the reductions of computability and complexity theory, where a computation problem is transformed into another problem.[7] In our case, we check whether finding a finite complete test suite in a given scenario can be achieved by means of finding a finite complete test suite in another one.

Examples illustrating $\leq_F$ are given in [14]. Some finitely constrained versions of FSMs, EFSMs (Extended FSMs), and TEFSMs (Temporal EFSMs) are considered. Under these finiteness conditions, (a) the problem of testing EFSMs is reduced to the problem of testing FSMs; (b) testing TEFSMs is reduced to testing EFSMs; (c) by the transitivity of $\leq_F$, testing TEFSMs is reduced to testing FSMs; (d) TEFSMs are finitely testable because FSMs are so. Other examples, showing that the finiteness of computation formalisms is neither sufficient nor necessary for achieving $\leq_F$ (at any side of $\leq_F$), are presented as well. They show that, in general, the testing reduction does not consist in mapping a computation formalism into another (as we do in the FSM-EFSM-TEFSM example), but in mapping the *border* between correctness and incorrectness from a problem to another. Next we show an example where computation formalisms are infinite and the reduction only considers factors affecting the correctness border.

Example 5. Let $\mathcal{D}$ be the trivial distinguishing, C_{TM} represent all deterministic terminating Turing Machines (TMs) from $\{0,1\}^*$ to $\{yes, no\}$, and C_{FA} represent all deterministic finite automata with the same type. Let $\mathcal{T}_{TM}$ consist of all triples where we have to distinguish some TM M from all TMs answering as M for at most $k \in \mathbb{N}$ inputs. Formally, $\mathcal{T}_{TM} = \{(\{f\} \cup C_{TM}^{f,k}, \{f\}, \mathcal{D}) | f \in C_{TM}, k \in \mathbb{N}\}$ where $C_{TM}^{f,k} = \{f' | f' \in C_{TM}, f'$ gives the same answer as f for at most k words$\}$. Similarly, let $\mathcal{T}_{FA} = \{(\{f\} \cup C_{FA}^{f,k}, \{f\}, \mathcal{D}) | f \in C_{FA}, k \in \mathbb{N}\}$ where $C_{FA}^{f,k} = \{f' | f' \in C_{FA}, f'$ gives the same answer as f for at most k words$\}$. Note that any triple in $\mathcal{T}_{TM}$ or $\mathcal{T}_{FA}$ is uniquely characterized by f

[7] In fact, a testing problem can also be regarded a kind of computation problem.

and k. We have $\mathcal{T}_{TM} \leq_F \mathcal{T}_{FA}$. In particular, the functions e and t considered in Definition 8 can be defined as follows. $e : \mathcal{T}_{TM} \longrightarrow \mathcal{T}_{FA}$ is such that, for all $p = (\{f\} \cup C^{f,k}_{TM}, \{f\}, \mathcal{D}) \in \mathcal{T}_{TM}$, we have $e(p) = (\{f'\} \cup C^{f',k}_{FA}, \{f'\}, \mathcal{D})$ where f' is *any* function in C_{FA} (say, the one answering *yes* for all inputs), and $t : 2^{\{0,1\}^*} \longrightarrow 2^{\{0,1\}^*}$ is the identity function. For all $e(p) = (\{f'\} \cup C^{f',k}_{FA}, \{f'\}, \mathcal{D})$, $\mathcal{I}$ is a complete test suite for $e(p)$ only if $\mathcal{I}$ consists of (any) $k+1$ or more different inputs. If it is so, $t(\mathcal{I}) = \mathcal{I}$ is also complete for $p = (\{f\} \cup C^{f,k}_{TM}, \{f\}, \mathcal{D}) \in \mathcal{T}_{TM}$. So, $\mathcal{T}_{TM} \leq_F \mathcal{T}_{FA}$. For all $q \in \mathcal{T}_{FA}$, for some $k \in \mathbb{N}$ we have that any set of $k+1$ inputs is a complete test suite for q, so $\mathcal{T}_{FA} \in$ `Class I`. By Theorem 3 (c) we conclude $\mathcal{T}_{TM} \in$ `Class I`. By using similar arguments we have $\mathcal{T}_{FA} \leq_F \mathcal{T}_{TM}$. $\square$

4 Conclusions and Future Work

Formal Testing Techniques have reached a high level of maturity during the last years. However, some common roots allowing to relate testing methods with each other are still missing. In particular, a classification of testing problems would allow us to use our expertise about old testing problems to solve new problems. This paper tries to contribute to this (long term) goal. We have presented some general notions of testability and we have identified five classes of testability. We have studied the properties of the first class, i.e. *finitely testable* problems, including the complexity of finding a minimum complete test suite or measuring the completeness degree of incomplete test suites, alternative characterizations, transformations keeping the testability, the effect of adding testing hypotheses, and methods to *reduce* a testing problem into another. From a theoretical point of view, these techniques allow to relate testing problems with each other as well as to classify them. From a practical point of view, they allow us to determine the (im-)possibility to find complete test suites in different scenarios, as well as to reason about how far an incomplete test suite is from being complete, thus providing an implicit way to compare and select incomplete test suites. The proposed framework endows us with these capabilities even though it is highly abstract (several examples of use have been presented, other examples can be found in [14]). Going one step further, the proposed general properties could be particularized and refined for specific computation formalisms.

More generally, properties presented in Section 3 allow us to envisage new paths to improve our understanding about testing in the future. In particular, the testing reduction notion could play a key role to organize the five proposed classes into a big variety of subclasses that should be defined, related, and studied as well: e.g. *bounded* finite testability (what is the size of minimum complete test suites with respect to the size of the system representation? e.g. polynomial or exponential?), *adaptive* testability, *probabilistic* testability, *heuristic* testability, etc. In addition, we wish to study the applicability of the proposed testability notions to the related problem of *learnability.*

Acknowledgements. I thank Fernando Rubio, Carlos Molinero, and Manuel Núñez for their careful reading of previous versions of this paper, Diego Rodríguez for his support, and the anonymous reviewers for their suggestions.

References

1. Bernot, G., Gaudel, M.-C., Marre, B.: Software testing based on formal specification: a theory and a tool. Software Engineering Journal 6, 387–405 (1991)
2. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–195. Springer, Heidelberg (2001)
3. Do, H., Rothermel, G., Kinneer, A.: Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. Empirical Software Engineering 11(1), 33–70 (2006)
4. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006)
5. Gaudel, M.-C.: Testing can be formal, too. In: Mosses, P.D., Schwartzbach, M.I., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995)
6. Hennessy, M.: Algebraic Theory of Processes. MIT Press, Cambridge (1988)
7. Hierons, R.M.: Comparing test sets and criteria in the presence of test hypotheses and fault domains. ACM Trans. on Software Engineering and Methodology 11(4), 427–448 (2002)
8. Hierons, R.M.: Verdict functions in testing with a fault domain or test hypotheses. In: ACM Transactions on Software Engineering and Methodology (2008) (to appear)
9. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM Journal on Computing 2(4), 225–231 (1973)
10. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines: A survey. Proceedings of the IEEE 84(8), 1090–1123 (1996)
11. López, N., Núñez, M., Rodríguez, I.: Specification, testing and implementation relations for symbolic-probabilistic systems. Theoretical Computer Science 353 (1–3), 228–248 (2006)
12. Merayo, M., Núñez, M., Rodríguez, I.: Extending EFSMs to specify and test timed systems with action durations and timeouts. IEEE Transactions on Computers 57(6), 835–844 (2008)
13. Petrenko, A.: Fault model-driven test derivation from finite state models: Annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 196–205. Springer, Heidelberg (2001)
14. Rodríguez, I.: A general testability theory: Extended version (2006), `http://kimba.mat.ucm.es/ismael/`
15. Rodríguez, I., Merayo, M.G., Núñez, M.: $\mathcal{HOTL}$: Hypotheses and observations testing logic. Journal of Logic and Algebraic Programming 74(2), 57–93 (2008)
16. Springintveld, J., Vaandrager, F., D'Argenio, P.R.: Testing timed automata. Theoretical Computer Science 254(1-2), 225–257 (2001); Previously appeared as Technical Report CTIT-97-17, University of Twente (1997)
17. Stoelinga, M., Vaandrager, F.: A testing scenario for probabilistic automata. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 464–477. Springer, Heidelberg (2003)
18. Tretmans, J.: Testing concurrent systems: A formal approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 46–65. Springer, Heidelberg (1999)

Counterexamples in Probabilistic LTL Model Checking for Markov Chains[⋆]

Matthias Schmalz[1], Daniele Varacca[2], and Hagen Völzer[3]

[1] ETH Zurich, Switzerland
[2] PPS - CNRS & Univ. Paris Diderot, France
[3] IBM Zurich Research Laboratory, Switzerland

Abstract. We propose a way of presenting and computing a counterexample in probabilistic LTL model checking for discrete-time Markov chains. In qualitative probabilistic model checking, we present a counterexample as a pair (α, γ), where α, γ are finite words such that all paths that extend α and have infinitely many occurrences of γ violate the specification. In quantitative probabilistic model checking, we present a counterexample as a pair (W, R), where W is a set of such finite words α and R is a set of such finite words γ. Moreover, we suggest how the counterexample presented helps the user identify the underlying error in the system by means of an interactive game with the model checker.

1 Introduction

A counterexample in LTL model checking is an execution path that violates the LTL specification. This counterexample path should help the user identify and repair an error in the system. However, a counterexample path is in general infinite, and therefore if we want to show it to the user, we must find a finite representation. In classical LTL model checking, we can exploit the fact that a *periodic* counterexample always exists (see e.g. [17]), i.e., an execution path of the form $\alpha\gamma^\omega$, where α and γ are finite words.

In the probabilistic LTL model checking problem that we consider here, we are given an LTL formula Φ and a discrete-time finite-state Markov chain generating a probability measure $\mathbb{P}$, and we want to check whether $\mathbb{P}[\Phi] > t$ (or $\mathbb{P}[\Phi] \geq t$). A counterexample witnessing the violation of this assertion is therefore a set Y of execution paths violating Φ such that $\mathbb{P}[Y] \geq 1-t$ (or $\mathbb{P}[Y] > 1-t$). In general such a set is not only infinite, but almost all of its paths are aperiodic. How can such a counterexample be presented to the user to provide useful debugging information?

In this paper, we show how a counterexample can be presented and computed and suggest how the user should interact with the model checker to find the error.

We start by considering the special case of qualitative probabilistic model checking, i.e., the question whether $\mathbb{P}[\Phi] = 1$. We propose to represent a qualitative counterexample as a pair (α, γ), where

– α is a finite path such that *almost all* paths extending α violate the specification and hence the specification is violated with at least the probability of α. Therefore, α shows where the probability is lost.

⋆ This work was partially supported by the EU project Deploy (N. 214158).

– γ is a finite word in a bottom strongly connected component such that *all* paths that extend α and that have infinitely many occurrences of γ violate the specification. The word γ witnesses that almost all paths extending α violate the specification.

The pair (α, γ) is presented to the user in an interactive game with the model checker. The user tries to construct a path extending α and satisfying the specification, whereas the model checker ensures that γ occurs infinitely often. By failing to construct such a path the user finds an error in the system.

We then show that this approach can be extended to the quantitative case ($t < 1$), where in general a set W of such finite paths α and a set R of such finite words γ has to be considered.

Finally, we show how such a counterexample can be computed; we build on a model checking algorithm by Courcoubetis and Yannakakis [8], which however has to be substantially complemented for our purposes.

We discuss related work in Section 6. Missing proofs can be found in [19].

2 Preliminaries

We assume that the reader is familiar with Kripke structures, discrete-time Markov chains, linear temporal logic (LTL) and ω-regular languages. We briefly recall the basic definitions to introduce conventions and fix the notation. References for further reading are also provided.

2.1 Words

Let Q be a set of *states*. The sets of infinite, finite and nonempty finite words over Q are denoted Q^ω, Q^* and Q^+, respectively. Usually q, p denote elements of Q; $\alpha, \beta, \gamma, \delta$ elements of Q^*, x an element of Q^ω, z an element of $Q^\omega \cup Q^*$, and λ the empty word.

We write $\alpha \sqsubseteq z$ if α is a prefix of z. If $\alpha \sqsubseteq z$, z is called an *extension* of α. We define $\alpha\uparrow := \{x \mid \alpha \sqsubseteq x\}$ and $z\downarrow := \{\alpha \mid \alpha \sqsubseteq z\}$. Similarly, given $W \subseteq Q^*$ and $Y \subseteq Q^\omega \cup Q^*$, $W\uparrow := \bigcup\{\alpha\uparrow \mid \alpha \in W\}$ (the set of *extensions of* W) and $Y\downarrow := \bigcup\{z\downarrow \mid z \in Y\}$.

2.2 Probabilistic Systems

A *system* (Kripke structure) $\Sigma = (Q, S, \rightarrow, v)$ consists of a finite set Q of *states*, a nonempty set $S \subseteq Q$ of *initial states*, a *state relation* $\rightarrow \subseteq Q \times Q$ and a *valuation function* $v : Q \rightarrow 2^{AP}$ mapping each state q to a set $v(q) \subseteq AP$ of *atomic propositions*. We assume here that for each $q \in Q$ there is a $p \in Q$ so that $q \rightarrow p$. The *size* of Σ is $|\Sigma| := |Q| + |\rightarrow|$.

A *path fragment (of* Σ*)* is a word $q_0 q_1 \ldots \in Q^\omega \cup Q^*$ such that $q_{i-1} \rightarrow q_i$, $i > 0$. A *path (of* Σ*)* is a path fragment qz with $q \in S$. The empty word is also a path and a path fragment of Σ. The set $path_{fin}(\Sigma)$ contains all finite, and $path_\omega(\Sigma)$ all infinite paths of Σ.

Often we view Σ as the directed graph $(Q, \rightarrow)$. A set $K \subseteq Q$ is a *strongly connected component (of* Σ*)* (scc for short) if it is a strongly connected component of $(Q, \rightarrow)$ (see e.g. [7]). A *bottom strongly connected component (of* Σ*)* (bscc) is an scc K without outgoing edges, i.e., if $q \in K$ and $q \rightarrow p$, then $p \in K$.

A *(labelled discrete-time) Markov chain* (see e.g. [6,15]) is a system $\Sigma = (Q, S, \rightarrow, v)$ equipped with *transition probabilities* given by $P : Q \times Q \rightarrow [0,1]$ and *initial probabilities* given by $P_{ini} : Q \rightarrow [0,1]$, where $P(q,p) > 0$ iff $q \rightarrow p$, $P_{ini}(q) > 0$ iff $q \in S$,

$\sum_{p\in Q} P(q,p) = 1$, and $\sum_{q\in Q} P_{ini}(q) = 1$. It is well known (see e.g. [6,15]) that a Markov chain induces a measure $\mathbb{P}$ on the σ-algebra $\mathcal{B}(Q^\omega)$ induced by the basic cylinder sets $\alpha\!\uparrow$, $\alpha \in Q^*$ with the property $\mathbb{P}[q_0 \dots q_n\!\uparrow] = P_{ini}(q_0)\prod_{i=1}^{n} P(q_{i-1},q_i)$, $q_0 \dots q_n \in Q^+$. A measure induced by a Markov chain is called *Markov measure*. We later refer to a Markov chain simply as $\Sigma, \mathbb{P}$.

2.3 Temporal Properties

A *(linear-time temporal) property*, denoted Y, Z, is a subset of Q^ω. We mainly consider properties expressible in LTL (linear temporal logic [16]); we use the notation introduced in [11]. A formula in LTL is built from atomic propositions in *AP*, *true*, *false* and the boolean and temporal connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ and X, U, G, F. The *size* $|\Phi|$ of a formula is the number of its temporal and boolean connectives.

An LTL formula Φ is interpreted in the context of a system $\Sigma = (Q, S, \rightarrow, v)$ over words $x \in Q^\omega$. For $i \in \mathbb{N}$, $x, i \models \Phi$ means that *x satisfies* Φ *at position i* (in the usual sense [11]). Moreover, $x \models \Phi$ (*"x satisfies* Φ*"*) abbreviates $x, 0 \models \Phi$; $\Sigma \models \Phi$ (*"*Σ *satisfies* Φ*"*) means $x \models \Phi$ for all $x \in path_\omega(\Sigma)$. We write $Sat(\Sigma, \Phi)$ for the set of all infinite paths of Σ satisfying Φ. For convenience, we often write $Sat(\Phi)$ or Φ instead of $Sat(\Sigma, \Phi)$. In particular, $Sat(true) = path_\omega(\Sigma)$. A formula ϕ without temporal connectives is a *state formula*. For $q \in Q$, $q \models \phi$ (*"q satisfies* ϕ*"*) iff $qx \models \phi$ for all (or equivalently some) $x \in Q^\omega$.

To simplify the presentation, we suppose that for each $q \in Q$ there is an atomic proposition a_q that holds in q and only there. In our examples, we do not explicitly mention such atomic propositions a_q. For better readability of formulas, we write q instead of a_q; $q_0q_1 \dots q_n$ instead of $q_0 \wedge \mathrm{X}(q_1 \wedge \dots \mathrm{X}(q_n)\dots)$ and λ instead of *true*. These assumptions do not affect the results of the paper.

An ω*-regular property* is a property that is accepted by some *Büchi automaton*. Any LTL formula expresses an ω-regular property. Any ω-regular property is *measurable*, i.e., a member of $\mathcal{B}(Q^\omega)$ (see [21]).

3 Qualitative Counterexamples

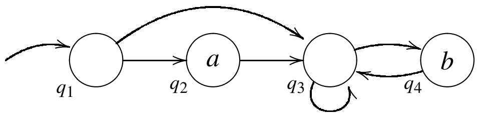
In this section, we consider the question whether $\mathbb{P}[\Phi] = 1$, where Φ is an LTL formula and $\mathbb{P}$ a Markov measure. Probabilistic satisfaction can be seen as a special form of quantification, but our traditional understanding of a counterexample is tightly connected with universal quantification. Say, we want to understand why the CTL* formula A.Φ does not hold, where Φ is an LTL formula. We display a classical linear counterexample path in this case. The system has more behaviour than expected, and the model checker displays the path as an example of the additional behaviour. The user can then replay the path to see where the actual behaviour deviates from her expectation, which is where she should find the error in the system.

The situation is different for existential quantification. Say we want to understand why a CTL* formula E.Φ is violated, again Φ being an LTL formula. For example, E.Φ could express the property that there exists a run of the system such that 'someone wins the jackpot'. If the formula is false, the answer of the model checker is just 'no'. The information to be returned could be the entire system showing the absence of a path satisfying Φ. Of course, that is not very informative. To find the error, we suggest that the proof burden should be reversed, i.e., the user should try to display a witness for the formula. The user should have an idea on what the path should look like; in the example: she knows *how* someone could win the jackpot in the system. She can then try to replay that path. In doing so, she will find a point where the behaviour of the system deviates from her expectations, because the desired path does not exist in the system.

The interaction between user and model checker that we propose for the probabilistic case will be a mixture of the universal and the existential case.

3.1 Examples of Counterexamples

To approach the problem for Markov chains, let us now consider some examples. By default, the examples are based on the system $\Sigma = (Q, S, \rightarrow, v)$ below. For the qualitative case the particular transition probabilities of the Markov chain are not relevant (see e.g. [20]); hence we do not display them. Each example considers an LTL formula Φ for which $\mathbb{P}[\Phi] < 1$, and in each case we will discuss how a counterexample should look.



Example 3.1. Let $\Phi = \mathrm{G}\,\neg a$. As Φ is a safety property, if it is violated, there is a finite path α such that each extension of α violates Φ. This is the case for the path $\alpha = q_1 q_2$. Because $\mathbb{P}[q_1 q_2 \uparrow] > 0$, we have $\mathbb{P}[\Phi] < 1$. As in classical model checking, it is sufficient to display the violating finite path α to the user as a counterexample.

Example 3.2. Let $\Phi = \mathrm{F}\,a$. There is a finite path $\alpha := q_1 q_3$ in the system such that each extension of α into $path_\omega(\Sigma)$ violates Φ, i.e., no extension of α into $path_\omega(\Sigma)$ contains an a-state. This clearly proves that $\mathbb{P}[\Phi] < 1$. In contrast to the previous example, not all extensions of α but only the extensions into $path_\omega(\Sigma)$ violate Φ. Hence, the inspection of α may not be sufficient to find the error; the user also has to take the structure of Σ into account. Similar to the CTL* case discussed above, the user who designed the system should have an idea on how to reach an a-state, once α has been executed. By trying to play such a path, which does not exist, she will eventually find the point in the system where the actual and the expected behaviour deviate.

Example 3.3. Let $\Phi = \mathrm{F}\,\mathrm{G}\,\neg b$. We recall that in any Markov chain a path eventually enters a bscc with probability one. For each reachable bscc K, a path eventually enters K with nonzero probability, and then, with probability 1, visits all states of K infinitely often. (These facts are well known and also follow from Lemma 4.3.) Any run that infinitely often visits a b-state violates Φ. The system above has a reachable bscc that contains a b-state, and therefore the specification is violated with nonzero probability.

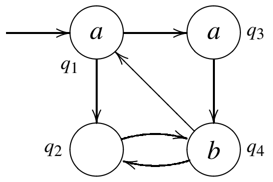
To show that to the user, we propose that the model checker returns a b-state within a bscc, namely q_4. The user then convinces herself that (i) the b-state indeed belongs to a reachable bscc and (ii) repeatedly visiting the b-state violates Φ. The latter point (ii) is straightforward in this case. To convince herself of (i), the user plays the following interactive game with the model checker: She tries to find a finite path fragment $q_4\beta$ so that q_4 is unreachable after β. If she believes that Φ has probability 1, she has an idea of how to do so. The model checker then goes back to q_4. The system must deviate from the expected behaviour in at least one of these two moves.

Example 3.4. Let $\Phi = \mathrm{G\,F}\,b \Rightarrow \mathrm{F}\,a$. Repeatedly visiting a b-state without visiting an a-state violates Φ. The specification does not have probability 1, as there is a bscc containing a b-state but no a-state, and that bscc can be reached without passing through an a-state. We propose that the model checker outputs q_4 and $\alpha := q_1q_3$. The user then convinces herself that (i) q_4 belongs to a bscc and that α leads to that bscc, and (ii) any path starting with α, and visiting q_4 infinitely often violates Φ. To this end, she plays the following game with the model checker: The model checker plays α. To convince herself of (i), the user tries to extend α so that q_4 becomes unreachable; the model checker then goes back to q_4. If that does not help discover the error, she tries to refute (ii) by extending α to $\alpha\beta \in path_{fin}(\Sigma)$ so that β visits an a-state.

In Examples 3.1 and 3.2, a counterexample is represented by a finite path α such that $\Sigma \models \alpha \Rightarrow \neg\Phi$. This representation is not sufficiently expressive for Examples 3.3 and 3.4. Therefore we use the more general representation (α, q), $\alpha \in path_{fin}(\Sigma)$, $q \in Q$ such that $\Sigma \models \alpha \wedge \mathrm{G\,F}\,q \Rightarrow \neg\Phi$. Note that, in the above examples, the formula Φ is violated after α with probability one, and q witnesses that. The state q is in particular important when α leads to a large bscc.

However, there still are situations, in which counterexamples of the form (α, q) cannot be found, and we need a path fragment instead of the single state q:

Example 3.5. Consider $\Phi = \mathrm{F\,G}(a \Rightarrow a\,\mathrm{U}\,b)$ with the following system: There are no α, q with $\Sigma \models (\alpha \wedge \mathrm{G\,F}\,q) \Rightarrow \neg\Phi$, but any path of Σ that infinitely often visits $\gamma := q_1q_2$ violates Φ.



We therefore consider counterexamples of the form $\alpha \wedge \mathrm{G\,F}\,\gamma$, $\alpha \in path_{fin}(\Sigma)$, $\gamma \in Q^*$. Below we prove that such a counterexample always exists when Φ has probability less than one.

3.2 Presenting a Qualitative Counterexample

According to the discussion in the preceding section, we propose to represent a qualitative counterexample as a pair (α, γ), where α is a finite path of the system and γ is a finite path fragment within some bscc of the system.

Definition 3.6. *A finite path fragment belonging to a bscc of a system Σ is called a* recurrent word (of Σ). *Let α be a finite path and γ a recurrent word of Σ. We say that γ* refutes *a property Y* in context α *iff the following conditions hold:*

1. *If $\gamma \neq \lambda$, then α leads to the bscc of γ, i.e., the bscc of γ is the unique bscc reachable after α.*
2. *$\alpha\!\uparrow \cap\, Sat(\mathrm{G\,F}\,\gamma) \cap Y = \varnothing$, i.e., any path starting with α and repeating γ infinitely often violates Y.*

If γ refutes Y in context α, then the pair (α, γ) represents the set of paths $\alpha\!\uparrow \cap\, Sat(\mathrm{G\,F}\,\gamma)$ violating Y; α describes how the violations begin and γ restricts their behaviour in the long run. The pair (α, γ) represents a qualitative counterexample because $\alpha\!\uparrow \cap\, Sat(\mathrm{G\,F}\,\gamma)$ has nonzero probability, as we will see in Section 3.3. In particular, almost all paths that extend α violate Y. In this sense, α is a 'bad' prefix of the system. The word γ witnesses that α is 'bad' in this sense.

We propose to use this representation of a qualitative counterexample in an interaction between the user and the model checker as follows. First the model checker outputs α and γ and claims that γ is a recurrent word refuting Φ in context α. Then the user can challenge that claim in the following ways:

1. If $\gamma = \lambda$, the user tries to construct a path that extends α and satisfies Φ. In failing to do so, she will find a point where the actual and the expected behaviour deviate.

2.1. She challenges that $\gamma \neq \lambda$ belongs to any bscc at all or that after α only that bscc is reachable by constructing a path $\alpha\beta$ after which, in her opinion, γ is unreachable. The model checker refutes this challenge by returning δ such that $\alpha\beta\delta\gamma \in path_{fin}(\Sigma)$.

2.2. She challenges $\alpha\!\uparrow \cap\, Sat(\mathrm{G\,F}\gamma) \cap Sat(\Phi) = \varnothing$, where $\gamma \neq \lambda$, by constructing a path $x = \alpha\beta_1\gamma\beta_2\gamma\ldots$, which she believes to satisfy Φ. In failing to construct such a path, she will observe that the expected and the actual behaviour of the system differ.

The path x can be constructed interactively: The model checker starts with α. The user wants to extend α to a path that ultimately satisfies Φ, but she may only append a finite word at a time, allowing the model checker to append γ in between. If the user appends a word that allows the model checker to append γ directly, the model checker will do so. Otherwise the model checker suggests some extension of the current finite path that allows it to append γ afterwards. This interaction continues until the user has found some unexpected behaviour of the system.

In practice, the user cannot play forever. But she can try to generate a periodic path, i.e., a path of the form $\alpha\beta_1(\gamma\beta_2)^\omega$. It is well known that an LTL formula is violated only if it has a periodic counterexample.

3.3 Soundness and Completeness

Let $\Sigma = (Q, S, \rightarrow, v), \mathbb{P}$ be a Markov chain and Y a property. In this section, we show that our proposal to present qualitative counterexamples is sound and complete, i.e., the existence of (α, γ) implies $\mathbb{P}[Y] < 1$ and vice versa. In fact, using results from [20], we can show that our proposal is sound for arbitrary properties and complete if the specification is ω-regular.

Theorem 3.7

1. *If γ is a recurrent word refuting Y in context α, then $\mathbb{P}[Y \mid \alpha\!\uparrow] = 0$ and $\mathbb{P}[Y] < 1$.*
2. *Suppose Y is ω-regular. If $\mathbb{P}[Y] < 1$, then there is an $\alpha \in path_{fin}(\Sigma)$ such that $\mathbb{P}[Y \mid \alpha\!\uparrow] = 0$. Moreover, if $\alpha \in path_{fin}(\Sigma)$ with $\mathbb{P}[Y \mid \alpha\!\uparrow] = 0$ and after α only one bscc is reachable, there is a recurrent word γ refuting Y in context α.*

The assumption in 2 that Y is ω-regular cannot be dropped. Take the Markov chain with two states q, p, both being initial states. From any state, the next state is q with probability $1/3$ and p with probability $2/3$. On the one hand, it can be shown by the Borel-Cantelli Lemma that the property Y, i.e., that "at infinitely many positions, the number of previous p's equals the number of previous q's", has probability zero. On the other hand, there is no recurrent word γ refuting Y in some context α: a path in $\alpha\!\uparrow \cap\, Sat(\mathrm{G\,F}\gamma) \cap Y$ can be constructed by extending α, visiting γ infinitely often and, between the γ's, making the number of previous p's equal the number of previous q's. A similar example shows that the theorem rests on the assumption that Q is finite.

We conclude this section by comparing our notion of recurrent word γ in a context α with the periodic paths $\tilde{\alpha}(\tilde{\gamma})^{\omega}$ used as counterexamples in classical model checking. The pair (α,γ) describes the set of all infinite paths extending α and executing γ infinitely often, which has nonzero probability. The periodic path $\tilde{\alpha}(\tilde{\gamma})^{\omega}$ in general has probability zero. (The probability is nonzero only if $\tilde{\gamma}$ belongs to a "ring-like" bscc.) Even the set of all periodic paths has probability zero in general, because it is a countable set. In the probabilistic setting, counterexamples must have nonzero probability; therefore periodic paths are unsuitable as counterexamples.

4 Quantitative Counterexamples

In this section, we discuss quantitative statements. In the following, let $\Sigma = (Q, S, \rightarrow, v)$, $\mathbb{P}$ be a Markov chain, Φ an LTL formula and Y a property. The corresponding question for a counterexample (or a witness) can take one of the following four shapes:

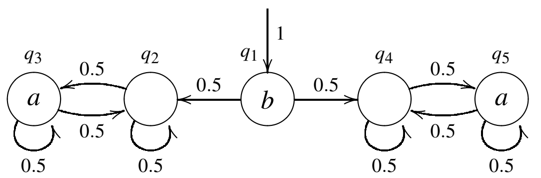
1. Why is $\mathbb{P}[\Phi] \leq t$? $(t < 1)$
2. Why is $\mathbb{P}[\Phi] \geq t$? $(t > 0)$
3. Why is $\mathbb{P}[\Phi] < t$? $(t > 0)$
4. Why is $\mathbb{P}[\Phi] > t$? $(t < 1)$

Questions 2 and 4 can be reduced to Questions 1 and 3, respectively, by negating the specification. Usually, quantitative probabilistic model checkers compute the probability of the specification. Hence, we know $\mathbb{P}[\Phi]$ before computing a counterexample, and can therefore reduce Question 3 to Question 1 by considering a bound between t and $\mathbb{P}[\Phi]$. We therefore restrict our attention to Question 1.

4.1 Presenting a Quantitative Counterexample

In some cases, a qualitative counterexample can be used as a quantitative counterexample. However, this is not always possible:

Example 4.1. Consider the Markov chain $\Sigma, \mathbb{P}$ below together with $\Phi = \mathrm{F}\,\mathrm{G}\,a$. Note that $\mathbb{P}[\Phi] = 0$. There is a recurrent word $\gamma = q_2$ refuting Φ in context $\alpha = q_1 q_2$. What does it tell us about the probability of Φ? As $\mathbb{P}[\Phi \mid \alpha\uparrow] = 0$, we have $\mathbb{P}[\Phi] \leq 1 - \mathbb{P}[\alpha\uparrow] = 0.5$. However, this does not answer the question of why is $\mathbb{P}[\Phi] \leq 0$. The problem is that the pair (α,γ) only provides information about one bscc, namely the left one, but a proof for $\mathbb{P}[\Phi] \leq 0$ must involve both bsccs. To overcome this problem, we will consider counterexamples with several recurrent words, so that different bsccs can be taken into account.



Definition 4.2. *A* recurrent set (of Σ) *is a set of recurrent words of* Σ*. Given a recurrent set R, a word* $x \in Q^{\omega}$ *is* R-fair (for Σ) *iff* $x \in path_{\omega}(\Sigma)$ *and for each* $\gamma \in R$ *either (i)* $x \vDash \mathrm{G}\,\mathrm{F}\,\gamma$ *or (ii) some prefix of x cannot be extended to a finite path of* Σ *with suffix* γ*. The set of R-fair paths is denoted as* $Fair_{\Sigma}(R)$*.*

Lemma 4.3. *Let R be a recurrent set. Then $\mathbb{P}[Fair_\Sigma(R)] = 1$.*

Proof. Let $\gamma \in R$. It can be checked that $Fair_\Sigma(\{\gamma\})$ is a fairness property according to [20,22]. Moreover, $Fair_\Sigma(\{\gamma\})$ is ω-regular. Varacca and Völzer [20] have shown that any ω-regular fairness property has probability one. The assertion then follows from the facts that R is countable and $Fair_\Sigma(R) = \bigcap_{\gamma \in R} Fair_\Sigma(\{\gamma\})$. $\square$

In the above example, consider $\alpha = q_1$ and the recurrent set $R = \{q_2, q_4\}$. Note that every R-fair run x violates the specification. Because of Lemma 4.3, $\mathbb{P}[\alpha\uparrow \cap Fair_\Sigma(R)] = \mathbb{P}[\alpha\uparrow] = 1$, and thus we have $\mathbb{P}[\Phi] \leq 1 - \mathbb{P}[\alpha\uparrow] = 0$. Together, α and R describe a set of paths violating Φ having probability 1. The prefix α describes how the violations begin. The recurrent set R describes what happens infinitely often in a violating path.

In the preceding example, R contains exactly one recurrent word for each bscc of the system, but in general it is possible that R contains no recurrent word or several recurrent words for some bscc. Consider, for instance, the specification $\Phi = \mathrm{G}\,\mathrm{F}\,b$; again $\mathbb{P}[\Phi] = 0$. A counterexample would be $\alpha = \lambda$ and $R = \{q_2\}$. In this case there are two kinds of R-fair paths: (i) paths going to the left bscc and visiting q_2 infinitely often; (ii) paths going to the right bscc, where q_2 can no longer be reached. As all R-fair paths violate Φ and $\mathbb{P}[Fair_\Sigma(R)] = 1$, we have $\mathbb{P}[\Phi] = 0$.

We now formalise this intuition.

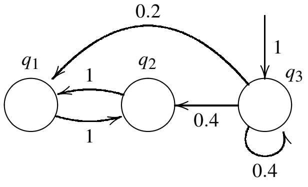
Definition 4.4
A recurrent set R refutes *Y* in context *$\alpha \in path_{fin}(\Sigma)$ iff $\alpha\uparrow \cap Fair_\Sigma(R) \cap Y = \varnothing$.*

Equivalently, R refutes Y in context α if every path of the system that extends α and is R-fair violates Y. In that case, Y is violated with at least the probability of $\alpha\uparrow$.

Corollary 4.5. *If there exists a recurrent set refuting Y in context α, then $\mathbb{P}[Y \mid \alpha\uparrow] = 0$, and therefore $\mathbb{P}[Y] \leq 1 - \mathbb{P}[\alpha\uparrow]$.*

It may also be necessary to consider several contexts:

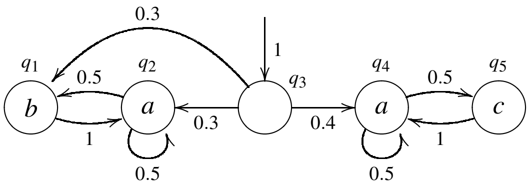
Example 4.6. Consider the Markov chain $\Sigma, \mathbb{P}$ below and $\Phi = q_3 \,\mathrm{U}\, q_2$.
To show that $\mathbb{P}[\Phi] \leq 0.7$, one context word α is not enough. For instance, any recurrent set refutes Φ in context q_3q_1, but $\mathbb{P}[q_3q_1\uparrow] = 0.2$. This counterexample only shows that $\mathbb{P}[\Phi] \leq 0.8$.



To gather enough weight, we need to use several contexts. For instance let $\alpha_1 = q_3q_1$, $\alpha_2 = q_3q_3q_1$ and $\alpha_3 = q_3q_3q_3q_1$. Clearly $\varnothing$ refutes Φ in context α_i. Then $\mathbb{P}[\Phi] \leq 1 - \mathbb{P}[\cup_i \alpha_i\uparrow]$. As the three sets are disjoint, $\mathbb{P}[\cup_i \alpha_i\uparrow] = 0.2 + 0.08 + 0.032 > 0.3$.

In this simple example, the recurrent sets do not matter. Different contexts in principle require different recurrent sets:

Example 4.7. Consider the Markov chain $\Sigma, \mathbb{P}$ below and $\Phi = \mathrm{G}\,\neg c \wedge (\mathrm{G}\,\mathrm{F}\,b \Rightarrow \mathrm{X}\,a)$. Let $\alpha_1 = q_3q_4$, $\alpha_2 = q_3q_1$ and $R_1 = \{q_5\}$, $R_2 = \{q_1\}$. Note that R_i refutes Φ in context α_i. First, any R_1-fair path extending q_3q_4 violates Φ, as it visits q_5. Second, any R_2-fair path extending q_3q_1 violates Φ, as it visits q_1 infinitely often,



but its second state does not satisfy a. Hence, $\mathbb{P}[\Phi] \le 1 - \mathbb{P}[\alpha_1\!\uparrow \cup \alpha_2\!\uparrow] = 0.3$.

This example also shows that whether a path almost certainly satisfies Φ depends not only on which bscc it visits; here, satisfaction also depends on the second state of the path. Therefore, in the case of general LTL properties, the bsccs cannot simply be partitioned into "accepting" and "rejecting".

If a recurrent set refutes a property in a context, a larger recurrent set will also do so. We can therefore suppose without loss of generality that all R_i are the same. In the above example, we can choose $R = R_1 \cup R_2$; then R refutes Φ in context α_i, $i = 1, 2$. Taking only one recurrent set is a design decision simplifying the theory. In practice, it might be desirable to have several recurrent sets.

Definition 4.8. *Let W be a set of finite paths of Σ. A recurrent set R refutes Y in context W iff $W\!\uparrow \cap \mathit{Fair}_\Sigma(R) \cap Y = \varnothing$.*

Corollary 4.9. *If there exists a recurrent set refuting Y in context W, then $\mathbb{P}[Y \mid W\!\uparrow] = 0$, and therefore $\mathbb{P}[Y] \le 1 - \mathbb{P}[W\!\uparrow]$.*

Thus, we present a quantitative counterexample explaining why $\mathbb{P}[\Phi] \le t$ by the sets W and R such that R is a recurrent set refuting Φ in context W and $\mathbb{P}[W\!\uparrow] \ge 1 - t$.

4.2 Completeness

Corollary 4.9 is a soundness result: if there is a recurrent set refuting Y in context W, then the property is violated with probability at least $\mathbb{P}[W\!\uparrow]$. It turns out that Definition 4.8 also gives us a complete representation of a counterexample: if a property is violated with some probability, there is a pair (W, R) witnessing it. In fact, there is a canonical set that can always be used as the context W.

Definition 4.10. *Let $I(\Sigma, Y)$ be the set of all $\alpha \in \mathit{path}_{\mathit{fin}}(\Sigma)$ such that there is a recurrent set refuting Y in context α. We call $I(\Sigma, Y)$ the* initial language (of Σ w.r.t. Y).

Note that $I(\Sigma, Y)$ by itself is a context, i.e., there is a recurrent set refuting Y in context $I(\Sigma, Y)$. To verify that, let R_α be the recurrent set refuting Y in context α, where $\alpha \in I(\Sigma, Y)$. Then $R := \bigcup_{\alpha \in I(\Sigma, Y)} R_\alpha$ refutes Y in context $I(\Sigma, Y)$.

Theorem 4.11. *For any LTL formula Φ, $I(\Sigma, \Phi)$ is regular.*

In Section 5.2, we will explain how to compute a finite automaton accepting $I(\Sigma, \Phi)$.

The next proposition states important properties of the initial language. Firstly, almost all elements of $I(\Sigma, Y)\!\uparrow$ are violations of Y. Moreover, if the property is given by an LTL formula Φ, almost all violations of Φ belong to $I(\Sigma, \Phi)\!\uparrow$. Hence, the probabilities of $\neg\Phi$ and $I(\Sigma, \Phi)\!\uparrow$ coincide.

Proposition 4.12

1. $\mathbb{P}[I(\Sigma,Y)\!\uparrow \cap Y] = 0$.
2. *For any LTL formula* Φ, $\mathbb{P}[I(\Sigma,\Phi)\!\uparrow \cup Sat(\Phi)] = 1$.
3. *For any LTL formula* Φ, $\mathbb{P}[I(\Sigma,\Phi)\!\uparrow] = \mathbb{P}[\neg\Phi]$.

We can now give some equivalent characterisations of the initial language. The first statement of Proposition 4.13 asserts that the initial language is the largest context in which a recurrent set refuting Y exists. The second statement provides an alternative definition of the initial language in terms of $\mathbb{P}$.

Proposition 4.13

1. *The initial language* $I(\Sigma,Y)$ *is the largest set* $W \subseteq path_{fin}(\Sigma)$ *such that there is a recurrent set refuting* Y *in context* W.
2. *For any LTL formula* Φ, $I(\Sigma,\Phi)$ *is the set of all* $\alpha \in path_{fin}(\Sigma)$ *so that* $\mathbb{P}[\Phi \mid \alpha\!\uparrow] = 0$.

Finally we prove completeness.

Theorem 4.14. *Let* Φ *be an LTL formula,* $0 \le t < 1$ *and* $\mathbb{P}[\Phi] \le t$. *Then there is a nonempty set* $W \subseteq path_{fin}(\Sigma)$ *such that* $\mathbb{P}[\Phi \mid W\!\uparrow] = 0$ *and* $\mathbb{P}[W\!\uparrow] \ge 1-t$. *Moreover, for any* $W \subseteq path_{fin}(\Sigma)$, $W \neq \varnothing$ *with* $\mathbb{P}[\Phi \mid W\!\uparrow] = 0$ *there is a recurrent set* R *refuting* Φ *in context* W.

We will see in Section 5.3 that the set R can be chosen to contain exactly one recurrent word per bscc. If the bound t is tight, i.e., $t = \mathbb{P}[\Phi]$, the context W is in general infinite. If $t > \mathbb{P}[\Phi]$, one can show – using standard results of measure theory – that it is always possible to choose W as a finite subset of $I(\Sigma,\Phi)$.

4.3 Interaction with the Model Checker

In this section, we discuss the interaction between user and model checker for quantitative counterexamples. The model checker computes $\mathbb{P}[\Phi]$ and presents $W \subseteq I(\Sigma,\Phi)$ such that $\mathbb{P}[W\!\uparrow] \ge t$, where t is given by the user. The user then inspects W, and may identify some $\alpha \in W$ for which she does not believe that $\mathbb{P}[\Phi \mid \alpha\!\uparrow] = 0$. To convince the user, the model checker computes a recurrent set R refuting Φ in context W, which contains at most one element for each bscc of the system (see Section 5.3). The interaction between user and model checker that follows is similar to the qualitative case. The user can challenge the following:

1. R is a recurrent set: each element $\gamma \in R$ can be checked as in the qualitative case.
2. R refutes Φ in context α: similar as in the qualitative case, the user interactively tries to construct a path in $\alpha\!\uparrow \cap Fair_\Sigma(R) \cap \Phi$ and fails. Note that the model checker can assure fairness while the user can concentrate on constructing a path that ultimately satisfies Φ. Once a bscc has been reached, the model checker can also output the $\gamma \in R$ associated with that bscc.

The set W may be too large or even infinite so that inspecting each element individually is not feasible (see [5,9]). This raises the question of how the user can understand what words are contained in W. Also, the reader may want evidence that indeed $\mathbb{P}[W\!\uparrow] \ge t$. Similar questions arise in the study of counterexamples for probabilistic CTL ([14]) model checking, and we refer to the literature for possible approaches [4,5,9,23]. We also discuss these issues further in Section 6.

5 Computing Counterexamples

In this section, we explain how the counterexamples defined above can be computed. Our algorithm is based on, but substantially complements an algorithm of Courcoubetis and Yannakakis [8][1]. We follow [18] in our presentation. In Section 5.1 we briefly recall the underlying model checking algorithm. In Sections 5.2 and 5.3 we address the computation of an automaton accepting the initial language and the computation of a recurrent set, respectively.

Throughout the entire section, $\Sigma = (Q, S, \rightarrow, v), \mathbb{P}$ is a Markov chain and Φ an LTL formula. Without loss of generality, we assume that Φ only contains the temporal connectives X and U.

It is well known that the assertion $\mathbb{P}[\Phi] = 1$ is independent from the underlying Markov measure $\mathbb{P}$ (see e.g. [20]). (It depends only on which transition probabilities are nonzero, which is uniquely determined by $\rightarrow$.) We therefore simply say that a formula Φ is *large (in* Σ*)* iff $\mathbb{P}[\Phi] = 1$.

5.1 Recalling Courcoubetis and Yannakakis

The algorithm presented in [8] works in steps. At each step, it eliminates one temporal operator from the specification and at the same time refines the system so that the largeness of the specification is preserved. After eliminating all operators, the specification becomes a state formula ϕ, for which largeness can easily be checked: ϕ is large iff all initial states satisfy ϕ. We now briefly recall how the transformation takes place.

If Φ is not a state formula, then it has a subformula of the form $\Theta = \psi \mathrm{U} \xi$ or $\Theta = \mathrm{X}\,\xi$, where ψ, ξ are state formulas. The algorithm chooses such a formula Θ and replaces it by a fresh atomic proposition d. We call the resulting formula Φ'.

The algorithm then partitions the set of states Q into three blocks Q^L_Θ, Q^S_Θ and Q^M_Θ. If the initial states of Σ are replaced by a state in Q^L_Θ, Θ becomes large. If the initial states of Σ are replaced by a state in Q^S_Θ, $\neg\Theta$ becomes large (Θ becomes "small"). If the initial states of Σ are replaced by a state in Q^M_Θ, neither Θ nor $\neg\Theta$ becomes large (Θ becomes "medium-sized").

The new system $\Sigma' = (Q', S', \rightarrow', v')$ has the set of states

$$Q' := Q^L_\Theta \times \{\Theta\} \quad \cup \quad Q^S_\Theta \times \{\neg\Theta\} \quad \cup \quad Q^M_\Theta \times \{\Theta, \neg\Theta\},$$

that is, the states in Q^L_Θ are annotated with Θ, the states in Q^S_Θ with $\neg\Theta$, and the states in Q^M_Θ are split into a copy with Θ and one with $\neg\Theta$. We denote the first projection as π so that, for instance, $\pi(q, \Theta) = q$. We extend π to words in the natural way. The initial states of the new system are the states that are projected to an initial state of the original system. The new valuation function v' is just like v, whereas d holds in the states annotated with Θ and only there. Finally, the transition relation of Σ' is defined so that Φ' is large in Σ' iff Φ is large in Σ (see [8,18]).

A single transformation step takes time $O(|\Sigma||\Phi|)$. Moreover, the size of Σ' is at most twice the size of Σ; hence, it can be shown that the overall complexity of the algorithm is $O(|\Sigma| 2^{|\Phi|})$.

[1] Note that we refer to the optimal algorithm in Section 3.1 of [8], and not to the automata-based algorithm in Section 4.1, which is non-optimal for LTL.

5.2 Computing the Initial Language

In this section, we explain how to compute a deterministic finite automaton (DFA) accepting $I(\Sigma, \Phi)$. The algorithm from 5.1 terminates after n transformation steps on Σ and Φ, resulting in the system Σ_n and state formula Φ_n. The n-fold projection on states and paths of Σ_n is denoted π^n, that is, π^n maps a state (path) of Σ_n to the corresponding state (path) of Σ. The following lemma shows how $I(\Sigma, \Phi)$ can be expressed in terms of $\mathit{Sat}(\Sigma_n, \Phi_n)$:

Lemma 5.1. *We have* $I(\Sigma, \Phi) = \mathit{path}_{\mathit{fin}}(\Sigma) \setminus \pi^n(\mathit{Sat}(\Sigma_n, \Phi_n)){\downarrow}$.

The elements of $\pi^n(\mathit{Sat}(\Sigma_n, \Phi_n)){\downarrow}$ are (modulo π^n) the finite paths of Σ_n starting in a state satisfying Φ_n. It is therefore straightforward to compute a non-deterministic finite automaton (NFA) accepting $\pi^n(\mathit{Sat}(\Sigma_n, \Phi_n)){\downarrow}$. It is also straightforward to compute a deterministic finite automaton (DFA) accepting $\mathit{path}_{\mathit{fin}}(\Sigma)$. By applying standard automata constructions, we obtain a DFA for $I(\Sigma, \Phi)$.

In Theorem 5.2, we provide the key points of our complexity analysis.

Theorem 5.2

1. *An NFA accepting* $\pi^n(\mathit{Sat}(\Sigma_n, \Phi_n)){\downarrow}$ *can be computed in time linear in* $|\Sigma|$ *and exponential in* $|\Phi|$.
2. *A DFA accepting* $\pi^n(\mathit{Sat}(\Sigma_n, \Phi_n)){\downarrow}$ *can be computed in time linear in* $|\Sigma|$ *and doubly exponential in* $|\Phi|$.
3. *A DFA accepting* $I(\Sigma, \Phi)$ *can be computed in time linear in* $|\Sigma|$ *and doubly exponential in* $|\Phi|$.

The overall running time is linear in $|\Sigma|$ and doubly exponential in $|\Phi|$, and we do not know whether an exponential algorithm can be found. In Section 5.3 we explain how to compute a single element of $I(\Sigma, \Phi)$ without computing the entire DFA; the running time of the latter approach is linear in $|\Sigma|$ and exponential in $|\Phi|$.

5.3 Computing a Recurrent Set

In this subsection, Σ' and Φ' denote the system and formula after one transformation step has been applied to Σ and Φ. Moreover, Θ is the subformula of Φ that has been replaced by the new atomic proposition d during the transformation.

We explain how to compute a recurrent set R refuting Φ in context $I(\Sigma, \Phi)$ and therefore in any context $W \subseteq I(\Sigma, \Phi)$. For each bscc K of Σ, our algorithm calls a function *computeRecurrentWord* to compute a path fragment $\gamma_K \in K^+$ such that $I(\Sigma, \Phi){\uparrow} \cap \mathit{Sat}(\mathrm{G\,F}\,\gamma_K) \cap \mathit{Sat}(\Phi) = \varnothing$. The result R is then defined as $R := \{\gamma_K \mid K \text{ bscc of } \Sigma\}$. Note that $\mathit{Fair}_\Sigma(R) = \bigcup_K \mathit{Sat}(\mathrm{G\,F}\,\gamma_K)$. Hence, $I(\Sigma, \Phi){\uparrow} \cap \mathit{Fair}_\Sigma(R) \cap \mathit{Sat}(\Phi) = \varnothing$, i.e., R refutes Φ in context $I(\Sigma, \Phi)$.

The function *computeRecurrentWord* is outlined in Figure 1. Correctness can be shown by induction over Φ.

Lemma 5.3

The function computeRecurrentWord terminates and establishes its postconditions.

We now explain how Lines 9-11 of Figure 1 can be implemented. Suppose $\Theta = \psi \,\mathrm{U}\, \xi$. Given γ' from Line 8, choose δ' minimal w.r.t. $\sqsubseteq$ such that the following holds:

```
1  **Precondition:** $\Sigma$ is a system, $\Phi$ a formula, $q$ a state of $\Sigma$.
2  **Postcondition:**
   (1) $\gamma$ is a finite path fragment of $\Sigma$ with first state $q$.
       (In particular, if $q$ belongs to the bscc $K$, then $\gamma \in K^+$.)
   (2) $I(\Sigma,\Phi)\!\uparrow \cap Sat(\mathrm{G\,F}\gamma) \cap Sat(\Phi) = \varnothing$.
3  **begin**
4      **if** $\Phi$ is a state formula **then**
5          $\gamma := q$;
6      **else**
7          **choose** a state $q'$ of $\Sigma'$ with $\pi(q') = q$;
8          $\gamma' := computeRecurrentWord(\Sigma', \Phi', q')$;
9          **choose** a finite path fragment $\gamma$ of $\Sigma$ such that
10         (1) for each path fragment $\tilde{\gamma}'$ of $\Sigma'$, if $\gamma = \pi(\tilde{\gamma}')$, then $\gamma' \sqsubseteq \tilde{\gamma}'$,
11         (2) for each $x' \in path_\omega(\Sigma')$, if $\pi(x') \models \mathrm{G\,F}\gamma$, then $x' \models \mathrm{G}(\Theta \Leftrightarrow d)$.
12     **end**
13 **end**
```

Fig. 1. Function: $\gamma =$ `computeRecurrentWord`(Σ, Φ, q)

1. $\gamma'\delta'$ is a finite path fragment of Σ'.
2. $\pi(\gamma'\delta')$ does not end in Q^M_Θ.
3. If $\pi(\gamma'\delta')$ visits a state satisfying d, then $\pi(\gamma'\delta')$ visits a state satisfying ξ.

Set $\gamma := \pi(\gamma'\delta')$.

It can be shown that from each state in Q^M_Θ both a state in Q^L_Θ satisfying ξ and a state in Q^S_Θ is reachable. An examination of the state relation of Σ' then yields that a δ' satisfying the above conditions exists and can therefore be computed by a breadth-first search.

Now suppose $\Theta = \mathrm{X}\,\xi$. Given γ' from Line 8, we construct γ as follows. If $\pi(\gamma')$ does not end in Q^M_Θ, we set $\gamma := \pi(\gamma')$. Otherwise, we extend γ' by one state q' to $\gamma' q' \in path_{fin}(\Sigma')$ and set $\gamma := \pi(\gamma' q')$.

A proof that γ satisfies the conditions in Lines 9-11 can be found in [19]. The running time of *computeRecurrentWord* is as follows:

Theorem 5.4. *Executing computeRecurrentWord*(Σ, Φ, q) *takes time* $O(|\Sigma||\Phi|2^{|\Phi|})$.

Proof. Let n be the number of transformation steps and Σ_i, $1 \le i \le n$, the system after the ith transformation step. The length of $\gamma = computeRecurrentWord(\Sigma, \Phi, q)$ is bounded by $O(\sum_{i=1}^n |\Sigma_i|)$, because the ith incarnation of *computeRecurrentWord* increases γ by at most $|\Sigma_i|$, $1 \le i \le n$. As $|\Sigma_i| \le |\Sigma| 2^i$, the length of γ is in $O(|\Sigma|2^{|\Phi|})$.

Computing γ from γ' includes reading γ' and computing some extension; both can be accomplished in time $O(|\Sigma|2^{|\Phi|})$. This has to be repeated n times; hence the overall running time is $O(|\Sigma||\Phi|2^{|\Phi|})$. □

The function *computeRecurrentWord* has to be executed once for each bscc of Σ; accordingly the overall running time is linear in the number of bsccs and $|\Sigma|$ and exponential in $|\Phi|$.

Note that the user does not need to compute the entire recurrent set at once. Instead, after computing one recurrent word, she can already inspect the bscc of the recurrent word. If she then wants to find an error in a different bscc, she can compute a recurrent word of that bscc. Thus, although the worst-case running time is quadratic in the size of the system, the user already obtains the first diagnostic feedback after $O(|\Sigma||\Phi|2^{|\Phi|})$ steps.

The function *computeRecurrentWord* can be adapted to compute a single element α of $I(\Sigma,\Phi)$, whereas the complexity remains the same. The details can be found in [19].

Theorem 5.5

If $I(\Sigma,\Phi) \neq \varnothing$, a single element of $I(\Sigma,\Phi)$ can be computed in $O(|\Sigma||\Phi|2^{|\Phi|})$ steps.

Theorems 5.5 and 5.4 mean that a representation of a qualitative counterexample can be computed in time linear in the system and exponential in the specification. This running time is optimal, because it is also the running time of the optimal probabilistic model checking algorithm in [8].

6 Conclusions and Related Work

We have proposed a way of presenting and computing counterexamples in probabilistic LTL model checking for Markov chains. Our notion is sound and complete, which means that a counterexample in our sense can be computed if and only if the specification is not met with the desired probability. We have also pointed out how such a counterexample can be utilised to find an error in the system.

Aljazzar and Leue [2] propose solutions for counterexamples in probabilistic model checking with respect to timed probabilistic reachability properties in Markov chains. Han and Katoen [12] and Wimmer *et al.* [23] present algorithms computing counterexamples for model checking PCTL (probabilistic CTL [14]) formulas in Markov chains. There are also suggestions of how to present such counterexamples to the user [4,9]. In [1,13], the problem has been tackled for continuous time Markov chains. In [3] Aljazzar and Leue generalise their proposal in [2] for (unnested, upwards-bounded) PCTL formulas and Markov decision processes.

Recently, Andrés *et al.* [5] proposed an approach for LTL formulas on Markov chains (and also Markov decision processes). They refer to the fact that probabilistic model checking of an LTL formula in a Markov chain M_1 can be reduced to probabilistic model checking of an upwards-bounded reachability property in a generated Markov chain M_2, which is doubly exponentially larger than M_1 in the size of the LTL formula [10]. Then they develop a counterexample representation in the style of Han and Katoen [12], which can be mapped to a subset of the initial language in M_1. The authors propose an interesting way of convincing the user that the upwards-bounded reachability property is indeed violated in the generated Markov chain M_2. However, in contrast to our approach, they do not address how to convince the user of the probability of the original LTL formula in the original system M_1.

The above approaches [2,4,5,9,12,23] have in common that a counterexample is *finitary*, i.e., a set of finite paths W so that any path of the system extending W violates the specification. In our terminology, W is a subset of the initial language. We have pointed out in Section 3.1 that sets of finite paths are not sufficient to refute general LTL properties – in particular liveness properties. Even so, the techniques of presenting finitary counterexamples to the user can be applied to what we have called a context *W* in our counterexample presentation. In future work, it would be interesting to combine these techniques with our approach. Another important direction is to carry out some case studies to evaluate the interaction between user and model checker.

Acknowledgement. We thank Husain Aljazzar, Christian Dax, Barbara Jobstmann and Felix Klaedtke for useful discussions and helpful comments. We also thank the reviewers for suggesting improvements to the paper.

References

1. Aljazzar, H., Hermanns, H., Leue, S.: Counterexamples for timed probabilistic reachability. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 177–195. Springer, Heidelberg (2005)
2. Aljazzar, H., Leue, S.: Extended directed search for probabilistic timed reachability. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 33–51. Springer, Heidelberg (2006)
3. Aljazzar, H., Leue, S.: Counterexamples for model checking of Markov decision processes. Tech. Report soft-08-01, University of Konstanz, Germany (2007)
4. Aljazzar, H., Leue, S.: Debugging of dependability models using interactive visualization of counterexamples. In: QEST 2008, pp. 189–198. IEEE, Los Alamitos (2008)
5. Andrés, M., D'Argenio, P., van Rossum, P.: Significant diagnostic counterexamples in probabilistic model checking. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 129–148. Springer, Heidelberg (2008)
6. Breiman, L.: Probability. Addison Wesley, Reading (1968)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (2001)
8. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM 42(4), 857–907 (1995)
9. Damman, B., Han, T., Katoen, J.-P.: Regular expressions for PCTL counterexamples. In: QEST 2008, pp. 179–188. IEEE, Los Alamitos (2008)
10. de Alfaro, L.: Temporal logics for the specification of performance and reliability. In: Reischuk, R., Morvan, M. (eds.) STACS 1997. LNCS, vol. 1200, pp. 165–176. Springer, Heidelberg (1997)
11. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, vol. B, ch. 16, pp. 995–1072. Elsevier Science, Amsterdam (1990)
12. Han, T., Katoen, J.-P.: Counterexamples in probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 72–86. Springer, Heidelberg (2007)
13. Han, T., Katoen, J.-P.: Providing evidence of likely being on time: Counterexample generation for CTMC model checking. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 331–346. Springer, Heidelberg (2007)
14. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Asp. Comput. 6(5), 512–535 (1994)
15. Kemeny, J.G., Snell, J.L., Knapp, A.W.: Denumerable Markov Chains. Springer, Heidelberg (1976)
16. Pnueli, A.: The temporal logic of programs. In: FOCS 1977, pp. 46–57. IEEE, Los Alamitos (1977)

17. Ravi, K., Bloem, R., Somenzi, F.: A comparative study of symbolic algorithms for the computation of fair cycles. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 143–160. Springer, Heidelberg (2000)
18. Schmalz, M.: Extensions of an algorithm for generalised fair model checking. Diploma Thesis, Lübeck, Germany (2007), http://www.infsec.ethz.ch/people/mschmalz/dt.pdf
19. Schmalz, M., Völzer, H., Varacca, D.: Counterexamples in probabilistic LTL model checking for Markov chains. Technical Report 627, ETH Zürich, Switzerland (2009), http://www.inf.ethz.ch/research/disstechreps/techreports
20. Varacca, D., Völzer, H.: Temporal logics and model checking for fairly correct systems. In: LICS 2006, pp. 389–398. IEEE, Los Alamitos (2006)
21. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state programs. In: FOCS 1985, pp. 327–338. IEEE, Los Alamitos (1985)
22. Völzer, H., Varacca, D., Kindler, E.: Defining fairness. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 458–472. Springer, Heidelberg (2005)
23. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time Markov chains using bounded model checking. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 366–380. Springer, Heidelberg (2009)

Query-Based Model Checking of Ad Hoc Network Protocols

Anu Singh, C.R. Ramakrishnan, and Scott A. Smolka

Department of Computer Science, Stony Brook University,
Stony Brook, NY 11794-4400, USA
{anusingh,cram,sas}@cs.sunysb.edu

Abstract. A prominent source of complexity in the verification of ad hoc network (AHN) protocols is the fact that the number of network topologies grows exponentially with the square of the number of nodes. To combat this *instance explosion* problem, we present a query-based verification framework for AHN protocols that utilizes symbolic reachability analysis. Specifically we consider AHN nodes of the form $P : I$, where P is a process and I is an interface: a set of groups, where each group represents a multicast port. Two processes can communicate if their interfaces share a common group. To achieve a *symbolic* representation of network topologies, we treat process interfaces as variables and introduce a constraint language for representing topologies. Terms of the language are simply conjunctions of *connection* and *disconnection* constraints of the form $conn(\mathcal{J}_i, \mathcal{J}_j)$ and $dconn(\mathcal{J}_i, \mathcal{J}_j)$, where $\mathcal{J}_i$ and $\mathcal{J}_j$ are interface variables. Our *symbolic reachability algorithm* explores the symbolic state space of an AHN in breadth-first order, accumulating topology constraints as multicast-transmit and multicast-receive transitions are encountered. We demonstrate the practical utility of our framework by applying it to the problem of detecting unresolved collisions in the LMAC protocol for sensor networks.

1 Introduction

An *ad-hoc network* (AHN) is a local area network (LAN) that is built spontaneously as wireless devices connect. Instead of relying on a base station to coordinate the flow of messages between nodes in the network, individual nodes forward packets to and from each other. Because of its ah-hoc nature, an n-node AHN can assume any one of the possible $O(2^{n^2})$ topologies. A number of network protocols have been developed for AHNs, including routing, MAC-layer, and transport protocols.

Due to the vast space of possible network topologies, the verification of AHN protocols is a computationally intensive if not intractable task. Consider, for example, the verification of the LMAC medium access control [13] protocol performed in [5]. (We also consider this protocol in Section 6.) The approach taken in [5] was to separately model check each of the possible network topologies (modulo isomorphism) for a fixed number of nodes in order to detect those that might lead to *unresolved collisions*. An unresolved collision occurs when neighboring nodes (connected by at most two links) without a common neighbor attempt to transmit within the same time slot; due to the lack of a common neighbor, the collision remains undetected. The problem with this approach is that as the number of nodes in the network grows, the number of possible topologies grows exponentially, posing an *instance explosion* problem for verification.

To combat instance explosion, we present in this paper a new, constraint-based *symbolic verification technique for ad-hoc network protocols*. The basic idea behind our approach is as follows. As in [11], we represent AHNs as a collection of nodes of the form $P : I$, where P is a sequential process and I is an *interface*. An interface is a set of *groups*, with each group corresponding to a clique in the network topology. Dually, a group is used as a local-broadcast (multi-cast) communication port. Two nodes in the network can communicate (are within each other's transmission range) only if there respective interfaces have a non-null intersection (share a common group).

To achieve a *symbolic* representation of an AHN, we treat process interfaces as variables and introduce a constraint language for representing topologies. Terms of the language are simply conjunctions of *connection* and *disconnection constraints* of the form $conn(\mathcal{I}_i, \mathcal{I}_j)$ and $dconn(\mathcal{I}_i, \mathcal{I}_j)$, respectively. Here, $\mathcal{I}_i$ and $\mathcal{I}_j$ are interface variables, and $conn(\mathcal{I}_i, \mathcal{I}_j)$ signifies that $\mathcal{I}_i$ and $\mathcal{I}_j$ are connected ($\mathcal{I}_i \cap \mathcal{I}_j \neq \emptyset$), while $dconn(\mathcal{I}_i, \mathcal{I}_j)$ means that $\mathcal{I}_i$ and $\mathcal{I}_j$ are disconnected ($\mathcal{I}_i \cap \mathcal{I}_j = \emptyset$). As such, each term of the language symbolically represents a *set* of possible topologies.

Given this symbolic representation of AHNs, one can now ask *model-checking queries* of the form: under what evaluations (i.e. topologies) of the symbolic interface variables does the reachability property in question hold? Our *symbolic reachability algorithm* explores the symbolic state space of an AHN. A *symbolic state* is a pair of the form (s, γ), where s is a network state comprising both the locations of the component processes and valuations of their local variables, and γ is a term from our topology constraint language. A symbolic transition from (s, γ) to (s', γ') is constructed by adding constraints to γ to obtain γ' whenever a communication (local broadcast) occurs. Assuming the communication involves process P_i as the broadcaster, the following constraints will be added: those of the form $conn(\mathcal{I}_i, \mathcal{I}_j)$, where P_j is a process capable of performing a corresponding receive action and deemed to fall within the transmission range of P_i; and those of the form $dconn(\mathcal{I}_i, \mathcal{I}_k)$, where P_k is also a process capable of performing a receive action and deemed *not* to fall within P_i's transmission range.

We describe an efficient symbolic reachability algorithm to verify reachability properties of symbolic AHNs. We moreover show that our symbolic reachability algorithm can be extended without major modification to query-based model checking of LTL properties. To demonstrate the practical utility of our symbolic verification technique for AHN protocols, we applied it to the problem of detecting unresolved collisions in the above-described LMAC protocol [13]. Our results show that our symbolic approach to query-based model checking is highly effective: in the case of a 6-node network, our symbolic reachability algorithm explored only 2,082 symbolic topologies, compared to a possible 32,768 actual topologies. Moreover, all 2,082 symbolic topologies were considered *in a single verification run*. In contrast, for the same property, the authors of [5] considered no more than a 5-node network, using 61 separate verification runs, one for each unique (modulo isomorphism) concrete topology.

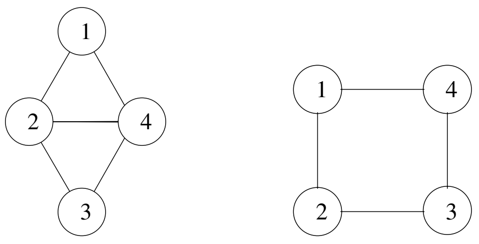
Main Contributions. The rest of the paper is organized around our main technical results, which include the following:

– Section 4 presents our modeling framework for AHNs, its concrete and symbolic semantics, and a correspondence result relating the two semantics.
– Section 5 considers our query-based verification technique based on symbolic reachability analysis, and its extension to LTL properties.
– Section 6 illustrates the practical utility of our technique by analyzing a formal model of the LMAC [13] protocol, a MAC layer protocol for sensor networks.

Additionally, Section 2 discusses our concrete and symbolic representations for AHN network topologies, Section 3 describes related work, and Section 7 offers our concluding remarks and directions for future work. Due to space limitations, complete proofs are omitted.

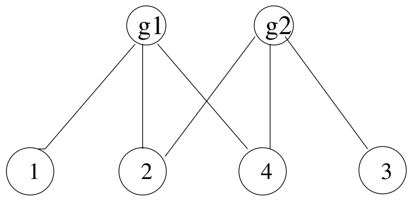
2 An Example of Topologies and Topology Constraints

Below we illustrate the use of a constraint language for representing sets of network topologies. In the LMAC protocol of [13], which is used to allocate transmission slots in a sensor network MAC layer, collision, i.e. simultaneous transmission between two nodes with overlapping ranges, is detected by neighbors common to both nodes. Fig. 1(a) shows a network topology for which a collision between nodes 1 and 2 can be



(a) Topology with detected collision (b) Topology with undetected collision

Fig. 1. Example topologies for collision and collision-detection in the LMAC protocol



(a) Group-Based View

AHN $\Pi_{1\leq i\leq 4} P_i : I_i$
$I_1 = \{g1\}$
$I_2 = \{g1, g2\}$
$I_3 = \{g2\}$
$I_4 = \{g1, g2\}$

(b) Concrete Representation of Interfaces

AHN $\Pi_{1\leq i\leq 4} P_i : \mathcal{J}_i$
$conn(\mathcal{J}_1, \mathcal{J}_2)$
$conn(\mathcal{J}_1, \mathcal{J}_4)$
$conn(\mathcal{J}_2, \mathcal{J}_3)$
$conn(\mathcal{J}_2, \mathcal{J}_4)$
$conn(\mathcal{J}_3, \mathcal{J}_4)$
$dconn(\mathcal{J}_1, \mathcal{J}_3)$

(c) Symbolic Representation of Interfaces

Fig. 2. Concrete and symbolic views of network topology of Fig. 1(a)

detected due to the presence of a common neighbor (node 4). Fig. 1(b) shows a topology for which a collision between 1 and 2 remains undetected since they do not share a neighbor.

As described in Section 1, we consider AHN nodes of the form $P : I$, where P is a process and I is an *interface*. Further, an interface is a set of *groups*, with each group g representing a shared communication channel and dually corresponding to a clique in the network topology [11]. Figs. 2(a) and 2(b) provide a group-based view and concrete representation based on process interfaces of the network topology of Fig. 1(a). A symbolic representation of the same topology is given in Fig. 2(c) using connection (*conn*) and disconnection (*dconn*) constraints over interface variables $\mathcal{J}_1$–$\mathcal{J}_4$, as mentioned in Sec. 1. The language in which symbolic topology constraints is expressed is formally described in Section 4. The symbolic representation permits us to compactly represent *sets* of topologies. For instance, consider the constraint $conn(\mathcal{J}_1, \mathcal{J}_2) \wedge conn(\mathcal{J}_1, \mathcal{J}_4) \wedge conn(\mathcal{J}_2, \mathcal{J}_3) \wedge conn(\mathcal{J}_3, \mathcal{J}_4)$. This represents topologies that contain edges $(1, 2), (1, 4), (2, 3)$ and $(3, 4)$. The topologies in this set may or may not contain edges $(1, 3)$ and/or $(2, 4)$. Hence the above constraint represents four 4-node topologies, including the ones in Fig. 1. We use topology constraints when constructing a symbolic verification proof (by reachability or model checking) to consider a set of topologies simultaneously. These constraints may get refined as needed as we progress in the proof, corresponding to case splits among the set of topologies. The constraint representation and lazy case-splitting enable us to consider a large number of topologies simultaneously within a single verification run.

3 Related Work

Our symbolic approach to query-based model checking of AHN protocols can be considered a form of *constraint-based model checking*. Traditionally this technique has been used for the verification of infinite-state systems [4,10], data-independent systems [12], systems with non-linear arithmetic constraints [3], timed automata [7], and imperative infinite-state programs [6]. In these works, constraints were used to compactly represent sets of states of a system being verified. In contrast to these, our approach uses variables in the system specification (to represent interconnections) and finds their valuations (in this case, topologies) for which a property holds. In this sense, our approach is closely related to temporal logic query checking, introduced in [2], which addresses the following problem: given a Kripke structure and a temporal logic formula with a placeholder, determine all propositional formulas ϕ such that when ϕ is inserted in the placeholder, the resulting temporal logic formula is satisfied by the Kripke structure. Query checking has been extended in a number of ways, including query checking of a wide range of temporal logics using a new class of alternating automata [1]; the application of query checking to a variety of model exploration tasks, ranging from invariant computation to test case generation [9]; and its adaptation to solving temporal queries in which formulas may contain integer variables [15].

Recently, symbolic representation of the set topologies has been used in [8] to analyze ad hoc networks. The constraint language in that work can only express the presence of connections between nodes, and not the absence of connections, in contrast to our work. It should be noted that the undetected collision problem in the LMAC protocol (see Section 6) is due to absence of connections, and cannot be detected using the constraint language of [8].

As mentioned in Section 1, the correctness of the 4-node and 5-node LMAC protocol [13] has been previously established in [5] using the UPPAAL model checker for timed automata. By systematically considering all 11 topologies for the 4-node case and all 61 topologies for the 5-node case (modulo isomorphism), they report all network topologies for which collisions may remain undetected in the LMAC protocol. They also iteratively improve the protocol model so that the number of topologies for which the protocol may fail is reduced. In contrast, our query-based approach verifies a property related to unresolved collisions using a single symbolic reachability run, thereby allowing us to additionally consider the 6-node case.

4 Modeling Framework

4.1 Syntax

We formally define the syntax and semantics of our framework. Systems in our framework are modeled as composition of *nodes*. Following the notion of separation of a node's communication and computation behavior presented in the ω-calculus [11], we consider a node to consist of a *process* (computational behavior) and an *interface* (communication capability). We present the notations used in defining our framework, followed by formal definitions of the components of our framework, namely a process, an interface, a node, and a system.

Let $\mathcal{D}$ be a non-empty *domain* with a set of *operations* F and *relations* R defined over it, and $\mathcal{V}ar$ be a countable set of *variables* over domain $\mathcal{D}$. For instance, $\mathcal{D}$ may be a set of finite integers, with F containing arithmetic operations, and R comprising equality, dis-equality and relational operations over integers. Symbols x, y (possibly subscripted) range over elements of $\mathcal{V}ar$. An *environment* $\theta : X \mapsto \mathcal{D}$, where $X \subseteq \mathcal{V}ar$ is a mapping from variables in $\mathcal{V}ar$ to values in domain $\mathcal{D}$. Symbol Θ is used to denote the set of all environments over $\mathcal{V}ar$ and $\mathcal{D}$. We use $\mathcal{E}$ to denote the set of *expressions*, which are terms over elements of $\mathcal{D} \cup \mathcal{V}ar \cup F$. Expressions are represented by symbol e (possibly subscripted). A *primitive condition* is a term with a symbol from R whose arguments are elements of $\mathcal{E}$. A *condition* is a conjunction of primitive conditions. An *assignment* is of the form $x := e$, where $x \in \mathcal{V}ar$ and $e \in \mathcal{E}$. Following traditional programming language semantics, we use $[\![.]\!]$ to represent semantics for expressions, conditions and assignments. For an expression e, condition *cond*, and assignment *asgn*, $[\![e]\!] : \Theta \mapsto \mathcal{D}$, $[\![cond]\!] : \Theta \mapsto Bool$, and $[\![asgn]\!] : \Theta \mapsto \Theta$ are mappings from an environment to domain $\mathcal{D}$, $Bool = \{\, \mathbf{true}\ ,\ \mathbf{false}\,\}$, and an environment, respectively. Semantics of a single assignment can be extended to a set of simultaneous assignments in the standard way.

The syntactic definition of a process is as follows.

Definition 1 (Process). *A* process = $\langle L, X, \Sigma, \delta, l_0, \eta_0 \rangle$, *is an extended finite state automaton over domain* $\mathcal{D}$, *where:*

- *L is a finite set of* locations.
- *$X \subseteq \mathcal{V}ar$ is a set of* local variables *for the process.*
- *Σ is a finite set of* action labels *containing*
  - $\mathbf{b}\ e$*, $e \in \mathcal{E}$ (*broadcast *action).*
  - $\mathbf{r}\ (x)$*, $x \in X$ (*receive *action).*
- *δ is a finite set of* transitions. *A transition is a tuple $(l, \alpha, l', \langle \rho, \eta \rangle)$, where*
  - *$l, l' \in L$ are* source *and* target *locations, respectively.*
  - *$\alpha \in \Sigma$ is an* action label.
  - *ρ, a condition, is a* transition guard.
  - *η is a set of* simultaneous assignments *of the form $x_1 := e_1, \ \ldots, \ x_n := e_n$, where the x_i are pairwise distinct.*
- *$l_0 \in L$ is the* start location.
- *η_0 is the set of* initial assignments *of the form $x := c$, $\forall x \in X$, and $c \in \mathcal{D}$.*

In the above definition of a process, we require that a variable that is used in a receive transition should not be assigned in the same transition.

An *interface*, represented by symbol I (possibly subscripted), is a finite set of names called *group names*. Group names are denoted by symbol g (possibly subscripted). We use $\mathcal{I}$ to denote the set of all interfaces. A *node* $P : I$ denotes a process P with interface I. Henceforth we use $\mathbf{n}$ to denote the set $\{1, \ldots, n\}$, and $P_i, i \in \mathbf{n}$, to denote the process $\langle L_i, X_i, \Sigma, \delta_i, l_{0,i}, \eta_{0,i} \rangle$ over domain $\mathcal{D}$.

Definition 2 (Ad Hoc Network, AHN). *For $i \in \mathbf{n}$, $P_i = \langle L_i, X_i, \Sigma, \delta_i, l_{0,i}, \eta_{0,i} \rangle$ s.t. $X_i \subseteq \mathcal{V}ar$ are pairwise disjoint, then $\Pi_{i \in \mathbf{n}} P_i : I_i$ is an* AHN.

4.2 Concrete Semantics

We provide a labeled transition system (LTS) based semantics for AHNs. An LTS is a 4-tuple $(S, \mathit{Act}, \longrightarrow, s_0)$, where S is a set of states, Act is a set of labels, $\longrightarrow \subseteq S \times \mathit{Act} \times S$ is a ternary relation of labeled transitions, and $s_0 \in S$ is the initial state. A labeled transition $(s, \alpha, t) \in \longrightarrow$, is also represented as $s \xrightarrow{\alpha} t$.

Definition 3 (Semantics of an AHN). *The semantics of an AHN $\Pi_{i \in \mathbf{n}} P_i : I_i$, denoted as $[\![\Pi_{i \in \mathbf{n}} P_i : I_i]\!]$, is the LTS $(S, \mathit{Act}, T, s_0)$ such that:*

- *$S = \overline{L} \times \Theta \times \Gamma$, where $\overline{L} = L_1 \times \ldots \times L_n$, Θ is the set of all possible environments $X \mapsto \mathcal{D}$, $X = X_1 \uplus \cdots \uplus X_n$.*
- *$\mathit{Act} = \{\mathbf{b}\ v \mid v \in \mathcal{D}\}$.*
- *$\rightsquigarrow$ is such that $(\bar{l}, \theta, \gamma) \overset{\mathbf{b}\ v}{\rightsquigarrow} (\bar{l}', \theta', \gamma')$, where $\bar{l} = (l_1, \ldots, l_n)$, $\bar{l}' = (l'_1, \ldots, l'_n)$, $\theta' = [\![\eta]\!]\theta$, $v = [\![e]\!]\theta$ if:*
  - *$\exists i \in \mathbf{n}$: $(l_i, \mathbf{b}\ e, l'_i, \langle \rho_i, \eta_i \rangle) \in \delta_i$, and*
  - *$\mathbf{k} = \{k | (l_k, \mathbf{r}\ (x_k), l'_k, \langle \rho_k, \eta_k \rangle) \in \delta_k, k \in \mathbf{n}, k \neq i\}$, $\exists \mathbf{k_c}, \mathbf{k_d} : \mathbf{k} = \mathbf{k_c} \uplus \mathbf{k_d}$ such that:*
    - *$\forall j \in \mathbf{n} \setminus (\mathbf{k_c} \cup \{i\})$: $l'_j = l_j$*
    - *$\rho = \rho_i \wedge \bigwedge_{k \in \mathbf{k_c}} \rho_k$, $[\![\rho]\!]\theta$ is $\mathbf{true}$*
    - *$\eta = \eta_i \cup \bigcup_{k \in \mathbf{k_c}} \eta_k[v/x_k] \cup \{x_k := v\}$*
    - *$\gamma' = \gamma \wedge \bigwedge_{k \in \mathbf{k_c}} conn(\mathcal{J}_i, \mathcal{J}_k) \wedge \bigwedge_{k \in \mathbf{k_d}} dconn(\mathcal{J}_i, \mathcal{J}_k)$ is satisfiable*
- *$s_0 = (\bar{l}_0, \theta_0, \mathbf{true})$, where $\bar{l}_0 = \langle l_{0,1}, \ldots, l_{0,n} \rangle$, $\theta_0 = [\![\bigcup_{i \in \mathbf{n}} \eta_{0,i}]\!]\theta_\epsilon$, and θ_ϵ is the empty environment.*

In the description of the transition relation ($\longrightarrow$) in Definition 3, i denotes the index of a process capable of performing a broadcast ($\mathbf{b}\ e$) action, and $\mathbf{k}$ denotes the set of indices of processes that are able to receive a value broadcast by process P_i. Note that processes not participating in the synchronization remain in the same location. For a transition to be enabled, the guards of synchronizing processes must be true. When a transition is taken, the value transmitted by the broadcaster is propagated to all receivers, and the assignments of the participating processes are performed.

4.3 Symbolic System Specification

We define a symbolic semantics for AHNs in which process interfaces are treated as variables. For example, for a node $P : I$, I is treated as a variable in contrast to the concrete semantics, where I represents a set of group names. We use $\mathbf{J}$ to denote the set of interface variables and $\mathcal{J}$ (possibly subscripted) to denote elements of $\mathbf{J}$.

Topology Constraint Language. Constraints on process interface variables are given by the following grammar. Symbol Γ represents the constraint language and γ (possibly subscripted) represents elements of Γ.

$$\Gamma \ ::= \ \mathbf{true} \ \mid \ \mathbf{false} \ \mid \ \mathit{conn}(\mathbf{J}, \mathbf{J}) \ \mid \ \mathit{dconn}(\mathbf{J}, \mathbf{J}) \ \mid \ \Gamma \wedge \Gamma$$

A valuation $\vartheta : \mathbf{J} \rightarrow \mathcal{I}$ maps an interface variable $\mathcal{J}$ to an interface I. A valuation ϑ is a model of a constraint γ, written as $\vartheta \models \gamma$, defined as follows: $\vartheta \models \mathbf{true}$

$\vartheta \not\models \mathbf{false}$
$\vartheta \models \mathit{conn}(\mathcal{J}_1, \mathcal{J}_2)$ if $\vartheta(\mathcal{J}_1) \cap \vartheta(\mathcal{J}_2) \neq \emptyset$
$\vartheta \models \mathit{dconn}(\mathcal{J}_1, \mathcal{J}_2)$ if $\vartheta(\mathcal{J}_1) \cap \vartheta(\mathcal{J}_2) = \emptyset$
$\vartheta \models \Gamma_1 \wedge \Gamma_2$ if $\vartheta \models \Gamma_1 \wedge \vartheta \models \Gamma_2$

A constraint of the form $\mathit{conn}(\mathcal{J}_1, \mathcal{J}_2)$ requires that nodes with interface variables $\mathcal{J}_1$ and $\mathcal{J}_2$ be connected, enabling them to communicate with each other. Constraint $\mathit{dconn}(\mathcal{J}_1, \mathcal{J}_2)$ requires nodes with interface variables $\mathcal{J}_1$ and $\mathcal{J}_2$ to be disconnected. A constraint γ is *satisfiable*, if there exists an interface valuation ϑ that assigns each interface variable in γ a value (set of group names) such that $\vartheta \models \gamma$. Two constraints γ_1 and γ_2 are *equivalent ($\equiv$)* if for every valuation ϑ s.t. $\vartheta \models \gamma_1$, it holds that $\vartheta \models \gamma_2$, and vice-versa.

Proposition 1. *Satisfiability of topology constraints is decidable.*

Proof Sketch: The following procedure determines the satisfiability of conjunction of primitive constraints over interface variables, and returns a satisfying assignment if there exists one.
Consider a constraint γ over interface variables $\mathcal{J}_1, \ldots, \mathcal{J}_n$.

- Step 1: For every constraint of the form $\mathit{conn}(\mathcal{J}_i, \mathcal{J}_j)$, add a fresh name g_{ij} to $\mathcal{J}_i$ and $\mathcal{J}_j$ (so that $\mathcal{J}_i \cap \mathcal{J}_j \neq \emptyset$).
- Step 2: For every $\mathcal{J}_i$ that is not assigned a value in Step 1, initialize $\mathcal{J}_i$ to singleton set $\{g_i\}$, such that g_i has not been assigned to any interface variable in Step 1.

– Step 3: For every constraint of the form $dconn(\mathcal{J}_i, \mathcal{J}_j)$, if $\mathcal{J}_i \cap \mathcal{J}_j = \emptyset$, then constraint γ is satisfiable, otherwise γ is unsatisfiable.

This procedure terminates and if γ is satisfiable, returns one satisfying assignment. □

For example, solution to the constraint $conn(\mathcal{J}_1, \mathcal{J}_2) \wedge conn(\mathcal{J}_1, \mathcal{J}_4) \wedge conn(\mathcal{J}_2, \mathcal{J}_3) \wedge conn(\mathcal{J}_3, \mathcal{J}_4)$, is $\mathcal{J}_1 = \{g_{1,2}, g_{1,4}\}, \mathcal{J}_2 = \{g_{1,2}, g_{2,3}\}, \mathcal{J}_3 = \{g_{2,3}, g_{3,4}\}, \mathcal{J}_4 = \{g_{1,4}, g_{3,4}\}$.

A symbolic AHN is an AHN for which topology is represented using interface variables.

Definition 4 (Symbolic AHN). *For $i \in \mathbf{n}$, $P_i = \langle L_i, X_i, \Sigma, \delta_i, l_{0,i}, \eta_{0,i}\rangle$ s.t. $X_i \subseteq \mathcal{V}ar$ are pairwise disjoint, then $\Pi_{i\in\mathbf{n}} P_i : \mathcal{J}_i$ is a* symbolic AHN.

Definition 5 (Semantics of a symbolic AHN). *The semantics of a symbolic AHN $\Pi_{i\in\mathbf{n}} P_i : \mathcal{J}_i$, denoted as $[\![\Pi_{i\in\mathbf{n}} P_i : \mathcal{J}_i]\!]$, is the symbolic LTS (S, Act, T, s_0), such that:*

- *$S = \overline{L} \times \Theta \times \Gamma$, where $\overline{L} = L_1 \times \ldots \times L_n$, Θ is the set of all possible environments $X \mapsto \mathcal{D}$, $X = X_1 \uplus \cdots \uplus X_n$.*
- *$Act = \{\mathbf{b}\ v \mid v \in \mathcal{D}\}$.*
- *$\rightsquigarrow$ is such that $(\bar{l}, \theta, \gamma) \overset{\mathbf{b}\ v}{\rightsquigarrow} (\bar{l}', \theta', \gamma')$, where $\bar{l} = (l_1, \ldots, l_n)$, $\bar{l}' = (l'_1, \ldots, l'_n)$, $\theta' = [\![\eta]\!]\theta$, $v = [\![e]\!]\theta$ if:*
  - *$\exists i \in \mathbf{n}$: $(l_i, \mathbf{b}\ e, l'_i, \langle \rho_i, \eta_i\rangle) \in \delta_i$, and*
  - *$\mathbf{k} = \{k | (l_k, \mathbf{r}\ (x_k), l'_k, \langle \rho_k, \eta_k\rangle) \in \delta_k, k \in \mathbf{n}, k \neq i\}$, $\exists \mathbf{k_c}, \mathbf{k_d} : \mathbf{k} = \mathbf{k_c} \uplus \mathbf{k_d}$ such that:*
    - *$\forall j \in \mathbf{n} \setminus (\mathbf{k_c} \cup \{i\})$: $l'_j = l_j$*
    - *$\rho = \rho_i \wedge \bigwedge_{k\in\mathbf{k_c}} \rho_k$, $[\![\rho]\!]\theta$ is $\mathbf{true}$*
    - *$\eta = \eta_i \cup \bigcup_{k\in\mathbf{k_c}} \eta_k[v/x_k] \cup \{x_k := v\}$*
    - *$\gamma' = \gamma \wedge \bigwedge_{k\in\mathbf{k_c}} conn(\mathcal{J}_i, \mathcal{J}_k) \wedge \bigwedge_{k\in\mathbf{k_d}} dconn(\mathcal{J}_i, \mathcal{J}_k)$ is satisfiable*
- *$s_0 = (\bar{l}_0, \theta_0, \mathbf{true})$, where $\bar{l}_0 = \langle l_{0,1}, \ldots, l_{0,n}\rangle$, $\theta_0 = [\![\bigcup_{i\in\mathbf{n}} \eta_{0,i}]\!]\theta_\epsilon$, and θ_ϵ is the empty environment.*

In the clause for transition relation ($\rightsquigarrow$) in Definition 5, i denotes the index of a process enabled to do a broadcast ($\mathbf{b}\ e$) action, and $\mathbf{k}$ denotes the set of indices of processes that are enabled to perform a receive action. $\mathbf{k_c}$ and $\mathbf{k_d}$ form a partition of $\mathbf{k}$ such that $\mathbf{k_c}$ is the set of indices of processes that synchronize with the P_i; thus $conn$ constraint is generated for processes in $\mathbf{k_c}$. Processes with indices in $\mathbf{k_d}$ do not synchronize with broadcast action of P_i, and thus are not connected to P_i, and $dconn$ constraint is generated for the transition. Note that, as in the concrete semantics, processes not involved in the synchronization remain in their locations. The guards and assignments are treated exactly as in the concrete semantics, considering only the synchronizing processes.

Theorem 2 (Correspondence). *The symbolic semantics is sound and complete w.r.t. the concrete semantics; i.e. $(s, \gamma) \overset{\alpha}{\rightsquigarrow} (s', \gamma')$ in $[\![\Pi_{i\in\mathbf{n}} P_i : \mathcal{J}_i]\!]$ iff $\forall$ interface valuations ϑ s.t. $\vartheta \models \gamma'$, $s \xrightarrow{\alpha} s'$ in $[\![\Pi_{i\in\mathbf{n}} P_i : \vartheta(\mathcal{J}_i)]\!]$.*

Proof Sketch:

- *Soundness:* Consider a symbolic transition $(s, \gamma) \overset{\alpha}{\leadsto} (s', \gamma')$ in $\Pi_{i \in \mathbf{n}} P_i : \mathcal{J}_i$. From the semantics of the symbolic transitions, $\gamma' \implies \gamma$. For all ϑ s.t. $\vartheta \models \gamma'$ (also $\vartheta \models \gamma$), there exists a concrete transition $s \xrightarrow{\alpha} s'$ in $\Pi_{i \in \mathbf{n}} P_i : \vartheta(\mathcal{J}_i)$.
- *Completeness:* Consider a concrete transition $s \xrightarrow{\alpha} s'$ in $\Pi_{i \in \mathbf{n}} P_i : I_i$. Let ϑ be an interface valuation, γ' be a constraint, and for $i \in \mathbf{n}$, $\mathcal{J}_i$ be interface variables, such that $\vartheta(\mathcal{J}_i) = I_i$, and $\vartheta \models \gamma'$. Then $\exists \gamma : \gamma \implies \gamma'$, and $(s, \gamma) \overset{\alpha}{\leadsto} (s', \gamma')$ in $\Pi_{i \in \mathbf{n}} P_i : \mathcal{J}_i$. □

5 Constraint-Based Verification

5.1 Verification of Reachability Properties

We first consider verification of symbolic AHNs for reachability properties, which is done by constructing and traversing the symbolic transition system.

Definition 6 (Reachability). *For an AHN $A_C = \Pi_{i \in \mathbf{n}} P_i : I_i$, the set of states reachable from a state s in $[\![A_C]\!]$, denoted by $Reach_C(s, A_C)$, is the smallest set such that $s \in Reach_C(s, A_C)$ and for every $s' \in Reach_C(s, A_C)$ and for every $\alpha \in Act$ if $s' \xrightarrow{\alpha} s'' \in [\![A_C]\!]$ then $s'' \in Reach_C(s, A_C)$.*

For a symbolic AHN $A_S = \Pi_{i \in \mathbf{n}} P_i : \mathcal{J}_i$, the set of states reachable from a symbolic state (s, γ) in the $[\![A_S]\!]$, denoted by $Reach_S((s, \gamma), A_S)$, is the smallest set such that $(s, \gamma) \in Reach_S((s, \gamma), A_S)$, and for every $(s', \gamma') \in Reach_S((s, \gamma), A_S)$ and for every $\alpha \in Act$ if $(s', \gamma') \overset{\alpha}{\leadsto} (s'', \gamma'')$ then $(s'', \gamma'') \in Reach_S((s, \gamma), A_S)$.

Satisfaction of a Property. A property over a concrete AHN A_C, denoted by ϕ is either a proposition, defined over the states of A_C, or of the form *EFp*, where p is a proposition. We use $s \models \phi$ to denote satisfaction of property ϕ in state s. We say that $s \models EFp$ if there is some state s' reachable from s such that $s' \models p$. The notion of satisfaction of a property is lifted to symbolic states, denoted as $(s, \gamma) \models \phi$, if γ is satisfiable, and ϕ is true in s in every topology ϑ such that $\vartheta \models \gamma$. The following proposition establishes that when verifying a reachability property for a symbolic AHN, it is sufficient to examine a subset of symbolic states. In particular, once (s, γ) is visited and $(s, \gamma) \models \phi$, all states (s, γ') such that $\gamma' \implies \gamma$ can be discarded from consideration.

Proposition 3. *For a given symbolic state (s_0, γ_0), symbolic AHN A_S, and property ϕ, if $\exists (s, \gamma) \in Reach_S((s_0, \gamma_0), A_S)$ s.t. $(s, \gamma) \models \phi$, then $\forall (s, \gamma') \in Reach_S((s_0, \gamma_0), A_S)$ s.t. $\gamma' \implies \gamma$, $(s, \gamma') \models \phi$.*

Algorithm **SymReach** (Fig. 3) uses Prop. 3 to prune the search space for proving reachability properties. For a given predicate p, a symbolic AHN and a start state (s_0, γ_0) in the AHN, Algorithm **SymReach** returns the set of most general constraints *CS* such that for all $\gamma \in CS$ $(s_0, \gamma) \models EFp$. The set of reachable states are stored in R

```
Algorithm SymReach
Input : predicate $p$ ; symbolic AHN $A_S$; initial symbolic state $(s_0, \gamma_0)$
Output : $CS$ the set of most general constraints in states that satisfy $p$ and are
         reachable from initial state $(s_0, \gamma_0)$
1.  $R := \{(s_0, \gamma_0)\}$
2.  $CS := \left\{ \begin{array}{ll} \{\gamma_0\} & \mathit{if}(s_0, \gamma_0) \models p \\ \emptyset & otherwise \end{array} \right\}$
3.  $WS := \{(s_0, \gamma_0)\}$ // working set (FIFO queue)
4.    while $(WS \neq \emptyset)$
5.      let $(s, \gamma) \in WS$
6.      $WS := WS \setminus (s, \gamma)$
7.      for each transition $(s, \gamma) \overset{\alpha}{\rightsquigarrow} (s', \gamma')$ in $[\![A_S]\!]$
8.        if $\gamma'$ not subsumed by any constraint in $CS$
9.          if there exists no $(s', \gamma'') \in R$ such that $\gamma' \implies \gamma''$
10.           $WS := WS \cup \{(s', \gamma')\}$
11.           $R := R \cup \{(s', \gamma')\}$
12.           if $(s', \gamma') \models p$
13.             $CS := mg(CS \cup \{\gamma'\})$
14.     return $CS$
```

Fig. 3. Symbolic Reachability Algorithm

and a working set WS is used to store unvisited states (Line 3) during a breadth-first traversal of the transition system. At the beginning of each iteration (Line 4) states in $R-WS$ have been completely explored. Since each transition only adds to the topology constraints, we discard symbolic states whose topologies are already known to satisfy the reachability property (Line 8). Line 9 uses Prop. 3 to prune the search space. In Line 13, mg chooses the most general set of constraints from a given set of constraints. Algorithm **SymReach** returns the CS set upon termination. It is easily shown that for a finite-state AHN Algorithm **SymReach** terminates.

The following theorem formally states the correctness of the algorithm: that the set of topology constraints computed by **SymReach** exactly covers the topology constraints in $Reach_S$ (Def. 6).

Theorem 4 (Correctness). *Let $CS' = \{\gamma \mid (s, \gamma) \in Reach_S((s_0, \gamma_0), A_S),\ (s, \gamma) \models \phi\}$ be the set of all constraints that are part of the reachable symbolic states (s, γ) for which ϕ holds. Let CS be the set returned by Algorithm* **SymReach** *(Figure 3). Then $\forall \gamma' \in CS'\ \exists \gamma \in CS : \gamma' \implies \gamma$, and $\forall \gamma \in CS\ \exists \gamma' \in CS' : \gamma \equiv \gamma'$.*

The choice of breadth-first search (BFS) in Algorithm **SymReach** is important for the following two reasons. First, subsumption-based pruning of search space is more effective with BFS because general constraints are visited before more specific constraints. Secondly, the use of BFS makes it easy to show the tight bound on the total number of symbolic transitions, used in the complexity analysis.

5.2 Complexity Analysis for the $\mathsf{SymReach}$ Algorithm

Consider a concrete AHN A_C with n nodes. Let the total number of states in A_C be $|S|$, and the total number of transitions in A_C be $|T| = O(|S|^2)$. The time for reachability analysis from a given initial state in A_C is bounded by the number of transitions and is equal to $|T| = O(|S|^2)$. The total number of topologies for an n-node AHN is $O(2^{n^2})$. Therefore, the time complexity for exploring states reachable from a given state in all n-node AHNs (all possible topologies) is $O(2^{n^2}) \times |T| = O(2^{n^2}|S|^2)$.

Let $A_S = \Pi_{i \in \mathbf{n}} P_i : \mathcal{J}_i$ be a symbolic AHN and $\mathcal{A}_C$ the set of all concrete AHNs $A_{C_j} = \Pi_{i \in \mathbf{n}} P_i : I_{i,j}$, where index j indicates one of the $O(2^{n^2})$ possible topologies for an n-node network. Recall that each state of A_S is of the form (s, γ), where s is a location-environment pair, and γ is a topology constraint. Let $|S|$ be the largest number of states of any concrete AHN $A_C \in \mathcal{A}_C$. Since the number of distinct γ's is $O(2^{n^2})$, the total number of symbolic states is bounded by $O(2^{n^2}|S|)$.

The number of symbolic transitions is bounded by the total number of concrete transitions for all possible topologies. We can establish this bound by defining a 1-1 mapping between symbolic transitions from a symbolic state (s, γ) in A_S to a transition from concrete state s in $\mathcal{A}_C$. Consider associating each state in R and *WS* with an index which is the length of the shortest path from the initial state to (s, γ). Now, let (s, γ) be the selected state with index i at some iteration of the algorithm. There is no state (s, γ') in $R - \mathit{WS}$ (i.e. visited state) such that $\gamma \implies \gamma'$ (due to the use of subsumption, line 9 of the algorithm). First consider the case when there is no other state (s, γ') in R with index i. It follows from Theorem 2 that for every concrete topology that satisfies γ, state s is reachable in i or fewer steps. In fact, there is a concrete topology $\vartheta \models \gamma$ for which the shortest path to reach s is of length i. The symbolic transition that placed (s, γ) in *WS* can then be mapped to the corresponding concrete transition in the topology given by ϑ. Now consider the case when there is another state (s, γ') in R with index i. If (s, γ) and (s, γ') can be reached using a single transition from a common state, say (s'', γ''), then the symbolic transition that placed (s, γ) in *WS* can then be mapped to the corresponding concrete transition in a topology that satisfies $\gamma \wedge \neg\gamma'$. Otherwise, (s, γ) and (s, γ') descend from two distinct states, both of which have the same index. We can then associate with the symbolic transition to (s, γ) the same concrete instance ϑ used to map the transition to its parent (and similarly with (s, γ')).

We now show that reachability computation over symbolic state space takes no additional time, in the asymptotic sense, than reachability over concrete state spaces. The main additional cost of symbolic reachability algorithm is constraint subsumption (line 9 of the algorithm). We can do this operation in amortized constant time, as follows. First, consider computing and storing the subsumption lattice for the constraints *a priori*. The construction cost of this lattice is $O(2^{n^2})$ but is paid only once. We can associate a set, initially empty, with each constraint in the lattice. To determine whether (s, γ) should be added to R, we check if s is in the set associated with γ in the lattice. This check can be done in constant time. When (s, γ) is added to R, we add s to the sets associated with constraints more specific than γ. This operation may take $O(2^{n^2})$ in the worst case, but note that an element s may be added to the set associated with γ at most once, and hence maintaining this data structure incurs a total cost of $O(2^{n^2}|S|)$ over the entire run of the algorithm. Hence symbolic reachability can be done in $O(2^{n^2}|S|^2)$, the same complexity as that of the concrete algorithm.

The space complexity is bounded by the size of the set of reachable states, R. The number of elements of this set is $2^{n^2}|S|$. The size of each element is $O(n^2)$ due to the size of the topology constraint, but this factor gets down-played in the asymptotic case. Hence the asymptotic space complexity for the symbolic algorithm is $O(2^{n^2}|S|)$.

5.3 Model Checking Symbolic AHNs

The symbolic transition system can be readily used for checking LTL properties of AHNs. We can use the standard procedure of constructing the product between a Büchi automaton (corresponding to the negation of a given LTL property) and the symbolic transition system and look for reachable accepting cycles in the product graph. Note that for every symbolic transition of the form $(s, \gamma) \rightsquigarrow (s', \gamma')$, it holds that $\gamma' \implies \gamma$. Hence it follows that if (s, γ) and (s, γ') are two states in a cycle, then $\gamma \equiv \gamma'$. Hence the constraint component of states in a cycle are all equivalent. Let $(s_1, \gamma), (s_2, \gamma), \ldots (s_n, \gamma)$ be states in an accepting cycle such that $(s_i, \gamma) \rightsquigarrow (s_{i+1}, \gamma)$ for $1 \leq i < n$, and $(s_n, \gamma) \rightsquigarrow (s_1, \gamma)$. It follows from Theorem 2 that for every concrete topology ϑ such that $\vartheta \models \gamma$, the states $s_1, s_2, \ldots, s_n$ will be in an accepting cycle. Hence reachable good cycles in the symbolic case mean that there are reachable good cycles in the concrete case. This forms the basis for LTL model checking of symbolic AHNs.

Model checking of other temporal logics such as CTL and CTL* can be performed over symbolic AHNs by using the standard algorithms over the symbolic transition system. From the complexity results for reachability checking, it follows that model checking for symbolic AHNs can be done in time and space comparable to the total time and space for model checking of concrete AHNs for all topologies.

6 Verification of the LMAC Protocol

We built a prototype implementation of **SymReach** in the XSB logic programming system [14]. XSB adds the capability of memoizing inferences to a traditional Prolog-based system, which simplifies the implementation of fixed point algorithms such as **SymReach**. Below we present the results of verifying the LMAC protocol [13], a medium access control protocol for wireless sensor networks, using this prototype.

LMAC protocol for Wireless Sensor Networks. The LMAC protocol aims to allocate each node in the sensor network a time slot during which the node can transmit without collisions. Note that for collision freedom, direct (one-hop) neighbors as well as two-hop neighbors must have pairwise different slots. The protocol works by nondeterministically assigning slots, and resolving any collisions that result from this assignment. We apply our query-based verification technique to this protocol to compute the set of topologies for which there are undetected and hence unresolved collisions.

Protocol Description [13]. In schedule-based MAC protocols, time is divided into slots, which are grouped into fixed length frames. Every node is allocated one time slot in which it can carry out its transmission in a frame without causing collision or interference with other transmissions. Each node broadcasts a set of time slots occupied by its (one-hop) neighbors and itself. When a node receives a message from a neighbor it marks the respective time slot as occupied. The four phases of the LMAC protocol involved in allocating time slots to nodes are as follows. ***Initialization phase:*** a node listens on the wireless medium to detect other nodes. On listening from a neighboring node, the node synchronizes by learning the current slot number and transitions to the wait phase. ***Wait phase:*** a node waits for a random period of time and then continues with the discover phase. ***Discover phase:*** a node listens to its one-hop neighbors during one entire frame and records the time slots occupied by them and their neighbors. On gathering information regarding the occupied time slots, the node randomly chooses a time slot from the available ones (time slots that do not interfere in its one-hop and two-hop neighborhood), and advances to the active phase. ***Active phase:*** a node transmits a message in its own time slot and listens during other time slots. When a neighboring node informs that there was a collision in the time slot of the node, the node transitions to the wait phase to discover a new time slot for itself. Collisions can occur when two or more one-hop or two-hop neighboring nodes choose the same time slot for transmission. Nodes causing a collision cannot detect the collision themselves, they need to be informed by their neighboring nodes about the collision. When a node detects a collision it transmits information about the collision in its time slot.

Modeling the LMAC protocol in our framework. Our encoding of the LMAC protocol in our framework follows the encoding used in [5]. We carry over the underlying assumption in the LMAC protocol, that the local clocks of nodes are synchronous. Since there is no support for modeling time in our prototype framework, we define a special *timer* node that informs other nodes about the end of a time slot by broadcasting an *end of slot* message. Nodes update their local information at the end of every time slot.

An encoding of a process in an AHN model of LMAC is presented in Fig. 4. At the beginning, we assume that one distinguished node is "active" (i.e. in `active` location) and the rest are "passive" (i.e. in `init` location). Note that the figure gives the definition of a passive node; the definition of the active node is identical except for its initial state. The (symbolic) system specification for a 3-node network is shown below.

$$\mathcal{A} = \textit{timer} : \mathcal{J}_1 \mid \textit{active\_node} : \mathcal{J}_2 \mid \textit{passive\_node} : \mathcal{J}_3 \mid \textit{passive\_node} : \mathcal{J}_4$$

Transitions in Fig. 4 are specified in the form $[\textit{label}]\ l\ \&\ \rho \rightarrow l'\ \&\ \eta$, where *label* is the label of the transition, l and l' are the source and destination locations, ρ is the (optional) guard and η is the set of simultaneous assignments. We use the standard notation of *primed* variables to denote variables in the destination state. We use "epsilon" transitions (denoted by action label `[ ]` in the figure) to simplify the encoding. We can derive the epsilon-free description (as in the formal definition of AHNs, Defn. 1) using standard automata construction techniques. In our model of LMAC, locations *init*, *init1* and *init2* correspond to the `initialization` phase; locations *listening0*, *recOne0*, *done0*, *choice0* and *choice* to the `discover` phase; and locations *active*, *sent*, *listening*, *recOne*, *recTwo*, and *collision_detected* to the `active` phase. It should be noted that the *wait* phase of the protocol is not modeled, since its function is to only separate the initialization and discover phases by an arbitrary period of time.

Passive LMAC Process : $< L, X, \Sigma, \delta, l_0, \eta_0 >$

$L = \{init, init1, init2, listening0, recOne0, done0, choice0, choice, active, sent,$
$\quad listening, recOne, recTwo, collision\_detected\}$

$X = \{Current, RecVec, Counter, SlotNo, First, Second, Col, Collision\}$

$\Sigma = \{\mathbf{r}\ (msg(Sslot, Scollision, Sfirst)), \mathbf{r}\ (eos), \mathbf{b}\ msg(slot, collision, first)\}$

$l_0 = init$

$\eta_0 = \{Current := -1, RecVec := \emptyset, Counter := 0, SlotNo := -1, First := \emptyset,$
$\quad Second := \emptyset, Col := -1, Collision := -1\}$

Transitions $(l, \alpha, l', \langle \rho, \eta \rangle) \in \delta$ are given below:

Init

$[\mathbf{r}\ (msg(Sslot, \_, \_))]\ init \rightarrow init1\ \&\ Current' := Sslot$

$[\mathbf{r}\ (eos)]\ init1 \rightarrow listening0\ \&\ Current' := (Current + 1)\%frame, Counter' := 0$

$[\mathbf{r}\ (msg(\_, \_, \_))]\ init1 \rightarrow init2$

$[\mathbf{r}\ (eos)]\ init2 \rightarrow init$

Discover

$[\mathbf{r}\ (msg(\_, \_, Sfirst))]\ listening0 \rightarrow recOne0\ \&\ RecVec' := Sfirst,$
$\qquad First' := \{Current\} \cup First$

$[\mathbf{r}\ (msg(\_, \_, \_))]\ recOne0 \rightarrow done0\ \&\ if\ Collision < 0\ then\ Collision' := Current,$
$\qquad RecVec' := \emptyset$

$[\mathbf{r}\ (eos)]\ done0 \rightarrow choice0\ \&\ Current' := (Current + 1)\%frame$

$[\mathbf{r}\ (eos)]\ recOne0 \rightarrow choice0\ \&\ Current' := (Current + 1)\%frame,$
$\qquad Second' := RecVec \cup Second,\ RecVec' := \emptyset$

$[\mathbf{r}\ (eos)]\ listening0 \rightarrow choice0\ \&\ Current' := (Current + 1)\%frame$

$[\,]\ choice0\ \&\ Counter < frame - 1 \rightarrow listening0\ \&\ Counter' := Counter + 1$

$[\,]\ choice0\ \&\ Counter >= frame - 1 \rightarrow choice\ \&\ Second' := First \cup Second$

Choice

$[\,]\ choice\ \&\ Second \neq AllSlots \rightarrow active\ \&\ SlotNo' \in AllSlots \setminus Second,$
$\qquad Second' := \emptyset$

$[\,]\ choice\ \&\ Second = AllSlots \rightarrow listening0\ \&\ Counter' := -1,\ Collision' := -1,$
$\qquad First' := \emptyset,\ Second' := \emptyset$

Active

$[\mathbf{b} msg(SlotNo, Collision, First)]\ active\ \&\ Current{=}SlotNo \rightarrow sent\ \&\ Collision' {:=} {-1}$

$[\,]\ active\ \&\ Current \neq SlotNo \rightarrow listening$

Send

$[\mathbf{r}\ (eos)]\ sent \rightarrow active\ \&\ Current' := (Current + 1)\%frame$

Listen

$[\mathbf{r}\ (msg(\_, Scollision, \_))]\ listening \rightarrow recOne\ \&\ Col' := Scollision,$
$\qquad First' := Current \cup First$

$[\mathbf{r}\ (eos)]\ listening \rightarrow active\ \&\ Current' := (Current + 1)\%frame$

$[\mathbf{r}\ (msg(\_, \_, \_))]\ recOne \rightarrow recTwo\ \&\ if\ Collision' < 0\ then\ Collision' := Current$

$[\mathbf{r}\ (eos)]\ recTwo \rightarrow active\ \&\ Current' := (Current + 1)\%frame$

$[\mathbf{r}\ (eos)]\ recOne\ \&\ Col \neq SlotNo \rightarrow active\ \&\ Current' := (Current + 1)\%frame$

Collision Reported

$[\mathbf{r}\ (eos)]\ recOne\ \&\ Col = SlotNo \rightarrow collision\_detected\ \&\ First' := \emptyset,\ RecVec' := \emptyset$
$\qquad Current' := (Current + 1)\%frame,$
$\qquad Counter' := 0,\ SlotNo' := -1,$
$\qquad Col' := -1,\ Collision' := -1$

$[\,]\ collision\_detected \rightarrow listening0$

Fig. 4. LMAC protocol model

Table 1. Verification statistics for the LMAC protocol for *detected collisions*

Nodes	# Topologies Symbolic/Concrete	# States	# Transitions	CPU Time	Memory (MB)
2	1/2	36	36	0.08 sec	2.42
3	5/8	110	123	0.24 sec	2.46
4	25/64	458	667	3.38 sec	3.05
5	181/1024	2204	5223	69.51 sec	5.09
6	2082/32768	29012	110194	2 hr 51 min 46 sec	49.79

The length of a time frame i.e. number of slots (= 5 for 5-node network) is represented by *frame*, and *AllSlots* denotes the set of all time slots. The state variables of a node are: *Current* (the current slot number w.r.t. the beginning of a frame), *RecVec* (auxiliary set to record the slots occupied by one-hop and two-hop neighbors), *Counter* (used to count the number of slots seen by the node in a frame), *SlotNo* (slot number of the node), *First* (set of slots occupied by one-hop neighbors of the node), *Second* (set of slots occupied by two-hop neighbors of the node), *Col* (collision slot reported by another node), *Collision* (slot in which the node detects a collision). The parameters of messages (*msg*) exchanged between nodes are: *Slot*, *Collision*, and *First* variables of the sender node.

Analysis of the LMAC protocol. The property "every collision is eventually detected" can be encoded in LTL as $G(\mathit{collision} \Rightarrow F\mathit{collision\_detected})$, where *collision* and *collision_detected* are propositions that are true in states where collision and collision detection occur, respectively. Although LTL model checking of symbolic AHNs can be done as outlined in [5], our current prototype implementation supports only reachability checking. We hence checked a related property "there is a detected collision" ($EF\mathit{collision\_detected}$). Let *CS* be the set of all topology constraints computed using algorithm $\mathbf{SymReach}$ when checking for reachability of proposition *collision_detected*. Let ϑ be a valuation such that $\vartheta \not\models \gamma$ for any $\gamma \in \mathit{CS}$. Note that in the LMAC protocol, there may be a collision between any two neighboring nodes. If γ does not represent a fully disconnected topology, then we can conclude that there is an undetected collision in γ. Hence, by checking for reachability of proposition *collision_detected*, we can compute (a subset of) topologies which have undetected collision. Moreover, using this method is sound: if there is an undetected collision in some topology, we will find at least one representative.

Verification Statistics and Results. We did symbolic reachability checking for 2- to 6-node networks. The performance results are shown in Table 1. The results were obtained on a machine with Intel Xeon 1.7GHz processor and 2Gb memory running Linux 2.6.18, and with XSB Prolog version 3.1. For 2- and 3-node cases there were no collisions. For 4-, 5- and 6-node cases, topologies containing one-hop neighboring (directly connected) node pairs that appeared in a ring in the topology and did not have a common direct neighbor were found to be in collision that remained undetected.

The second column in the table gives two numbers ξ_s/ξ_c, where ξ_s is the number of symbolic topology constraints explored in a reachability run, i.e. the number of distinct

γ such that $(s, \gamma) \in R$ as per the algorithm in Fig. 3; and ξ_c is the total number of possible concrete topologies. Observe that for the 6-node case the number of symbolic topology constraints examined is smaller than the number of concrete topologies by a factor of more than 5. It should also be noted that the same property was verified for a 5-node network in [5] by using 61 separate verification runs, one for each unique (modulo isomorphism) concrete topology. In contrast, we verified a related property using a single symbolic reachability run.

The third and fourth columns in Table 1 give the number of symbolic states and transitions explored, respectively; and the last two columns give the CPU time and total memory used. Observe that the performance of our prototype implementation is efficient enough to be used for topologies of reasonable size (e.g. 6 nodes). It should be noted that our technique and its implementation does not exploit the symmetry inherent in the problem by identifying isomorphic topologies. At a high level, symmetry reduction can be incorporated by using a check in line 9 of **SymReach** that recognizes constraints representing the same set of topologies modulo isomorphism. Doing so will enable the technique to scale to large network sizes.

7 Conclusions

We presented an efficient query-based verification technique for ad hoc network protocols. Network topologies are represented symbolically using interface variables, and the model-checking process generates constraints on the topology under which a system specification satisfies a specified property. As such, a term in our constraint language compactly represents a set of concrete topologies that may lead to the satisfaction of the property in question. We demonstrated the practical utility of our approach by considering the verification of a medium access control protocol for sensor networks (LMAC) [13], identifying topologies under which collision may remain unresolved.

The basic data structure for query-based verification is the symbolic transition system, where each state carries with it a topology constraint. If a symbolic state is reachable, then, for every topology satisfying its constraint, the corresponding concrete state is reachable. This structure makes it possible to infer topologies under which reachability properties hold. As described in the paper, it is also possible to verify properties specified in temporal logics such as LTL over symbolic transition systems, inferring sets of topologies under which the properties hold. Extending our prototype implementation to handle verification with an expressive temporal logic is a topic of future work. There are several avenues for further improving the efficiency of the symbolic verification technique. Some of these are optimizations to common low-level operations, subsumption checks, while others are high-level state-space reductions, e.g. by exploiting symmetries in systems and topologies.

In this work, the focus is on a verification technique and not on the modeling language. We considered ad hoc networks whose topology does not change with time. We deliberately considered only closed systems and chose a simple language that uses interfaces to separate node behavior from network topology as in the ω-calculus [11]. As part of our future work, we plan to extend this work to open systems specified in the ω-calculus, and consider compositional verification in that setting.

Acknowledgements. We thank the anonymous reviewers for their valuable comments on an earlier version of this paper. This work was supported in part by NSF grants CNS-0509230, CNS-0627447, CNS-0721665, and ONR grant N000140710928.

References

1. Bruns, G., Godefroid, P.: Temporal logic query checking. In: LICS, pp. 409–417 (2001)
2. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000)
3. Chan, W., Anderson, R., Beame, P., Notkin, D.: Combining constraint solving and symbolic model checking for a class of a systems with non-linear constraints. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 316–327. Springer, Heidelberg (1997)
4. Delzanno, G., Podelski, A.: Model checking in CLP. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 223–239. Springer, Heidelberg (1999)
5. 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)
6. Flanagan, C.: Automatic software model checking via constraint logic. Sci. Comput. Program. 50(1-3), 253–270 (2004)
7. Fribourg, L.: Constraint logic programming applied to model checking. In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817, pp. 30–41. Springer, Heidelberg (2000)
8. Ghassemi, F., Fokkink, W., Movaghar, A.: Equational reasoning on ad hoc networks. In: Proceedings of the Third International Conference on Fundamentals of Software Engineering, FSEN (2009)
9. Gurfinkel, A., Chechik, M., Devereux, B.: Temporal logic query checking: A tool for model exploration. IEEE Trans. Software Eng. 29(10), 898–914 (2003)
10. Podelski, A.: Model checking as constraint solving. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 22–37. Springer, Heidelberg (2000)
11. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 296–314. Springer, Heidelberg (2008)
12. Starosta, B.S., Ramakrishnan, C.R.: Constraint-based model checking of data-independent systems. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 579–598. Springer, Heidelberg (2003)
13. van Hoesel, L., Havinga, P.: A lightweight medium access protocol (LMAC) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In: 1st International Workshop on Networked Sensing Systems (INSS), pp. 205–208 (2004)
14. XSB. The XSB logic programming system, http://xsb.sourceforge.net
15. Zhang, D., Cleaveland, R.: Efficient temporal-logic query checking for presburger systems. In: ASE, pp. 24–33. ACM, New York (2005)

Strict Divergence for Probabilistic Timed Automata[⋆]

Jeremy Sproston

Dipartimento di Informatica, Università di Torino, 10149 Torino, Italy
sproston@di.unito.it

Abstract. Probabilistic timed automata are an extension of timed automata with discrete probability distributions. In previous work, a probabilistic notion of time divergence for probabilistic timed automata has been considered, which requires the divergence of time with probability 1. We show that this notion can lead to cases in which the probabilistic timed automaton satisfies a correctness requirement by making an infinite number of probabilistic transitions in a finite amount of time. To avoid such cases, we consider strict time divergence which concerns the divergence of time over all paths, rather than time divergence of paths with probability 1. We present new model-checking algorithms for probabilistic timed automata both for probabilistic and strict divergence. The algorithms have the same complexity as the previous model-checking algorithms for probabilistic timed automata.

1 Introduction

Model checking is an automatic verification technique for establishing that a model of a system satisfies a formally-specified property [1]. Two particular classes of systems have been subject to extensions of the basic model-checking paradigm. Firstly, methods for *timed systems*, in which the durations of system behaviours is critical for the system's correctness, have been developed, with particular emphasis on techniques for the system-description formalism of timed automata [2]. Secondly, methods for *probabilistic systems*, in which system behaviours have associated probabilities of occurrence, have been introduced, in this case concentrating on techniques for Markov chains (in which the choice between transitions is probabilistic) or Markov decision processes (in which the choice between transitions is both nondeterministic and probabilistic). In this paper, we consider methods for *probabilistic timed systems*, in which both timed and probabilistic behaviour coexist. In the context of probabilistic timed systems, a correctness requirement typically combines probabilistic and timing thresholds, such as "a request is followed by a response within 5 time units with probability 0.99 or greater". A number of model-checking methods for system-description formalisms for such systems, which generally can differ in terms of the way in

[⋆] Supported in part by the MIUR-PRIN project PaCo - Performability-Aware Computing: Logics, Models and Languages.

which the interaction of probability and time is modelled both in the system and in the correctness requirements, have been presented [3,4,5,6,7,8,9]. Our focus is on the system-description formalism of *probabilistic timed automata* [10,6], which can be regarded as an extension of timed automata with discrete probability distributions, or, equivalently, an extension of Markov decision processes with timed automata-like clocks, constraints and resets. Probabilistic timed automata have been used to model systems such as the IEEE 1394 root contention protocol, the backoff procedure in IEEE 802.11 Wireless LANs, and the IPv4 Zeroconf protocol [11].

When modelling timed systems, the issue of time divergence is of importance. Roughly speaking, behaviours of the model which correspond to the case in which the amount of time elapsed converges do not correspond to phenomena that a real system can exhibit, and therefore should be excluded from model-checking analyses. Methods for model checking timed automata therefore are defined in such a way as to consider divergent behaviours only [12,13,14]. Recall that, for probabilistic timed automata (as for Markov decision processes), a strategy is a function which resolves the nondeterminism of the system, by mapping finite system behaviours to nondeterministic alternative transitions available in the last state of the behaviour. For probabilistic timed automata, a probabilistic notion of time divergence has been presented [6], which requires that time diverges with probability 1 for all strategies of the model. We henceforth refer to this notion as *probabilistic divergence*. Furthermore, a model-checking algorithm for the probabilistic timed temporal logic PTCTL based on the standard region graph construction [2,12] is presented in [6], and which computes the correct probability of property satisfaction in the case in which all strategies of the model are probabilistically divergent. This is guaranteed when all structural loops in the graph of the probabilistic automaton require that at least one time unit elapses, which holds for many case studies considered [11]. The algorithm runs in EXPTIME, which is optimal by the results of [15]. Furthermore, a symbolic probabilistic model-checking method for probabilistic timed automata has been presented which can be applied also if not all strategies of the system are probabilistically divergent [16], although this algorithm does not run in EXPTIME.

There remain two questions in this context. The first concerns whether there exists an EXPTIME algorithm for PTCTL model checking of probabilistic timed automata with probabilistically divergent strategies. The second question concerns whether the notion of probabilistic divergence is generally applicable. Consider the probabilistic timed automaton of Figure 1 (where edges without a probability label correspond to probability 1). From the location l_0, there exists a probabilistically divergent strategy to reach l_2 with probability 1. An example of such a strategy is the following: the first time l_0 is visited, let $\frac{1}{2}$ time units elapse, then select the rightmost transition; if the probabilistic choice is resolved so that l_0 is then visited, let $\frac{1}{4}$ time units elapse, then take the rightmost transition again; if l_0 is visited for a third time, then let $\frac{1}{8}$ time units elapse, then take the rightmost transition again, and so on. Then the value of the clock x will never be 1, and the strategy will be able to take the rightmost transition an

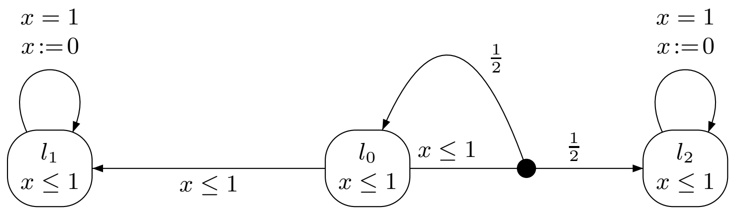


Fig. 1. Maximal reachability probabilities under probabilistic and strict divergence

infinite number of times, resulting in the probability of $\sum_{k \geq 1} \frac{1}{2^i} = 1$ of reaching l_2. However, in the case in which we assume that the selection of the rightmost transition corresponds to the change of some physical state in the system, the behaviour exhibited by this strategy should be excluded from the analysis of the system. Therefore we argue that it is important to have an alternative to probabilistic divergence. We propose *strict divergence*, which requires that *all behaviours* (rather than probability 1 of the behaviours) of a strategy should be time divergent. Note that, with the requirement of strict divergence, the maximum probability of reaching l_2 from l_0 can be made to be arbitrarily close to 1 (by making arbitrarily large the upper bound on the number of times the rightmost transition is selected before the leftmost transition is taken). However, there is no strictly divergent strategy which reaches l_2 from l_0 actually with probability 1. Therefore, the correctness property which specifies that l_2 is reached with probability 1 for some strategy is satisfied under probabilistic divergence, but not under strict divergence.

In this paper, after first recalling the definition of probabilistic timed automata and PTCTL, we present an EXPTIME algorithm for PTCTL model checking with probabilistic divergence in Section 3. Then, in Section 4, we present an EXPTIME algorithm for PTCTL model checking with strict divergence.

Related work. The distinction between probabilistic and strict divergence is inspired by the distinction between fairness and strict fairness, as introduced by Baier and Kwiatkowska in the context of model checking Markov decision processes [17,18]. We note that [19] features randomized strategies in the context of 2-player timed games which are required to be either time divergent, or blameless for the convergence of time, over all paths. In [20], the probability of behaviours satisfying a (Büchi) correctness requirement *and* are divergent can be computed. We do not follow this approach, in which the correctness property is adapted to encode also the divergence of time, because it does not exclude strategies in which time converges with positive probability. We note that, in contrast, the correctness property can be adapted to encode the divergence of time in timed automata [21] and timed games [22].

2 Probabilistic Timed Automata

Preliminaries. We use $\mathbb{R}_{\geq 0}$ to denote the set of non-negative real numbers, $\mathbb{N}$ to denote the set of natural numbers, and AP to denote a set of atomic propositions. Given a set Q and a function $\mu : Q \to \mathbb{R}_{\geq 0}$, we define $\mathsf{support}(\mu) = \{q \in Q \mid \mu(q) > 0\}$. A (discrete) probability *distribution* over a countable set Q is a function $\mu : Q \to [0,1]$ such that $\sum_{q \in Q} \mu(q) = 1$. Let $\mathsf{Dist}(Q)$ be the set of distributions over Q. If Q is an uncountable set, we define $\mathsf{Dist}(Q)$ to be the set of functions $\mu : Q \to [0,1]$, such that $\mathsf{support}(\mu)$ is a countable set and μ restricted to $\mathsf{support}(\mu)$ is a (discrete) probability distribution.

A *timed Markov decision process* (TMDP) $\mathsf{T} = (S, \to, lab)$ comprises the following components: a (possibly uncountable) set of *states* S; a (possibly uncountable) *timed probabilistic, nondeterministic transition relation* $\to \subseteq S \times \mathbb{R}_{\geq 0} \times \mathsf{Dist}(S)$; and a *labelling function* $lab : S \to 2^{AP}$. The transitions from state to state of a TMDP are performed in two steps: given that the current state is s, the first step concerns a nondeterministic selection of $(s, d, \mu) \in \to$, where d corresponds to the duration of the transition; the second step comprises a probabilistic choice, made according to the distribution μ, as to which state to make the transition to (that is, we make a transition to a state $s' \in S$ with probability $\mu(s')$). We often denote such a completed transition by $s \xrightarrow{d,\mu} s'$. A TMDP is *total* if, for each state $s \in S$, there exists at least one transition $(s, \_, \_) \in \to$.

An *infinite path* of T is an infinite sequence of transitions $r = s_0 \xrightarrow{d_0,\mu_0} s_1 \xrightarrow{d_1,\mu_1} \cdots$ such that the target state of one transition is the source state of the next. Similarly, a *finite path* of T is a finite sequence of consecutive transitions $r = s_0 \xrightarrow{d_0,\mu_0} s_1 \xrightarrow{d_1,\mu_1} \cdots \xrightarrow{d_{n-1},\mu_{n-1}} s_n$. If r is finite, the length of r, denoted by $|r|$, is equal to the number of transitions along r. If r is infinite, we let $|r| = \infty$. We use $Path^{\mathsf{T}}_{ful}$ to denote the set of infinite paths of T, and $Path^{\mathsf{T}}_{fin}$ the set of finite paths of T. When clear from the context, we omit the superscript T. If r is a finite path, we denote by $last(r)$ the last state of r. For any path r and $i \leq |r|$, let $r(i) = s_i$ be the $(i+1)$th state along r, and let $step(r, i) = \mu_i$ be the $(i+1)$th distribution taken along r. Let $Path^{\mathsf{T}}_{ful}(s)$ and $Path^{\mathsf{T}}_{fin}(s)$ refer to the sets of infinite and finite paths of T, respectively, commencing in state $s \in S$.

A *strategy* of a TMDP T is a function σ mapping every finite path $r \in Path_{fin}$ to a transition $(last(r), d, \mu) \in \to$. Let Σ_{T} be the set of strategies of T (when the context is clear, we write simply Σ). For any strategy $\sigma \in \Sigma$, let $Path^{\sigma}_{ful}$ and $Path^{\sigma}_{fin}$ denote the sets of infinite and finite paths, respectively, resulting from the choices of σ. For a state $s \in S$, let $Path^{\sigma}_{ful}(s) = Path^{\sigma}_{ful} \cap Path_{ful}(s)$ and $Path^{\sigma}_{fin}(s) = Path^{\sigma}_{fin} \cap Path_{fin}(s)$. Given a strategy $\sigma \in \Sigma$ and a state $s \in S$, we define the probability measure $Prob^{\sigma}_{s}$ over $Path^{\sigma}_{ful}(s)$ in the standard way [23].

An *untimed Markov decision process* (MDP) $\mathsf{M} = (S, \to, lab)$ is defined as a TMDP, but for which $\to \subseteq S \times \mathsf{Dist}(S)$ (that is, the transition relation $\to$ does not contain timing information). A sub-MDP $(S', \to', lab|_{S'})$ of M is an MDP such that $S' \subseteq S$, $\to' \subseteq \to$, and $lab|_{S'}$ is equal to lab restricted to S'. Let $T \subseteq S$. The *sub-MDP of* M *induced by* T is the sub-MDP $(T, \to|_T, lab|_T)$ of M, where $\to|_T = \{(s, \nu) \in \to \mid s \in T \wedge \mathsf{support}(\nu) \subseteq T\}$. Occasionally we omit the labelling function lab for MDPs. The graph of an MDP $(S, \to)$ is the pair (S, E) where $(s, s') \in E$ if and only if there exists $(s, \mu) \in \to$ such that $s' \in \mathsf{support}(\mu)$. An *end component* (EC) of an MDP M is a sub-MDP $(C, D) \in 2^S \times 2^{\to}$ such that

(1) if $(s, \mu) \in D$, then $s \in C$ and $\mathsf{support}(\mu) \subseteq C$, and (2) the graph of (C, D) is strongly connected [5]. An end component (C, D) of M is *maximal* if there does not exist any EC (C', D') of M such that $(C, D) \neq (C', D')$, $C \subseteq C'$ and $D \subseteq D'$.

Probabilistic timed automata. Let $\mathcal{X}$ be a finite set of real-valued variables called *clocks*, the values of which increase at the same rate as real-time. The set $CC(\mathcal{X})$ of *clock constraints* over $\mathcal{X}$ is defined as the set of conjunctions over atomic formulae of the form $x \sim c$, where $x, y \in \mathcal{X}$, $\sim \in \{<, \leq, >, \geq, =\}$, and $c \in \mathbb{N}$. A *probabilistic timed automaton* (PTA) $\mathsf{P} = (L, \mathcal{X}, inv, prob, \mathcal{L})$ consists of the following components: a finite set L of *locations*; a finite set $\mathcal{X}$ of clocks; a function $inv : L \to CC(\mathcal{X})$ associating an *invariant condition* with each location; a finite set $prob \subseteq L \times CC(\mathcal{X}) \times \mathsf{Dist}(2^{\mathcal{X}} \times L)$ of *probabilistic edges* such that, for each $l \in L$, there exists at least one $(l, \_, \_) \in prob$; and a *labelling function* $\mathcal{L} : L \to 2^{AP}$. A probabilistic edge $(l, g, p) \in prob$ is a triple containing (1) a source location l, (2) a clock constraint g, called a *guard*, and (3) a probability distribution p which assigns probability to pairs of the form (X, l'), where $X \subseteq \mathcal{X}$ is a clock set X and $l' \in L$ is a location. The behaviour of a probabilistic timed automaton takes a similar form to that of a timed automaton [2]: in any location time can advance as long as the invariant holds, and a probabilistic edge can be taken if its guard is satisfied by the current values of the clocks. However, probabilistic timed automata generalize timed automata in the sense that, once a probabilistic edge is nondeterministically selected, the choice of which clocks to reset and which target location to make the transition to is *probabilistic*.

We refer to a mapping $v : \mathcal{X} \to \mathbb{R}_{\geq 0}$ as a *clock valuation*. Let $\mathbb{R}_{\geq 0}^{\mathcal{X}}$ denote the set of clock valuations. For a clock valuation $v \in \mathbb{R}_{\geq 0}^{\mathcal{X}}$ and a value $d \in \mathbb{R}_{\geq 0}$, we use $v + d$ to denote the clock valuation such that $(v + d)(x) = v(x) + d$ for all clocks $x \in \mathcal{X}$. For a clock set $X \subseteq \mathcal{X}$, we let $v[X := 0]$ be the clock valuation obtained from v by resetting all clocks within X to 0; formally, we let $v[X := 0](x) = 0$ for all $x \in X$, and let $v[X := 0](x) = v(x)$ for all $x \in \mathcal{X} \setminus X$. The clock valuation v *satisfies* the clock constraint $\psi \in CC(\mathcal{X})$, written $v \models \psi$, if and only if ψ resolves to true after substituting each clock $x \in \mathcal{X}$ with the corresponding clock value $v(x)$.

The semantics of the probabilistic timed automaton $\mathsf{P} = (L, \mathcal{X}, inv, prob, \mathcal{L})$ is the TMDP $\mathsf{T}[\mathsf{P}] = (S, \to, lab)$ where:

- $S = \{(l, v) \mid l \in L$ and $v \in \mathbb{R}_{\geq 0}^{\mathcal{X}}$ s.t. $v \models inv(l)\}$;
- $\to$ is the smallest set such that $((l, v), d, \mu) \in \to$ if there exist $d \in \mathbb{R}_{\geq 0}$ and a probabilistic edge $(l, g, p) \in prob$ where:
  1. $v + d \models g$, and $v + d' \models inv(l)$ for all $0 \leq d' \leq d$;
  2. for any $(X, l') \in 2^{\mathcal{X}} \times L$, we have that $p(X, l') > 0$ implies $(v + d)[X := 0] \models inv(l')$;
  3. for any $(l', v') \in S$, we have that $\mu(l', v') = \sum_{X \in \mathsf{Reset}(v, d, v')} p(X, l')$, where $\mathsf{Reset}(v, d, v') = \{X \subseteq \mathcal{X} \mid (v + d)[X := 0] = v'\}$.
- lab is such that $lab(l, v) = \mathcal{L}(l)$ for each state $(l, v) \in S$.

We restrict our attention to PTA P with a semantic TMDP $\mathsf{T}[\mathsf{P}]$ which is total. This can be guaranteed by a syntactic condition on PTA which has been presented in [24], and which holds generally for PTA models in practice [11].

We say that σ is a strategy of P if σ is a strategy of $\Sigma_{\mathsf{T}[\mathsf{P}]}$. Given a path $r = (l_0, v_0) \xrightarrow{d_0,\mu_0} (l_1, v_1) \xrightarrow{d_1,\mu_1} \cdots$ of $\mathsf{T}[\mathsf{P}]$, for every $i \in \mathbb{N}$, we use $r(i, d)$, with $0 \leq d \leq d_i$, to denote the state $(l_i, v_i + d)$ reached from (l_i, v_i) after delaying d time units. A pair (i, d) is called a *position* of r. We define a total order on positions of r: given positions $(i, d), (j, d')$ of r, the position (i, d) precedes (j, d') — denoted $(i, d) \prec_r (j, d')$ — if and only if either $i < j$, or $i = j$ and $d < d'$.

To reason about time divergence in the remainder of the paper, we construct a modified PTA in the following manner [21,22]. First we add a new atomic proposition *tick* to *AP*. Given a PTA $\mathsf{P} = (L, \mathcal{X}, \mathit{inv}, \mathit{prob}, \mathcal{L})$, its *enlarged PTA* $\mathsf{P}' = (L', \mathcal{X}', \mathit{inv}', \mathit{prob}', \mathcal{L}')$ is constructed as follows. For each location $l \in L$, we introduce a new location $\bar{l}$. Let $L' = L \cup \{\bar{l} \mid l \in L\}$, and $\mathcal{X}' = \mathcal{X} \cup \{\mathfrak{z}\}$. For each $l \in L$, let $\mathit{inv}'(l) = \mathit{inv}'(\bar{l}) = \mathit{inv}(l) \wedge (\mathfrak{z} \leq 1)$. Let $\mathit{prob}' = \mathit{prob} \cup \{(l, (\mathfrak{z} = 1), p_{(\emptyset,\bar{l})}), (\bar{l}, (\mathfrak{z} = 1), p_{(\{\mathfrak{z}\},l)}) \mid l \in L\}$, where $p_{(\emptyset,\bar{l})}$ and $p_{(\{\mathfrak{z}\},l)}$ are the distributions assigning probability 1 to the elements $(\emptyset, \bar{l})$ and $(\{\mathfrak{z}\}, l)$, respectively. Finally, let $\mathcal{L}'(l) = \mathcal{L}(l)$ and $\mathcal{L}'(\bar{l}) = \mathcal{L}(l) \cup \{\mathit{tick}\}$ for each $l \in L$. Note that *tick* becomes true at all natural numbered time points after the start of execution of the PTA. In the remainder of the paper, we assume that all considered PTA are enlarged.

Probabilistic timed temporal logic. We now describe a *probabilistic*, *timed* temporal logic which combines PCTL [4,25] and TCTL [12,13], and which can be used to specify properties of probabilistic timed automata [6]. Assume that the PTA $\mathsf{P} = (L, \mathcal{X}, \mathit{inv}, \mathit{prob}, \mathcal{L})$ is fixed. Let $\mathcal{Z}$ be a finite set of clocks disjoint from $\mathcal{X}$. Clocks in the set $\mathcal{Z}$ are called *formula clocks*. Valuations of formula clocks are denoted by $w : \mathcal{Z} \to \mathbb{R}_{\geq 0}$. The formulae of PTCTL (Probabilistic Timed Computation Tree Logic) are given by the following grammar:

$$\Phi ::= a \mid z \sim c \mid z \cdot \Phi \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathbb{P}_{\bowtie\zeta}(\Phi \mathcal{U} \Phi)$$

where $a \in AP$ is an atomic proposition, $z \in \mathcal{Z}$ is a formula clock, $\sim \in \{<, \leq, =, \geq, >\}$, $c \in \mathbb{N}$ is a natural number, $\bowtie \in \{<, \leq, \geq, >\}$, and $\zeta \in [0, 1]$ is a probability.

We proceed to define the satisfaction relation of PTCTL for TMDPs. Given the infinite path $r = s_0 \xrightarrow{d_0,\mu_0} s_1 \xrightarrow{d_1,\mu_1} \cdots$ of the TMDP T, let $\mathsf{Dur}(r, i, d) = d + \sum_{0 \leq k < i} d_k$ be the accumulated duration along r until position (i, d). Given a set of strategies $\Sigma \subseteq \Sigma_{\mathsf{T}[\mathsf{P}]}$ of P, and a PTCTL formula Φ, we define the satisfaction relation $\models_\Sigma$ of PTCTL as in Figure 2.

In the following, for simplicity, we generally encode formula clock valuations within the state of a PTA: that is, a state $(l, v) \in S$ consists of a location l and a clock valuation $v \in \mathbb{R}_{\geq 0}^{(\mathcal{X} \cup \mathcal{Z})}$. This allows us to write $s \models_\Sigma \Phi$ rather than $s, w \models_\Sigma \Phi$.

The model-checking problem for a PTA P and a PTCTL formula Φ, given a set $\Sigma \subseteq \Sigma_{\mathsf{T}[\mathsf{P}]}$ of strategies, consists of computing the set $[\![\Phi]\!]_\Sigma = \{s \in S \mid s \models_\Sigma \Phi\}$.

$$
\begin{array}{ll}
s, w \models_\Sigma a & \text{iff } a \in \mathit{lab}(s) \\
s, w \models_\Sigma z \sim c & \text{iff } w(z) \sim c \\
s, w \models_\Sigma z \cdot \Phi & \text{iff } s, w[z := 0] \models_\Sigma \Phi \\
s, w \models_\Sigma \Phi_1 \wedge \Phi_2 & \text{iff } s, w \models_\Sigma \Phi_1 \text{ and } s, w \models_\Sigma \Phi_2 \\
s, w \models_\Sigma \neg\Phi & \text{iff } s, w \not\models_\Sigma \Phi \\
s, w \models_\Sigma \mathbb{P}_{\bowtie\zeta}(\varphi) & \text{iff } \mathit{Prob}_s^\sigma\{r \in \mathit{Path}_{\mathit{ful}}^\sigma(s) \mid r, w \models_\Sigma \varphi\} \bowtie \zeta \text{ for all } \sigma \in \Sigma \\
r, w \models_\Sigma \Phi_1 \mathcal{U} \Phi_2 & \text{iff } \exists \text{ position } (i, \delta) \text{ of } r \text{ s.t. } r(i, \delta), w + \mathsf{Dur}(r, i, \delta) \models_\Sigma \Phi_2, \\
& \quad \text{and } \forall \text{ positions } (j, \delta') \text{ of } r \text{ s.t. } (j, \delta') \prec_r (i, \delta) \text{ we have} \\
& \quad r(j, \delta'), w + \mathsf{Dur}(r, j, \delta') \models_\Sigma \Phi_1 \vee \Phi_2 \ .
\end{array}
$$

Fig. 2. Semantics of PTCTL

When clear from the context, we write $[\![\Phi]\!]$ rather than $[\![\Phi]\!]_\Sigma$. From [6,15], the PTCTL model-checking problem with respect to the full set $\Sigma_{\mathsf{T[P]}}$ of strategies is EXPTIME-complete.

3 Probabilistic Divergence

In this section, we give an EXPTIME algorithm for PTCTL model checking of PTA with the definition of probabilistically-divergent strategies of [6]. Throughout this section, we assume that the PTA $\mathsf{P} = (L, \mathcal{X}, \mathit{inv}, \mathit{prob}, \mathcal{L})$ and the PTCTL formula Φ, which has a set $\mathcal{Z}$ of formula clocks, are fixed. A path $r \in \mathit{Path}_{\mathit{ful}}$ is *divergent* if $\lim_{k\to\infty} \mathsf{Dur}(r, k, 0) = \infty$. Let $\mathsf{Timediv}$ be the set of divergent paths. A strategy $\sigma \in \Sigma_{\mathsf{T[P]}}$ is *probabilistically divergent* if, for all states $s \in S$, we have $\mathit{Prob}_s^\sigma(\mathsf{Timediv}) = 1$. The set of all probabilistically divergent strategies of P is denoted by $\Sigma_{\mathsf{P}}^{\mathsf{Pd}}$.

Region MDP. Our first task is to construct an MDP from P and Φ by using the standard region graph construction [2,6]. For $t \in \mathbb{R}_{\geq 0}$, we let $\mathsf{frac}(t) = t - \lfloor t \rfloor$. For each clock $x \in \mathcal{X} \cup \mathcal{Z}$, we let c_x be the maximal constant to which x is compared in any of the guards of probabilistic edges or invariants of P, or in a clock constraint in the formula Φ (if x is not involved in any clock constraint of P or Φ, we let $c_x = 1$). Two clock valuations $v, v' \in \mathbb{R}_{\geq 0}^{(\mathcal{X} \cup \mathcal{Z})}$ are *clock equivalent* if the following conditions are satisfied: (1) for all clocks $x \in \mathcal{X} \cup \mathcal{Z}$, we have $v(x) \leq c_x$ if and only if $v'(x) \leq c_x$; (2) for all clocks $x \in \mathcal{X} \cup \mathcal{Z}$ with $v(x) \leq c_x$, we have $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$; (3) for all clocks $x, y \in \mathcal{X} \cup \mathcal{Z}$ with $v(x) \leq c_x$ and $v(y) \leq c_y$, we have $\mathsf{frac}(v(x)) \leq \mathsf{frac}(v(y))$ if and only if $\mathsf{frac}(v'(x)) \leq \mathsf{frac}(v'(y))$; and (4) for all clocks $x \in \mathcal{X} \cup \mathcal{Z}$ with $v(x) \leq c_x$, we have $\mathsf{frac}(v(x)) = 0$ if and only if $\mathsf{frac}(v'(x)) = 0$. We use α and β to refer to classes of clock equivalence.

Two states $(l, v), (l', v')$ are *region equivalent* if (1) $l = l'$, and (2) v and v' are clock equivalent. A *region* is an equivalence class of region equivalence. Let $\mathsf{Regions}$ be the set of regions of P and Φ. The number of regions corresponding to the PTA P and the PTCTL formula Φ is bounded by $|L| \cdot \prod_{x \in \mathcal{X} \cup \mathcal{Z}} (c_x + 1) \cdot |\mathcal{X} \cup \mathcal{Z}|! \cdot 2^{|\mathcal{X} \cup \mathcal{Z}|}$.

The set of regions of a PTA P and the PTCTL formula Φ can be used to construct an untimed, finite-state MDP $\mathsf{Reg}[\mathsf{P},\Phi] = (\mathsf{Regions}, \rightarrow_{\mathsf{Reg}}, lab_{\mathsf{Reg}})$ in the following way. The set of states of $\mathsf{Reg}[\mathsf{P},\Phi]$ is the set $\mathsf{Regions}$ of regions. The transition relation $\rightarrow_{\mathsf{Reg}} \subseteq \mathsf{Regions} \times \mathsf{Dist}(\mathsf{Regions})$ is the smallest set such that $((l,\alpha),\nu) \in \rightarrow_{\mathsf{Reg}}$ if there exists $((l,v),d,\mu) \in \rightarrow$ such that (1) $v \in \alpha$, and (2) for each $(l',\beta) \in \mathsf{Regions}$ such that there exists $(l',v') \in \mathsf{support}(\mu)$ and $v' \in \beta$ (by definition, this (l',v') will be unique), we have $\nu(l',\beta) = \mu(l',v')$, otherwise $(l',\beta) = 0$. For each region $(l,\alpha) \in \mathsf{Regions}$, we let $lab_{\mathsf{Reg}}(l,\alpha) = \mathcal{L}(l)$.

Given a clock valuation v, the unique clock equivalence class to which v belongs is denoted by $[v]$. Given a state $(l,v) \in S$, the unique region to which (l,v) belongs is $(l,[v])$, and is denoted by $[(l,v)]$. An infinite path $r = s_0 \xrightarrow{d_0,\mu_0} s_1 \xrightarrow{d_1,\mu_1} \cdots$ of $\mathsf{T}[\mathsf{P}]$ corresponds to a unique infinite path $[r] = [s_0] \xrightarrow{\nu_0} [s_1] \xrightarrow{\nu_1} \cdots$. Similarly, a finite path $r = s_0 \xrightarrow{d_0,\mu_0} s_1 \xrightarrow{d_1,\mu_1} \cdots \xrightarrow{d_{n-1},\mu_{n-1}} s_n$ of $\mathsf{T}[\mathsf{P}]$ corresponds to a unique finite path $[r] = [s_0] \xrightarrow{\nu_0} [s_1] \xrightarrow{\nu_1} \cdots \xrightarrow{\nu_{n-1}} [s_n]$.

Probabilistically divergent strategies on $\mathsf{Reg}[\mathsf{P},\Phi]$. In the following, we use LTL notation (see, for example, [1]), which is interpreted on paths of $\mathsf{Reg}[\mathsf{P},\Phi]$ in the standard way. An infinite path r of $\mathsf{Reg}[\mathsf{P},\Phi]$ is *region divergent* if it satisfies the condition $\Box\Diamond tick$. Note that an infinite path r of $\mathsf{T}[\mathsf{P}]$ is divergent if and only if $[r]$ is region divergent. Hence $[\mathsf{Timediv}] = \bigcup_{r \in \mathsf{Timediv}} [r] = \{\mathsf{r} \in Path_{ful}^{\mathsf{Reg}[\mathsf{P},\Phi]} \mid \mathsf{r} \models \Box\Diamond tick\}$ is the set of all region divergent runs (where $\models$ is the standard satisfaction for LTL properties on finite-state systems [1]). A strategy $\sigma \in \Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}$ is *probabilistically region divergent* if, for all regions $R \in \mathsf{Regions}$, we have $Prob_R^\sigma(\Box\Diamond tick) = 1$. The set of all probabilistically region divergent strategies of $\mathsf{Reg}[\mathsf{P},\Phi]$ is denoted by $\Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}^{\mathsf{Pd}}$.

We can check whether there exists a probabilistically region divergent strategy of $\mathsf{Reg}[\mathsf{P},\Phi]$ by computing the set of regions from which it is possible to satisfy $\Box\Diamond tick$ with probability 1, then comparing this set to $\mathsf{Regions}$. Formally, we compute the set of regions of $\mathsf{Reg}[\mathsf{P},\Phi]$, denoted by $[\![\neg\mathbb{P}_{<1}(\Box\Diamond tick)]\!]$, for which $R \in [\![\neg\mathbb{P}_{<1}(\Box\Diamond tick)]\!]$ if and only if there exists a strategy $\sigma \in \Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}$ such that $Prob_R^\sigma(\Box\Diamond tick) = 1$. If $[\![\neg\mathbb{P}_{<1}(\Box\Diamond tick)]\!] \neq \mathsf{Regions}$, then $\mathsf{Reg}[\mathsf{P},\Phi]$ does not have a probabilistically region divergent strategy. We note that, for the region $R = (l,\alpha)$, we have $(l,\alpha) \in [\![\neg\mathbb{P}_{<1}(\Box\Diamond tick)]\!]$ if and only if there exists $\sigma \in \Sigma_{\mathsf{T}[\mathsf{P}]}$ such that $Prob_{(l,v)}^\sigma(\mathsf{Timediv}) = 1$ for all $v \in \alpha$. The set $[\![\neg\mathbb{P}_{<1}(\Box\Diamond tick)]\!]$ can be computed on $\mathsf{Reg}[\mathsf{P},\Phi]$ using polynomial-time algorithms for Büchi objectives of Markov decision processes [26]. In the remainder of this section, we assume that $\mathsf{Reg}[\mathsf{P},\Phi]$ has at least one probabilistically region divergent strategy; that is, $[\![\neg\mathbb{P}_{<1}(\Box\Diamond tick)]\!] = \mathsf{Regions}$. If this is not the case, we compute the sub-MDP of $\mathsf{Reg}[\mathsf{P},\Phi]$ induced by $[\![\neg\mathbb{P}_{<1}(\Box\Diamond tick)]\!]$ and use it in the place of $\mathsf{Reg}[\mathsf{P},\Phi]$.

PTCTL **model checking with probabilistic divergence.** The only formula which depends on the notion of strategy is $\mathbb{P}_{\bowtie\zeta}(\Phi_1 \mathcal{U} \Phi_2)$, and hence we consider sub-formulae of Φ of this form. We assume that, for each state $s \in S$, we have $s \models_{\Sigma_{\mathsf{P}}^{\mathsf{Pd}}} \Phi_i$ if and only if $[s] \models_{\Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}^{\mathsf{Pd}}} \Phi_i$ for $i \in \{1,2\}$. From standard reasoning, for any path $r \in Path_{ful}$ of $\mathsf{T}[\mathsf{P}]$, we have $r \models_{\Sigma} \Phi_1 \mathcal{U} \Phi_2$ if and only if $[r] \models \Phi_1 \mathcal{U} \Phi_2$, where Σ is an arbitrary set of strategies.

Proposition 1. *(1) Let $\sigma \in \Sigma_{\mathsf{P}}^{\mathsf{Pd}}$ be a probabilistically divergent strategy. Then there exists a probabilistically region divergent strategy $\sigma' \in \Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}^{\mathsf{Pd}}$ such that $Prob_s^{\sigma}(\Phi_1 \mathcal{U} \Phi_2) = Prob_{[s]}^{\sigma'}(\Phi_1 \mathcal{U} \Phi_2)$ for all states $s \in S$. (2) Let $\sigma \in \Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}^{\mathsf{Pd}}$ be a probabilistically region divergent strategy. Then there exists a probabilistically divergent strategy $\sigma' \in \Sigma_{\mathsf{P}}^{\mathsf{Pd}}$ such that $Prob_{[s]}^{\sigma}(\Phi_1 \mathcal{U} \Phi_2) = Prob_s^{\sigma'}(\Phi_1 \mathcal{U} \Phi_2)$ for all states $s \in S$.*

Corollary 1. *For any $s \in S$, we have $s \models_{\Sigma_{\mathsf{P}}^{\mathsf{Pd}}} \mathbb{P}_{\bowtie\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ if and only if $[s] \models_{\Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}^{\mathsf{Pd}}} \mathbb{P}_{\bowtie\zeta}(\Phi_1 \mathcal{U} \Phi_2)$.*

Corollary 1 follows from Proposition 1 and the semantics of PTCTL. Therefore it suffices to consider resolving properties of the form $\mathbb{P}_{\bowtie\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ on $\mathsf{Reg}[\mathsf{P},\Phi]$. We now make a case distinction based on whether $\mathbb{P}_{\bowtie\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ is of the form (A) $\mathbb{P}_{\leq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ or $\mathbb{P}_{<\zeta}(\Phi_1 \mathcal{U} \Phi_2)$, or (B) $\mathbb{P}_{\geq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ or $\mathbb{P}_{>\zeta}(\Phi_1 \mathcal{U} \Phi_2)$. In the remainder of this section, we generally omit the subscript from the sets of strategies of $\mathsf{Reg}[\mathsf{P},\Phi]$, and write Σ for $\Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}$, and Σ^{Pd} for $\Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}^{\mathsf{Pd}}$.

Case (A): properties of the form $\mathbb{P}_{\leq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ or $\mathbb{P}_{<\zeta}(\Phi_1 \mathcal{U} \Phi_2)$. The following proposition states that any strategy of $\mathsf{Reg}[\mathsf{P},\Phi]$ can be transformed into a probabilistically region divergent strategy which assigns the same or greater probability to the satisfaction of $\Phi_1 \mathcal{U} \Phi_2$.

Proposition 2. *Let $\sigma \in \Sigma$ be a strategy of $\mathsf{Reg}[\mathsf{P},\Phi]$ and $R \in \mathsf{Regions}$ be a region. There exists a probabilistically region divergent strategy $\sigma' \in \Sigma^{\mathsf{Pd}}$ such that $Prob_R^{\sigma}(\Phi_1 \mathcal{U} \Phi_2) \leq Prob_R^{\sigma'}(\Phi_1 \mathcal{U} \Phi_2)$.*

Proposition 2, together with the fact that $\Sigma^{\mathsf{Pd}} \subseteq \Sigma$, establishes that there exists a probabilistically region divergent strategy which assigns the same probability to satisfying $\Phi_1 \mathcal{U} \Phi_2$ as a maximal – but not necessarily divergent – strategy of $\mathsf{Reg}[\mathsf{P},\Phi]$. Combining this fact with standard methods for finite-state MDPs [25,17], we conclude that $\mathsf{Reg}[\mathsf{P},\Phi]$ can be used directly to compute maximal probabilities of until formulae.

Theorem 1. *Let $R \in \mathsf{Regions}$ be a region. Then $R \models_{\Sigma^{\mathsf{Pd}}} \mathbb{P}_{\leq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ (respectively, $\mathbb{P}_{<\zeta}(\Phi_1 \mathcal{U} \Phi_2)$) if and only if $R \models_{\Sigma} \mathbb{P}_{\leq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ (respectively, $\mathbb{P}_{<\zeta}(\Phi_1 \mathcal{U} \Phi_2)$).*

Case (B): properties of the form $\mathbb{P}_{\geq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ or $\mathbb{P}_{>\zeta}(\Phi_1 \mathcal{U} \Phi_2)$. To avoid overloading the subsequent notation, we consider properties in the form $\mathbb{P}_{\geq\zeta}(\neg\Phi_1 \mathcal{U} \neg\Phi_2)$ or $\mathbb{P}_{>\zeta}(\neg\Phi_1 \mathcal{U} \neg\Phi_2)$. Observe that $\neg(\neg\Phi_1 \mathcal{U} \neg\Phi_2) \equiv \Phi_2 \mathcal{U}(\Phi_1 \wedge \Phi_2) \vee \Box(\neg\Phi_1 \wedge \Phi_2)$ (from classical reasoning about temporal logic). Therefore, the maximal probability over probabilistically region divergent strategies of satisfying $\Phi_2 \mathcal{U}(\Phi_1 \wedge \Phi_2) \vee \Box(\neg\Phi_1 \wedge \Phi_2)$ equals 1 minus the minimal probability over probabilistically region divergent strategies of satisfying $\neg\Phi_1 \mathcal{U} \neg\Phi_2$. Hence, our aim is to compute the maximal probability over probabilistically region divergent strategies of satisfying $\Phi_2 \mathcal{U}(\Phi_1 \wedge \Phi_2) \vee \Box(\neg\Phi_1 \wedge \Phi_2)$.

We introduce a notion of *time-divergent EC*. A time-divergent EC (C, D) is an EC of $\mathsf{Reg}[\mathsf{P}, \Phi]$ such that $tick \in lab_{\mathsf{Reg}}(R)$ for some region $R \in C$ (a similar definition is featured in [5]). For an infinite path $\mathsf{r} \in Path_{ful}^{\mathsf{Reg}[\mathsf{P},\Phi]}$, let $C_{\mathsf{r}} = \{R \mid \overset{\infty}{\exists}\, i \geq 0 . \mathsf{r}(i) = R\}$ and $D_{\mathsf{r}} = \{(R, \nu) \mid \overset{\infty}{\exists}\, i \geq 0 . R \in C_{\mathsf{r}} \wedge step(\mathsf{r}, i) = \nu\}$. Let $\mathrm{Inf}(\mathsf{r}) = (C_{\mathsf{r}}, D_{\mathsf{r}})$. Note that a path $\mathsf{r} \in Path_{ful}^{\mathsf{Reg}[\mathsf{P},\Phi]}$ of $\mathsf{Reg}[\mathsf{P}, \Phi]$ is region divergent if and only if $\mathrm{Inf}(\mathsf{r})$ is a time-divergent EC. For $C \subseteq \mathsf{Regions}$ and $D \subseteq \rightarrow_{\mathsf{Reg}}$, let $Path_{ful}^{(C,D)}(R) = \{\mathsf{r} \in Path_{ful}^{\mathsf{Reg}[\mathsf{P},\Phi]}(R) \mid \mathrm{Inf}(\mathsf{r}) = (C, D)\}$. The next lemma adapts to probabilistic divergence a fundamental result for ECs [5], and states that a probabilistically region divergent strategy will be confined eventually to time-divergent ECs with probability 1.

Lemma 1. *Let $\mathcal{E}$ be the set of time-divergent ECs of $\mathsf{Reg}[\mathsf{P}, \Phi]$, let $R \in \mathsf{Regions}$ and let $\sigma \in \Sigma^{\mathsf{Pd}}$. Then $Prob_R^{\sigma}(\bigcup_{(C,D) \in \mathcal{E}} Path_{ful}^{(C,D)}(R)) = 1$.*

Let $\mathsf{U} \subseteq \mathsf{Regions}$ be a set of regions. The set $\mathcal{M}_{\mathsf{U}}$ of time-divergent maximal ECs within U can be computed as follows: first compute the set of maximal ECs of the sub-MDP of $\mathsf{Reg}[\mathsf{P}, \Phi]$ induced by U by the standard maximal EC computation algorithm of [5], then include in $\mathcal{M}_{\mathsf{U}}$ only those maximal ECs with at least one region labelled by *tick*.

We compute the set $\mathcal{M}_{[\![\neg\Phi_1 \wedge \Phi_2]\!]}$ of time-divergent maximal ECs within the set of states satisfying $\neg\Phi_1 \wedge \Phi_2$. Let $\mathsf{U}_{\neg\Phi_1 \wedge \Phi_2} = \bigcup_{(C,D) \in \mathcal{M}_{[\![\neg\Phi_1 \wedge \Phi_2]\!]}} C$ be the set of regions corresponding to $\mathcal{M}_{[\![\neg\Phi_1 \wedge \Phi_2]\!]}$. By abuse of notation, we use $\mathsf{U}_{\neg\Phi_1 \wedge \Phi_2}$ as an atomic proposition such that $R \models_{\Sigma} \mathsf{U}_{\neg\Phi_1 \wedge \Phi_2}$ if and only if $R \in \mathsf{U}_{\neg\Phi_1 \wedge \Phi_2}$. Note that, by Proposition 2, for any strategy $\sigma \in \Sigma$ and $R \in \mathsf{Regions}$, there exists a probabilistically region divergent strategy $\sigma' \in \Sigma^{\mathsf{Pd}}$ such that:

$$Prob_R^{\sigma}(\Phi_2 \mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{U}_{\neg\Phi_1 \wedge \Phi_2})) \leq Prob_R^{\sigma'}(\Phi_2 \mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{U}_{\neg\Phi_1 \wedge \Phi_2})) \; .$$

Proposition 3. *Let $R \in \mathsf{Regions}$ be a region and $\sigma \in \Sigma^{\mathsf{Pd}}$ be a probabilistically region divergent strategy of $\mathsf{Reg}[\mathsf{P}, \Phi]$. There exists a probabilistically region divergent strategy $\sigma' \in \Sigma^{\mathsf{Pd}}$ such that:*

$$Prob_R^{\sigma}(\Phi_2 \mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{U}_{\neg\Phi_1 \wedge \Phi_2})) \leq Prob_R^{\sigma'}(\Phi_2 \mathcal{U}(\Phi_1 \wedge \Phi_2) \vee \Box(\neg\Phi_1 \wedge \Phi_2)) \; .$$

Proposition 4. *Let $R \in \mathsf{Regions}$ be a region and $\sigma \in \Sigma^{\mathsf{Pd}}$ be a probabilistically region divergent strategy of $\mathsf{Reg}[\mathsf{P}, \Phi]$. Then:*

$$Prob_R^{\sigma}(\Phi_2 \mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{U}_{\neg\Phi_1 \wedge \Phi_2})) \geq Prob_R^{\sigma}(\Phi_2 \mathcal{U}(\Phi_1 \wedge \Phi_2) \vee \Box(\neg\Phi_1 \wedge \Phi_2)) \; .$$

The subsequent theorem then follows from Proposition 2, Proposition 3, Proposition 4, and the fact that $\Sigma^{\mathsf{Pd}} \subseteq \Sigma$.

Theorem 2. *Let $R \in \mathsf{Regions}$ be a region. Then $R \models_{\Sigma^{\mathsf{Pd}}} \mathbb{P}_{\geq \zeta}(\neg\Phi_1 \mathcal{U} \neg\Phi_2)$ (respectively, $\mathbb{P}_{>\zeta}(\neg\Phi_1 \mathcal{U} \neg\Phi_2)$) if and only if $R \models_{\Sigma} \mathbb{P}_{\leq 1-\zeta}(\Phi_2 \mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{U}_{\neg\Phi_1 \wedge \Phi_2}))$ (respectively, $\mathbb{P}_{<1-\zeta}(\Phi_2 \mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{U}_{\neg\Phi_1 \wedge \Phi_2}))$).*

The following theorem is a consequence of Corollary 1, Theorem 1, Theorem 2 and the following facts: the size of $\mathsf{Reg}[\mathsf{P}, \Phi]$ is exponential in the size of P and Φ; computing the maximal probability of a formula of the form $\Phi_1 \mathcal{U} \Phi_2$ on MDPs is in polynomial-time [25,17]; model checking PTA against properties of the form $\neg\mathbb{P}_{<1}(\Diamond a)$ is EXPTIME-hard [15].

Theorem 3. *Let* P *be a PTA and* Φ *be a formula of* PTCTL. *Then the problem of computing the set* $[\![\Phi]\!]$ *for* P *under probabilistically divergent strategies is EXPTIME-complete.*

4 Strict Divergence

We now extend the model-checking algorithm for probabilistically divergent strategies to provide a model-checking algorithm for strictly divergent adversaries. Given a PTA P, a strategy $\sigma \in \Sigma_{\mathsf{T}[\mathsf{P}]}$ is *strictly divergent* if, for all states $s \in S$, we have $Path^{\sigma}_{ful}(s) \subseteq \mathsf{Timediv}$. The set of all probabilistically divergent strategies of P is denoted by $\Sigma^{\mathsf{Sd}}_{\mathsf{P}}$. An example of the difference between probabilistic and strict divergence for maximal reachability probabilities (or maximal probabilities of satisfying formulae of the form $\Phi_1 \mathcal{U} \Phi_2$) has been presented in the introduction. For minimal reachability (or $\Phi_1 \mathcal{U} \Phi_2$) probabilities, note that, in the PTA of Figure 3, the minimum probability of reaching l_2 from l_1 is 0 under probabilistically divergent strategies, but is 1 under strictly divergent strategies.

Strictly divergent strategies on $\mathsf{Reg}[\mathsf{P}, \Phi]$. A strategy $\sigma \in \Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}$ is *strictly region divergent* if, for all regions $R \in \mathsf{Regions}$, all paths $\mathsf{r} \in Path^{\sigma}_{ful}(R)$ are region divergent. The set of strictly region divergent strategies of $\mathsf{Reg}[\mathsf{P}, \Phi]$ is denoted by $\Sigma^{\mathsf{Sd}}_{\mathsf{Reg}[\mathsf{P},\Phi]}$. We generally write Σ instead of $\Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}$, and Σ^{Sd} instead of $\Sigma^{\mathsf{Sd}}_{\mathsf{Reg}[\mathsf{P},\Phi]}$.

Similarly to the case of Section 3, we can check whether there exists a strictly region divergent strategy of $\mathsf{Reg}[\mathsf{P}, \Phi]$ by computing the set of regions from which it is possible to satisfy $\Box\Diamond tick$ on all paths. For this purpose, we compute the set of regions satisfying the ATL [27] formula $\langle\!\langle N \rangle\!\rangle(\Box\Diamond tick)$, where $\mathsf{Reg}[\mathsf{P}, \Phi]$ is interpreted as a turn-based game with 2 players: player N corresponds to nondeterministic choice between transitions from a region, whereas player P refers to choice between probabilistic alternatives corresponding to a

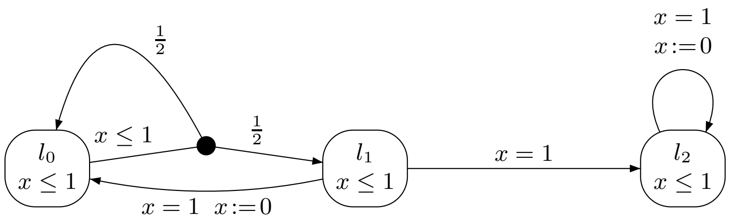


Fig. 3. Minimal reachability probabilities under probabilistic and strict divergence

transition. Then the formula $\langle\langle N \rangle\rangle(\Box\Diamond \mathit{tick})$ expresses the property that player N has the aim of ensuring region time divergence, regardless of the choices of player P. Formally, $[\![\langle\langle N \rangle\rangle(\Box\Diamond \mathit{tick})]\!] = \{R \in \mathsf{Regions} \mid \exists \sigma \in \Sigma_{\mathsf{Reg}[\mathsf{P},\Phi]}.\forall \mathsf{r} \in \mathit{Path}^{\sigma}_{\mathit{ful}}(R).\mathsf{r} \models_{\Sigma} \Box\Diamond \mathit{tick}\}$. We note that, for the region $R = (l, \alpha)$, we have $(l, \alpha) \in [\![\langle\langle N \rangle\rangle(\Box\Diamond \mathit{tick})]\!]$ if and only if there exists $\sigma \in \Sigma_{\mathsf{T}[\mathsf{P}]}$ such that r is divergent for all paths $r \in \mathit{Path}^{\sigma}_{\mathit{ful}}(l, v)$, for all $v \in \alpha$. In order to compute $[\![\langle\langle N \rangle\rangle(\Box\Diamond \mathit{tick})]\!]$, we rely on standard methods for obtaining the winning states in 2-player turn-based games with Büchi objectives [26]. In the remainder of this section, we assume that $\mathsf{Reg}[\mathsf{P},\Phi]$ has at least one strictly region divergent strategy; that is, $[\![\langle\langle N \rangle\rangle(\Box\Diamond \mathit{tick})]\!] = \mathsf{Regions}$. If this is not the case, we compute the sub-MDP of $\mathsf{Reg}[\mathsf{P},\Phi]$ induced by $[\![\langle\langle N \rangle\rangle(\Box\Diamond \mathit{tick})]\!]$ and use the new sub-MDP in the place of $\mathsf{Reg}[\mathsf{P},\Phi]$.

PTCTL **model checking with strict divergence.** We now describe a PTCTL model-checking algorithm for the semantics under strictly divergent strategies. The mechanism that we add to the PTCTL model-checking algorithm in order to cater for strict divergence is inspired by similar results of [17,18], and takes the form of the following: a set $\mathsf{T}^{\max}$ of regions of $\mathsf{Reg}[\mathsf{P},\Phi]$ is computed from which it is guaranteed that there exists an optimal (maximal or minimal probability), strictly divergent strategy. From regions not in $\mathsf{T}^{\max}$, there does not exist such a strategy. However, from regions not in $\mathsf{T}^{\max}$, we show that we can approximate arbitrarily closely an optimal, probabilistically divergent strategy.

We first present an analogue of Proposition 1 adapted to strict divergence.

Proposition 5. *(1) Let $\sigma \in \Sigma^{\mathsf{Sd}}_{\mathsf{P}}$ be a strictly divergent strategy. Then there exists a strictly region divergent strategy $\sigma' \in \Sigma^{\mathsf{Sd}}_{\mathsf{Reg}[\mathsf{P},\Phi]}$ such that $\mathit{Prob}^{\sigma}_{s}(\Phi_1 \mathcal{U} \Phi_2) = \mathit{Prob}^{\sigma'}_{[s]}(\Phi_1 \mathcal{U} \Phi_2)$ for all states $s \in S$. (2) Let $\sigma \in \Sigma^{\mathsf{Sd}}_{\mathsf{Reg}[\mathsf{P},\Phi]}$ be a strictly region divergent strategy. Then there exists a strictly divergent strategy $\sigma' \in \Sigma^{\mathsf{Sd}}_{\mathsf{P}}$ such that $\mathit{Prob}^{\sigma}_{[s]}(\Phi_1 \mathcal{U} \Phi_2) = \mathit{Prob}^{\sigma'}_{s}(\Phi_1 \mathcal{U} \Phi_2)$ for all states $s \in S$.*

Corollary 2. *For any $s \in S$, we have $s \models_{\Sigma^{\mathsf{Sd}}_{\mathsf{P}}} \mathbb{P}_{\bowtie\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ if and only if $[s] \models_{\Sigma^{\mathsf{Sd}}_{\mathsf{Reg}[\mathsf{P},\Phi]}} \mathbb{P}_{\bowtie\zeta}(\Phi_1 \mathcal{U} \Phi_2)$.*

Therefore, as in Section 3, it suffices to resolve properties of the form $\mathbb{P}_{\bowtie\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ on $\mathsf{Reg}[\mathsf{P},\Phi]$. We make a case distinction based on whether $\mathbb{P}_{\bowtie\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ is of the form (A) $\mathbb{P}_{\leq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ or $\mathbb{P}_{<\zeta}(\Phi_1 \mathcal{U} \Phi_2)$, or (B) $\mathbb{P}_{\geq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ or $\mathbb{P}_{>\zeta}(\Phi_1 \mathcal{U} \Phi_2)$.

Case (A): properties of the form $\mathbb{P}_{\leq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ or $\mathbb{P}_{<\zeta}(\Phi_1 \mathcal{U} \Phi_2)$. Recall that the example in the introduction shows that a PTA may exhibit a strategy with a certain probability of reaching a location under probabilistic divergence, but, under strict divergence, there may not exist a strategy with the same probability. However, we show that strictly region divergent strategies can approximate from below the probability of satisfying $\Phi_1 \mathcal{U} \Phi_2$ of an arbitrary strategy on $\mathsf{Reg}[\mathsf{P},\Phi]$.

Lemma 2. *Let $R \in \mathsf{Regions}$ and let $\sigma \in \Sigma$ be a strategy of $\mathsf{Reg}[\mathsf{P},\Phi]$. Then, for every $n \in \mathbb{N}$, there exists a strictly region divergent strategy $\sigma_n \in \Sigma^{\mathsf{Sd}}$ such that $\mathit{Prob}^{\sigma_n}_{R}(\Phi_1 \mathcal{U} \Phi_2) \geq \mathit{Prob}^{\sigma}_{R}(\Phi_1 \mathcal{U} \Phi_2) - \frac{1}{n}$.*

The intuition underlying the proof of the lemma is that we construct σ_n to behave the same as σ on all paths whose length does not exceed some constant c_n which depends on n. From the last regions of paths of length c_n, the strategy σ_n then behaves as a strictly region divergent strategy. From Lemma 1 of [28], given σ and n, such a constant can be found such that the lemma holds.

Given that it is possible to approximate arbitrarily closely using strictly divergent strategies the probability of satisfying a property $\Phi_1 \mathcal{U} \Phi_2$ of an arbitrary strategy, the case $\mathbb{P}_{\leq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ is not of interest: a maximal arbitrary strategy satisfies $\Phi_1 \mathcal{U} \Phi_2$ with probability greater than ζ if and only if there exists a strictly divergent strategy satisfying $\Phi_1 \mathcal{U} \Phi_2$ with probability greater than ζ. Hence, we concentrate on the case of $\mathbb{P}_{<\zeta}(\Phi_1 \mathcal{U} \Phi_2)$.

Let $R \in \mathsf{Regions}$, and $p_R^{\max}(\Phi_1 \mathcal{U} \Phi_2) = \max_{\sigma \in \Sigma} \mathit{Prob}_R^\sigma(\Phi_1 \mathcal{U} \Phi_2)$. Following [17,18], for each region $R \in \mathsf{Regions}$ of $\mathsf{Reg}[\mathsf{P}, \Phi]$, we define the set $\mathit{Max}(R, \Phi_1 \mathcal{U} \Phi_2)$. If $R \in \mathsf{Regions} \setminus [\![\Phi_1]\!]$, then we let $\mathit{Max}(R, \Phi_1 \mathcal{U} \Phi_2) = \{(R', \nu) \in \rightarrow_{\mathsf{Reg}} \mid R = R'\}$. If $R \in [\![\Phi_1]\!]$, then we let $\mathit{Max}(R, \Phi_1 \mathcal{U} \Phi_2)$ equal:

$$\left\{ (R, \nu) \in \rightarrow_{\mathsf{Reg}} \mid p_R^{\max}(\Phi_1 \mathcal{U} \Phi_2) = \sum_{R'' \in \mathsf{Regions}} \nu(R'') \cdot p_{R''}^{\max}(\Phi_1 \mathcal{U} \Phi_2) \right\} .$$

We use $\mathsf{M}^{\max}$ to denote the MDP obtained by removing from $\mathsf{Reg}[\mathsf{P}, \Phi]$ all transitions which are not in $\mathit{Max}(\_, \Phi_1 \mathcal{U} \Phi_2)$. Formally, let $\mathsf{M}^{\max} = (\mathsf{Regions}, \rightarrow_{\mathsf{Reg}}^{\max}, \mathit{lab}_{\mathsf{Reg}})$, where $\rightarrow_{\mathsf{Reg}}^{\max} = \bigcup_{R \in \mathsf{Regions}} \mathit{Max}(R, \Phi_1 \mathcal{U} \Phi_2)$.

Let $\mathsf{T} = [\![\Phi_2]\!] \cup \mathsf{Regions} \setminus \mathsf{Regions}^+(\Phi_1, \Phi_2)$, and let $[\![\exists \Diamond \mathsf{T}]\!] = \{R \in \mathsf{Regions} \mid \exists \mathsf{r} \in \mathit{Path}_{\mathit{fin}}(R) \text{ s.t. } \mathit{last}(\mathsf{r}) \in \mathsf{T}\}$. We apply the following algorithm to $\mathsf{M}^{\max}$.

1. Let U equal $\mathsf{Regions}$ and M equal $\mathsf{M}^{\max}$.
2. Repeat the following:
   (a) Let U equal either $[\![\langle\!\langle N \rangle\!\rangle(\Diamond \mathit{tick})]\!]$ or $[\![\exists \Diamond \mathsf{T}]\!]$, computed on M.
   (b) Compute the sub-MDP of M induced by U, and call this sub-MDP M.
   Until M cannot be changed by the above.

Let $\mathsf{T}^{\max}$ be the set of regions of the MDP obtained on termination of the algorithm. The strategies of this MDP do not select non-optimal transitions (from the definition of $\mathsf{M}^{\max}$), and can be both strictly region divergent *and* reach T with probability 1. We state this formally in relation to strategies of $\mathsf{Reg}[\mathsf{P}, \Phi]$ in the following lemma.

Lemma 3. *Let $R \in \mathsf{Regions}$. Then $R \in \mathsf{T}^{\max}$ if and only if there exists a strategy $\sigma \in \Sigma$ such that (1) for each $\mathsf{r} \in \mathit{Path}_{\mathit{fin}}^\sigma(R)$, we have $\sigma(\mathsf{r}) \in \mathit{Max}(\mathit{last}(\mathsf{r}), \Phi_1 \mathcal{U} \Phi_2)$, (2) $\sigma \in \Sigma^{\mathsf{Sd}}$ (σ is strictly region divergent) and (3) $\mathit{Prob}_R^\sigma(\Diamond \mathsf{T}) = 1$.*

We note the importance of reaching T with probability 1: strategies satisfying this requirement do not idle in a cycle of transitions. Such cycles, in which a strategy does not attempt to reach $[\![\Phi_2]\!]$, may be locally optimal (in the sense that transitions of $\mathsf{M}^{\max}$ are taken), but will be globally sub-optimal: another strategy which does not cycle, but attempts to reach $[\![\Phi_2]\!]$, will correspond to a higher probability of satisfying $\Phi_1 \mathcal{U} \Phi_2$.

Our next task is to show that, from regions in $\mathsf{T}^{\max}$, there exists a strictly region divergent strategy with the same probability of satisfying $\Phi_1 \mathcal{U} \Phi_2$ as for an optimal strategy which is not necessarily strictly region divergent.

Proposition 6. *Let $R \in \mathsf{T}^{\max}$ and $\sigma \in \Sigma^{\mathsf{Sd}}$ be such that $Prob^{\sigma}_{R}(\Diamond \mathsf{T}) = 1$ and, for each $\mathsf{r} \in Path^{\sigma}_{fin}(R)$, we have $\sigma(\mathsf{r}) \in Max(last(\mathsf{r}), \Phi_1 \mathcal{U} \Phi_2)$. Then $p^{\max}_{R}(\Phi_1 \mathcal{U} \Phi_2) = Prob^{\sigma}_{R}(\Phi_1 \mathcal{U} \Phi_2)$.*

Next, we show that, from regions not in $\mathsf{T}^{\max}$, it is not possible to find strictly region divergent strategies which obtain the probability of satisfying $\Phi_1 \mathcal{U} \Phi_2$ computed over arbitrary strategies on $\mathsf{Reg}[\mathsf{P}, \Phi]$.

Lemma 4. *Let $R \in \mathsf{Regions} \setminus \mathsf{T}^{\max}$. Then, for $\sigma \in \Sigma^{\mathsf{Sd}}$, we have $p^{\max}_{R}(\Phi_1 \mathcal{U} \Phi_2) > Prob^{\sigma}_{R}(\Phi_1 \mathcal{U} \Phi_2)$.*

The combination of Lemma 2, Proposition 6 and Lemma 4 allows us to obtain the following result.

Theorem 4. *Let $R \in \mathsf{Regions}$ be a region. Then:*

$$R \models_{\Sigma^{\mathsf{Sd}}} \mathbb{P}_{\leq \zeta}(\Phi_1 \mathcal{U} \Phi_2) \Leftrightarrow p^{\max}_{R}(\Phi_1 \mathcal{U} \Phi_2) \leq \zeta$$
$$R \models_{\Sigma^{\mathsf{Sd}}} \mathbb{P}_{< \zeta}(\Phi_1 \mathcal{U} \Phi_2) \Leftrightarrow \begin{cases} p^{\max}_{R}(\Phi_1 \mathcal{U} \Phi_2) < \zeta \text{ if } R \in \mathsf{T}^{\max} \\ p^{\max}_{R}(\Phi_1 \mathcal{U} \Phi_2) \leq \zeta \text{ otherwise.} \end{cases}$$

Case (B): properties of the form $\mathbb{P}_{\geq \zeta}(\Phi_1 \mathcal{U} \Phi_2)$ or $\mathbb{P}_{> \zeta}(\Phi_1 \mathcal{U} \Phi_2)$. Analogously to Section 3, we use the equivalence $\neg(\neg\Phi_1 \mathcal{U} \neg\Phi_2) \equiv \Phi_2 \mathcal{U}(\Phi_1 \wedge \Phi_2) \vee \Box(\neg\Phi_1 \wedge \Phi_2)$, and then compute the maximal probability over strictly region divergent strategies of satisfying the formula $\Phi_2 \mathcal{U}(\Phi_1 \wedge \Phi_2) \vee \Box(\neg\Phi_1 \wedge \Phi_2)$. This resulting probability corresponds to the 1 minus the minimal probability of strictly region divergent strategies satisfying $\neg\Phi_1 \mathcal{U} \neg\Phi_2$.

In Section 3, we computed the set of time-divergent maximal ECs in which $\Box(\neg\Phi_1 \wedge \Phi_2)$ was guaranteed. In the context of strict region divergence, time-divergent ECs do not suffice, because a strategy which confines itself to a time-divergent EC could exhibit a path (albeit with probability 0) which is not time divergent. Therefore we introduce strictly-divergent ECs. A *strictly-divergent EC* (C, D) is an EC of $\mathsf{Reg}[\mathsf{P}, \Phi]$ such that, for all regions $R \in C$, we have $R \in [\![\langle\langle N \rangle\rangle(\Diamond tick)]\!]$ in the sub-MDP (C, D) (hence the strategy witnessing $\Diamond tick$ for all paths from R is a strategy of (C, D)). Intuitively, a strategy can guarantee strict region divergence and remain within a strictly-divergent EC (C, D) by choosing transitions according to a strategy of the sub-MDP (C, D) which witnesses $\langle\langle N \rangle\rangle(\Diamond tick)$; then, after a $tick$-region is visited, the strategy again chooses transitions according to a strategy which, starting from the current region, witnesses $\langle\langle N \rangle\rangle(\Diamond tick)$, and so on. We also obtain an analogue of Lemma 1.

Lemma 5. *Let $\mathcal{F}$ be the set of strictly-divergent ECs of $\mathsf{Reg}[\mathsf{P}, \Phi]$, let $R \in \mathsf{Regions}$ and let $\sigma \in \Sigma^{\mathsf{Sd}}$. Then $Prob^{\sigma}_{R}(\bigcup_{(C,D) \in \mathcal{F}} Path^{(C,D)}_{ful}(R)) = 1$.*

Let $\mathsf{U} \subseteq \mathsf{Regions}$. The set of strictly-divergent maximal ECs within the set U can be computed using the following algorithm.

1. Compute the set $\mathcal{M}_{\mathsf{U}}$ of maximal ECs of $\mathsf{Reg}[\mathsf{P},\Phi]$ within U.[1] Let $\mathcal{M} = \mathcal{M}_{\mathsf{U}}$.
2. Repeat the following:
   (a) Remove some (C, D) from $\mathcal{M}$.
   (b) Compute $[\![\langle\!\langle N\rangle\!\rangle(\diamond \mathit{tick})]\!]$ obtained from the sub-MDP (C, D).
   (c) Compute the maximal ECs $(C_1, D_1), ..., (C_n, D_n)$ of the sub-MDP (C, D) induced by $[\![\langle\!\langle N\rangle\!\rangle(\diamond \mathit{tick})]\!]$, and add them to $\mathcal{M}$.

   Until $\mathcal{M}$ cannot be changed by the above iteration.

At the termination of the algorithm, the set $\mathcal{M}$ will be the set of strictly-divergent maximal ECs of $\mathsf{Reg}[\mathsf{P},\Phi]$ induced by U. We use $\mathcal{S}_{\mathsf{U}}$ to denote this set.

We then follow the approach of Section 3 by computing the set $\mathcal{S}_{[\![\neg\Phi_1\wedge\Phi_2]\!]}$ of strictly-divergent maximal ECs within the set of states satisfying $\neg\Phi_1 \wedge \Phi_2$. Let $\mathsf{V}_{\neg\Phi_1\wedge\Phi_2} = \bigcup_{(C,D)\in\mathcal{S}_{[\![\neg\Phi_1\wedge\Phi_2]\!]}} C$ be the set of regions corresponding to $\mathcal{S}_{[\![\neg\Phi_1\wedge\Phi_2]\!]}$. We use $\mathsf{V}_{\neg\Phi_1\wedge\Phi_2}$ as an atomic proposition such that $R \models_\Sigma \mathsf{V}_{\neg\Phi_1\wedge\Phi_2}$ if and only if $R \in \mathsf{V}_{\neg\Phi_1\wedge\Phi_2}$. Then we consider the path formula $\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})$. We derive the following facts from Case (A). In the following, the set $\mathsf{T}^{\max}$ is defined with respect to the until formula $\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})$ (rather than with respect to $\Phi_1\mathcal{U}\Phi_2$, as in Case (A)). Let $\mathsf{T} = [\![(\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2}]\!] \cup \mathsf{Regions} \setminus \mathsf{Regions}^+(\Phi_2, (\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})$. From Proposition 6, for $R \in \mathsf{T}^{\max}$ and $\sigma \in \Sigma^{\mathsf{Sd}}$ such that $\mathit{Prob}^\sigma_R(\diamond\mathsf{T}) = 1$ and $\sigma(\mathsf{r}) \in \mathit{Max}(\mathit{last}(\mathsf{r}), \Phi_1\mathcal{U}\Phi_2)$ for each $\mathsf{r} \in \mathit{Path}^\sigma_{\mathit{fin}}(R)$, we have that:

$$p^{\max}_R(\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})) = \mathit{Prob}^\sigma_R(\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})) \ .$$

From Lemma 4, $R \in \mathsf{Regions} \setminus \mathsf{T}^{\max}$ implies that for $\sigma \in \Sigma^{\mathsf{Sd}}$ we have:

$$p^{\max}_R(\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2}))) > \mathit{Prob}^\sigma_R(\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})) \ .$$

We present analogues of Proposition 3 and Proposition 4 adapted for strict region divergence.

Proposition 7. *Let $R \in$ Regions be a region and $\sigma \in \Sigma^{\mathsf{Sd}}$ be a strictly region divergent strategy of $\mathsf{Reg}[\mathsf{P},\Phi]$ such that $\mathit{Prob}^\sigma_R(\diamond\mathsf{T}) = 1$. There exists a strictly region divergent strategy $\sigma' \in \Sigma^{\mathsf{Sd}}$ such that:*

$$\mathit{Prob}^\sigma_R(\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})) \leq \mathit{Prob}^{\sigma'}_R(\Phi_2\mathcal{U}(\Phi_1 \wedge \Phi_2) \vee \Box(\neg\Phi_1 \wedge \Phi_2)) \ .$$

Proposition 8. *Let $R \in$ Regions and $\sigma \in \Sigma^{\mathsf{Sd}}$ be a strictly region divergent strategy of $\mathsf{Reg}[\mathsf{P},\Phi]$. Then:*

$$\mathit{Prob}^\sigma_R(\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})) \geq \mathit{Prob}^\sigma_R(\Phi_2\mathcal{U}(\Phi_1 \wedge \Phi_2) \vee \Box(\neg\Phi_1 \wedge \Phi_2)) \ .$$

Applying Proposition 7, Proposition 8 and Theorem 4, we then obtain the following result.

Theorem 5. *Let $R \in$ Regions be a region. Then:*

$$R \models_{\Sigma^{\mathsf{Sd}}} \mathbb{P}_{\geq\zeta}(\neg\Phi_1\mathcal{U}\neg\Phi_2) \Leftrightarrow 1 - p^{\max}_R(\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})) \geq \zeta$$

$$R \models_{\Sigma^{\mathsf{Sd}}} \mathbb{P}_{>\zeta}(\neg\Phi_1\mathcal{U}\neg\Phi_2) \Leftrightarrow \begin{cases} 1 - p^{\max}_R(\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})) > \zeta \ \textit{if}\ [s] \in \mathsf{T}^{\max} \\ 1 - p^{\max}_R(\Phi_2\mathcal{U}((\Phi_1 \wedge \Phi_2) \vee \mathsf{V}_{\neg\Phi_1\wedge\Phi_2})) \geq \zeta \ \textit{otherwise.} \end{cases}$$

[1] Recall that an algorithm for this purpose can be found in [5].

From Corollary 2, Theorem 4 and Theorem 5, and from the fact that the procedures for computing $\mathsf{T}^{\max}$ and the set of strictly-divergent maximal ECs presented above run in time polynomial in the size of $\mathsf{Reg}[\mathsf{P}, \Phi]$, we then obtain the following result.

Theorem 6. *Let* P *be a PTA and* Φ *be a formula of* PTCTL. *Then the problem of computing the set* $[\![\Phi]\!]$ *for* P *under strictly divergent strategies is EXPTIME-complete.*

5 Conclusions

We have presented optimal model-checking algorithms for PTA for two notions of time divergence. As in previous methods [6], the algorithms rely on a computation of probabilities of satisfying an until property on a finite-state MDP resulting from the classical region graph construction. For probabilistic divergence and properties of the form $\mathbb{P}_{\geq\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ and $\mathbb{P}_{>\zeta}(\Phi_1 \mathcal{U} \Phi_2)$, the algorithms rely also on the computation of maximal ECs in which a strategy can ensure time divergence with probability 1. For strict divergence and for properties of the form $\mathbb{P}_{<\zeta}(\Phi_1 \mathcal{U} \Phi_2)$, we compute the set of states in which the maximal until probability can be obtained by a strictly-divergent strategy; for all other states, strictly-divergent strategies can approximate arbitrarily closely the maximal probability. A similar technique can be used for properties of the form $\mathbb{P}_{>\zeta}(\Phi_1 \mathcal{U} \Phi_2)$ in combination with the computation of maximal ECs in which a strategy can ensure time divergence on all paths. The techniques of this paper are useful when considering models in which there are no lower time bounds on structural loops: these include abstract models of embedded controllers in which lower bounds on certain reaction times are left unspecified. In future work we intend to extend our notions of divergence to controller synthesis, and to consider symbolic, zone-based algorithms for strict divergence.

References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press, Cambridge (1999)
2. Alur, R., Dill, D.L.: A theory of timed automata. TCS 126(2), 183–235 (1994)
3. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking for probabilistic real-time systems. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 115–136. Springer, Heidelberg (1991)
4. Hansson, H.A., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
5. de Alfaro, L.: Formal verification of probabilistic systems. PhD thesis, Stanford University, Department of Computer Science (1997)
6. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. TCS 286, 101–150 (2002)
7. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. TSE 29(6), 524–541 (2003)

8. Donatelli, S., Haddad, S., Sproston, J.: Model checking stochastic and timed properties with CSL^{TA}. TSE 35(2), 224–240 (2009)
9. Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Quantitative model checking of continuous-time Markov chains against timed automata specifications. In: Proc. LICS 2009. IEEE, Los Alamitos (2009)
10. Jensen, H.E.: Model checking probabilistic real time systems. In: Proc. of the 7th Nordic Work. on Progr. Theory, Chalmers Institute of Technology, pp. 247–261 (1996)
11. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. FMSD 29, 33–78 (2006)
12. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. I & C 104(1), 2–34 (1993)
13. Henzinger, T., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. I & C 111(2), 193–244 (1994)
14. Tripakis, S., Yovine, S., Bouajjani, A.: Checking timed Büchi automata emptiness efficiently. FMSD 26(3), 267–292 (2005)
15. Laroussinie, F., Sproston, J.: State explosion in almost-sure probabilistic reachability. IPL 102(6), 236–241 (2007)
16. Kwiatkowska, M., Norman, G., Sproston, J., Wang, F.: Symbolic model checking for probabilistic timed automata. I & C 205(7), 1027–1077 (2007)
17. Baier, C., Kwiatkowska, M.: Model checking for a probabilistic branching time logic with fairness. Dist. Comp. 11(3), 125–155 (1998)
18. Baier, C.: On the algorithmic verification of probabilistic systems, Habilitation thesis, Universität Mannheim (1998)
19. Chatterjee, K., Henzinger, T.A., Prabhu, V.S.: Trading infinite memory for uniform randomness in timed games. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 87–100. Springer, Heidelberg (2008)
20. Beauquier, D.: On probabilistic timed automata. TCS 292(1), 65–84 (2003)
21. Alur, R., Henzinger, T.: Real-Time System = Discrete System + Clock Variables. STTT 1, 86–109 (1997)
22. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: The element of surprise in timed games. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 144–158. Springer, Heidelberg (2003)
23. Kemeny, J.G., Snell, J.L., Knapp, A.W.: Denumerable Markov Chains, 2nd edn. Graduate Texts in Mathematics. Springer, Heidelberg (1976)
24. Jurdziński, M., Laroussinie, F., Sproston, J.: Model checking probabilistic timed automata with one or two clocks. LMCS 4(3), 1–28 (2008)
25. Bianco, A., Alfaro, L.d.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
26. Chatterjee, K., Jurdziński, M., Henzinger, T.: Simple stochastic parity games. In: Baaz, M., Makowsky, J.A. (eds.) CSL 2003. LNCS, vol. 2803, pp. 100–113. Springer, Heidelberg (2003)
27. Alur, R., Henzinger, T., Kupferman, O.: Alternating-time temporal logic. JACM 49, 672–713 (2002)
28. Fecher, H., Huth, M., Piterman, N., Wagner, D.: Hintikka games for PCTL on labeled Markov chains. In: Proc. QEST 2008, pp. 169–178. IEEE, Los Alamitos (2008)

Author Index